Using GNU Fortran

Release 12.0.1 (experimental 20210414)

The gfortran team

Aug 06, 2021
# CONTENTS

1 Copyright  
2 Introduction  
  2.1 About GNU Fortran  
  2.2 GNU Fortran and GCC  
  2.3 Preprocessing and conditional compilation  
  2.4 GNU Fortran and G77  
  2.5 Project Status  
  2.6 Standards  
3 GNU Fortran Command Options  
  3.1 Description  
  3.2 Option summary  
  3.3 Options controlling Fortran dialect  
  3.4 Enable and customize preprocessing  
  3.5 Options to request or suppress errors and warnings  
  3.6 Options for debugging your program or GNU Fortran  
  3.7 Options for directory search  
  3.8 Influencing the linking step  
  3.9 Influencing runtime behavior  
  3.10 Options for code generation conventions  
  3.11 Options for interoperability with other languages  
  3.12 Environment variables affecting gfortran  
4 Runtime: Influencing runtime behavior with environment variables  
  4.1 TMPDIR—Directory for scratch files  
  4.2 GFORTRAN_STDIN_UNIT—Unit number for standard input  
  4.3 GFORTRAN_STDOUT_UNIT—Unit number for standard output  
  4.4 GFORTRAN STDERR_UNIT—Unit number for standard error  
  4.5 GFORTRAN UNBUFFERED ALL—Do not buffer I/O on all units  
  4.6 GFORTRAN UNBUFFERED PRECONNECTED—Do not buffer I/O on preconnected units  
  4.7 GFORTRAN SHOW LOCUS—Show location for runtime errors  
  4.8 GFORTRAN OPTIONAL PLUS—Print leading + where permitted  
  4.9 GFORTRAN LIST_SEPARATOR—Separator for list output  
  4.10 GFORTRAN CONVERT_UNIT—Set endianness for unformatted I/O  
  4.11 GFORTRAN ERROR BACKTRACE—Show backtrace on run-time errors  
  4.12 GFORTRAN FORMATTED BUFFER_SIZE—Set buffer size for formatted I/O  
  4.13 GFORTRAN UNFORMATTED BUFFER_SIZE—Set buffer size for unformatted I/O  
5 Fortran standards status
5.1 Fortran 2003 status ........................................ 35
5.2 Fortran 2008 status ........................................ 37
5.3 Status of Fortran 2018 support .......................... 38

6 Compiler Characteristics .................................. 41
  6.1 KIND Type Parameters ................................... 41
  6.2 Internal representation of LOGICAL variables .... 42
  6.3 Evaluation of logical expressions .................... 42
  6.4 MAX and MIN intrinsics with REAL NaN arguments 42
  6.5 Thread-safety of the runtime library ............... 42
  6.6 Data consistency and durability .................... 43
  6.7 Files opened without an explicit ACTION= specifier 44
  6.8 File operations on symbolic links .................. 44
  6.9 File format of unformatted sequential files ....... 44
  6.10 Asynchronous I/O .................................... 45

7 Extensions .................................................. 47
  7.1 Extensions implemented in GNU Fortran ............ 47
  7.2 Extensions not implemented in GNU Fortran ....... 65

8 Mixed-Language Programming ............................ 69
  8.1 Interoperability with C ............................... 69
  8.2 GNU Fortran Compiler Directives .................... 75
  8.3 Non-Fortran Main Program ............................ 77
  8.4 Naming and argument-passing conventions ......... 81

9 Coarray Programming ..................................... 85
  9.1 Type and enum ABI Documentation .................. 85
  9.2 Function ABI Documentation .......................... 88

10 Intrinsic Procedures .................................. 107
  10.1 Introduction to intrinsic procedures ............... 107
  10.2 ABORT — Abort the program ....................... 107
  10.3 ABS — Absolute value ................................ 108
  10.4 ACCESS — Checks file access modes ............... 109
  10.5 ACHAR — Character in ASCII collating sequence 109
  10.6 ACOS — Arcosine function .......................... 110
  10.7 ACOSD — Arcosine function, degrees ............... 111
  10.8 ACOSH — Inverse hyperbolic cosine function .... 111
  10.9 ADJUSTL — Left adjust a string ................... 112
  10.10 ADJUSTR — Right adjust a string ................. 112
  10.11 AIMAG — Imaginary part of complex number ...... 113
  10.12 AINT — Truncate to a whole number ............... 114
  10.13 ALARM — Execute a routine after a given delay 114
  10.14 ALL — All values in MASK along DIM are true 115
  10.15 ALLOCATED — Status of an allocatable entity .. 116
  10.16 AND — Bitwise logical AND ....................... 117
  10.17 ANINT — Nearest whole number ................... 117
  10.18 ANY — Any value in MASK along DIM is true .... 118
  10.19 ASIN — Arcsine function .......................... 119
  10.20 ASIND — Arcsine function, degrees ............... 120
  10.21 ASINH — Inverse hyperbolic sine function ....... 120
  10.22 ASSOCIATED — Status of a pointer or pointer/target pair 121
  10.23 ATAN — Arctangent function ....................... 122
  10.24 ATAND — Arctangent function, degrees .......... 122
10.25 ATAN2 — Arctangent function .................................................. 123
10.26 ATAN2D — Arctangent function, degrees .................................. 124
10.27 ATANH — Inverse hyperbolic tangent function .......................... 125
10.28 ATOMIC_ADD — Atomic ADD operation .................................... 125
10.29 ATOMIC_AND — Atomic bitwise AND operation ......................... 126
10.30 ATOMIC_CAS — Atomic compare and swap ................................ 127
10.31 ATOMIC_DEFINE — Setting a variable atomically ....................... 127
10.32 ATOMIC_FETCH_ADD — Atomic ADD operation with prior fetch .... 128
10.33 ATOMIC_FETCH_AND — Atomic bitwise AND operation with prior fetch 129
10.34 ATOMIC_FETCH_OR — Atomic bitwise OR operation with prior fetch 130
10.35 ATOMIC_FETCH_XOR — Atomic bitwise XOR operation with prior fetch 130
10.36 ATOMIC_OR — Atomic bitwise OR operation ................................ 131
10.37 ATOMIC_REF — Obtaining the value of a variable atomically .......... 132
10.38 ATOMIC_XOR — Atomic bitwise OR operation ............................ 133
10.39 BACKTRACE — Show a backtrace ............................................. 133
10.40 BESSEL_J0 — Bessel function of the first kind of order 0 .......... 134
10.41 BESSEL_J1 — Bessel function of the first kind of order 1 .......... 134
10.42 BESSEL_JN — Bessel function of the first kind ......................... 135
10.43 BESSEL_Y0 — Bessel function of the second kind of order 0 ....... 136
10.44 BESSEL_Y1 — Bessel function of the second kind of order 1 ....... 136
10.45 BESSEL_YN — Bessel function of the second kind ..................... 137
10.46 BGE — Bitwise greater than or equal to .................................... 138
10.47 BGT — Bitwise greater than ..................................................... 138
10.48 BIT_SIZE — Bit size inquiry function ....................................... 139
10.49 BLE — Bitwise less than or equal to ......................................... 139
10.50 BLT — Bitwise less than .......................................................... 140
10.51 BTTEST — Bit test function ..................................................... 140
10.52 C_ASSOCIATED — Status of a C pointer ................................... 141
10.53 C_F_POINTER — Convert C into Fortran pointer......................... 142
10.54 C_F_PROCPOINTER — Convert C into Fortran procedure pointer .... 142
10.55 C_FUNLOC — Obtain the C address of a procedure ..................... 143
10.56 C_LOC — Obtain the C address of an object ................................ 144
10.57 C_SIZEOF — Size in bytes of an expression ................................ 145
10.58 CEILING — Integer ceiling function ........................................ 145
10.59 CHAR — Character conversion function .................................... 146
10.60 CHDIR — Change working directory ......................................... 147
10.61 CHMOD — Change access permissions of files ........................... 147
10.62 CMPLX — Complex conversion function ..................................... 148
10.63 CO_BROADCAST — Copy a value to all images the current set of images .................................................. 149
10.64 CO_MAX — Maximal value on the current set of images .................. 150
10.65 CO_MIN — Minimal value on the current set of images ................ 150
10.66 CO_REDUCE — Reduction of values on the current set of images .... 151
10.67 CO_SUM — Sum of values on the current set of images .............. 151
10.68 COMMAND_ARGUMENT_COUNT — Get number of command line arguments .................................................. 153
10.69 COMPILER_OPTIONS — Options passed to the compiler ............. 154
10.70 COMPILER_VERSION — Compiler version string ........................ 154
10.71 COMPLEX — Complex conversion function .................................. 155
10.72 CONJG — Complex conjugate function ...................................... 155
10.73 COS — Cosine function ............................................................ 156
10.74 COSD — Cosine function, degrees ............................................ 157
10.75 COSH — Hyperbolic cosine function ......................................... 157
10.76 COTAN — Cotangent function .................................................. 158
10.77 COTAND — Cotangent function, degrees .................................. 159
10.78 COUNT — Count function ........................................................ 159
10.79 CPU_TIME — CPU elapsed time in seconds .................................................. 160
10.80 CSHEFF — Circular shift elements of an array .............................................. 161
10.81 CTIME — Convert a time into a string ......................................................... 162
10.82 DATE_AND_TIME — Date and time subroutine .............................................. 162
10.83 DBLE — Double conversion function .......................................................... 163
10.84 DCMLPX — Double complex conversion function ........................................ 164
10.85 DIGITS — Significant binary digits function ................................................. 164
10.86 DIM — Positive difference ........................................................................... 165
10.87 DOT_PRODUCT — Dot product function ....................................................... 166
10.88 DPROD — Double product function ............................................................ 166
10.89 DREAL — Double real part function ............................................................ 167
10.90 DSHIFT — Combined left shift .................................................................... 168
10.91 DSHIFTR — Combined right shift .................................................................. 168
10.92 DTIME — Execution time subroutine (or function) ..................................... 169
10.93 EOSHEF — End-off shift elements of an array ............................................. 170
10.94 EPSILON — Epsilon function ....................................................................... 170
10.95 ERF — Error function .................................................................................. 171
10.96 ERFC — Error function ............................................................................... 172
10.97 ERF_SCALED — Error function .................................................................. 172
10.98 ETIME — Execution time subroutine (or function) .................................... 173
10.99 EVENT_QUERY — Query whether a coarray event has occurred ............... 173
10.100 EXECUTE_COMMAND_LINE — Execute a shell command ..................... 174
10.101 EXIT — Exit the program with status ....................................................... 175
10.102 EXP — Exponential function ..................................................................... 176
10.103 EXPS — Exponent function ....................................................................... 176
10.104 EXTENDS_TYPE_OF — Query dynamic type for extension .................... 177
10.105 GETDATE — Get the current time as a string ........................................... 177
10.106 FGET — Read a single character in stream mode from stdin .................. 178
10.107 FGETC — Read a single character in stream mode ................................... 179
10.108 FINDLOC — Search an array for a value .................................................. 179
10.109 FLOOR — Integer floor function ............................................................... 180
10.110 FLUSH — Flush I/O unit(s) ....................................................................... 181
10.111 FNUM — File number function .................................................................. 182
10.112 FPUT — Write a single character in stream mode to stdout ...................... 182
10.113 FPUTC — Write a single character in stream mode ................................... 183
10.114 FRACTION — Fractional part of the model representation ....................... 184
10.115 FREE — Frees memory ............................................................................. 184
10.116 FSEEK — Low level file positioning subroutine ......................................... 185
10.117 FSTAT — Get file status ............................................................................ 186
10.118 FTELL — Current stream position .............................................................. 186
10.119 GAMMA — Gamma function ..................................................................... 187
10.120 GERROR — Get last system error message ............................................... 187
10.121 GETARG — Get command line arguments .................................................. 188
10.122 GET_COMMAND — Get the entire command line ..................................... 189
10.123 GET_COMMAND_ARGUMENT — Get command line arguments .......... 189
10.124 GETCWD — Get current working directory .............................................. 190
10.125 GETENV — Get an environmental variable .............................................. 191
10.126 GET_ENVIRONMENT_VARIABLE — Get an environmental variable ...... 191
10.127 GETGID — Group ID function .................................................................. 192
10.128 GETLOG — Get login name ...................................................................... 192
10.129 GETPID — Process ID function .................................................................. 193
10.130 GETUID — User ID function .................................................................... 193
10.131 GMTIME — Convert time to GMT info ...................................................... 194
10.132 HOSTNM — Get system host name ............................................................ 194
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{HUGE}</td>
<td>Largest number of a kind</td>
</tr>
<tr>
<td>\texttt{HYPOT}</td>
<td>Euclidean distance function</td>
</tr>
<tr>
<td>\texttt{ACHAR}</td>
<td>Code in ASCII collating sequence</td>
</tr>
<tr>
<td>\texttt{ALL}</td>
<td>Bitwise AND of array elements</td>
</tr>
<tr>
<td>\texttt{AND}</td>
<td>Bitwise logical and</td>
</tr>
<tr>
<td>\texttt{ANY}</td>
<td>Bitwise OR of array elements</td>
</tr>
<tr>
<td>\texttt{ARGC}</td>
<td>Get the number of command line arguments</td>
</tr>
<tr>
<td>\texttt{BCLR}</td>
<td>Clear bit</td>
</tr>
<tr>
<td>\texttt{BITEX}</td>
<td>Bit extraction</td>
</tr>
<tr>
<td>\texttt{BSET}</td>
<td>Set bit</td>
</tr>
<tr>
<td>\texttt{CHAR}</td>
<td>Character-to-integer conversion function</td>
</tr>
<tr>
<td>\texttt{DATE}</td>
<td>Get current local time subroutine (day/month/year)</td>
</tr>
<tr>
<td>\texttt{EOR}</td>
<td>Bitwise logical exclusive or</td>
</tr>
<tr>
<td>\texttt{ERRNO}</td>
<td>Get the last system error number</td>
</tr>
<tr>
<td>\texttt{IMAGE_INDEX}</td>
<td>Function that converts a co-subscript to an image index</td>
</tr>
<tr>
<td>\texttt{INDEX}</td>
<td>Position of a substring within a string</td>
</tr>
<tr>
<td>\texttt{INT}</td>
<td>Convert to integer type</td>
</tr>
<tr>
<td>\texttt{INT2}</td>
<td>Convert to 16-bit integer type</td>
</tr>
<tr>
<td>\texttt{INT8}</td>
<td>Convert to 64-bit integer type</td>
</tr>
<tr>
<td>\texttt{OR}</td>
<td>Bitwise logical or</td>
</tr>
<tr>
<td>\texttt{PARITY}</td>
<td>Bitwise XOR of array elements</td>
</tr>
<tr>
<td>\texttt{RAND}</td>
<td>Integer pseudo-random number</td>
</tr>
<tr>
<td>\texttt{IS_CONTIGUOUS}</td>
<td>Test whether an array is contiguous</td>
</tr>
<tr>
<td>\texttt{IOSTAT_END}</td>
<td>Test for end-of-file value</td>
</tr>
<tr>
<td>\texttt{IOSTAT_EOR}</td>
<td>Test for end-of-record value</td>
</tr>
<tr>
<td>\texttt{SATTY}</td>
<td>Whether a unit is a terminal device</td>
</tr>
<tr>
<td>\texttt{SHIFT}</td>
<td>Shift bits</td>
</tr>
<tr>
<td>\texttt{SHFTC}</td>
<td>Shift bits circularly</td>
</tr>
<tr>
<td>\texttt{SNAN}</td>
<td>Test for a NaN</td>
</tr>
<tr>
<td>\texttt{TIME}</td>
<td>Get current local time subroutine (hour/minutes/seconds)</td>
</tr>
<tr>
<td>\texttt{KILL}</td>
<td>Send a signal to a process</td>
</tr>
<tr>
<td>\texttt{KIND}</td>
<td>Kind of an entity</td>
</tr>
<tr>
<td>\texttt{BOUND}</td>
<td>Lower dimension bounds of an array</td>
</tr>
<tr>
<td>\texttt{COBOUND}</td>
<td>Lower codimension bounds of an array</td>
</tr>
<tr>
<td>\texttt{LEADZ}</td>
<td>Number of leading zero bits of an integer</td>
</tr>
<tr>
<td>\texttt{LEN}</td>
<td>Length of a character entity</td>
</tr>
<tr>
<td>\texttt{LEN_TRIM}</td>
<td>Length of a character entity without trailing blank characters</td>
</tr>
<tr>
<td>\texttt{LGE}</td>
<td>Lexical greater than or equal</td>
</tr>
<tr>
<td>\texttt{LGT}</td>
<td>Lexical greater than</td>
</tr>
<tr>
<td>\texttt{LINK}</td>
<td>Create a hard link</td>
</tr>
<tr>
<td>\texttt{LE}</td>
<td>Lexical less than or equal</td>
</tr>
<tr>
<td>\texttt{LT}</td>
<td>Lexical less than</td>
</tr>
<tr>
<td>\texttt{NBLNK}</td>
<td>Index of the last non-blank character in a string</td>
</tr>
<tr>
<td>\texttt{LOC}</td>
<td>Returns the address of a variable</td>
</tr>
<tr>
<td>\texttt{LOG}</td>
<td>Natural logarithm function</td>
</tr>
<tr>
<td>\texttt{LOG10}</td>
<td>Base 10 logarithm function</td>
</tr>
<tr>
<td>\texttt{LOG_GAMMA}</td>
<td>Logarithm of the Gamma function</td>
</tr>
<tr>
<td>\texttt{LOGICAL}</td>
<td>Convert to logical type</td>
</tr>
<tr>
<td>\texttt{LONG}</td>
<td>Convert to integer type</td>
</tr>
<tr>
<td>\texttt{SHIFT}</td>
<td>Left shift bits</td>
</tr>
<tr>
<td>\texttt{STAT}</td>
<td>Get file status</td>
</tr>
<tr>
<td>\texttt{TIME}</td>
<td>Convert time to local time info</td>
</tr>
<tr>
<td>\texttt{SMALLOC}</td>
<td>Allocate dynamic memory</td>
</tr>
<tr>
<td>\texttt{MASKL}</td>
<td>Left justified mask</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>10.187 MASKR</td>
<td>Right justified mask</td>
</tr>
<tr>
<td>10.188 MATMUL</td>
<td>Matrix multiplication</td>
</tr>
<tr>
<td>10.189 MAX</td>
<td>Maximum value of an argument list</td>
</tr>
<tr>
<td>10.190 MAXEXPONENT</td>
<td>Maximum exponent of a real kind</td>
</tr>
<tr>
<td>10.191 MAXLOC</td>
<td>Location of the maximum value within an array</td>
</tr>
<tr>
<td>10.192 MAXVAL</td>
<td>Maximum value of an array</td>
</tr>
<tr>
<td>10.193 MCLOCK</td>
<td>Time function</td>
</tr>
<tr>
<td>10.194 MCLOCK8</td>
<td>Time function (64-bit)</td>
</tr>
<tr>
<td>10.195 MERGE</td>
<td>Merge variables</td>
</tr>
<tr>
<td>10.196 MERGE_BITS</td>
<td>Merge of bits under mask</td>
</tr>
<tr>
<td>10.197 MIN</td>
<td>Minimum value of an argument list</td>
</tr>
<tr>
<td>10.198 MINEXONENT</td>
<td>Minimum exponent of a real kind</td>
</tr>
<tr>
<td>10.199 MINLOC</td>
<td>Location of the minimum value within an array</td>
</tr>
<tr>
<td>10.200 MINVAL</td>
<td>Minimum value of an array</td>
</tr>
<tr>
<td>10.201 MOD</td>
<td>Remainder function</td>
</tr>
<tr>
<td>10.202 MODULO</td>
<td>Modulo function</td>
</tr>
<tr>
<td>10.203 MOVE_ALLOC</td>
<td>Move allocation from one object to another</td>
</tr>
<tr>
<td>10.204 MVBITS</td>
<td>Move bits from one integer to another</td>
</tr>
<tr>
<td>10.205 NEAREST</td>
<td>Nearest representable number</td>
</tr>
<tr>
<td>10.206 NEW_LINE</td>
<td>New line character</td>
</tr>
<tr>
<td>10.207 NINT</td>
<td>Nearest whole number</td>
</tr>
<tr>
<td>10.208 NORM2</td>
<td>Euclidean vector norms</td>
</tr>
<tr>
<td>10.209 NOT</td>
<td>Logical negation</td>
</tr>
<tr>
<td>10.210 NULL</td>
<td>Function that returns an disassociated pointer</td>
</tr>
<tr>
<td>10.211 NUM_IMAGES</td>
<td>Function that returns the number of images</td>
</tr>
<tr>
<td>10.212 OR</td>
<td>Bitwise logical OR</td>
</tr>
<tr>
<td>10.213 PACK</td>
<td>Pack an array into an array of rank one</td>
</tr>
<tr>
<td>10.214 PARITY</td>
<td>Reduction with exclusive OR</td>
</tr>
<tr>
<td>10.215 PERROR</td>
<td>Print system error message</td>
</tr>
<tr>
<td>10.216 POPCNT</td>
<td>Number of bits set</td>
</tr>
<tr>
<td>10.217 POPPAR</td>
<td>Parity of the number of bits set</td>
</tr>
<tr>
<td>10.218 PRECISION</td>
<td>Decimal precision of a real kind</td>
</tr>
<tr>
<td>10.219 PRESENT</td>
<td>Determine whether an optional dummy argument is specified</td>
</tr>
<tr>
<td>10.220 PRODUCT</td>
<td>Product of array elements</td>
</tr>
<tr>
<td>10.221 RADIX</td>
<td>Base of a model number</td>
</tr>
<tr>
<td>10.222 RAND</td>
<td>Real pseudo-random number</td>
</tr>
<tr>
<td>10.223 RAND</td>
<td>Real pseudo-random number</td>
</tr>
<tr>
<td>10.224 RANDOM_INIT</td>
<td>Initialize a pseudo-random number generator</td>
</tr>
<tr>
<td>10.225 RANDOM_NUMBER</td>
<td>Pseudo-random number</td>
</tr>
<tr>
<td>10.226 RANDOM_SEED</td>
<td>Initialize a pseudo-random number sequence</td>
</tr>
<tr>
<td>10.227 RANGE</td>
<td>Decimal exponent range</td>
</tr>
<tr>
<td>10.228 RANK</td>
<td>Rank of a data object</td>
</tr>
<tr>
<td>10.229 REAL</td>
<td>Convert to real type</td>
</tr>
<tr>
<td>10.230 RENAME</td>
<td>Rename a file</td>
</tr>
<tr>
<td>10.231 RESHAPE</td>
<td>Function to reshape an array</td>
</tr>
<tr>
<td>10.232 RRSPPACING</td>
<td>Reciprocal of the relative spacing</td>
</tr>
<tr>
<td>10.233 RSHIFT</td>
<td>Right shift bits</td>
</tr>
<tr>
<td>10.234 SAME_TYPE_AS</td>
<td>Query dynamic types for equality</td>
</tr>
<tr>
<td>10.235 SCALE</td>
<td>Scale a real value</td>
</tr>
<tr>
<td>10.236 SCAN</td>
<td>Scan a string for the presence of a set of characters</td>
</tr>
<tr>
<td>10.237 SECONDS</td>
<td>Time function</td>
</tr>
<tr>
<td>10.238 SECOND</td>
<td>CPU time function</td>
</tr>
<tr>
<td>10.240 SELECTED_CHAR_KIND</td>
<td>Choose character kind</td>
</tr>
</tbody>
</table>
10.24 SELECTED_INT_KIND — Choose integer kind .................................................. 258
10.24 SELECTED_REAL_KIND — Choose real kind ...................................................... 259
10.24 SET_EXPONENT — Set the exponent of the model .............................................. 259
10.24 SHAPE — Determine the shape of an array ...................................................... 260
10.24 SHIFTA — Right shift with fill ........................................................................... 261
10.24 SHIFTL — Left shift ............................................................................................. 261
10.24 SHIFTR — Right shift ......................................................................................... 262
10.24 SIGN — Sign copying function ........................................................................... 262
10.24 SIGNAL — Signal handling subroutine (or function) ........................................... 263
10.25 SIN — Sine function ............................................................................................ 264
10.25 SIND — Sine function, degrees ........................................................................... 264
10.25 SINH — Hyperbolic sine function ....................................................................... 265
10.25 SIZE — Determine the size of an array .............................................................. 266
10.25 SIZEOF — Size in bytes of an expression ......................................................... 266
10.25 SLEEP — Sleep for the specified number of seconds .......................................... 267
10.25 SPACING — Smallest distance between two numbers of a given type ............... 267
10.25 SPREAD — Add a dimension to an array ......................................................... 268
10.25 SQRT — Square-root function ............................................................................ 269
10.25 RAN — Reinitialize the random number generator ............................................. 269
10.26 STAT — Get file status ....................................................................................... 270
10.26 STORAGE_SIZE — Storage size in bits ............................................................... 271
10.26 SUM — Sum of array elements ........................................................................... 271
10.26 SYMLNK — Create a symbolic link .................................................................. 272
10.26 SYSTEM — Execute a shell command ............................................................... 272
10.26 SYSTEM_CLOCK — Time function .................................................................. 273
10.26 TAN — Tangent function ................................................................................. 274
10.26 TANH — Hyperbolic tangent function ............................................................... 274
10.26 TANH — Hyperbolic tangent function ................................................................ 275
10.26 THIS_IMAGE — Function that returns the cosubscript index of this image ....... 275
10.27 TIME — Time function ..................................................................................... 276
10.27 TIME8 — Time function (64-bit) ...................................................................... 277
10.27 TINY — Smallest positive number of a real kind .............................................. 277
10.27 TRAILZ — Number of trailing zero bits of an integer ....................................... 278
10.27 TRANSFER — Transfer bit patterns .................................................................. 278
10.27 TRANSPOSE — Transpose an array of rank two ............................................... 279
10.27 TRIM — Remove trailing blank characters of a string ..................................... 279
10.27 TTYNAME — Get the name of a terminal device ................................................ 280
10.27 UBOUND — Upper dimension bounds of an array .......................................... 280
10.27 UCOBOUND — Upper codimension bounds of an array .................................... 281
10.28 UMASK — Set the file creation mask ................................................................. 281
10.28 URLINK — Remove a file from the file system .................................................. 282
10.28 UNPACK — Unpack an array of rank one into an array .................................... 282
10.28 VERIFY — Scan a string for characters not a given set .................................... 283
10.28 XOR — Bitwise logical exclusive OR ............................................................... 284

11 Intrinsic Modules 285
11.1 OpenACC Module OPENACC ............................................................................... 285
11.2 ISO_FORTRAN_ENV ............................................................................................ 285
11.3 ISO_C_BINDING ...................................................................................................... 286
11.4 IEEE modules: IEEE_EXCEPTIONS, IEEE_ARITHMETIC, and IEEE_FEATURES .... 288
11.5 OpenMP Modules OMP_LIB and OMP_LIB_KINDS ........................................ 288

12 Contributing 291
12.1 Contributors to GNU Fortran ................................................................................ 291
This manual documents the use of `gfortran`, the GNU Fortran compiler. You can find in this manual how to invoke `gfortran`, as well as its features and incompatibilities.

**Warning:** This document, and the compiler it describes, are still under development. While efforts are made to keep it up-to-date, it might not accurately reflect the status of the most recent GNU Fortran compiler.
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with the Invariant Sections being GNU General Public License and Funding Free Software, the Front-Cover texts being (a) (see below), and with the Back-Cover Texts being (b) (see below). A copy of the license is included in the gfdl(7) man page.

(a) The FSF’s Front-Cover Text is:

A GNU Manual

(b) The FSF’s Back-Cover Text is:

You have freedom to copy and modify this GNU Manual, like GNU software. Copies published by the Free Software Foundation raise funds for GNU development.
The GNU Fortran compiler front end was designed initially as a free replacement for, or alternative to, the Unix `f95` command; `gfortran` is the command you will use to invoke the compiler.

### 2.1 About GNU Fortran

The GNU Fortran compiler supports the Fortran 77, 90 and 95 standards completely, parts of the Fortran 2003, 2008 and 2018 standards, and several vendor extensions. The development goal is to provide the following features:

- Read a user’s program, stored in a file and containing instructions written in Fortran 77, Fortran 90, Fortran 95, Fortran 2003, Fortran 2008 or Fortran 2018. This file contains *source code*.
- Translate the user’s program into instructions a computer can carry out more quickly than it takes to translate the instructions in the first place. The result after compilation of a program is *machine code*, code designed to be efficiently translated and processed by a machine such as your computer. Humans usually are not as good writing machine code as they are at writing Fortran (or C++, Ada, or Java), because it is easy to make tiny mistakes writing machine code.
- Provide the user with information about the reasons why the compiler is unable to create a binary from the source code. Usually this will be the case if the source code is flawed. The Fortran 90 standard requires that the compiler can point out mistakes to the user. An incorrect usage of the language causes an *error message*.
  
  The compiler will also attempt to diagnose cases where the user’s program contains a correct usage of the language, but instructs the computer to do something questionable. This kind of diagnostics message is called a *warning message*.
- Provide optional information about the translation passes from the source code to machine code. This can help a user of the compiler to find the cause of certain bugs which may not be obvious in the source code, but may be more easily found at a lower level compiler output. It also helps developers to find bugs in the compiler itself.
- Provide information in the generated machine code that can make it easier to find bugs in the program (using a debugging tool, called a *debugger*, such as the GNU Debugger `gdb`).
- Locate and gather machine code already generated to perform actions requested by statements in the user’s program. This machine code is organized into *modules* and is located and *linked* to the user program.

The GNU Fortran compiler consists of several components:

- A version of the `gcc` command (which also might be installed as the system’s `cc` command) that also understands and accepts Fortran source code. The `gcc` command is the *driver* program for all the languages in the GNU Compiler Collection (GCC); With `gcc`, you can compile the source code of any language for which a front end is available in GCC.
• The `gfortran` command itself, which also might be installed as the system’s `f95` command. `gfortran` is just another driver program, but specifically for the Fortran compiler only. The difference with `gcc` is that `gfortran` will automatically link the correct libraries to your program.

• A collection of run-time libraries. These libraries contain the machine code needed to support capabilities of the Fortran language that are not directly provided by the machine code generated by the `gfortran` compilation phase, such as intrinsic functions and subroutines, and routines for interaction with files and the operating system.

• The Fortran compiler itself, (`f95`). This is the GNU Fortran parser and code generator, linked to and interfaced with the GCC backend library. `f95` ‘translates’ the source code to assembler code. You would typically not use this program directly; instead, the `gcc` or `gfortran` driver programs will call it for you.

2.2 GNU Fortran and GCC

GNU Fortran is a part of GCC, the GNU Compiler Collection. GCC consists of a collection of front ends for various languages, which translate the source code into a language-independent form called GENERIC. This is then processed by a common middle end which provides optimization, and then passed to one of a collection of back ends which generate code for different computer architectures and operating systems.

Functionally, this is implemented with a driver program (`gcc`) which provides the command-line interface for the compiler. It calls the relevant compiler front-end program (e.g., `f95` for Fortran) for each file in the source code, and then calls the assembler and linker as appropriate to produce the compiled output. In a copy of GCC which has been compiled with Fortran language support enabled, `gcc` will recognize files with `.f`, `.for`, `.ftn`, `.f90`, `.f95`, `.f03` and `.f08` extensions as Fortran source code, and compile it accordingly. A `gfortran` driver program is also provided, which is identical to `gcc` except that it automatically links the Fortran runtime libraries into the compiled program.

Source files with `.f`, `.for`, `.fpp`, `.ftn`, `.F`, `.FOR`, `.FPP`, and `.FTN` extensions are treated as fixed form. Source files with `.f90`, `.f95`, `.f03`, `.f08`, `.F90`, `.F95`, `.F03` and `.F08` extensions are treated as free form. The capitalized versions of either form are run through pre-processing. Source files with the lower case `.fpp` extension are also run through pre-processing.

This manual specifically documents the Fortran front end, which handles the programming language’s syntax and semantics. The aspects of GCC which relate to the optimization passes and the back-end code generation are documented in the GCC manual. The two manuals together provide a complete reference for the GNU Fortran compiler.

2.3 Preprocessing and conditional compilation

Many Fortran compilers including GNU Fortran allow passing the source code through a C preprocessor (CPP; sometimes also called the Fortran preprocessor, FPP) to allow for conditional compilation. In the case of GNU Fortran, this is the GNU C Preprocessor in the traditional mode. On systems with case-preserving file names, the preprocessor is automatically invoked if the filename extension is `.F`, `.FOR`, `.FTN`, `.fpp`, `.FPP`, `.F90`, `.F95`, `.F03` or `.F08`. To manually invoke the preprocessor on any file, use `-cpp`, to disable preprocessing on files where the preprocessor is run automatically, use `nocpp`.

If a preprocessed file includes another file with the Fortran INCLUDE statement, the included file is not preprocessed. To preprocessed included files, use the equivalent preprocessor statement `#include`.

If GNU Fortran invokes the preprocessor, `__GNUC__` is defined. The macros `__GNUC__, __GNUC_MINOR__` and `__GNUC_PATCHLEVEL__` can be used to determine the version of the compiler. See TopOverviewcppThe C Preprocessor for details.

GNU Fortran supports a number of INTEGER and REAL kind types in additional to the kind types required by the Fortran standard. The availability of any given kind type is architecture dependent. The following pre-defined
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

preprocessor macros can be used to conditionally include code for these additional kind types: __GFC_INT_1__, __GFC_INT_2__, __GFC_INT_8__, __GFC_INT_16__, __GFC_REAL_10__, and __GFC_REAL_16__.

While CPP is the de-facto standard for preprocessing Fortran code, Part 3 of the Fortran 95 standard (ISO/IEC 1539-3:1998) defines Conditional Compilation, which is not widely used and not directly supported by the GNU Fortran compiler. You can use the program coco to preprocess such files (http://www.daniellnagle.com/coco.html).

2.4 GNU Fortran and G77

The GNU Fortran compiler is the successor to g77, the Fortran 77 frontend included in GCC prior to version 4. It is an entirely new program that has been designed to provide Fortran 95 support and extensibility for future Fortran language standards, as well as providing backwards compatibility for Fortran 77 and nearly all of the GNU language extensions supported by g77.

2.5 Project Status

As soon as gfortran can parse all of the statements correctly, it will be in the 'larva' state. When we generate code, the 'puppa' state. When gfortran is done, we'll see if it will be a beautiful butterfly, or just a big bug….

–Andy Vaught, April 2000

The start of the GNU Fortran 95 project was announced on the GCC homepage in March 18, 2000 (even though Andy had already been working on it for a while, of course).

The GNU Fortran compiler is able to compile nearly all standard-compliant Fortran 95, Fortran 90, and Fortran 77 programs, including a number of standard and non-standard extensions, and can be used on real-world programs. In particular, the supported extensions include OpenMP, Cray-style pointers, some old vendor extensions, and several Fortran 2003 and Fortran 2008 features, including TR 15581. However, it is still under development and has a few remaining rough edges. There also is initial support for OpenACC.

At present, the GNU Fortran compiler passes the NIST Fortran 77 Test Suite, and produces acceptable results on the LAPACK Test Suite. It also provides respectable performance on the Polyhedron Fortran compiler benchmarks and the Livermore Fortran Kernels test. It has been used to compile a number of large real-world programs, including the HARMONIE and HIRLAM weather forecasting code and the Tonto quantum chemistry package; see https://gcc.gnu.org/wiki/GfortranApps for an extended list.

Among other things, the GNU Fortran compiler is intended as a replacement for G77. At this point, nearly all programs that could be compiled with G77 can be compiled with GNU Fortran, although there are a few minor known regressions.

The primary work remaining to be done on GNU Fortran falls into three categories: bug fixing (primarily regarding the treatment of invalid code and providing useful error messages), improving the compiler optimizations and the performance of compiled code, and extending the compiler to support future standards—in particular, Fortran 2003, Fortran 2008 and Fortran 2018.
2.6 Standards

The GNU Fortran compiler implements ISO/IEC 1539:1997 (Fortran 95). As such, it can also compile essentially all standard-compliant Fortran 90 and Fortran 77 programs. It also supports the ISO/IEC TR-15581 enhancements to allocatable arrays.

GNU Fortran also have a partial support for ISO/IEC 1539-1:2004 (Fortran 2003), ISO/IEC 1539-1:2010 (Fortran 2008), the Technical Specification Further Interoperability of Fortran with C (ISO/IEC TS 29113:2012). Full support of those standards and future Fortran standards is planned. The current status of the support is can be found in the Fortran 2003 status, Fortran 2008 status and Fortran 2018 status sections of the documentation.

Additionally, the GNU Fortran compilers supports the OpenMP specification (version 4.5 and partial support of the features of the 5.0 version, http://openmp.org/openmp-specifications/). There also is support for the OpenACC specification (targeting version 2.6, http://www.openacc.org/). See https://gcc.gnu.org/wiki/OpenACC for more information.

2.6.1 Varying Length Character Strings

The Fortran 95 standard specifies in Part 2 (ISO/IEC 1539-2:2000) varying length character strings. While GNU Fortran currently does not support such strings directly, there exist two Fortran implementations for them, which work with GNU Fortran. They can be found at http://www.fortran.com/iso_varying_string.f95 and at ftp://ftp.nag.co.uk/sc22wg5/ISO_VARYING_STRING/.

Deferred-length character strings of Fortran 2003 supports part of the features of ISO_VARYING_STRING and should be considered as replacement. (Namely, allocatable or pointers of the type character(len=:).)

Only the most useful options are listed here; see below for the remainder.

For instructions on reporting bugs, see https://gcc.gnu.org/bugs/.

See the Info entry for gfortran for contributors to GCC and GNU Fortran.
CHAPTER THREE

GNU FORTRAN COMMAND OPTIONS

3.1 Description

The gfortran command supports all the options supported by the gcc command. Only options specific to GNU Fortran are documented here.

See GCC Command Options, for information on the non-Fortran-specific aspects of the gcc command (and, therefore, the gfortran command).

All GCC and GNU Fortran options are accepted both by gfortran and by gcc (as well as any other drivers built at the same time, such as g++), since adding GNU Fortran to the GCC distribution enables acceptance of GNU Fortran options by all of the relevant drivers.

In some cases, options have positive and negative forms; the negative form of -ffoo would be -fno-foo. This manual documents only one of these two forms, whichever one is not the default.

3.2 Option summary

3.2.1 Options

Here is a summary of all the options specific to GNU Fortran, grouped by type. Explanations are in the following sections.

*Fortran Language Options*  See *Options controlling Fortran dialect*.

- fall-intrinsics -fallow-argument-mismatch -fallow-invalid-boz -fbackslash

*Preprocessing Options*  See *Enable and customize preprocessing*.

Error and Warning Options  See Options to request or suppress errors and warnings.


Debugging Options  See Options for debugging your program or GNU Fortran.


Directory Options  See Options for directory search.

-Idir  -Jdir  -fintrinsic-modules-path dir

Link Options  See Options for influencing the linking step.

-static-libgfortran

Runtime Options  See Options for influencing runtime behavior.

-fconvert=conversion  -fmax-subrecord-length=length  -frecord-marker=length  -fsign-zero

Interoperability Options  See Options for interoperability.

-fc-prototypes  -fc-prototypes-external

Code Generation Options  See Options for code generation conventions.


3.3 Options controlling Fortran dialect

The following options control the details of the Fortran dialect accepted by the compiler:

-ffree-form, -ffixed-form

Specify the layout used by the source file. The free form layout was introduced in Fortran 90. Fixed form was traditionally used in older Fortran programs. When neither option is specified, the source form is determined by the file extension.

-fall-intrinsics

This option causes all intrinsic procedures (including the GNU-specific extensions) to be accepted. This can be useful with std=f95 to force standard-compliance but get access to the full range of intrinsics available with gfortran. As a consequence, -Wintrinsics-std will be ignored and no user-defined procedure with the same name as any intrinsic will be called except when it is explicitly declared EXTERNAL.
-fallow-argument-mismatch
Some code contains calls to external procedures with mismatches between the calls and the procedure definition, or with mismatches between different calls. Such code is non-conforming, and will usually be flagged with an error. This option degrades the error to a warning, which can only be disabled by disabling all warnings via -w. Only a single occurrence per argument is flagged by this warning. -fallow-argument-mismatch is implied by -std=legacy.

Using this option is strongly discouraged. It is possible to provide standard-conforming code which allows different types of arguments by using an explicit interface and TYPE(*).

-fallow-invalid-boz
A BOZ literal constant can occur in a limited number of contexts in standard conforming Fortran. This option degrades an error condition to a warning, and allows a BOZ literal constant to appear where the Fortran standard would otherwise prohibit its use.

-fd-lines-as-code, -fd-lines-as-comments
Enable special treatment for lines beginning with d or D in fixed form sources. If the -fd-lines-as-code option is given they are treated as if the first column contained a blank. If the -fd-lines-as-comments option is given, they are treated as comment lines.

-fdec
DEC compatibility mode. Enables extensions and other features that mimic the default behavior of older compilers (such as DEC). These features are non-standard and should be avoided at all costs. For details on GNU Fortran's implementation of these extensions see the full documentation.

Other flags enabled by this switch are: -fdollar-ok -fcray-pointer -fdec-char-conversions -fdec-structure -fdec-intrinsic-ints -fdec-static -fdec-math -fdec-include -fdec-blank-format-item -fdec-format-defaults

If -fd-lines-as-code / -fd-lines-as-comments are unset, then -fdec also sets -fd-lines-as-comments.

-fdec-char-conversions
Enable the use of character literals in assignments and DATA statements for non-character variables.

-fdec-structure
Enable DEC STRUCTURE and RECORD as well as UNION, MAP, and dot (‘.’) as a member separator (in addition to ‘%’). This is provided for compatibility only; Fortran 90 derived types should be used instead where possible.

-fdec-intrinsic-ints
Enable B/I/J/K kind variants of existing integer functions (e.g. BIAND, IIAND, JIAND, etc…). For a complete list of intrinsics see the full documentation.

-fdec-math
Enable legacy math intrinsics such as COTAN and degree-valued trigonometric functions (e.g. TAND, ATAND, etc…) for compatability with older code.

-fdec-static
Enable DEC-style STATIC and AUTOMATIC attributes to explicitly specify the storage of variables and other objects.

-fdec-include
Enable parsing of INCLUDE as a statement in addition to parsing it as INCLUDE line. When parsed as INCLUDE statement, INCLUDE does not have to be on a single line and can use line continuations.

-fdec-format-defaults
Enable format specifiers F, G and I to be used without width specifiers, default widths will be used instead.

-fdec-blank-format-item
Enable a blank format item at the end of a format specification i.e. nothing following the final comma.

3.3. Options controlling Fortran dialect
-fdollar-ok
Allow $ as a valid non-first character in a symbol name. Symbols that start with $ are rejected since it is unclear which rules to apply to implicit typing as different vendors implement different rules. Using $ in IMPLICIT statements is also rejected.

-fbackslash
Change the interpretation of backslashes in string literals from a single backslash character to 'C-style' escape characters. The following combinations are expanded \a, \b, \f, \n, \r, \t, \v, \w, and \0 to the ASCII characters alert, backspace, form feed, newline, carriage return, horizontal tab, vertical tab, backslash, and NUL, respectively. Additionally, \xnn, \xnnnn and \xnnnnnnnn (where each n is a hexadecimal digit) are translated into the Unicode characters corresponding to the specified code points. All other combinations of a character preceded by are unexpanded.

-module-private
Set the default accessibility of module entities to PRIVATE. Use-associated entities will not be accessible unless they are explicitly declared as PUBLIC.

-fixed-line-length-n
Set column after which characters are ignored in typical fixed-form lines in the source file, and, unless -fno-pad-source, through which spaces are assumed (as if padded to that length) after the ends of short fixed-form lines.

Popular values for n include 72 (the standard and the default), 80 (card image), and 132 (corresponding to 'extended-source' options in some popular compilers). n may also be none, meaning that the entire line is meaningful and that continued character constants never have implicit spaces appended to them to fill out the line. -ffixed-line-length-0 means the same thing as -ffixed-line-length-none.

-fno-pad-source
By default fixed-form lines have spaces assumed (as if padded to that length) after the ends of short fixed-form lines. This is not done either if -ffixed-line-length-0, -ffixed-line-length-none or if -fno-pad-source option is used. With any of those options continued character constants never have implicit spaces appended to them to fill out the line.

-free-line-length-n
Set column after which characters are ignored in typical free-form lines in the source file. The default value is 132. n may be none, meaning that the entire line is meaningful. -ffree-line-length-0 means the same thing as -ffree-line-length-none.

-fmax-identifier-length=n
Specify the maximum allowed identifier length. Typical values are 31 (Fortran 95) and 63 (Fortran 2003 and Fortran 2008).

-fimplicit-none
Specify that no implicit typing is allowed, unless overridden by explicit IMPLICIT statements. This is the equivalent of adding implicit none to the start of every procedure.

-fcay-pointer
Enable the Cray pointer extension, which provides C-like pointer functionality.

-fopenacc
Enable the OpenACC extensions. This includes OpenACC !$acc directives in free form and c$acc, *$acc and !$acc directives in fixed form, !$ conditional compilation sentinels in free form and c$, *, $ and !$ sentinels in fixed form, and when linking arranges for the OpenACC runtime library to be linked in.

-fopenmp
Enable the OpenMP extensions. This includes OpenMP !$omp directives in free form and c$omp, *$omp and !$omp directives in fixed form, !$ conditional compilation sentinels in free form and c$, *, $ and !$ sentinels in fixed form, and when linking arranges for the OpenMP runtime library to be linked in. The option -fopenmp implies -frecursive.
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

- **-fno-range-check**
  Disable range checking on results of simplification of constant expressions during compilation. For example, GNU Fortran will give an error at compile time when simplifying a = 1. / 0. With this option, no error will be given and a will be assigned the value +Infinity. If an expression evaluates to a value outside of the relevant range of [−HUGE():HUGE()], then the expression will be replaced by −Inf or +Inf as appropriate. Similarly, DATA i/Z'FFFFFFFF'/ will result in an integer overflow on most systems, but with -fno-range-check the value will 'wrap around' and i will be initialized to -1 instead.

- **-fdefault-integer-8**
  Set the default integer and logical types to an 8 byte wide type. This option also affects the kind of integer constants like 42. Unlike -finteger-4-integer-8, it does not promote variables with explicit kind declaration.

- **-fdefault-real-8**
  Set the default real type to an 8 byte wide type. This option also affects the kind of non-double real constants like 1.0. This option promotes the default width of DOUBLE PRECISION and double real constants like 1.d0 to 16 bytes if possible. If -fdefault-double-8 is given along with fdefault-real-8, DOUBLE PRECISION and double real constants are not promoted. Unlike -freal-4-real-8, fdefault-real-8 does not promote variables with explicit kind declarations.

- **-fdefault-real-10**
  Set the default real type to an 10 byte wide type. This option also affects the kind of non-double real constants like 1.0. This option promotes the default width of DOUBLE PRECISION and double real constants like 1.d0 to 16 bytes if possible. If -fdefault-double-8 is given along with fdefault-real-10, DOUBLE PRECISION and double real constants are not promoted. Unlike -freal-4-real-10, fdefault-real-10 does not promote variables with explicit kind declarations.

- **-fdefault-real-16**
  Set the default real type to an 16 byte wide type. This option also affects the kind of non-double real constants like 1.0. This option promotes the default width of DOUBLE PRECISION and double real constants like 1.d0 to 16 bytes if possible. If -fdefault-double-8 is given along with fdefault-real-16, DOUBLE PRECISION and double real constants are not promoted. Unlike -freal-4-real-16, fdefault-real-16 does not promote variables with explicit kind declarations.

- **-fdefault-double-8**
  Set the DOUBLE PRECISION type and double real constants like 1.d0 to an 8 byte wide type. Do nothing if this is already the default. This option prevents -fdefault-real-8, -fdefault-real-10, and -fdefault-real-16, from promoting DOUBLE PRECISION and double real constants like 1.d0 to 16 bytes.

- **-finteger-4-integer-8**
  Promote all INTEGER(KIND=4) entities to an INTEGER(KIND=8) entities. If KIND=8 is unavailable, then an error will be issued. This option should be used with care and may not be suitable for your codes. Areas of possible concern include calls to external procedures, alignment in EQUIVALENCE and/or COMMON, generic interfaces, BOZ literal constant conversion, and I/O. Inspection of the intermediate representation of the translated Fortran code, produced by -fdump-tree-original, is suggested.

- **-freal-4-real-8, -freal-4-real-10, -freal-4-real-16, -freal-8-real-4, -freal-8-real-10, -freal-8-real-16**
  Promote all REAL(KIND=M) entities to REAL(KIND=N) entities. If REAL(KIND=N) is unavailable, then an error will be issued. The -freal-4- flags also affect the default real kind and the -freal-8- flags also the double-precision real kind. All other real-kind types are unaffected by this option. The promotion is also applied to real literal constants of default and double-precision kind and a specified kind number of 4 or 8, respectively. However, -fdefault-real-8, -fdefault-real-10, -fdefault-real-16, and -fdefault-double-8 take precedence for the default and double-precision real kinds, both for real literal constants and for declarations without a kind number. Note that for REAL(KIND=KIND(1.0)) the literal may get promoted and then the result may get promoted again. These options should be used with care and may not be suitable for your codes. Areas of possible concern include calls to external procedures, alignment in EQUIVALENCE
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

and/or COMMON, generic interfaces, BOZ literal constant conversion, and I/O and calls to intrinsic procedures when passing a value to the kind= dummy argument. Inspection of the intermediate representation of the translated Fortran code, produced by -fdump-fortran-original or -fdump-tree-original, is suggested.

-std=std
Specify the standard to which the program is expected to conform, which may be one of f95, f2003, f2008, f2018, gnu, or legacy. The default value for std is gnu, which specifies a superset of the latest Fortran standard that includes all of the extensions supported by GNU Fortran, although warnings will be given for obsolete extensions not recommended for use in new code. The legacy value is equivalent but without the warnings for obsolete extensions, and may be useful for old non-standard programs. The f95, f2003, f2008, and f2018 values specify strict conformance to the Fortran 95, Fortran 2003, Fortran 2008 and Fortran 2018 standards, respectively; errors are given for all extensions beyond the relevant language standard, and warnings are given for the Fortran 77 features that are permitted but obsolescent in later standards. The deprecated option -std=f2008ts acts as an alias for -std=f2018. It is only present for backwards compatibility with earlier gfortran versions and should not be used any more.

-ftest-forall-temp
Enhance test coverage by forcing most forall assignments to use temporary.

3.4 Enable and customize preprocessing

Preprocessor related options. See section Preprocessing and conditional compilation for more detailed information on preprocessing in gfortran.

-cpp, -nocpp
Enable preprocessing. The preprocessor is automatically invoked if the file extension is .fpp, .FPP, .F, .FOR, .FTN, .F90, .F95, .F03 or .F08. Use this option to manually enable preprocessing of any kind of Fortran file.

To disable preprocessing of files with any of the above listed extensions, use the negative form: -nocpp.

The preprocessor is run in traditional mode. Any restrictions of the file-format, especially the limits on line length, apply for preprocessed output as well, so it might be advisable to use the -ffree-line-length-none or -ffixed-line-length-none options.

-dM
Instead of the normal output, generate a list of '#define' directives for all the macros defined during the execution of the preprocessor, including predefined macros. This gives you a way of finding out what is predefined in your version of the preprocessor. Assuming you have no file foo.f90, the command

```
touch foo.f90; gfortran -cpp -E -dM foo.f90
```

will show all the predefined macros.

-dD
Like -dM except in two respects: it does not include the predefined macros, and it outputs both the #define directives and the result of preprocessing. Both kinds of output go to the standard output file.

-dN
Like -dD, but emit only the macro names, not their expansions.

-dU
Like dD except that only macros that are expanded, or whose definedness is tested in preprocessor directives, are output; the output is delayed until the use or test of the macro; and '#undef' directives are also output for macros tested but undefined at the time.

-dI
Output '#include' directives in addition to the result of preprocessing.
-fworking-directory
Enable generation of linemarkers in the preprocessor output that will let the compiler know the current working directory at the time of preprocessing. When this option is enabled, the preprocessor will emit, after the initial linemarker, a second linemarker with the current working directory followed by two slashes. GCC will use this directory, when it is present in the preprocessed input, as the directory emitted as the current working directory in some debugging information formats. This option is implicitly enabled if debugging information is enabled, but this can be inhibited with the negated form -fno-working-directory. If the -P flag is present in the command line, this option has no effect, since no #line directives are emitted whatsoever.

-idirafter dir
Search dir for include files, but do it after all directories specified with -I and the standard system directories have been exhausted. dir is treated as a system include directory. If dir begins with =, then the = will be replaced by the sysroot prefix; see --sysroot and -isysroot.

-imultilib dir
Use dir as a subdirectory of the directory containing target-specific C++ headers.

-iprefix prefix
Specify prefix as the prefix for subsequent -i with prefix options. If the prefix represents a directory, you should include the final '/'.

-isysroot dir
This option is like the --sysroot option, but applies only to header files. See the --sysroot option for more information.

-iquote dir
Search dir only for header files requested with #include "file"; they are not searched for #include <file>, before all directories specified by -I and before the standard system directories. If dir begins with =, then the = will be replaced by the sysroot prefix; see --sysroot and -isysroot.

-isystem dir
Search dir for header files, after all directories specified by -I but before the standard system directories. Mark it as a system directory, so that it gets the same special treatment as is applied to the standard system directories. If dir begins with =, then the = will be replaced by the sysroot prefix; see --sysroot and -isysroot.

-nostdinc
Do not search the standard system directories for header files. Only the directories you have specified with -I options (and the directory of the current file, if appropriate) are searched.

-undef
Do not predefine any system-specific or GCC-specific macros. The standard predefined macros remain defined.

-A predicate=answer
Make an assertion with the predicate predicate and answer answer. This form is preferred to the older form -A predicate(answer), which is still supported, because it does not use shell special characters.

-A-predicate=answer
Cancel an assertion with the predicate predicate and answer answer.

-C
Do not discard comments. All comments are passed through to the output file, except for comments in processed directives, which are deleted along with the directive.

You should be prepared for side effects when using -C; it causes the preprocessor to treat comments as tokens in their own right. For example, comments appearing at the start of what would be a directive line have the effect of turning that line into an ordinary source line, since the first token on the line is no longer a ' #'.

Warning: this currently handles C-Style comments only. The preprocessor does not yet recognize Fortran-style comments.
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

-CC
Do not discard comments, including during macro expansion. This is like -C, except that comments contained within macros are also passed through to the output file where the macro is expanded.

In addition to the side-effects of the -C option, the -CC option causes all C++-style comments inside a macro to be converted to C-style comments. This is to prevent later use of that macro from inadvertently commenting out the remainder of the source line. The -CC option is generally used to support lint comments.

Warning: this currently handles C- and C++-Style comments only. The preprocessor does not yet recognize Fortran-style comments.

-Dname
Predefine name as a macro, with definition 1.

-Dname=definition
The contents of definition are tokenized and processed as if they appeared during translation phase three in a '#define' directive. In particular, the definition will be truncated by embedded newline characters.

If you are invoking the preprocessor from a shell or shell-like program you may need to use the shell’s quoting syntax to protect characters such as spaces that have a meaning in the shell syntax.

If you wish to define a function-like macro on the command line, write its argument list with surrounding parentheses before the equals sign (if any). Parentheses are meaningful to most shells, so you will need to quote the option. With sh and csh, -D'name(args...)=definition' works.

-D and -U options are processed in the order they are given on the command line. All -imacros file and -include file options are processed after all -D and -U options.

-H
Print the name of each header file used, in addition to other normal activities. Each name is indented to show how deep in the '#include' stack it is.

-P
Inhibit generation of linemarkers in the output from the preprocessor. This might be useful when running the preprocessor on something that is not C code, and will be sent to a program which might be confused by the linemarkers.

-Uname
Cancel any previous definition of name, either built in or provided with a -D option.

3.5 Options to request or suppress errors and warnings

Errors are diagnostic messages that report that the GNU Fortran compiler cannot compile the relevant piece of source code. The compiler will continue to process the program in an attempt to report further errors to aid in debugging, but will not produce any compiled output.

Warnings are diagnostic messages that report constructions which are not inherently erroneous but which are risky or suggest there is likely to be a bug in the program. Unless -Werror is specified, they do not prevent compilation of the program.

You can request many specific warnings with options beginning -W, for example -Wimplicit to request warnings on implicit declarations. Each of these specific warning options also has a negative form beginning -Wno- to turn off warnings; for example, -Wno-implicit. This manual lists only one of the two forms, whichever is not the default.

These options control the amount and kinds of errors and warnings produced by GNU Fortran:

-fmax-errors=n
Limits the maximum number of error messages to n, at which point GNU Fortran bails out rather than attempting to continue processing the source code. If n is 0, there is no limit on the number of error messages produced.
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

-**fsyntax-only**

Check the code for syntax errors, but do not actually compile it. This will generate module files for each module present in the code, but no other output file.

-**-Wpedantic, -pedantic**

Issue warnings for uses of extensions to Fortran. -pedantic also applies to C-language constructs where they occur in GNU Fortran source files, such as use of `\e` in a character constant within a directive like `#include`.

Valid Fortran programs should compile properly with or without this option. However, without this option, certain GNU extensions and traditional Fortran features are supported as well. With this option, many of them are rejected.

Some users try to use -pedantic to check programs for conformance. They soon find that it does not do quite what they want—it finds some nonstandard practices, but not all. However, improvements to GNU Fortran in this area are welcome.

This should be used in conjunction with -std=f95, -std=f2003, -std=f2008 or -std=f2018.

-**-pedantic-errors**

Like -pedantic, except that errors are produced rather than warnings.

-**-Wall**

Enables commonly used warning options pertaining to usage that we recommend avoiding and that we believe are easy to avoid. This currently includes -Walliasing, -Wampersand, -Wconversion, -Wsurprising, -Wc-binding-type, -Wintrinsics-std, -Wtabs, -Wintrinsic-shadow, -Wline-truncation, -Wtarget-lifetime, -Winteger-division, -Wreal-q-constant, -Wunused and -Wundefined-do-loop.

-**-Walliasing**

Warn about possible aliasing of dummy arguments. Specifically, it warns if the same actual argument is associated with a dummy argument with `INTENT(IN)` and a dummy argument with `INTENT(OUT)` in a call with an explicit interface.

The following example will trigger the warning.

```fortran
interface
  subroutine bar(a,b)
    integer, intent(in) :: a
    integer, intent(out) :: b
  end subroutine
end interface
integer :: a

call bar(a,a)
```

-**-Wampersand**

Warn about missing ampersand in continued character constants. The warning is given with -Wampersand, -pedantic, -std=f95, -std=f2003, -std=f2008 and -std=f2018. Note: With no ampersand given in a continued character constant, GNU Fortran assumes continuation at the first non-comment, non-whitespace character after the ampersand that initiated the continuation.

-**-Warray-temporaries**

Warn about array temporaries generated by the compiler. The information generated by this warning is sometimes useful in optimization, in order to avoid such temporaries.

-**-Wc-binding-type**

Warn if the a variable might not be C interoperable. In particular, warn if the variable has been declared using an intrinsic type with default kind instead of using a kind parameter defined for C interoperability in the intrinsic ISO_C_Binding module. This option is implied by -Wall.
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

-Wcharacter-truncation
Warn when a character assignment will truncate the assigned string.

-Wline-truncation
Warn when a source code line will be truncated. This option is implied by -Wall. For free-form source code, the default is -Werror=line-truncation such that truncations are reported as error.

-Wconversion
Warn about implicit conversions that are likely to change the value of the expression after conversion. Implied by -Wall.

-Wconversion-extra
Warn about implicit conversions between different types and kinds. This option does not imply -Wconversion.

-Wextra
Enables some warning options for usages of language features which may be problematic. This currently includes -Wcompare-reals, -Wunused-parameter and -Wdo-subscript.

-Wfrontend-loop-interchange
Warn when using -ffrontend-loop-interchange for performing loop interchanges.

-Wimplicit-interface
Warn if a procedure is called without an explicit interface. Note this only checks that an explicit interface is present. It does not check that the declared interfaces are consistent across program units.

-Wimplicit-procedure
Warn if a procedure is called that has neither an explicit interface nor has been declared as EXTERNAL.

-Winteger-division
Warn if a constant integer division truncates its result. As an example, 3/5 evaluates to 0.

-Wintrinsics-std
Warn if gfortran finds a procedure named like an intrinsic not available in the currently selected standard (with -std) and treats it as EXTERNAL procedure because of this. -fall-intrinsics can be used to never trigger this behavior and always link to the intrinsic regardless of the selected standard.

-Wno-overwrite-recursive
Do not warn when -fno-automatic is used with -frecursive. Recursion will be broken if the relevant local variables do not have the attribute AUTOMATIC explicitly declared. This option can be used to suppress the warning when it is known that recursion is not broken. Useful for build environments that use -Werror.

-Wreal-q-constant
Produce a warning if a real-literal-constant contains a q exponent-letter.

-Wsurprising
Produce a warning when ‘suspicous’ code constructs are encountered. While technically legal these usually indicate that an error has been made.

This currently produces a warning under the following circumstances:

• An INTEGER SELECT construct has a CASE that can never be matched as its lower value is greater than its upper value.

• A LOGICAL SELECT construct has three CASE statements.

• A TRANSFER specifies a source that is shorter than the destination.

• The type of a function result is declared more than once with the same type. If -pedantic or standard-conforming mode is enabled, this is an error.

• A CHARACTER variable is declared with negative length.
-Wtabs
By default, tabs are accepted as whitespace, but tabs are not members of the Fortran Character Set. For continuation lines, a tab followed by a digit between 1 and 9 is supported. `-Wtabs` will cause a warning to be issued if a tab is encountered. Note, `-Wtabs` is active for `--pedantic`, `--std=f95`, `--std=f2003`, `--std=f2008`, `--std=f2018` and `--Wall`.

-Wundefined-do-loop
Warn if a DO loop with step either 1 or -1 yields an underflow or an overflow during iteration of an induction variable of the loop. This option is implied by `--Wall`.

-Wunderflow
Produce a warning when numerical constant expressions are encountered, which yield an UNDERFLOW during compilation. Enabled by default.

-Wintrinsic-shadow
Warn if a user-defined procedure or module procedure has the same name as an intrinsic; in this case, an explicit interface or `EXTERNAL` or `INTRINSIC` declaration might be needed to get calls later resolved to the desired intrinsic/procedure. This option is implied by `--Wall`.

-Wuse-without-only
Warn if a `USE` statement has no `ONLY` qualifier and thus implicitly imports all public entities of the used module.

-Wunused-dummy-argument
Warn about unused dummy arguments. This option is implied by `--Wall`.

-Wunused-parameter
Contrary to gcc's meaning of `--unused-parameter`, gfortran's implementation of this option does not warn about unused dummy arguments (see `--unused-dummy-argument`), but about unused PARAMETER values. `-Wunused-parameter` is implied by `--Wextra` if also `--Wunused` or `--Wall` is used.

-Walign-commons
By default, gfortran warns about any occasion of variables being padded for proper alignment inside a COMMON block. This warning can be turned off via `--no-align-commons`. See also `--falign-commons`.

-Wfunction-elimination
Warn if any calls to impure functions are eliminated by the optimizations enabled by the `--ffrontend-optimize` option. This option is implied by `--Wextra`.

-Wrealloc-lhs
Warn when the compiler might insert code to for allocation or reallocation of an allocatable array variable of intrinsic type in intrinsic assignments. In hot loops, the Fortran 2003 reallocation feature may reduce the performance. If the array is already allocated with the correct shape, consider using a whole-array array-spec (e.g. `(i,j,:,:)`) for the variable on the left-hand side to prevent the reallocation check. Note that in some cases the warning is shown, even if the compiler will optimize reallocation checks away. For instance, when the right-hand side contains the same variable multiplied by a scalar. See also `--frealloc-lhs`.

-Wrealloc-lhs-all
Warn when the compiler inserts code to for allocation or reallocation of an allocatable variable; this includes scalars and derived types.

-Wcompare-reals
Warn when comparing real or complex types for equality or inequality. This option is implied by `--Wextra`.

-Wtarget-lifetime
Warn if the pointer in a pointer assignment might be longer than the its target. This option is implied by `--Wall`.

-Wzerotrip
Warn if a DO loop is known to execute zero times at compile time. This option is implied by `--Wall`. 

3.5. Options to request or suppress errors and warnings
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

- **-Wdo-subscript**
  Warn if an array subscript inside a DO loop could lead to an out-of-bounds access even if the compiler cannot prove that the statement is actually executed, in cases like

  ```fortran
  real a(3)
do i=1,4
   if (condition(i)) then
      a(i) = 1.2
   end if
  end do
  ```

  This option is implied by **-Wextra**.

- **-Werror**
  Turns all warnings into errors.

See Options to Request or Suppress Errors and Warnings, for information on more options offered by the GBE shared by `gfortran`, `gcc` and other GNU compilers.

Some of these have no effect when compiling programs written in Fortran.

### 3.6 Options for debugging your program or GNU Fortran

GNU Fortran has various special options that are used for debugging either your program or the GNU Fortran compiler.

- **-fdump-fortran-original**
  Output the internal parse tree after translating the source program into internal representation. This option is mostly useful for debugging the GNU Fortran compiler itself. The output generated by this option might change between releases. This option may also generate internal compiler errors for features which have only recently been added.

- **-fdump-fortran-optimized**
  Output the parse tree after front-end optimization. Mostly useful for debugging the GNU Fortran compiler itself. The output generated by this option might change between releases. This option may also generate internal compiler errors for features which have only recently been added.

- **-fdump-parse-tree**
  Output the internal parse tree after translating the source program into internal representation. Mostly useful for debugging the GNU Fortran compiler itself. The output generated by this option might change between releases. This option may also generate internal compiler errors for features which have only recently been added. This option is deprecated; use **-fdump-fortran-original** instead.

- **-fdebug-aux-vars**
  Renames internal variables created by the gfortran front end and makes them accessible to a debugger. The name of the internal variables then start with upper-case letters followed by an underscore. This option is useful for debugging the compiler's code generation together with **-fdump-tree-original** and enabling debugging of the executable program by using `-g` or `-ggdb3`.

- **-fdump-fortran-global**
  Output a list of the global identifiers after translating into middle-end representation. Mostly useful for debugging the GNU Fortran compiler itself. The output generated by this option might change between releases. This option may also generate internal compiler errors for features which have only recently been added.

- **-ffpe-trap=list**
  Specify a list of floating point exception traps to enable. On most systems, if a floating point exception occurs and the trap for that exception is enabled, a SIGFPE signal will be sent and the program being aborted, producing a core file useful for debugging. `list` is a (possibly empty) comma-separated list of the following exceptions: `invalid` (invalid floating point operation, such as `SQRT(-1.0)`), `zero` (division by zero), `overflow` (overflow in a
floating point operation), underflow (underflow in a floating point operation), inexact (loss of precision during operation), and denormal (operation performed on a denormal value). The first five exceptions correspond to the five IEEE 754 exceptions, whereas the last one (denormal) is not part of the IEEE 754 standard but is available on some common architectures such as x86.

The first three exceptions (invalid, zero, and overflow) often indicate serious errors, and unless the program has provisions for dealing with these exceptions, enabling traps for these three exceptions is probably a good idea.

If the option is used more than once in the command line, the lists will be joined: ‘ffpe-trap=list1 ffpe-trap=list2’ is equivalent to ffpe-trap=list1,list2.

Note that once enabled an exception cannot be disabled (no negative form).

Many, if not most, floating point operations incur loss of precision due to rounding, and hence the ffpe-trap=inexact is likely to be uninteresting in practice.

By default no exception traps are enabled.

-ffpe-summary=list

Specify a list of floating-point exceptions, whose flag status is printed to ERROR_UNIT when invoking STOP and ERROR STOP. list can be either none, all or a comma-separated list of the following exceptions: invalid, zero, overflow, underflow, inexact and denormal. (See -ffpe-trap for a description of the exceptions.)

If the option is used more than once in the command line, only the last one will be used.

By default, a summary for all exceptions but inexact is shown.

-fno-backtrace

When a serious runtime error is encountered or a deadly signal is emitted (segmentation fault, illegal instruction, bus error, floating-point exception, and the other POSIX signals that have the action core), the Fortran runtime library tries to output a backtrace of the error. -fno-backtrace disables the backtrace generation. This option only has influence for compilation of the Fortran main program.

See Options for Debugging Your Program or GCC, for more information on debugging options.

3.7 Options for directory search

These options affect how GNU Fortran searches for files specified by the INCLUDE directive and where it searches for previously compiled modules.

It also affects the search paths used by cpp when used to preprocess Fortran source.

-Idir

These affect interpretation of the INCLUDE directive (as well as of the #include directive of the cpp preprocessor).

Also note that the general behavior of -I and INCLUDE is pretty much the same as of −I with #include in the cpp preprocessor, with regard to looking for header.gcc files and other such things.

This path is also used to search for .mod files when previously compiled modules are required by a USE statement.

See Options for Directory Search, for information on the −I option.

-Jdir

This option specifies where to put .mod files for compiled modules. It is also added to the list of directories to searched by an USE statement.

The default is the current directory.
-fintrinsic-modules-path dir

This option specifies the location of pre-compiled intrinsic modules, if they are not in the default location expected by the compiler.

3.8 Influencing the linking step

These options come into play when the compiler links object files into an executable output file. They are meaningless if the compiler is not doing a link step.

- static-libgfortran

On systems that provide libgfortran as a shared and a static library, this option forces the use of the static version. If no shared version of libgfortran was built when the compiler was configured, this option has no effect.

3.9 Influencing runtime behavior

These options affect the runtime behavior of programs compiled with GNU Fortran.

- fconvert=conversion

Specify the representation of data for unformatted files. Valid values for conversion are: native, the default; swap, swap between big- and little-endian; big-endian, use big-endian representation for unformatted files; little-endian, use little-endian representation for unformatted files.

This option has an effect only when used in the main program. The CONVERT specifier and the GFORTRAN_CONVERT_UNIT environment variable override the default specified by -fconvert.

- frecord-marker=length

Specify the length of record markers for unformatted files. Valid values for length are 4 and 8. Default is 4. This is different from previous versions of gfortran, which specified a default record marker length of 8 on most systems. If you want to read or write files compatible with earlier versions of gfortran, use -frecord-marker=8.

- fmax-subrecord-length=length

Specify the maximum length for a subrecord. The maximum permitted value for length is 2147483639, which is also the default. Only really useful for use by the gfortran testsuite.

- fsign-zero

When enabled, floating point numbers of value zero with the sign bit set are written as negative number in formatted output and treated as negative in the SIGN intrinsic. -fno-sign-zero does not print the negative sign of zero values (or values rounded to zero for I/O) and regards zero as positive number in the SIGN intrinsic for compatibility with Fortran 77. The default is -fsign-zero.

3.10 Options for code generation conventions

These machine-independent options control the interface conventions used in code generation.

Most of them have both positive and negative forms; the negative form of -ffoo would be -fno-foo. In the table below, only one of the forms is listed—the one which is not the default. You can figure out the other form by either removing no- or adding it.

- fno-automatic

Treat each program unit (except those marked as RECURSIVE) as if the SAVE statement were specified for every local variable and array referenced in it. Does not affect common blocks. (Some Fortran compilers provide this
option under the name \texttt{-static} or \texttt{-save}.) The default, which is \texttt{-fautomatic}, uses the stack for local variables smaller than the value given by \texttt{-fmax-stack-var-size}. Use the option \texttt{-frecurursive} to use no static memory.

Local variables or arrays having an explicit \texttt{SAVE} attribute are silently ignored unless the \texttt{-pedantic} option is added.

\textbf{-ff2c}

Generate code designed to be compatible with code generated by \texttt{g77} and \texttt{f2c}.

The calling conventions used by \texttt{g77} (originally implemented in \texttt{f2c}) require functions that return type default \texttt{REAL} to actually return the C type \texttt{double}, and functions that return type \texttt{COMPLEX} to return the values via an extra argument in the calling sequence that points to where to store the return value. Under the default GNU calling conventions, such functions simply return their results as they would in GNU C—default \texttt{REAL} functions return the C type \texttt{float}, and \texttt{COMPLEX} functions return the GNU C type \texttt{complex}. Additionally, this option implies the \texttt{-fsecond-underscore} option, unless \texttt{-fno-second-underscore} is explicitly requested.

This does not affect the generation of code that interfaces with the \texttt{libgfortran} library.

\textit{Caution:} It is not a good idea to mix Fortran code compiled with \texttt{-ff2c} with code compiled with the default \texttt{-fno-f2c} calling conventions as, calling \texttt{COMPLEX} or default \texttt{REAL} functions between program parts which were compiled with different calling conventions will break at execution time.

\textit{Caution:} This will break code which passes intrinsic functions of type default \texttt{REAL} or \texttt{COMPLEX} as actual arguments, as the library implementations use the \texttt{-fno-f2c} calling conventions.

\textbf{-fno-underscoring}

Do not transform names of entities specified in the Fortran source file by appending underscores to them.

With \texttt{-funderscoring} in effect, GNU Fortran appends one underscore to external names with no underscores. This is done to ensure compatibility with code produced by many UNIX Fortran compilers.

\textit{Caution:} The default behavior of GNU Fortran is incompatible with \texttt{f2c} and \texttt{g77}, please use the \texttt{-ff2c} option if you want object files compiled with GNU Fortran to be compatible with object code created with these tools.

Use of \texttt{-fno-underscoring} is not recommended unless you are experimenting with issues such as integration of GNU Fortran into existing system environments (vis-à-vis existing libraries, tools, and so on).

For example, with \texttt{-funderscoring}, and assuming that \texttt{j()} and \texttt{max_count()} are external functions while \texttt{my_var} and \texttt{lvar} are local variables, a statement like

\begin{verbatim}
I = J() + MAX_COUNT (MY_VAR, LVAR)
\end{verbatim}

is implemented as something akin to:

\begin{verbatim}
i = j_() + max_count__(&my_var__, &lvar);
\end{verbatim}

With \texttt{-fno-underscoring}, the same statement is implemented as:

\begin{verbatim}
i = j() + max_count(&my_var, &lvar);
\end{verbatim}

Use of \texttt{-fno-underscoring} allows direct specification of user-defined names while debugging and when interfacing GNU Fortran code with other languages.

Note that just because the names match does \textit{not} mean that the interface implemented by GNU Fortran for an external name matches the interface implemented by some other language for that same name. That is, getting code produced by GNU Fortran to link to code produced by some other compiler using this or any other method can be only a small part of the overall solution—getting the code generated by both compilers to agree on issues other than naming can require significant effort, and, unlike naming disagreements, linkers normally cannot detect disagreements in these other areas.
Also, note that with `--fno-underscoring`, the lack of appended underscores introduces the very real possibility that a user-defined external name will conflict with a name in a system library, which could make finding unresolved-reference bugs quite difficult in some cases—they might occur at program run time, and show up only as buggy behavior at run time.

In future versions of GNU Fortran we hope to improve naming and linking issues so that debugging always involves using the names as they appear in the source, even if the names as seen by the linker are mangled to prevent accidental linking between procedures with incompatible interfaces.

`-fsecond-underscore`

By default, GNU Fortran appends an underscore to external names. If this option is used GNU Fortran appends two underscores to names with underscores and one underscore to external names with no underscores. GNU Fortran also appends two underscores to internal names with underscores to avoid naming collisions with external names.

This option has no effect if `--fno-underscoring` is in effect. It is implied by the `--ff2c` option.

Otherwise, with this option, an external name such as `MAX_COUNT` is implemented as a reference to the link-time external symbol `max_count__`, instead of `max_count_`. This is required for compatibility with `g77` and `f2c`, and is implied by use of the `--ff2c` option.

`-fcoarray=<keyword>`

- `none` Disable coarray support; using coarray declarations and image-control statements will produce a compile-time error. (Default)
- `single` Single-image mode, i.e. `num_images()` is always one.
- `lib` Library-based coarray parallelization; a suitable GNU Fortran coarray library needs to be linked.

`-fcheck=<keyword>`

Enable the generation of run-time checks; the argument shall be a comma-delimited list of the following keywords. Prefixing a check with `no-` disables it if it was activated by a previous specification.

- `all` Enable all run-time test of `-fcheck`.
- `array-temps` Warns at run time when for passing an actual argument a temporary array had to be generated. The information generated by this warning is sometimes useful in optimization, in order to avoid such temporaries.
  
  Note: The warning is only printed once per location.
- `bits` Enable generation of run-time checks for invalid arguments to the bit manipulation intrinsics.
- `bounds` Enable generation of run-time checks for array subscripts and against the declared minimum and maximum values. It also checks array indices for assumed and deferred shape arrays against the actual allocated bounds and ensures that all string lengths are equal for character array constructors without an explicit type-spec.
  
  Some checks require that `-fcheck=bounds` is set for the compilation of the main program.
  
  Note: In the future this may also include other forms of checking, e.g., checking substring references.
- `do` Enable generation of run-time checks for invalid modification of loop iteration variables.
- `mem` Enable generation of run-time checks for memory allocation. Note: This option does not affect explicit allocations using the `ALLOCATE` statement, which will be always checked.
- `pointer` Enable generation of run-time checks for pointers and allocatables.
- `recursion` Enable generation of run-time checks for recursively called subroutines and functions which are not marked as recursive. See also `-frecursive`. Note: This check does not work for OpenMP programs and is disabled if used together with `-frecursive` and `-fopenmp`.

Example: Assuming you have a file `foo.f90`, the command
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

```plaintext
gfortran -fcheck=all,no-array-temps foo.f90
```

will compile the file with all checks enabled as specified above except warnings for generated array temporaries.

\textbf{-fbounds-check}

Deprecated alias for \texttt{-fcheck=bounds}.

\textbf{-ftail-call-workaround, -ftail-call-workaround=n}

Some C interfaces to Fortran codes violate the gfortran ABI by omitting the hidden character length arguments as described in See \texttt{Argument passing conventions}. This can lead to crashes because pushing arguments for tail calls can overflow the stack.

To provide a workaround for existing binary packages, this option disables tail call optimization for gfortran procedures with character arguments. With \texttt{-ftail-call-workaround=2} tail call optimization is disabled in all gfortran procedures with character arguments, with \texttt{-ftail-call-workaround=1} or equivalent \texttt{-ftail-call-workaround=n} only in gfortran procedures with character arguments that call implicitly prototyped procedures.

Using this option can lead to problems including crashes due to insufficient stack space.

It is \textit{very strongly} recommended to fix the code in question. The \texttt{-fc-prototypes-external} option can be used to generate prototypes which conform to gfortran’s ABI, for inclusion in the source code.

Support for this option will likely be withdrawn in a future release of gfortran.

The negative form, \texttt{-fno-tail-call-workaround} or equivalent \texttt{-ftail-call-workaround=0}, can be used to disable this option.

Default is currently \texttt{-ftail-call-workaround}, this will change in future releases.

\textbf{-fcheck-array-temporaries}

Deprecated alias for \texttt{-fcheck=array-temps}.

\textbf{-fmax-array-constructor=n}

This option can be used to increase the upper limit permitted in array constructors. The code below requires this option to expand the array at compile time.

```
program test
  implicit none
  integer j
  integer, parameter :: n = 100000
  integer, parameter :: i(n) = (/ (2*j, j = 1, n) /)
  print '(10(I0,1X))', i
end program test
```

\textit{Caution: This option can lead to long compile times and excessively large object files.}

The default value for \textit{n} is 65535.

\textbf{-fmax-stack-var-size=n}

This option specifies the size in bytes of the largest array that will be put on the stack; if the size is exceeded static memory is used (except in procedures marked as RECURSIVE). Use the option \texttt{-frecursive} to allow for recursive procedures which do not have a RECURSIVE attribute or for parallel programs. Use \texttt{-fno-automatic} to never use the stack.

This option currently only affects local arrays declared with constant bounds, and may not apply to all character variables. Future versions of GNU Fortran may improve this behavior.

The default value for \textit{n} is 65536.

\textbf{-fstack-arrays}

Adding this option will make the Fortran compiler put all arrays of unknown size and array temporaries onto stack
memory. If your program uses very large local arrays it is possible that you will have to extend your runtime limits for stack memory on some operating systems. This flag is enabled by default at optimization level -Ofast unless -fmax-stack-var-size is specified.

-fpack-derived
This option tells GNU Fortran to pack derived type members as closely as possible. Code compiled with this option is likely to be incompatible with code compiled without this option, and may execute slower.

-frepack-arrays
In some circumstances GNU Fortran may pass assumed shape array sections via a descriptor describing a noncontiguous area of memory. This option adds code to the function prologue to repack the data into a contiguous block at runtime.

This should result in faster access to the array. However it can introduce significant overhead to the function call, especially when the passed data is noncontiguous.

-fshort-enums
This option is provided for interoperability with C code that was compiled with the -fshort-enums option. It will make GNU Fortran choose the smallest INTEGER kind a given enumerator set will fit in, and give all its enumerators this kind.

-finline-arg-packing
When passing an assumed-shape argument of a procedure as actual argument to an assumed-size or explicit size or as argument to a procedure that does not have an explicit interface, the argument may have to be packed, that is put into contiguous memory. An example is the call to foo in

```
subroutine foo(a)
  real, dimension(*) :: a
end subroutine foo
subroutine bar(b)
  real, dimension(*) :: b
  call foo(b)
end subroutine bar
```

When -finline-arg-packing is in effect, this packing will be performed by inline code. This allows for more optimization while increasing code size.

-fexternal-blas
This option will make gfortran generate calls to BLAS functions for some matrix operations like MATMUL, instead of using our own algorithms, if the size of the matrices involved is larger than a given limit (see -fblas-matmul-limit). This may be profitable if an optimized vendor BLAS library is available. The BLAS library will have to be specified at link time.

-fblas-matmul-limit=n
Only significant when -fexternal-blas is in effect. Matrix multiplication of matrices with size larger than (or equal to) n will be performed by calls to BLAS functions, while others will be handled by gfortran internal algorithms. If the matrices involved are not square, the size comparison is performed using the geometric mean of the dimensions of the argument and result matrices.

The default value for n is 30.

-finline-matmul-limit=n
When front-end optimization is active, some calls to the MATMUL intrinsic function will be inlined. This may result in code size increase if the size of the matrix cannot be determined at compile time, as code for both cases is generated. Setting -finline-matmul-limit=0 will disable inlining in all cases. Setting this option with a
value of \( n \) will produce inline code for matrices with size up to \( n \). If the matrices involved are not square, the size comparison is performed using the geometric mean of the dimensions of the argument and result matrices.

The default value for \( n \) is 30. The `-fblas-matmul-limit` can be used to change this value.

- **-frecursive**
  Allow indirect recursion by forcing all local arrays to be allocated on the stack. This flag cannot be used together with `-fmax-stack-var-size` or `-fno-automatic`.

- **-finit-local-zero, -finit-derived, -finit-integer=n, -finit-real=\(<zero|inf|-inf|nan|snan>\), -finit-logical=<true|false>, -finit-character=n**
  The `-finit-local-zero` option instructs the compiler to initialize local INTEGER, REAL, and COMPLEX variables to zero, LOGICAL variables to false, and CHARACTER variables to a string of null bytes. Finer-grained initialization options are provided by the `-finit-integer=n, -finit-real=\(<zero|inf|-inf|nan|snan>\)` (which also initializes the real and imaginary parts of local COMPLEX variables), `-finit-logical=<true|false>`, and `-finit-character=n` (where \( n \) is an ASCII character value) options.

  With `-finit-derived`, components of derived type variables will be initialized according to these flags. Components whose type is not covered by an explicit `-finit-*` flag will be treated as described above with `-finit-local-zero`.

  These options do not initialize
  
  - objects with the POINTER attribute
  - allocatable arrays
  - variables that appear in an EQUIVALENCE statement.

  (These limitations may be removed in future releases).

  Note that the `-finit-real=nan` option initializes REAL and COMPLEX variables with a quiet NaN. For a signalling NaN use `-finit-real=snan`; note, however, that compile-time optimizations may convert them into quiet NaN and that trapping needs to be enabled (e.g. via `-ffpe-trap`).

  The `-finit-integer` option will parse the value into an integer of type `INTEGER(kind=C_LONG)` on the host. Said value is then assigned to the integer variables in the Fortran code, which might result in wraparound if the value is too large for the kind.

  Finally, note that enabling any of the `-finit-*` options will silence warnings that would have been emitted by `-Wuninitialized` for the affected local variables.

- **-falign-commons**
  By default, `gfortran` enforces proper alignment of all variables in a COMMON block by padding them as needed. On certain platforms this is mandatory, on others it increases performance. If a COMMON block is not declared with consistent data types everywhere, this padding can cause trouble, and `-fno-align-commons` can be used to disable automatic alignment. The same form of this option should be used for all files that share a COMMON block.

  To avoid potential alignment issues in COMMON blocks, it is recommended to order objects from largest to smallest.

- **-fno-protect-parens**
  By default the parentheses in expression are honored for all optimization levels such that the compiler does not do any re-association. Using `-fno-protect-parens` allows the compiler to reorder REAL and COMPLEX expressions to produce faster code. Note that for the re-association optimization `-fno-signed-zeros` and `-fno-trapping-math` need to be in effect. The parentheses protection is enabled by default, unless `-O0` is given.

- **-frealloc-lhs**
  An allocatable left-hand side of an intrinsic assignment is automatically (re)allocated if it is either unallocated or has a different shape. The option is enabled by default except when `-std=f95` is given. See also `-Wrealloc-lhs`.

3.10. Options for code generation conventions 27
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

-fflaggressive-function-elimination
Functions with identical argument lists are eliminated within statements, regardless of whether these functions are marked PURE or not. For example, in

\[
a = f(b, c) + f(b, c)
\]

there will only be a single call to \( f \). This option only works if `-ffrontend-optimize` is in effect.

-ffrontend-optimize
This option performs front-end optimization, based on manipulating parts the Fortran parse tree. Enabled by default by any `-O` option except `-O0` and `-Og`. Optimizations enabled by this option include:

- inlining calls to `MATMUL`,
- elimination of identical function calls within expressions,
- removing unnecessary calls to `TRIM` in comparisons and assignments,
- replacing \( \text{TRIM}(a) \) with \( a(1: \text{LEN_TRIM}(a)) \) and
- short-circuiting of logical operators (`.AND.` and `.OR.`).

It can be deselected by specifying `-fno-frontend-optimize`.

-ffrontend-loop-interchange
Attempt to interchange loops in the Fortran front end where profitable. Enabled by default by any `-O` option. At the moment, this option only affects `FORALL` and `DO CONCURRENT` statements with several forall triplets.

See `Options for Code Generation Conventions`, for information on more options offered by the GBE shared by `gfortran`, `gcc`, and other GNU compilers.

### 3.11 Options for interoperability with other languages

-fflag-c-prototypes
This option will generate C prototypes from `BIND(C)` variable declarations, types and procedure interfaces and writes them to standard output. `ENUM` is not yet supported.

The generated prototypes may need inclusion of an appropriate header, such as `<stdint.h>` or `<stdlib.h>`. For types which are not specified using the appropriate kind from the `iso_c_binding` module, a warning is added as a comment to the code.

For function pointers, a pointer to a function returning `int` without an explicit argument list is generated.

Example of use:

```
$ gfortran -fflag-c-prototypes -fsyntax-only foo.f90 > foo.h
```

where the C code intended for interoperating with the Fortran code then uses `#include "foo.h"`.

-fflag-c-prototypes-external
This option will generate C prototypes from external functions and subroutines and write them to standard output. This may be useful for making sure that C bindings to Fortran code are correct. This option does not generate prototypes for `BIND(C)` procedures, use `-fflag-c-prototypes` for that.

The generated prototypes may need inclusion of an appropriate header, such as `<stdint.h>` or `<stdlib.h>`.

This is primarily meant for legacy code to ensure that existing C bindings match what `gfortran` emits. The generated C prototypes should be correct for the current version of the compiler, but may not match what other
compilers or earlier versions of gfortran need. For new developments, use of the BIND(C) features is recommended.

Example of use:

```
gfortran -fc-prototypes-external -fsyntax-only foo.f > foo.h
```

where the C code intended for interoperating with the Fortran code then uses `#include "foo.h"`.

### 3.12 Environment variables affecting gfortran

#### 3.12.1 Environment

The gfortran compiler currently does not make use of any environment variables to control its operation above and beyond those that affect the operation of gcc.

See Environment Variables Affecting GCC, for information on environment variables.

See Runtime: Influencing runtime behavior with environment variables, for environment variables that affect the run-time behavior of programs compiled with GNU Fortran.
CHAPTER FOUR

RUNTIME: INFLUENCING RUNTIME BEHAVIOR WITH ENVIRONMENT VARIABLES

The behavior of the *gfortran* can be influenced by environment variables. Malformed environment variables are silently ignored.

### 4.1 TMPDIR—Directory for scratch files

When opening a file with `STATUS='SCRATCH'`, GNU Fortran tries to create the file in one of the potential directories by testing each directory in the order below.

- The environment variable `TMPDIR`, if it exists.
- On the MinGW target, the directory returned by the `GetTempPath` function. Alternatively, on the Cygwin target, the `TMP` and `TEMP` environment variables, if they exist, in that order.
- The `P_tmpdir` macro if it is defined, otherwise the directory `/tmp`.

### 4.2 GFORTRAN_STDIN_UNIT—Unit number for standard input

This environment variable can be used to select the unit number preconnected to standard input. This must be a positive integer. The default value is 5.

### 4.3 GFORTRAN_STDOUT_UNIT—Unit number for standard output

This environment variable can be used to select the unit number preconnected to standard output. This must be a positive integer. The default value is 6.
4.4 **GFORTRAN_STDERR_UNIT**—Unit number for standard error

This environment variable can be used to select the unit number preconnected to standard error. This must be a positive integer. The default value is 0.

4.5 **GFORTRAN_UNBUFFERED_ALL**—Do not buffer I/O on all units

This environment variable controls whether all I/O is unbuffered. If the first letter is `y`, `Y` or `1`, all I/O is unbuffered. This will slow down small sequential reads and writes. If the first letter is `n`, `N` or `0`, I/O is buffered. This is the default.

4.6 **GFORTRAN_UNBUFFERED_PRECONNECTED**—Do not buffer I/O on preconnected units

The environment variable named `GFORTRAN_UNBUFFERED_PRECONNECTED` controls whether I/O on a preconnected unit (i.e. STDOUT or STDERR) is unbuffered. If the first letter is `y`, `Y` or `1`, I/O is unbuffered. This will slow down small sequential reads and writes. If the first letter is `n`, `N` or `0`, I/O is buffered. This is the default.

4.7 **GFORTRAN_SHOW_LOCUS**—Show location for runtime errors

If the first letter is `y`, `Y` or `1`, filename and line numbers for runtime errors are printed. If the first letter is `n`, `N` or `0`, do not print filename and line numbers for runtime errors. The default is to print the location.

4.8 **GFORTRAN_OPTIONAL_PLUS**—Print leading + where permitted

If the first letter is `y`, `Y` or `1`, a plus sign is printed where permitted by the Fortran standard. If the first letter is `n`, `N` or `0`, a plus sign is not printed in most cases. Default is not to print plus signs.

4.9 **GFORTRAN_LIST_SEPARATOR**—Separator for list output

This environment variable specifies the separator when writing list-directed output. It may contain any number of spaces and at most one comma. If you specify this on the command line, be sure to quote spaces, as in

```
$ GFORTRAN_LIST_SEPARATOR=' , ' ./a.out
```

when `a.out` is the compiled Fortran program that you want to run. Default is a single space.
4.10 GFORTRAN_CONVERT_UNIT—Set endianness for unformatted I/O

By setting the GFORTRAN_CONVERT_UNIT variable, it is possible to change the representation of data for unformatted files. The syntax for the GFORTRAN_CONVERT_UNIT variable is:

```
GFORTRAN_CONVERT_UNIT: mode | mode ';\' exception | exception ;
mode: 'native' | 'swap' | 'big_endian' | 'little_endian';
exception: mode ';\' unit_list | unit_list ;
unit_list: unit_spec | unit_list unit_spec ;
unit_spec: INTEGER | INTEGER '-\' INTEGER ;
```

The variable consists of an optional default mode, followed by a list of optional exceptions, which are separated by semicolons from the preceding default and each other. Each exception consists of a format and a comma-separated list of units. Valid values for the modes are the same as for the CONVERT specifier:

- **NATIVE** Use the native format. This is the default.
- **SWAP** Swap between little- and big-endian.
- **LITTLE_ENDIAN** Use the little-endian format for unformatted files.
- **BIG_ENDIAN** Use the big-endian format for unformatted files.

A missing mode for an exception is taken to mean BIG_ENDIAN. Examples of values for GFORTRAN_CONVERT_UNIT are:

- `'big_endian'` Do all unformatted I/O in big_endian mode.
- `'little_endian;native:10-20,25'` Do all unformatted I/O in little_endian mode, except for units 10 to 20 and 25, which are in native format.
- `'10-20'` Units 10 to 20 are big-endian, the rest is native.

Setting the environment variables should be done on the command line or via the `export` command for sh-compatible shells and via `setenv` for csh-compatible shells.

Example for **sh**:

```
$ gfortran foo.f90
$ GFORTRAN_CONVERT_UNIT='big_endian;native:10-20' ./a.out
```

Example code for **csh**:

```
gfortran foo.f90
setenv GFORTRAN_CONVERT_UNIT 'big_endian;native:10-20'
./a.out
```

Using anything but the native representation for unformatted data carries a significant speed overhead. If speed in this area matters to you, it is best if you use this only for data that needs to be portable.

See **CONVERT specifier**, for an alternative way to specify the data representation for unformatted files. See **Influencing runtime behavior**, for setting a default data representation for the whole program. The CONVERT specifier overrides the `-fconvert` compile options.

Note that the values specified via the GFORTRAN_CONVERT_UNIT environment variable will override the CONVERT specifier in the open statement. This is to give control over data formats to users who do not have the source code of their program available.
4.11 GFORTRAN_ERROR_BACKTRACE—Show backtrace on run-time errors

If the GFORTRAN_ERROR_BACKTRACE variable is set to y, Y or 1 (only the first letter is relevant) then a backtrace is printed when a serious run-time error occurs. To disable the backtracing, set the variable to n, N, 0. Default is to print a backtrace unless the -fno-backtrace compile option was used.

4.12 GFORTRAN_FORMATTED_BUFFER_SIZE—Set buffer size for formatted I/O

The GFORTRAN_FORMATTED_BUFFER_SIZE environment variable specifies buffer size in bytes to be used for formatted output. The default value is 8192.

4.13 GFORTRAN_UNFORMATTED_BUFFER_SIZE—Set buffer size for unformatted I/O

The GFORTRAN_UNFORMATTED_BUFFER_SIZE environment variable specifies buffer size in bytes to be used for unformatted output. The default value is 131072.
5.1 Fortran 2003 status

GNU Fortran supports several Fortran 2003 features; an incomplete list can be found below. See also the wiki page about Fortran 2003.

- Procedure pointers including procedure-pointer components with \texttt{PASS} attribute.
- Procedures which are bound to a derived type (type-bound procedures) including \texttt{PASS}, \texttt{PROCEDURE} and \texttt{GENERIC}, and operators bound to a type.
- Abstract interfaces and type extension with the possibility to override type-bound procedures or to have deferred binding.
- Polymorphic entities (\texttt{`CLASS`}) for derived types and unlimited polymorphism (\texttt{`CLASS(*)`}) – including \texttt{SAME_TYPE_AS}, \texttt{EXTENDS_TYPE_OF} and \texttt{SELECT TYPE} for scalars and arrays and finalization.
- Generic interface names, which have the same name as derived types, are now supported. This allows one to write constructor functions. Note that Fortran does not support static constructor functions. For static variables, only default initialization or structure-constructor initialization are available.
- The \texttt{ASSOCIATE} construct.
- Interoperability with C including enumerations,
- In structure constructors the components with default values may be omitted.
- Extensions to the \texttt{ALLOCATE} statement, allowing for a type-specification with type parameter and for allocation and initialization from a \texttt{SOURCE=} expression; \texttt{ALLOCATE} and \texttt{DEALLOCATE} optionally return an error message string via \texttt{ERRMSG=}.
- Reallocation on assignment: If an intrinsic assignment is used, an allocatable variable on the left-hand side is automatically allocated (if unallocated) or reallocated (if the shape is different). Currently, scalar deferred character length left-hand sides are correctly handled but arrays are not yet fully implemented.
- Deferred-length character variables and scalar deferred-length character components of derived types are supported. (Note that array-valued components are not yet implemented.)
- Transferring of allocations via \texttt{MOVE_ALLOC}.
- The \texttt{PRIVATE} and \texttt{PUBLIC} attributes may be given individually to derived-type components.
- In pointer assignments, the lower bound may be specified and the remapping of elements is supported.
- For pointers an \texttt{INTENT} may be specified which affect the association status not the value of the pointer target.
- \texttt{Intrinsics} \texttt{command_argument_count}, \texttt{get_command}, \texttt{get_command_argument}, and \texttt{get_environment_variable}. 


Using GNU Fortran, Release 12.0.1 (experimental 20210414)

- Support for Unicode characters (ISO 10646) and UTF-8, including the SELECTED_CHAR_KIND and NEW_LINE intrinsic functions.
- Support for binary, octal and hexadecimal (BOZ) constants in the intrinsic functions INT, REAL, CMPLX and DBLE.
- Support for namelist variables with allocatable and pointer attribute and nonconstant length type parameter.
- Array constructors using square brackets. That is, \([...]\) rather than \((/.../). Type-specification for array constructors like \((/ \text{some-type} :: ... /).\)
- Extensions to the specification and initialization expressions, including the support for intrinsics with real and complex arguments.
- Support for the asynchronous input/output.
- FLUSH statement.
- IOMSG= specifier for I/O statements.
- Support for the declaration of enumeration constants via the ENUM and ENUMERATOR statements. Interoperability with gcc is guaranteed also for the case where the -fshort-enums command line option is given.
- TR 15581:
  - ALLOCATABLE dummy arguments.
  - ALLOCATABLE function results
  - ALLOCATABLE components of derived types
- The OPEN statement supports the ACCESS='STREAM' specifier, allowing I/O without any record structure.
- Namelist input/output for internal files.
- Minor I/O features: Rounding during formatted output, using of a decimal comma instead of a decimal point, setting whether a plus sign should appear for positive numbers. On systems where strtod honours the rounding mode, the rounding mode is also supported for input.
- The PROTECTED statement and attribute.
- The VALUE statement and attribute.
- The VOLATILE statement and attribute.
- The IMPORT statement, allowing to import host-associated derived types.
- The intrinsic modules ISO_FORTRAN_ENVIRONMENT is supported, which contains parameters of the I/O units, storage sizes. Additionally, procedures for C interoperability are available in the ISO_C_BINDING module.
- USE statement with INTRINSIC and NON_INTRINSIC attribute; supported intrinsic modules: ISO_FORTRAN_ENV, ISO_C_BINDING, OMP_LIB and OMP_LIB_KINDS, and OPENACC.
- Renaming of operators in the USE statement.
5.2 Fortran 2008 status

The latest version of the Fortran standard is ISO/IEC 1539-1:2010, informally known as Fortran 2008. The official version is available from International Organization for Standardization (ISO) or its national member organizations. The final draft (FDIS) can be downloaded free of charge from [http://www.nag.co.uk/sc22wg5/links.html](http://www.nag.co.uk/sc22wg5/links.html). Fortran is developed by the Working Group 5 of Sub-Committee 22 of the Joint Technical Committee 1 of the International Organization for Standardization and the International Electrotechnical Commission (IEC). This group is known as WG5.

The GNU Fortran compiler supports several of the new features of Fortran 2008; the wiki has some information about the current Fortran 2008 implementation status. In particular, the following is implemented.

- The `-std=f2008` option and support for the file extensions `.f08` and `.F08`.
- The `OPEN` statement now supports the `NEWUNIT=` option, which returns a unique file unit, thus preventing inadvertent use of the same unit in different parts of the program.
- The `g0` format descriptor and unlimited format items.
- The mathematical intrinsics `ASINH`, `ACOSH`, `ATANH`, `ERF`, `ERFC`, `GAMMA`, `LOG_GAMMA`, `BESSEL_J0`, `BESSEL_J1`, `BESSEL_JN`, `BESSEL_Y0`, `BESSEL_Y1`, `BESSEL_YN`, `HYPOT`, `NORM2`, and `ERFC_SCALED`.
- Using complex arguments with `TAN`, `SINH`, `COSH`, `TANH`, `ASIN`, `ACOS`, and `ATAN` is now possible; `ATAN(Y, X)` is now an alias for `ATAN2(Y, X)`.
- Support of the `PARITY` intrinsic functions.
- The following bit intrinsics: `LEADZ` and `TRAILZ` for counting the number of leading and trailing zero bits, `POPCNT` and `POPPAR` for counting the number of one bits and returning the parity; `BGE`, `BGT`, `BLE`, and `BLT` for bitwise comparisons; `DSHIFTL` and `DSHIFTR` for combined left and right shifts, `MASKL` and `MASKR` for simple left and right justified masks, `MERGE_BITS` for a bitwise merge using a mask, `SHIFTA`, `SHIFTL` and `SHIFTR` for shift operations, and the transformational bit intrinsics `IALL`, `IANY` and `IPARITY`.
- Support of the `EXECUTE_COMMAND_LINE` intrinsic subroutine.
- Support for the `STORAGE_SIZE` intrinsic inquiry function.
- The `INT{8,16,32}` and `REAL{32,64,128}` kind type parameters and the array-valued named constants `INTEGER_KINDS`, `LOGICAL_KINDS`, `REAL_KINDS` and `CHARACTER_KINDS` of the intrinsic module `ISO_FORTRAN_ENV`.
- The module procedures `C_SIZEOF` of the intrinsic module `ISO_C_BINDINGS` and `COMPILER_VERSION` and `COMPILER_OPTIONS` of `ISO_FORTRAN_ENV`.
- Coarray support for serial programs with `-fcoarray=single` flag and experimental support for multiple images with the `-fcoarray=lib` flag.
- Submodules are supported. It should noted that `MODULEs` do not produce the smod file needed by the descendent `SUBMODULEs` unless they contain at least one `MODULE` `PROCEDURE` interface. The reason for this is that `SUBMODULEs` are useless without `MODULE` `PROCEDUREs`. See [http://j3-fortran.org/doc/meeting/207/15-209.txt](http://j3-fortran.org/doc/meeting/207/15-209.txt) for a discussion and a draft interpretation. Adopting this interpretation has the advantage that code that does not use submodules does not generate smod files.
- The `DO CONCURRENT` construct is supported.
- The `BLOCK` construct is supported.
- The `STOP` and the new `ERROR STOP` statements now support all constant expressions. Both show the signals which were signaling at termination.
- Support for the `CONTIGUOUS` attribute.
- Support for `ALLOCATE` with `MOLD`.

5.2. Fortran 2008 status
- Support for the **IMPURE** attribute for procedures, which allows for **ELEMENTAL** procedures without the restrictions of **PURE**.
- Null pointers (including **NULL()**) and not-allocated variables can be used as actual argument to optional non-pointer, non-allocatable dummy arguments, denoting an absent argument.
- Non-pointer variables with **TARGET** attribute can be used as actual argument to **POINTER** dummies with **IN-TENT** (**IN**).
- Pointers including procedure pointers and those in a derived type (pointer components) can now be initialized by a target instead of only by **NULL**.
- The **EXIT** statement (with construct-name) can be now be used to leave not only the **DO** but also the **ASSOCIATE**, **BLOCK**, **IF**, **SELECT CASE** and **SELECT TYPE** constructs.
- Internal procedures can now be used as actual argument.
- Minor features: obsolesce diagnostics for **ENTRY** with `-std=f2008`; a line may start with a semicolon; for internal and module procedures **END** can be used instead of **END SUBROUTINE** and **END FUNCTION**; **SELECTED_REAL_KIND** now also takes a **RADIX** argument; intrinsic types are supported for **TYPE**(intrinsic-type-spec); multiple type-bound procedures can be declared in a single **PROCEDURE** statement; implied-shape arrays are supported for named constants (**PARAMETER**).

### 5.3 Status of Fortran 2018 support

- **ERROR STOP** in a **PURE** procedure An **ERROR STOP** statement is permitted in a **PURE** procedure.
- **IMPLICIT NONE** with a spec-list Support the **IMPLICIT NONE** statement with an **implicit-none-spec-list**.
- Behavior of **INQUIRE** with the **RECL=** specifier
  
  The behavior of the **INQUIRE** statement with the **RECL=** specifier now conforms to Fortran 2018.

#### 5.3.1 TS 29113 Status (Further Interoperability with C)

GNU Fortran supports some of the new features of the Technical Specification (TS) 29113 on Further Interoperability of Fortran with C. The wiki has some information about the current TS 29113 implementation status. In particular, the following is implemented.

See also Further Interoperability of Fortran with C.

- The **OPTIONAL** attribute is allowed for dummy arguments of **BIND(C)** procedures.
- The **RANK** intrinsic is supported.
- GNU Fortran's implementation for variables with **ASYNCHRONOUS** attribute is compatible with TS 29113.
- Assumed types (**TYPE(*)**).
- Assumed-rank (**DIMENSION(*)**).
- **ISO_Fortran_binding** (now in Fortran 2018 18.4) is implemented such that conversion of the array descriptor for assumed type or assumed rank arrays is done in the library. The include file **ISO_Fortran_binding.h** is can be found in `-prefix/lib/gcc/$target/$version`. 

---
5.3.2 TS 18508 Status (Additional Parallel Features)

GNU Fortran supports the following new features of the Technical Specification 18508 on Additional Parallel Features in Fortran:

- The new atomic ADD, CAS, FETCH and ADD/OR/XOR, OR and XOR intrinsics.
- The `CO_MIN` and `CO_MAX` and `SUM` reduction intrinsics. And the `CO.Broadcast` and `CO.Reduce` intrinsic, except that those do not support polymorphic types or types with allocatable, pointer or polymorphic components.
- Events (EVENT POST, EVENT WAIT, EVENT_QUERY)
- Failed images (FAIL IMAGE, IMAGE_STATUS, FAILED IMAGES, STOPPED IMAGES)
This chapter describes certain characteristics of the GNU Fortran compiler, that are not specified by the Fortran standard, but which might in some way or another become visible to the programmer.

6.1 KIND Type Parameters

The KIND type parameters supported by GNU Fortran for the primitive data types are:

**INTEGER**
1, 2, 4, 8\(^1\), 16\(^7\), default: 4\(^2\)

**LOGICAL**
1, 2, 4, 8\(^7\), 16\(^7\), default: 4\(^7\)

**REAL**
4, 8, 10\(^7\), 16\(^7\), default: 4\(^7\)

**COMPLEX**
4, 8, 10\(^7\), 16\(^7\), default: 4\(^7\)

**DOUBLE PRECISION**
4, 8, 10\(^7\), 16\(^7\), default: 8\(^7\)

**CHARACTER**
1, 4, default: 1

The KIND value matches the storage size in bytes, except for COMPLEX where the storage size is twice as much (or both real and imaginary part are a real value of the given size). It is recommended to use the SELECTED_CHAR_KIND, SELECTED_INT_KIND and SELECTED_REAL_KIND intrinsics or the INT8, INT16, INT32, INT64, REAL32, REAL64, and REAL128 parameters of the ISO_FORTRAN_ENV module instead of the concrete values. The available kind parameters can be found in the constant arrays CHARACTER_KINDS, INTEGER_KINDS, LOGICAL_KINDS and REAL_KINDS in the ISO_FORTRAN_ENV module. For C interoperability, the kind parameters of the ISO_C_BINDING module should be used.

---

\(^1\) not available on all systems
\(^2\) unless -fdefault-integer-8 is used
\(^3\) unless -fdefault-real-8 is used (see Fortran Dialect Options)
6.2 Internal representation of LOGICAL variables

The Fortran standard does not specify how variables of LOGICAL type are represented, beyond requiring that LOGICAL variables of default kind have the same storage size as default INTEGER and REAL variables. The GNU Fortran internal representation is as follows.

A \texttt{LOGICAL(KIND=N)} variable is represented as an \texttt{INTEGER(KIND=N)} variable, however, with only two permissible values: 1 for \texttt{.TRUE.} and 0 for \texttt{.FALSE.}. Any other integer value results in undefined behavior.

See also Argument passing conventions and Interoperability with C.

6.3 Evaluation of logical expressions

The Fortran standard does not require the compiler to evaluate all parts of an expression, if they do not contribute to the final result. For logical expressions with \texttt{.AND.} or \texttt{.OR.} operators, in particular, GNU Fortran will optimize out function calls (even to impure functions) if the result of the expression can be established without them. However, since not all compilers do that, and such an optimization can potentially modify the program flow and subsequent results, GNU Fortran throws warnings for such situations with the \texttt{-Wfunction-elimination} flag.

6.4 MAX and MIN intrinsics with REAL NaN arguments

The Fortran standard does not specify what the result of the \texttt{MAX} and \texttt{MIN} intrinsics are if one of the arguments is a NaN. Accordingly, the GNU Fortran compiler does not specify that either, as this allows for faster and more compact code to be generated. If the programmer wishes to take some specific action in case one of the arguments is a NaN, it is necessary to explicitly test the arguments before calling \texttt{MAX} or \texttt{MIN}, e.g. with the \texttt{IEEE_IS_NAN} function from the intrinsic module \texttt{IEEE_ARITHMETIC}.

6.5 Thread-safety of the runtime library

GNU Fortran can be used in programs with multiple threads, e.g. by using OpenMP, by calling OS thread handling functions via the \texttt{ISO_C_BINDING} facility, or by GNU Fortran compiled library code being called from a multi-threaded program.

The GNU Fortran runtime library, (libgfortran), supports being called concurrently from multiple threads with the following exceptions.

During library initialization, the C \texttt{getenv} function is used, which need not be thread-safe. Similarly, the \texttt{getenv} function is used to implement the \texttt{GET_ENVIRONMENT_VARIABLE} and \texttt{GETENV} intrinsics. It is the responsibility of the user to ensure that the environment is not being updated concurrently when any of these actions are taking place.

The \texttt{EXECUTE_COMMAND_LINE} and \texttt{SYSTEM} intrinsics are implemented with the \texttt{system} function, which need not be thread-safe. It is the responsibility of the user to ensure that \texttt{system} is not called concurrently.

For platforms not supporting thread-safe POSIX functions, further functionality might not be thread-safe. For details, please consult the documentation for your operating system.

The GNU Fortran runtime library uses various C library functions that depend on the locale, such as \texttt{strtod} and \texttt{snprintf}. In order to work correctly in locale-aware programs that set the locale using \texttt{setlocale}, the locale is reset to the default ‘C’ locale while executing a formatted \texttt{READ} or \texttt{WRITE} statement. On targets supporting the POSIX 2008 per-thread locale functions (e.g. \texttt{newlocale}, \texttt{uselocale}, \texttt{freelocale}), these are used and thus the global locale set using \texttt{setlocale} or the per-thread locales in other threads are not affected. However, on targets lacking this
functionality, the global LC_NUMERIC locale is set to 'C' during the formatted I/O. Thus, on such targets it's not safe to call setlocale concurrently from another thread while a Fortran formatted I/O operation is in progress. Also, other threads doing something dependent on the LC_NUMERIC locale might not work correctly if a formatted I/O operation is in progress in another thread.

6.6 Data consistency and durability

This section contains a brief overview of data and metadata consistency and durability issues when doing I/O.

With respect to durability, GNU Fortran makes no effort to ensure that data is committed to stable storage. If this is required, the GNU Fortran programmer can use the intrinsic FNUM to retrieve the low level file descriptor corresponding to an open Fortran unit. Then, using e.g. the ISO_C_BINDING feature, one can call the underlying system call to flush dirty data to stable storage, such as fsync on POSIX, _commit on MingW, or fcntl(fd, F_FULLSYNC, 0) on Mac OS X. The following example shows how to call fsync:

```
! Declare the interface for POSIX fsync function
interface
  function fsync (fd) bind(c,name="fsync")
    use iso_c_binding, only: c_int
    integer(c_int), value :: fd
    integer(c_int) :: fsync
  end function fsync
end interface

! Variable declaration
integer :: ret

! Opening unit 10
open (10,file="foo")

! Perform I/O on unit 10
!
!
!

! Flush and sync
flush(10)
ret = fsync(fnum(10))

! Handle possible error
if (ret /= 0) stop "Error calling FSYNC"
```

With respect to consistency, for regular files GNU Fortran uses buffered I/O in order to improve performance. This buffer is flushed automatically when full and in some other situations, e.g. when closing a unit. It can also be explicitly flushed with the FLUSH statement. Also, the buffering can be turned off with the GFORTRAN_UNBUFFERED_ALL and GFORTRAN_UNBUFFERED_PRECONNECTED environment variables. Special files, such as terminals and pipes, are always unbuffered. Sometimes, however, further things may need to be done in order to allow other processes to see data that GNU Fortran has written, as follows.

The Windows platform supports a relaxed metadata consistency model, where file metadata is written to the directory lazily. This means that, for instance, the dir command can show a stale size for a file. One can force a directory metadata update by closing the unit, or by calling _commit on the file descriptor. Note, though, that _commit will force all dirty data to stable storage, which is often a very slow operation.

The Network File System (NFS) implements a relaxed consistency model called open-to-close consistency. Closing a file forces dirty data and metadata to be flushed to the server, and opening a file forces the client to contact the server in order to revalidate cached data. fsync will also force a flush of dirty data and metadata to the server. Similar to open
and close, acquiring and releasing fcntl file locks, if the server supports them, will also force cache validation and flushing dirty data and metadata.

6.7 Files opened without an explicit ACTION= specifier

The Fortran standard says that if an OPEN statement is executed without an explicit ACTION= specifier, the default value is processor dependent. GNU Fortran behaves as follows:

- Attempt to open the file with ACTION='READWRITE'
- If that fails, try to open with ACTION='READ'
- If that fails, try to open with ACTION='WRITE'
- If that fails, generate an error

6.8 File operations on symbolic links

This section documents the behavior of GNU Fortran for file operations on symbolic links, on systems that support them.

- Results of INQUIRE statements of the ‘inquire by file’ form will relate to the target of the symbolic link. For example, INQUIRE(FILE="foo",EXIST=ex) will set ex to .true. if foo is a symbolic link pointing to an existing file, and .false. if foo points to a non-existing file ('dangling' symbolic link).
- Using the OPEN statement with a STATUS="NEW" specifier on a symbolic link will result in an error condition, whether the symbolic link points to an existing target or is dangling.
- If a symbolic link was connected, using the CLOSE statement with a STATUS="DELETE" specifier will cause the symbolic link itself to be deleted, not its target.

6.9 File format of unformatted sequential files

Unformatted sequential files are stored as logical records using record markers. Each logical record consists of one of more subrecords.

Each subrecord consists of a leading record marker, the data written by the user program, and a trailing record marker. The record markers are four-byte integers by default, and eight-byte integers if the -fmax-subrecord-length=8 option (which exists for backwards compatibility only) is in effect.

The representation of the record markers is that of unformatted files given with the -fconvert option, the CONVERT specifier in an open statement or the GFORTRAN_CONVERT_UNIT environment variable.

The maximum number of bytes of user data in a subrecord is 2147483639 (2 GiB - 9) for a four-byte record marker. This limit can be lowered with the -fmax-subrecord-length option, although this is rarely useful. If the length of a logical record exceeds this limit, the data is distributed among several subrecords.

The absolute of the number stored in the record markers is the number of bytes of user data in the corresponding subrecord. If the leading record marker of a subrecord contains a negative number, another subrecord follows the current one. If the trailing record marker contains a negative number, then there is a preceding subrecord.

In the most simple case, with only one subrecord per logical record, both record markers contain the number of bytes of user data in the record.

The format for unformatted sequential data can be duplicated using unformatted stream, as shown in the example program for an unformatted record containing a single subrecord:
program main
  use iso_fortran_env, only: int32
  implicit none
  integer(int32) :: i
  real, dimension(10) :: a, b
  call random_number(a)
  open (10, file='test.dat', form='unformatted', access='stream')
  inquire (iolength=i) a
  write (10) i, a, i
  close (10)
  open (10, file='test.dat', form='unformatted')
  read (10) b
  if (all (a == b)) print '*', 'success!'
end program main

6.10 Asynchronous I/O

Asynchronous I/O is supported if the program is linked against the POSIX thread library. If that is not the case, all I/O is performed as synchronous. On systems which do not support pthread condition variables, such as AIX, I/O is also performed as synchronous.

On some systems, such as Darwin or Solaris, the POSIX thread library is always linked in, so asynchronous I/O is always performed. On other systems, such as Linux, it is necessary to specify `-pthread`, `-lpthread` or `-fopenmp` during the linking step.
The two sections below detail the extensions to standard Fortran that are implemented in GNU Fortran, as well as some of the popular or historically important extensions that are not (or not yet) implemented. For the latter case, we explain the alternatives available to GNU Fortran users, including replacement by standard-conforming code or GNU extensions.

7.1 Extensions implemented in GNU Fortran

GNU Fortran implements a number of extensions over standard Fortran. This chapter contains information on their syntax and meaning. There are currently two categories of GNU Fortran extensions, those that provide functionality beyond that provided by any standard, and those that are supported by GNU Fortran purely for backward compatibility with legacy compilers. By default, -std=gnu allows the compiler to accept both types of extensions, but to warn about the use of the latter. Specifying either -std=f95, -std=f2003, -std=f2008, or -std=f2018 disables both types of extensions, and -std=legacy allows both without warning. The special compile flag -fdec enables additional compatibility extensions along with those enabled by -std=legacy.

7.1.1 Old-style kind specifications

GNU Fortran allows old-style kind specifications in declarations. These look like:

```
TYPESPEC*size x,y,z
```

where TYPESPEC is a basic type (INTEGER, REAL, etc.), and where size is a byte count corresponding to the storage size of a valid kind for that type. (For COMPLEX variables, size is the total size of the real and imaginary parts.) The statement then declares x, y and z to be of type TYPESPEC with the appropriate kind. This is equivalent to the standard-conforming declaration

```
TYPESPEC(k) x,y,z
```

where k is the kind parameter suitable for the intended precision. As kind parameters are implementation-dependent, use the KIND, SELECTED_INT_KIND and SELECTED_REAL_KIND intrinsics to retrieve the correct value, for instance REAL*8 x can be replaced by:

```
INTEGER, PARAMETER :: dbl = KIND(1.0d0)
REAL(KIND=dbl) :: x
```
7.1.2 Old-style variable initialization

GNU Fortran allows old-style initialization of variables of the form:

```fortran
INTEGER i/1/, j/2/
REAL x(2,2) /3*0.,1./
```

The syntax for the initializers is as for the DATA statement, but unlike in a DATA statement, an initializer only applies to the variable immediately preceding the initialization. In other words, something like `INTEGER I,J/2,3/` is not valid.

This style of initialization is only allowed in declarations without double colons (`::`); the double colons were introduced in Fortran 90, which also introduced a standard syntax for initializing variables in type declarations.

Examples of standard-conforming code equivalent to the above example are:

```fortran
! Fortran 90
INTEGER :: i = 1, j = 2
REAL :: x(2,2) = RESHAPE((/0.,0..0.,1./),SHAPE(x))
! Fortran 77
INTEGER i, j
REAL x(2,2)
DATA i/1/, j/2/, x/3*0.,1./
```

Note that variables which are explicitly initialized in declarations or in DATA statements automatically acquire the SAVE attribute.

7.1.3 Extensions to namelist

GNU Fortran fully supports the Fortran 95 standard for namelist I/O including array qualifiers, substrings and fully qualified derived types. The output from a namelist write is compatible with namelist read. The output has all names in upper case and indentation to column 1 after the namelist name. Two extensions are permitted:

Old-style use of $ instead of &

```
$MYNML
X(:)%Y(2) = 1.0 2.0 3.0
CH(1:4) = "abcd"
$END
```

It should be noted that the default terminator is `/` rather than `&END`.

Querying of the namelist when inputting from stdin. After at least one space, entering `?` sends to stdout the namelist name and the names of the variables in the namelist:

```
? 
&mynml 
x 
x%y 
ch 
&end
```

Entering `=?` outputs the namelist to stdout, as if WRITE(*,NML = mynml) had been called:

```
=?
&MYNML
X(1)%Y= 0.000000 , 1.000000 , 0.000000 ,
```

(continues on next page)
X(2)%Y= 0.000000 , 2.000000 , 0.000000 ,
X(3)%Y= 0.000000 , 3.000000 , 0.000000 ,
CH=abcd, /

To aid this dialog, when input is from stdin, errors send their messages to stderr and execution continues, even if IOSTAT
is set.
PRINT namelist is permitted. This causes an error if -std=f95 is used.

```
PROGRAM test_print
  REAL, dimension (4) :: x = (/1.0, 2.0, 3.0, 4.0/)
  NAMELIST /mynml/ x
  PRINT mynml
END PROGRAM test_print
```

Expanded namelist reads are permitted. This causes an error if -std=f95 is used. In the following example, the first
element of the array will be given the value 0.00 and the two succeeding elements will be given the values 1.00 and 2.00.

```
&MYNML
  X(1,1) = 0.00 , 1.00 , 2.00
/
```

When writing a namelist, if no DELIM= is specified, by default a double quote is used to delimit character strings. If
-std=F95, F2003, or F2008, etc, the delim status is set to 'none'. Defaulting to quotes ensures that namelists with character
strings can be subsequently read back in accurately.

### 7.1.4 X format descriptor without count field

To support legacy codes, GNU Fortran permits the count field of the X edit descriptor in FORMAT statements to be
omitted. When omitted, the count is implicitly assumed to be one.

```
PRINT 10, 2, 3
10 FORMAT (I1, X, I1)
```

### 7.1.5 Commas in FORMAT specifications

To support legacy codes, GNU Fortran allows the comma separator to be omitted immediately before and after character
string edit descriptors in FORMAT statements. A comma with no following format descriptor is permitted if the -fdec-
blank-format-item is given on the command line. This is considered non-conforming code and is discouraged.

```
PRINT 10, 2, 3
10 FORMAT ('FOO='I1' BAR='I2)
print 20, 5, 6
20 FORMAT (I3, I3)
```
7.1.6 Missing period in FORMAT specifications

To support legacy codes, GNU Fortran allows missing periods in format specifications if and only if `-std=legacy` is given on the command line. This is considered non-conforming code and is discouraged.

```
REAL :: value
READ(*,10) value
10 FORMAT ('F4')
```

7.1.7 Default widths for F, G and I format descriptors

To support legacy codes, GNU Fortran allows width to be omitted from format specifications if and only if `-fdec-format-defaults` is given on the command line. Default widths will be used. This is considered non-conforming code and is discouraged.

```
REAL :: value1
INTEGER :: value2
WRITE(*,10) value1, value1, value2
10 FORMAT ('F, G, I')
```

7.1.8 I/O item lists

To support legacy codes, GNU Fortran allows the input item list of the `READ` statement, and the output item lists of the `WRITE` and `PRINT` statements, to start with a comma.

7.1.9 Q exponent-letter

GNU Fortran accepts real literal constants with an exponent-letter of `Q`, for example, `1.23Q45`. The constant is interpreted as a `REAL(16)` entity on targets that support this type. If the target does not support `REAL(16)` but has a `REAL(10)` type, then the real-literal-constant will be interpreted as a `REAL(10)` entity. In the absence of `REAL(16)` and `REAL(10)`, an error will occur.

7.1.10 BOZ literal constants

Besides decimal constants, Fortran also supports binary (b), octal (o) and hexadecimal (z) integer constants. The syntax is: `prefix quote digits quote`, where the prefix is either b, o or z, quote is either ' or " and the digits are 0 or 1 for binary, between 0 and 7 for octal, and between 0 and F for hexadecimal. (Example: b'01011101'.)

Up to Fortran 95, BOZ literal constants were only allowed to initialize integer variables in DATA statements. Since Fortran 2003 BOZ literal constants are also allowed as actual arguments to the `REAL`, `DBLE`, `INT` and `CMPLX` intrinsic functions. The BOZ literal constant is simply a string of bits, which is padded or truncated as needed, during conversion to a numeric type. The Fortran standard states that the treatment of the sign bit is processor dependent. Gfortran interprets the sign bit as a user would expect.

As a deprecated extension, GNU Fortran allows hexadecimal BOZ literal constants to be specified using the X prefix. That the BOZ literal constant can also be specified by adding a suffix to the string, for example, Z'ABC' and 'ABC'X are equivalent. Additionally, as extension, BOZ literals are permitted in some contexts outside of DATA and the intrinsic functions listed in the Fortran standard. Use `-fallow-invalid-boz` to enable the extension.
7.1.11 Real array indices

As an extension, GNU Fortran allows the use of REAL expressions or variables as array indices.

7.1.12 Unary operators

As an extension, GNU Fortran allows unary plus and unary minus operators to appear as the second operand of binary arithmetic operators without the need for parenthesis.

\[ X = Y \times -Z \]

7.1.13 Implicitly convert LOGICAL and INTEGER values

As an extension for backwards compatibility with other compilers, GNU Fortran allows the implicit conversion of LOGICAL values to INTEGER values and vice versa. When converting from a LOGICAL to an INTEGER, .FALSE. is interpreted as zero, and .TRUE. is interpreted as one. When converting from INTEGER to LOGICAL, the value zero is interpreted as .FALSE. and any nonzero value is interpreted as .TRUE..

```
LOGICAL :: l
  l = 1

INTEGER :: i
  i = .TRUE.
```

However, there is no implicit conversion of INTEGER values in if-statements, nor of LOGICAL or INTEGER values in I/O operations.

7.1.14 Hollerith constants support

GNU Fortran supports Hollerith constants in assignments, DATA statements, function and subroutine arguments. A Hollerith constant is written as a string of characters preceded by an integer constant indicating the character count, and the letter H or h, and stored in bytewise fashion in a numeric (INTEGER, REAL, or COMPLEX), LOGICAL or CHARACTER variable. The constant will be padded with spaces or truncated to fit the size of the variable in which it is stored.

Examples of valid uses of Hollerith constants:

```
complex*16 x(2)
data x /16Habcdefhijklmnop, 16Hqrstuvwxyz012345/
x(1) = 16HABCDEFGHIJKLMNOP
call foo (4h abc)
```

Examples of Hollerith constants:

```
integer*4 a
  a = 0H ! Invalid, at least one character is needed.
  a = 4HAB12 ! Valid
  a = 8H12345678 ! Valid, but the Hollerith constant will be truncated.
  a = 3Hxyz ! Valid, but the Hollerith constant will be padded.
```

In general, Hollerith constants were used to provide a rudimentary facility for handling character strings in early Fortran compilers, prior to the introduction of CHARACTER variables in Fortran 77; in those cases, the standard-compliant equivalent is to convert the program to use proper character strings. On occasion, there may be a case where the intent is
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

specifically to initialize a numeric variable with a given byte sequence. In these cases, the same result can be obtained by using the `TRANSFER` statement, as in this example.

```
integer(kind=4) :: a
a = transfer("abcd", a)  ! equivalent to: a = 4Habcd
```

The use of the `-fdec` option extends support of Hollerith constants to comparisons:

```
integer*4 a
a = 4HABCD
if (a .ne. 4habcd) then
  write(*,*) "no match"
end if
```

Supported types are numeric (INTEGER, REAL, or COMPLEX), and CHARACTER.

### 7.1.15 Character conversion

Allowing character literals to be used in a similar way to Hollerith constants is a non-standard extension. This feature is enabled using `-fdec-char-conversions` and only applies to character literals of `kind=1`.

Character literals can be used in DATA statements and assignments with numeric (INTEGER, REAL, or COMPLEX) or LOGICAL variables. Like Hollerith constants they are copied byte-wise fashion. The constant will be padded with spaces or truncated to fit the size of the variable in which it is stored.

Examples:

```
integer*4 x
data x / 'abcd' /

x = 'A'  ! Will be padded.
x = 'ab123'  ! Will be truncated.
```

### 7.1.16 Cray pointers

Cray pointers are part of a non-standard extension that provides a C-like pointer in Fortran. This is accomplished through a pair of variables: an integer “pointer” that holds a memory address, and a “pointee” that is used to dereference the pointer.

Pointer/pointee pairs are declared in statements of the form:

```
pointer ( <pointer> , <pointee> )
```

or,

```
pointer ( <pointer1> , <pointee1> ), ( <pointer2> , <pointee2> ), ...
```

The pointer is an integer that is intended to hold a memory address. The pointee may be an array or scalar. If an assumed-size array is permitted within the scoping unit, a pointee can be an assumed-size array. That is, the last dimension may be left unspecified by using a * in place of a value. A pointee cannot be an assumed shape array. No space is allocated for the pointee.

The pointee may have its type declared before or after the pointer statement, and its array specification (if any) may be declared before, during, or after the pointer statement. The pointer may be declared as an integer prior to the pointer statement. However, some machines have default integer sizes that are different than the size of a pointer, and so the following code is not portable:
integer ipt
pointer (ipt, iarr)

If a pointer is declared with a kind that is too small, the compiler will issue a warning; the resulting binary will probably not work correctly, because the memory addresses stored in the pointers may be truncated. It is safer to omit the first line of the above example; if explicit declaration of ipt's type is omitted, then the compiler will ensure that ipt is an integer variable large enough to hold a pointer.

Pointer arithmetic is valid with Cray pointers, but it is not the same as C pointer arithmetic. Cray pointers are just ordinary integers, so the user is responsible for determining how many bytes to add to a pointer in order to increment it. Consider the following example:

real target(10)
real pointee(10)
pointer (ipt, pointee)
ipt = loc (target)
ipt = ipt + 1

The last statement does not set ipt to the address of target(1), as it would in C pointer arithmetic. Adding 1 to ipt just adds one byte to the address stored in ipt.

Any expression involving the pointee will be translated to use the value stored in the pointer as the base address.

To get the address of elements, this extension provides an intrinsic function LOC(). The LOC() function is equivalent to the & operator in C, except the address is cast to an integer type:

real ar(10)
pointer (ipt, arpte(10))
real arpte
ipt = loc (ar)  ! Makes arpte is an alias for ar
arpte(1) = 1.0 ! Sets ar(1) to 1.0

The pointer can also be set by a call to the MALLOC intrinsic (see MALLOC).

Cray pointees often are used to alias an existing variable. For example:

integer target(10)
integer iarr(10)
pointer (ipt, iarr)
ipt = loc (target)

As long as ipt remains unchanged, iarr is now an alias for target. The optimizer, however, will not detect this aliasing, so it is unsafe to use iarr and target simultaneously. Using a pointee in any way that violates the Fortran aliasing rules or assumptions is illegal. It is the user's responsibility to avoid doing this; the compiler works under the assumption that no such aliasing occurs.

Cray pointers will work correctly when there is no aliasing (i.e., when they are used to access a dynamically allocated block of memory), and also in any routine where a pointee is used, but any variable with which it shares storage is not used. Code that violates these rules may not run as the user intends. This is not a bug in the optimizer; any code that violates the aliasing rules is illegal. (Note that this is not unique to GNU Fortran; any Fortran compiler that supports Cray pointers will ‘incorrectly’ optimize code with illegal aliasing.)

There are a number of restrictions on the attributes that can be applied to Cray pointers and pointees. Pointees may not have the ALLOCATABLE, INTENT, OPTIONAL, DUMMY, TARGET, INTRINSIC, or POINTER attributes. Pointers may not have the DIMENSION, POINTER, TARGET, ALLOCATABLE, EXTERNAL, or INTRINSIC attributes, nor may they be function results. Pointees may not occur in more than one pointer statement. A pointee cannot be a pointer. Pointees cannot occur in equivalence, common, or data statements.

A Cray pointer may also point to a function or a subroutine. For example, the following excerpt is valid:

7.1. Extensions implemented in GNU Fortran
A pointer may be modified during the course of a program, and this will change the location to which the pointee refers. However, when pointees are passed as arguments, they are treated as ordinary variables in the invoked function. Subsequent changes to the pointer will not change the base address of the array that was passed.

### 7.1.17 CONVERT specifier

GNU Fortran allows the conversion of unformatted data between little- and big-endian representation to facilitate moving of data between different systems. The conversion can be indicated with the `CONVERT` specifier on the `OPEN` statement. See [GFORTRAN_CONVERT_UNIT—Set endianness for unformatted I/O](#), for an alternative way of specifying the data format via an environment variable.

Valid values for `CONVERT` are:

- `CONVERT='NATIVE'` Use the native format. This is the default.
- `CONVERT='SWAP'` Swap between little- and big-endian.
- `CONVERT='LITTLE_ENDIAN'` Use the little-endian representation for unformatted files.
- `CONVERT='BIG_ENDIAN'` Use the big-endian representation for unformatted files.

Using the option could look like this:

```fortran
open(file='big.dat',form='unformatted',access='sequential', &
     convert='big_endian')
```

The value of the conversion can be queried by using `INQUIRE(CONVERT=ch)`. The values returned are 'BIG_ENDIAN' and 'LITTLE_ENDIAN'.

`CONVERT` works between big- and little-endian for `INTEGER` values of all supported kinds and for `REAL` on IEEE systems of kinds 4 and 8. Conversion between different 'extended double' types on different architectures such as m68k and x86_64, which GNU Fortran supports as `REAL(KIND=10)` and `REAL(KIND=16)`, will probably not work.

*Note that the values specified via the `GFORTRAN_CONVERT_UNIT` environment variable will override the `CONVERT` specifier in the open statement.* This is to give control over data formats to users who do not have the source code of their program available.

Using anything but the native representation for unformatted data carries a significant speed overhead. If speed in this area matters to you, it is best if you use this only for data that needs to be portable.
7.1.18 OpenMP

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C/C++ and Fortran on many architectures, including Unix and Microsoft Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

GNU Fortran strives to be compatible to the OpenMP Application Program Interface v4.5.

To enable the processing of the OpenMP directive !$omp in free-form source code; the c$omp, *$omp and !$omp directives in fixed form; the !$ conditional compilation sentinels in free form; and the c$, *$ and !$ sentinels in fixed form, gfortran needs to be invoked with the -fopenmp. This also arranges for automatic linking of the GNU Offloading and Multi Processing Runtime Library ToplibgomplibgompGNU Offloading and Multi Processing Runtime Library.

The OpenMP Fortran runtime library routines are provided both in a form of a Fortran 90 module named omp_lib and in a form of a Fortran include file named omp_lib.h.

An example of a parallelized loop taken from Appendix A.1 of the OpenMP Application Program Interface v2.5:

```fortran
SUBROUTINE A1(N, A, B)
   INTEGER I, N
   REAL B(N), A(N)
   !$OMP PARALLEL DO !I is private by default
   DO I = 2, N
       B(I) = (A(I) + A(I-1)) / 2.0
   ENDDO
   !$OMP END PARALLEL DO
END SUBROUTINE A1
```

Please note:

- -fopenmp implies -frecursive, i.e., all local arrays will be allocated on the stack. When porting existing code to OpenMP, this may lead to surprising results, especially to segmentation faults if the stacksize is limited.

- On glibc-based systems, OpenMP enabled applications cannot be statically linked due to limitations of the underlying pthreads-implementation. It might be possible to get a working solution if -Wl,--whole-archive -lpthread -Wl,--no-whole-archive is added to the command line. However, this is not supported by gcc and thus not recommended.

7.1.19 OpenACC

OpenACC is an application programming interface (API) that supports offloading of code to accelerator devices. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

GNU Fortran strives to be compatible to the OpenACC Application Programming Interface v2.6.

To enable the processing of the OpenACC directive !$acc in free-form source code; the c$acc, *$acc and !$acc directives in fixed form; the !$ conditional compilation sentinels in free form; and the c$, *$ and !$ sentinels in fixed form, gfortran needs to be invoked with the -fopenacc. This also arranges for automatic linking of the GNU Offloading and Multi Processing Runtime Library ToplibgomplibgompGNU Offloading and Multi Processing Runtime Library.

The OpenACC Fortran runtime library routines are provided both in a form of a Fortran 90 module named openacc and in a form of a Fortran include file named openacc_lib.h.
7.1.20 Argument list functions %VAL, %REF and %LOC

GNU Fortran supports argument list functions %VAL, %REF and %LOC statements, for backward compatibility with g77. It is recommended that these should be used only for code that is accessing facilities outside of GNU Fortran, such as operating system or windowing facilities. It is best to constrain such uses to isolated portions of a program—portions that deal specifically and exclusively with low-level, system-dependent facilities. Such portions might well provide a portable interface for use by the program as a whole, but are themselves not portable, and should be thoroughly tested each time they are rebuilt using a new compiler or version of a compiler.

%VAL passes a scalar argument by value, %REF passes it by reference and %LOC passes its memory location. Since gfortran already passes scalar arguments by reference, %REF is in effect a do-nothing. %LOC has the same effect as a Fortran pointer.

An example of passing an argument by value to a C subroutine foo.:

```c
C
C prototype void foo_ (float x);
C

extern foo

real*4 x
x = 3.14159
call foo (%VAL (x))
end
```

For details refer to the g77 manual https://gcc.gnu.org/onlinedocs/gcc-3.4.6/g77/index.html#Top.

Also, c_by_val.f and its partner c_by_val.c of the GNU Fortran testsuite are worth a look.

7.1.21 Read/Write after EOF marker

Some legacy codes rely on allowing READ or WRITE after the EOF file marker in order to find the end of a file. GNU Fortran normally rejects these codes with a run-time error message and suggests the user consider BACKSPACE or REWIND to properly position the file before the EOF marker. As an extension, the run-time error may be disabled using -std=legacy.

7.1.22 STRUCTURE and RECORD

Record structures are a pre-Fortran-90 vendor extension to create user-defined aggregate data types. Support for record structures in GNU Fortran can be enabled with the -fdec-structure compile flag. If you have a choice, you should instead use Fortran 90’s ‘derived types’, which have a different syntax.

In many cases, record structures can easily be converted to derived types. To convert, replace STRUCTURE /structure-name / by TYPE type-name. Additionally, replace RECORD /structure-name / by TYPE(type-name ). Finally, in the component access, replace the period (.) by the percent sign (%).

Here is an example of code using the non portable record structure syntax:

```fortran
! Declaring a structure named "item" and containing three fields:
! an integer ID, an description string and a floating-point price.
STRUCTURE /item/
   INTEGER id
   CHARACTER(LEN=200) description
   REAL price
END STRUCTURE

! Define two variables, an single record of type "item"
! named "pear", and an array of items named "store_catalog"
```

(continues on next page)
RECORD /item/ pear, store_catalog(100)

! We can directly access the fields of both variables
pear.id = 92316
pear.description = "juicy D'Anjou pear"
pear.price = 0.15
store_catalog(7).id = 7831
store_catalog(7).description = "milk bottle"
store_catalog(7).price = 1.2

! We can also manipulate the whole structure
store_catalog(12) = pear
print *, store_catalog(12)

This code can easily be rewritten in the Fortran 90 syntax as following:

! ``STRUCTURE /name/ ... END STRUCTURE'' becomes
! ``TYPE name ... END TYPE''

TYPE item
   INTEGER id
   CHARACTER(LEN=200) description
   REAL price
END TYPE

! ``RECORD /name/ variable'' becomes ``TYPE(name) variable''

TYPE(item) pear, store_catalog(100)

! Instead of using a dot (.) to access fields of a record, the
! standard syntax uses a percent sign (%)
pear%id = 92316
pear%description = "juicy D'Anjou pear"
pear%price = 0.15
store_catalog(7)%id = 7831
store_catalog(7)%description = "milk bottle"
store_catalog(7)%price = 1.2

! Assignments of a whole variable do not change
store_catalog(12) = pear
print *, store_catalog(12)

GNU Fortran implements STRUCTURES like derived types with the following rules and exceptions:

- Structures act like derived types with the SEQUENCE attribute. Otherwise they may contain no specifiers.

- Structures may contain a special field with the name %FILL. This will create an anonymous component which cannot be accessed but occupies space just as if a component of the same type was declared in its place, useful for alignment purposes. As an example, the following structure will consist of at least sixteen bytes:

    structure /padded/
    character(4) start
    character(8) %FILL
    character(4) end
    end structure

- Structures may share names with other symbols. For example, the following is invalid for derived types, but valid for structures:

7.1. Extensions implemented in GNU Fortran
structure /header/
   ! ...
end structure
record /header/ header

• Structure types may be declared nested within another parent structure. The syntax is:

structure /type-name/
   ...
   structure [/<type-name>/] <field-list>
...

The type name may be omitted, in which case the structure type itself is anonymous, and other structures of the same type cannot be instantiated. The following shows some examples:

structure /appointment/
   ! nested structure definition: app_time is an array of two 'time'
structure /time/ app_time (2)
   integer(1) hour, minute
end structure
character(10) memo
end structure

! The 'time' structure is still usable
record /time/ now
now = time(5, 30)
...

structure /appointment/
   ! anonymous nested structure definition
structure start, end
   integer(1) hour, minute
end structure
character(10) memo
end structure

• Structures may contain UNION blocks. For more detail see the section on UNION and MAP.

• Structures support old-style initialization of components, like those described in Old-style variable initialization. For array initializers, an initializer may contain a repeat specification of the form `<literal-integer> * <constant-initializer>`. The value of the integer indicates the number of times to repeat the constant initializer when expanding the initializer list.

7.1.23 UNION and MAP

Unions are an old vendor extension which were commonly used with the non-standard STRUCTURE and RECORD extensions. Use of UNION and MAP is automatically enabled with -fdec-structure.

A UNION declaration occurs within a structure; within the definition of each union is a number of MAP blocks. Each MAP shares storage with its sibling maps (in the same union), and the size of the union is the size of the largest map within it, just as with unions in C. The major difference is that component references do not indicate which union or map the component is in (the compiler gets to figure that out).

Here is a small example:
structure /myunion/
union
  map
    character(2) w0, w1, w2
  end map
map
  character(6) long
end map
end union
end structure

record /myunion/ rec
! After this assignment...
rec.long = 'hello!'

! The following is true:
! rec.w0 === 'he'
! rec.w1 === 'll'
! rec.w2 === 'o!'

The two maps share memory, and the size of the union is ultimately six bytes:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>Byte offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>--------------</td>
</tr>
</tbody>
</table>

\--------/ \--------/ \--------/
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\
\      \      \      \\n
Following is an example mirroring the layout of an Intel x86_64 register:

structure /reg/
union ! U0 ! rax
  map
    character(16) rx
  end map
map
  character(8) rh ! rah
union ! U1
  map
    character(8) rl ! ral
  end map
map
  character(8) ex ! eax
end map
map
  character(4) eh ! eah
union ! U2
  map
    character(4) el ! eal
end map
map
  character(4) x ! ax

(continues on next page)
end map
map
    character(2) h ! ah
    character(2) l ! al
end map
end union
end map
end union
end union
end union
end structure
record /reg/ a

! After this assignment...
a.rx = 'AAAAAAAA.BBB.C.D'

! The following is true:
a.rx  === 'AAAAAAAA.BBB.C.D'
a.rh  === 'AAAAAAAA'
a.rl  === '.BBB.C.D'
a.ex  === '.BBB.C.D'
a.eh  === '.BBB'
a.el  === '.C.D'
a.x   === '.C.D'
a.h   === '.C'
a.l   === '.D'

### 7.1.24 Type variants for integer intrinsics

Similar to the D/C prefixes to real functions to specify the input/output types, GNU Fortran offers B/I/J/K prefixes to integer functions for compatibility with DEC programs. The types implied by each are:

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>INTEGER(kind=1)</td>
</tr>
<tr>
<td>I</td>
<td>INTEGER(kind=2)</td>
</tr>
<tr>
<td>J</td>
<td>INTEGER(kind=4)</td>
</tr>
<tr>
<td>K</td>
<td>INTEGER(kind=8)</td>
</tr>
</tbody>
</table>

GNU Fortran supports these with the flag `-fdec-intrinsic-ints`. Intrinsics for which prefixed versions are available and in what form are noted in Intrinsic Procedures. The complete list of supported intrinsics is here:
### 7.1.25 AUTOMATIC and STATIC attributes

With `-fdec-static` GNU Fortran supports the DEC extended attributes `STATIC` and `AUTOMATIC` to provide explicit specification of entity storage. These follow the syntax of the Fortran standard `SAVE` attribute.

`STATIC` is exactly equivalent to `SAVE`, and specifies that an entity should be allocated in static memory. As an example, `STATIC` local variables will retain their values across multiple calls to a function.

Entities marked `AUTOMATIC` will be stack automatic whenever possible. `AUTOMATIC` is the default for local variables smaller than `-fmax-stack-var-size`, unless `-fno-automatic` is given. This attribute overrides `-fno-automatic`, `-fmax-stack-var-size`, and blanket `SAVE` statements.

Examples:

```fortran
subroutine f
  integer, automatic :: i    ! automatic variable
  integer x, y               ! static variables
  save
  ...                      
end subroutine

subroutine f
  integer a, b, c, x, y, z
  static :: x
  save y
  automatic z, c
  ! a, b, c, and z are automatic
  ! x and y are static
end subroutine

! Compiled with -fno-automatic
subroutine f
  integer a, b, c, d
  automatic :: a
  ! a is automatic; b, c, and d are static
end subroutine
```
7.1.26 Extended math intrinsics

GNU Fortran supports an extended list of mathematical intrinsics with the compile flag `-fdec-math` for compatibility with legacy code. These intrinsics are described fully in Intrinsic Procedures where it is noted that they are extensions and should be avoided whenever possible.

Specifically, `-fdec-math` enables the COTAN intrinsic, and trigonometric intrinsics which accept or produce values in degrees instead of radians. Here is a summary of the new intrinsics:

<table>
<thead>
<tr>
<th>Radians</th>
<th>Degrees</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACOS</td>
<td>ACOSD *</td>
</tr>
<tr>
<td>ASIN</td>
<td>ASIND *</td>
</tr>
<tr>
<td>ATAN</td>
<td>ATAND *</td>
</tr>
<tr>
<td>ATAN2</td>
<td>ATAN2D *</td>
</tr>
<tr>
<td>COS</td>
<td>COSD *</td>
</tr>
<tr>
<td>COTAN</td>
<td>COTAND *</td>
</tr>
<tr>
<td>SIN</td>
<td>SIND *</td>
</tr>
<tr>
<td>TAN</td>
<td>TAND *</td>
</tr>
</tbody>
</table>

* Enabled with `-fdec-math`.

For advanced users, it may be important to know the implementation of these functions. They are simply wrappers around the standard radian functions, which have more accurate builtin versions. These functions convert their arguments (or results) to degrees (or radians) by taking the value modulus 360 (or 2*pi) and then multiplying it by a constant radian-to-degree (or degree-to-radian) factor, as appropriate. The factor is computed at compile-time as 180/pi (or pi/180).

7.1.27 Form feed as whitespace

Historically, legacy compilers allowed insertion of form feed characters (‘f’, ASCII 0xC) at the beginning of lines for formatted output to line printers, though the Fortran standard does not mention this. GNU Fortran supports the interpretation of form feed characters in source as whitespace for compatibility.

7.1.28 TYPE as an alias for PRINT

For compatibility, GNU Fortran will interpret TYPE statements as PRINT statements with the flag `-fdec`. With this flag asserted, the following two examples are equivalent:

```fortran
TYPE *, 'hello world'

PRINT *, 'hello world'
```

7.1.29 %LOC as an rvalue

Normally `%LOC` is allowed only in parameter lists. However the intrinsic function LOC does the same thing, and is usable as the right-hand-side of assignments. For compatibility, GNU Fortran supports the use of `%LOC` as an alias for the builtin LOC with `-std=legacy`. With this feature enabled the following two examples are equivalent:

```fortran
integer :: i, l
l = %loc(i)
call sub(l)
```
integer :: i
call sub(%loc(i))

7.1.30 .XOR. operator

GNU Fortran supports .XOR. as a logical operator with -std=legacy for compatibility with legacy code. .XOR. is equivalent to .NEQV.. That is, the output is true if and only if the inputs differ.

7.1.31 Bitwise logical operators

With -fdec, GNU Fortran relaxes the type constraints on logical operators to allow integer operands, and performs the corresponding bitwise operation instead. This flag is for compatibility only, and should be avoided in new code. Consider:

```
INTEGER :: i, j
i = z'33'
j = z'cc'
print *, i .AND. j
```

In this example, compiled with -fdec, GNU Fortran will replace the .AND. operation with a call to the intrinsic IAND function, yielding the bitwise-and of i and j.

Note that this conversion will occur if at least one operand is of integral type. As a result, a logical operand will be converted to an integer when the other operand is an integer in a logical operation. In this case, .TRUE. is converted to 1 and .FALSE. to 0.

Here is the mapping of logical operator to bitwise intrinsic used with -fdec:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Intrinsic</th>
<th>Bitwise operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>.NOT.</td>
<td>NOT</td>
<td>complement</td>
</tr>
<tr>
<td>.AND.</td>
<td>IAND</td>
<td>intersection</td>
</tr>
<tr>
<td>.OR.</td>
<td>IOR</td>
<td>union</td>
</tr>
<tr>
<td>.NEQV.</td>
<td>IEOR</td>
<td>exclusive or</td>
</tr>
<tr>
<td>.EQV.</td>
<td>NOT(IEOR)</td>
<td>complement of exclusive or</td>
</tr>
</tbody>
</table>

7.1.32 Extended I/O specifiers

GNU Fortran supports the additional legacy I/O specifiers CARRIAGECONTROL, READONLY, and SHARE with the compile flag -fdec, for compatibility.

CARRIAGECONTROL

The CARRIAGECONTROL specifier allows a user to control line termination settings between output records for an I/O unit. The specifier has no meaning for readonly files. When CARRIAGECONTROL is specified upon opening a unit for formatted writing, the exact CARRIAGECONTROL setting determines what characters to write between output records. The syntax is:

```
OPEN(..., CARRIAGECONTROL=cc)
```

Where cc is a character expression that evaluates to one of the following values:

<table>
<thead>
<tr>
<th>cc</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'LIST'</td>
<td>One line feed between records (default)</td>
</tr>
<tr>
<td>'FORTRAN'</td>
<td>Legacy interpretation of the first character (see below)</td>
</tr>
<tr>
<td>'NONE'</td>
<td>No separator between records</td>
</tr>
</tbody>
</table>
With \texttt{CARRIAGECONTROL='FORTRAN'}, when a record is written, the first character of the input record is not written, and instead determines the output record separator as follows:

<table>
<thead>
<tr>
<th>Leading character</th>
<th>Meaning</th>
<th>Output separating character(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>'+'</td>
<td>Overprinting</td>
<td>Carriage return only</td>
</tr>
<tr>
<td>'-'</td>
<td>New line</td>
<td>Line feed and carriage return</td>
</tr>
<tr>
<td>'0'</td>
<td>Skip line</td>
<td>Two line feeds and carriage return</td>
</tr>
<tr>
<td>'1'</td>
<td>New page</td>
<td>Form feed and carriage return</td>
</tr>
<tr>
<td>'$'</td>
<td>Prompting</td>
<td>Line feed (no carriage return)</td>
</tr>
<tr>
<td>\texttt{CHAR(0)}</td>
<td>Overprinting (no advance)</td>
<td>None</td>
</tr>
</tbody>
</table>

**READONLY**

The \texttt{READONLY} specifier may be given upon opening a unit, and is equivalent to specifying \texttt{ACTION='READ'}, except that the file may not be deleted on close (\texttt{i.e.} \texttt{CLOSE with STATUS="DELETE"}). The syntax is:

\begin{verbatim}
OPEN(..., READONLY)
\end{verbatim}

**SHARE**

The \texttt{SHARE} specifier allows system-level locking on a unit upon opening it for controlled access from multiple processes/threads. The \texttt{SHARE} specifier has several forms:

\begin{verbatim}
OPEN(..., SHARE=sh)
OPEN(..., SHARED)
OPEN(..., NOSHARED)
\end{verbatim}

Where \texttt{sh} in the first form is a character expression that evaluates to a value as seen in the table below. The latter two forms are aliases for particular values of \texttt{sh}:

<table>
<thead>
<tr>
<th>Explicit form</th>
<th>Short form</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{SHARE='DENYRW'}</td>
<td>NOSHARED</td>
<td>Exclusive (write) lock</td>
</tr>
<tr>
<td>\texttt{SHARE='DENYNONE'}</td>
<td>SHARED</td>
<td>Shared (read) lock</td>
</tr>
</tbody>
</table>

In general only one process may hold an exclusive (write) lock for a given file at a time, whereas many processes may hold shared (read) locks for the same file.

The behavior of locking may vary with your operating system. On POSIX systems, locking is implemented with \texttt{fcntl}. Consult your corresponding operating system's manual pages for further details. Locking via \texttt{SHARE=} is not supported on other systems.

### 7.1.33 Legacy \texttt{PARAMETER} statements

For compatibility, GNU Fortran supports legacy \texttt{PARAMETER} statements without parentheses with \texttt{-std=legacy}. A warning is emitted if used with \texttt{-std=gnu}, and an error is acknowledged with a real Fortran standard flag (\texttt{-std=f95}, etc...). These statements take the following form:

\begin{verbatim}
implicit real (E)
parameter e = 2.718282
real c
parameter c = 3.0e8
\end{verbatim}
7.1.34 Default exponents

For compatibility, GNU Fortran supports a default exponent of zero in real constants with `-fdec`. For example, `9e` would be interpreted as `9e0`, rather than an error.

7.2 Extensions not implemented in GNU Fortran

The long history of the Fortran language, its wide use and broad userbase, the large number of different compiler vendors and the lack of some features crucial to users in the first standards have lead to the existence of a number of important extensions to the language. While some of the most useful or popular extensions are supported by the GNU Fortran compiler, not all existing extensions are supported. This section aims at listing these extensions and offering advice on how best make code that uses them running with the GNU Fortran compiler.

7.2.1 ENCODE and DECODE statements

GNU Fortran does not support the `ENCODE` and `DECODE` statements. These statements are best replaced by `READ` and `WRITE` statements involving internal files (`CHARACTER` variables and arrays), which have been part of the Fortran standard since Fortran 77. For example, replace a code fragment like

```
INTEGER*1 LINE(80)
REAL A, B, C
c ... Code that sets LINE
DECODER (80, 9000, LINE) A, B, C
9000 FORMAT (1X, 3(F10.5))
```

with the following:

```
CHARACTER(LEN=80) LINE
REAL A, B, C
c ... Code that sets LINE
READ (UNIT=LINE, FMT=9000) A, B, C
9000 FORMAT (1X, 3(F10.5))
```

Similarly, replace a code fragment like

```
INTEGER*1 LINE(80)
REAL A, B, C
c ... Code that sets A, B and C
ENCODER (80, 9000, LINE) A, B, C
9000 FORMAT (1X, 'OUTPUT IS ', 3(F10.5))
```

with the following:

```
CHARACTER(LEN=80) LINE
REAL A, B, C
c ... Code that sets A, B and C
WRITE (UNIT=LINE, FMT=9000) A, B, C
9000 FORMAT (1X, 'OUTPUT IS ', 3(F10.5))
```
7.2.2 Variable FORMAT expressions

A variable FORMAT expression is format statement which includes angle brackets enclosing a Fortran expression: \texttt{FORMAT(I<N>)}). GNU Fortran does not support this legacy extension. The effect of variable format expressions can be reproduced by using the more powerful (and standard) combination of internal output and string formats. For example, replace a code fragment like this:

```fortran
WRITE(6,20) INT1
20 FORMAT(I<N+1>)
```

with the following:

```fortran
c Variable declaration
    CHARACTER(LEN=20) FMT

c Other code here...

c WRITE(FMT,'("I", I0, ")")' N+1
WRITE(6,FMT) INT1
```

or with:

```fortran
    WRITE(FMT,* N+1
WRITE(6,"(I" // ADJUSTL(FMT) // ")") INT1
```

7.2.3 Alternate complex function syntax

Some Fortran compilers, including \texttt{g77}, let the user declare complex functions with the syntax \texttt{COMPLEX FUNCTION name*16()}, as well as \texttt{COMPLEX*16 FUNCTION name()}. Both are non-standard, legacy extensions. \texttt{gfortran} accepts the latter form, which is more common, but not the former.

7.2.4 Volatile COMMON blocks

Some Fortran compilers, including \texttt{g77}, let the user declare \texttt{COMMON} with the \texttt{VOLATILE} attribute. This is invalid standard Fortran syntax and is not supported by \texttt{gfortran}. Note that \texttt{gfortran} accepts \texttt{VOLATILE} variables in \texttt{COMMON} blocks since revision 4.3.

7.2.5 OPEN( ... NAME=)

Some Fortran compilers, including \texttt{g77}, let the user declare \texttt{OPEN( ... NAME=)}. This is invalid standard Fortran syntax and is not supported by \texttt{gfortran}. \texttt{OPEN( ... NAME=)} should be replaced with \texttt{OPEN( ... FILE=)}. 
7.2.6 Q edit descriptor

Some Fortran compilers provide the Q edit descriptor, which transfers the number of characters left within an input record into an integer variable.

A direct replacement of the Q edit descriptor is not available in gfortran. How to replicate its functionality using standard-conforming code depends on what the intent of the original code is.

Options to replace Q may be to read the whole line into a character variable and then counting the number of non-blank characters left using LEN_TRIM. Another method may be to use formatted stream, read the data up to the position where the Q descriptor occurred, use INQUIRE to get the file position, count the characters up to the next NEW_LINE and then start reading from the position marked previously.
8.1 Interoperability with C

Since Fortran 2003 (ISO/IEC 1539-1:2004(E)) there is a standardized way to generate procedure and derived-type declarations and global variables which are interoperable with C (ISO/IEC 9899:1999). The bind(C) attribute has been added to inform the compiler that a symbol shall be interoperable with C; also, some constraints are added. Note, however, that not all C features have a Fortran equivalent or vice versa. For instance, neither C’s unsigned integers nor C’s functions with variable number of arguments have an equivalent in Fortran.

Note that array dimensions are reversely ordered in C and that arrays in C always start with index 0 while in Fortran they start by default with 1. Thus, an array declaration \( A(n,m) \) in Fortran matches \( A[m][n] \) in C and accessing the element \( A(i,j) \) matches \( A[j-1][i-1] \). The element following \( A(i,j) \) (C: \( A[j-1][i-1] \); assuming \( i < n \)) in memory is \( A(i+1,j) \) (C: \( A[j-1][i] \)).

8.1.1 Intrinsic Types

In order to ensure that exactly the same variable type and kind is used in C and Fortran, the named constants shall be used which are defined in the ISO_C_BINDING intrinsic module. That module contains named constants for kind parameters and character named constants for the escape sequences in C. For a list of the constants, see ISO_C_BINDING.

For logical types, please note that the Fortran standard only guarantees interoperability between C99’s _Bool and Fortran’s C_Bool -kind logicals and C99 defines that true has the value 1 and false the value 0. Using any other integer value with GNU Fortran’s LOGICAL (with any kind parameter) gives an undefined result. (Passing other integer values than 0 and 1 to GCC’s _Bool is also undefined, unless the integer is explicitly or implicitly casted to _Bool.)

8.1.2 Derived Types and struct

For compatibility of derived types with struct, one needs to use the BIND(C) attribute in the type declaration. For instance, the following type declaration

```
USE ISO_C_BINDING
TYPE, BIND(C) :: myType
    INTEGER(C_INT) :: i1, i2
    INTEGER(C_SIGNED_CHAR) :: i3
    REAL(C_DOUBLE) :: d1
    COMPLEX(C_FLOAT_COMPLEX) :: c1
    CHARACTER(KIND=C_CHAR) :: str(5)
END TYPE
```

matches the following struct declaration in C
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

```fortran
struct {
  int i1, i2;
  /* Note: "char" might be signed or unsigned. */
  signed char i3;
  double d1;
  float _Complex c1;
  char str[5];
} myType;
```

Derived types with the C binding attribute shall not have the `sequence` attribute, type parameters, the `extends` attribute, nor type-bound procedures. Every component must be of interoperable type and kind and may not have the `pointer` or `allocatable` attribute. The names of the components are irrelevant for interoperability.

As there exist no direct Fortran equivalents, neither unions nor structs with bit field or variable-length array members are interoperable.

### 8.1.3 Interoperable Global Variables

Variables can be made accessible from C using the C binding attribute, optionally together with specifying a binding name. Those variables have to be declared in the declaration part of a `MODULE`, be of interoperable type, and have neither the `pointer` nor the `allocatable` attribute.

```fortran
MODULE m
  USE myType_module
  USE ISO_C_BINDING
  integer(C_INT), bind(C, name="_MyProject_flags") :: global_flag
  type(myType), bind(C) :: tp
END MODULE
```

Here, `_MyProject_flags` is the case-sensitive name of the variable as seen from C programs while `global_flag` is the case-insensitive name as seen from Fortran. If no binding name is specified, as for `tp`, the C binding name is the (lowercase) Fortran binding name. If a binding name is specified, only a single variable may be after the double colon. Note of warning: You cannot use a global variable to access `errno` of the C library as the C standard allows it to be a macro. Use the `IERRNO` intrinsic (GNU extension) instead.

### 8.1.4 Interoperable Subroutines and Functions

Subroutines and functions have to have the `BIND(C)` attribute to be compatible with C. The dummy argument declaration is relatively straightforward. However, one needs to be careful because C uses call-by-value by default while Fortran behaves usually similar to call-by-reference. Furthermore, strings and pointers are handled differently. Note that in Fortran 2003 and 2008 only explicit size and assumed-size arrays are supported but not assumed-shape or deferred-shape (i.e. allocatable or pointer) arrays. However, those are allowed since the Technical Specification 29113, see Further Interoperability of Fortran with C.

To pass a variable by value, use the `VALUE` attribute. Thus, the following C prototype

```c
int func(int i, int *j)
```

matches the Fortran declaration

```fortran
integer(c_int) function func(i,j)
  use iso_c_binding, only: c_int
  integer(c_int), VALUE :: i
  integer(c_int) :: j
```

Chapter 8. Mixed-Language Programming
Note that pointer arguments also frequently need the `VALUE` attribute, see Working with Pointers.

Strings are handled quite differently in C and Fortran. In C a string is a NUL-terminated array of characters while in Fortran each string has a length associated with it and is thus not terminated (by e.g. NUL). For example, if one wants to use the following C function,

```c
#include <stdio.h>
void print_C(char *string) /* equivalent: char string[] */
{
    printf("%s\n", string);
}
```

to print ‘Hello World’ from Fortran, one can call it using

```fortran
use iso_c_binding, only: C_CHAR, C_NULL_CHAR
interface
    subroutine print_c(string) bind(C, name="print_C")
        use iso_c_binding, only: c_char
        character(kind=c_char) :: string(*)
    end subroutine print_c
end interface
call print_c(C_CHAR_"Hello World"//C_NULL_CHAR)
```

As the example shows, one needs to ensure that the string is NUL terminated. Additionally, the dummy argument `string` of `print_C` is a length-one assumed-size array; using `character(len=*)` is not allowed. The example above uses `c_char_"Hello World"` to ensure the string literal has the right type; typically the default character kind and `c_char` are the same and thus "Hello World" is equivalent. However, the standard does not guarantee this.

The use of strings is now further illustrated using the C library function `strncpy`, whose prototype is

```c
char *strncpy(char *restrict s1, const char *restrict s2, size_t n);
```

The function `strncpy` copies at most `n` characters from string `s2` to `s1` and returns `s1`. In the following example, we ignore the return value:

```fortran
use iso_c_binding
implicit none
character(len=30) :: str,str2
interface
    ! Ignore the return value of strncpy -> subroutine
    ! "restrict" is always assumed if we do not pass a pointer
    subroutine strncpy(dest, src, n) bind(C)
        import
        character(kind=c_char), intent(out) :: dest(*)
        character(kind=c_char), intent(in) :: src(*)
        integer(c_size_t), value, intent(in) :: n
    end subroutine strncpy
end interface
str = repeat('X',30) ! Initialize whole string with 'X'
call strncpy(str, c_char_"Hello World"//C_NULL_CHAR, &
      len(c_char_"Hello World",kind=c_size_t))
print '(a)', str ! prints: "Hello WorldXXXXXXXXXXXXXXXXXXX"
end
```

The intrinsic procedures are described in Intrinsic Procedures.

8.1. Interoperability with C 71
8.1.5 Working with Pointers

C pointers are represented in Fortran via the special opaque derived type `type(c_ptr)` (with private components). Thus one needs to use intrinsic conversion procedures to convert from or to C pointers.

For some applications, using an assumed type `(TYPE(*))` can be an alternative to a C pointer; see Further Interoperability of Fortran with C.

For example,

```fortran
use iso_c_binding
type(c_ptr) :: cptr1, cptr2
integer, target :: array(7), scalar
integer, pointer :: pa(:), ps
cptr1 = c_loc(array(1)) ! The programmer needs to ensure that the ! array is contiguous if required by the C ! procedure
cptr2 = c_loc(scalar)
call c_f_pointer(cptr2, ps)
call c_f_pointer(cptr2, pa, shape=[7])
```

When converting C to Fortran arrays, the one-dimensional `SHAPE` argument has to be passed.

If a pointer is a dummy-argument of an interoperable procedure, it usually has to be declared using the `VALUE` attribute. `void*` matches TYPE `(C_PTR)`, `VALUE`, while `TYPE(C_PTR)` alone matches `void**`.

Procedure pointers are handled analogously to pointers; the C type is `TYPE(C_FUNPTR)` and the intrinsic conversion procedures are `C_F_PROCPOINTER` and `C_FUNLOC`.

Let us consider two examples of actually passing a procedure pointer from C to Fortran and vice versa. Note that these examples are also very similar to passing ordinary pointers between both languages. First, consider this code in C:

```c
/* Procedure implemented in Fortran. */
void get_values (void (*)(double));

/* Call-back routine we want called from Fortran. */
void print_it (double x)
{
    printf ("Number is %f.\n", x);
}

/* Call Fortran routine and pass call-back to it. */
void foobar ()
{
    get_values (&print_it);
}
```

A matching implementation for `get_values` in Fortran, that correctly receives the procedure pointer from C and is able to call it, is given in the following `MODULE`:

```fortran
MODULE m
    IMPLICIT NONE
    ! Define interface of call-back routine.
    ABSTRACT INTERFACE
        SUBROUTINE callback (x)
            USE, INTRINSIC :: ISO_C_BINDING
    END SUBROUTINE
END MODULE
```

(continues on next page)
Next, we want to call a C routine that expects a procedure pointer argument and pass it a Fortran procedure (which clearly must be interoperable!). Again, the C function may be:

```c
int call_it (int (*func)(int), int arg)
{
    return func (arg);
}
```

It can be used as in the following Fortran code:

```fortran
MODULE m
USE, INTRINSIC :: ISO_C_BINDING
IMPLICIT NONE

! Define interface of C function.
INTERFACE
    INTEGER(KIND=C_INT) FUNCTION call_it (func, arg) BIND(C)
    USE, INTRINSIC :: ISO_C_BINDING
    TYPE(C_FUNPTR), INTENT(IN), VALUE :: func
    INTEGER(KIND=C_INT), INTENT(IN), VALUE :: arg
END FUNCTION call_it
END INTERFACE

CONTAINS

! Define procedure passed to C function.
! It must be interoperable!
INTEGER(KIND=C_INT) FUNCTION double_it (arg) BIND(C)
    INTEGER(KIND=C_INT), INTENT(IN), VALUE :: arg
    double_it = arg + arg
END FUNCTION double_it
```

(continues on next page)
! Call C function.
SUBROUTINE foobar ()
  TYPE(C_FUNPTR) :: cproc
  INTEGER(KIND=C_INT) :: i

! Get C procedure pointer.
cproc = C_FUNLOC (double_it)

! Use it.
 DO i = 1_C_INT, 10_C_INT
   PRINT *, call_it (cproc, i)
 END DO
END SUBROUTINE foobar
END MODULE m

8.1.6 Further Interoperability of Fortran with C

The Technical Specification ISO/IEC TS 29113:2012 on further interoperability of Fortran with C extends the interoperability support of Fortran 2003 and Fortran 2008. Besides removing some restrictions and constraints, it adds assumed-type (TYPE(*)) and assumed-rank (dimension) variables and allows for interoperability of assumed-shape, assumed-rank and deferred-shape arrays, including allocatables and pointers.

Note: Currently, GNU Fortran does not use internally the array descriptor (dope vector) as specified in the Technical Specification, but uses an array descriptor with different fields. Assumed type and assumed rank formal arguments are converted in the library to the specified form. The ISO_Fortran_binding API functions (also Fortran 2018 18.4) are implemented in libgfortran. Alternatively, the Chasm Language Interoperability Tools, http://chasm-interop.sourceforge.net/, provide an interface to GNU Fortran’s array descriptor.

The Technical Specification adds the following new features, which are supported by GNU Fortran:

- The ASYNCHRONOUS attribute has been clarified and extended to allow its use with asynchronous communication in user-provided libraries such as in implementations of the Message Passing Interface specification.
- Many constraints have been relaxed, in particular for the C_LOC and C_F_POINTER intrinsics.
- The OPTIONAL attribute is now allowed for dummy arguments; an absent argument matches a NULL pointer.
- Assumed types (TYPE(*)) have been added, which may only be used for dummy arguments. They are unlimited polymorphic but contrary to CLASS(*) they do not contain any type information, similar to C’s void* pointers. Expressions of any type and kind can be passed; thus, it can be used as replacement for TYPE(C_PTR), avoiding the use of C_LOC in the caller.

Note, however, that TYPE(*) only accepts scalar arguments, unless the DIMENSION is explicitly specified. As DIMENSION(*) only supports array (including array elements) but no scalars, it is not a full replacement for C_LOC. On the other hand, assumed-type assumed-rank dummy arguments (TYPE(*), DIMENSION(..)) allow for both scalars and arrays, but require special code on the callee side to handle the array descriptor.

- Assumed-rank arrays (DIMENSION(..)) as dummy argument allow that scalars and arrays of any rank can be passed as actual argument. As the Technical Specification does not provide for direct means to operate with them, they have to be used either from the C side or be converted using C_LOC and C_F_POINTER to scalars or arrays of a specific rank. The rank can be determined using the RANK intrinsic.

Currently unimplemented:

- GNU Fortran always uses an array descriptor, which does not match the one of the Technical Specification. The ISO_Fortran_binding.h header file and the C functions it specifies are not available.
• Using assumed-shape, assumed-rank and deferred-shape arrays in `BIND(C)` procedures is not fully supported. In particular, C interoperable strings of other length than one are not supported as this requires the new array descriptor.

8.2 GNU Fortran Compiler Directives

8.2.1 ATTRIBUTES directive

The Fortran standard describes how a conforming program shall behave; however, the exact implementation is not standardized. In order to allow the user to choose specific implementation details, compiler directives can be used to set attributes of variables and procedures which are not part of the standard. Whether a given attribute is supported and its exact effects depend on both the operating system and on the processor; see TopC ExtensionsgccUsing the GNU Compiler Collection (GCC) for details.

For procedures and procedure pointers, the following attributes can be used to change the calling convention:

• `CDECL` – standard C calling convention
• `STDCALL` – convention where the called procedure pops the stack
• `fastcall` – part of the arguments are passed via registers instead using the stack

Besides changing the calling convention, the attributes also influence the decoration of the symbol name, e.g., by a leading underscore or by a trailing at-sign followed by the number of bytes on the stack. When assigning a procedure to a procedure pointer, both should use the same calling convention.

On some systems, procedures and global variables (module variables and COMMON blocks) need special handling to be accessible when they are in a shared library. The following attributes are available:

• `DLLExport` – provide a global pointer to a pointer in the DLL
• `DLLImport` – reference the function or variable using a global pointer

For dummy arguments, the `NO_ARG_CHECK` attribute can be used; in other compilers, it is also known as `IGNORE_TKR`. For dummy arguments with this attribute actual arguments of any type and kind (similar to `TYPE(*)`), scalars and arrays of any rank (no equivalent in Fortran standard) are accepted. As with `TYPE(*)`, the argument is unlimited polymorphic and no type information is available. Additionally, the argument may only be passed to dummy arguments with the `NO_ARG_CHECK` attribute and as argument to the `PRESENT` intrinsic function and to `C_LOC` of the `ISO_C_BINDING` module.

Variables with `NO_ARG_CHECK` attribute shall be of assumed-type (`TYPE(*)`; recommended) or of type `INTEGER`, `LOGICAL`, `REAL` or `COMPLEX`. They shall not have the `ALLOCATE`, `CODIMENSION`, `INTENT(OUT)`, `POINTER` or `VALUE` attribute; furthermore, they shall be either scalar or of assumed-size (`dimension(*)`). As `TYPE(*)`, the `NO_ARG_CHECK` attribute requires an explicit interface.

• `NO_ARG_CHECK` – disable the type, kind and rank checking
• `DEPRECATED` – print a warning when using a such-tagged deprecated procedure, variable or parameter; the warning can be suppressed with `-Wno-deprecated-declarations`

The attributes are specified using the syntax

```
!GCC$ ATTRIBUTES attribute-list :: variable-list
```

where in free-form source code only whitespace is allowed before `!GCC$` and in fixed-form source code `!GCC$, cGCC$` or `*GCC$` shall start in the first column.

For procedures, the compiler directives shall be placed into the body of the procedure; for variables and procedure pointers, they shall be in the same declaration part as the variable or procedure pointer.

8.2. GNU Fortran Compiler Directives 75
8.2.2 UNROLL directive

The syntax of the directive is

!GCC$ unroll N

You can use this directive to control how many times a loop should be unrolled. It must be placed immediately before a DO loop and applies only to the loop that follows. N is an integer constant specifying the unrolling factor. The values of 0 and 1 block any unrolling of the loop.

8.2.3 BUILTIN directive

The syntax of the directive is

!GCC$ BUILTIN (B) attributes simd FLAGS IF('target')

You can use this directive to define which middle-end built-ins provide vector implementations. B is name of the middle-end built-in. FLAGS are optional and must be either “(inbranch)” or “(notinbranch)”. IF statement is optional and is used to filter multilib ABIs for the built-in that should be vectorized. Example usage:

!GCC$ builtin (sinf) attributes simd (notinbranch) if('x86_64')

The purpose of the directive is to provide an API among the GCC compiler and the GNU C Library which would define vector implementations of math routines.

8.2.4 IVDEP directive

The syntax of the directive is

!GCC$ ivdep

This directive tells the compiler to ignore vector dependencies in the following loop. It must be placed immediately before a DO loop and applies only to the loop that follows.

Sometimes the compiler may not have sufficient information to decide whether a particular loop is vectorizable due to potential dependencies between iterations. The purpose of the directive is to tell the compiler that vectorization is safe.

This directive is intended for annotation of existing code. For new code it is recommended to consider OpenMP SIMD directives as potential alternative.

8.2.5 VECTOR directive

The syntax of the directive is

!GCC$ vector

This directive tells the compiler to vectorize the following loop. It must be placed immediately before a DO loop and applies only to the loop that follows.
8.2.6 NOVECTOR directive

The syntax of the directive is

!GCC$ novector

This directive tells the compiler to not vectorize the following loop. It must be placed immediately before a DO loop and applies only to the loop that follows.

8.3 Non-Fortran Main Program

Even if you are doing mixed-language programming, it is very likely that you do not need to know or use the information in this section. Since it is about the internal structure of GNU Fortran, it may also change in GCC minor releases.

When you compile a PROGRAM with GNU Fortran, a function with the name main (in the symbol table of the object file) is generated, which initializes the libgfortran library and then calls the actual program which uses the name MAIN__, for historic reasons. If you link GNU Fortran compiled procedures to, e.g., a C or C++ program or to a Fortran program compiled by a different compiler, the libgfortran library is not initialized and thus a few intrinsic procedures do not work properly, e.g. those for obtaining the command-line arguments.

Therefore, if your PROGRAM is not compiled with GNU Fortran and the GNU Fortran compiled procedures require intrinsics relying on the library initialization, you need to initialize the library yourself. Using the default options, gfortran calls _gfortran_set_args and _gfortran_set_options. The initialization of the former is needed if the called procedures access the command line (and for backtracing); the latter sets some flags based on the standard chosen or to enable backtracing. In typical programs, it is not necessary to call any initialization function.

If your PROGRAM is compiled with GNU Fortran, you shall not call any of the following functions. The libgfortran initialization functions are shown in C syntax but using C bindings they are also accessible from Fortran.

8.3.1 _gfortran_set_args — Save command-line arguments

void _gfortran_set_args (int argc, char *argv[])

_gfortran_set_args saves the command-line arguments; this initialization is required if any of the command-line intrinsics is called. Additionally, it shall be called if backtracing is enabled (see _gfortran_set_options).

Parameters

• argc – number of command line argument strings
• argv – the command-line argument strings; argv[0] is the pathname of the executable itself.

Example:

```c
int main (int argc, char *argv[])
{
    /* Initialize libgfortran. */
    _gfortran_set_args (argc, argv);
    return 0;
}
```
8.3.2 _gfortran_set_options — Set library option flags

void _gfortran_set_options (int num, int options[])

_gfortran_set_options sets several flags related to the Fortran standard to be used, whether backtracing should be enabled and whether range checks should be performed. The syntax allows for upward compatibility since the number of passed flags is specified; for non-passed flags, the default value is used. See also see Options for code generation conventions. Please note that not all flags are actually used.

Parameters

- num – number of options passed
- argv – The list of flag values

option flag list:
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

| Option 0 | Allowed standard; can give run-time errors if e.g. an input-output edit descriptor is invalid in a given standard. Possible values are (bitwise or-ed) GFC_STD_F77 (1), GFC_STD_F95_OBS (2), GFC_STD_F95_DEL (4), GFC_STD_F95 (8), GFC_STD_F2003 (16), GFC_STD_GNU (32), GFC_STD_LEGACY (64), GFC_STD_F2008 (128), GFC_STD_F2008_OBS (256), GFC_STD_F2008_TS (512), GFC_STD_F2018 (1024), GFC_STD_F2018_OBS (2048), and GFC_STD=F2018_DEL (4096). Default: GFC_STD_F95_OBS | GFC_STD_F95_DEL | GFC_STD_F95 | GFC_STD_F2003 | GFC_STD_F2008 | GFC_STD_F2008_TS | GFC_STD_F2018 | GFC_STD_F2018_DEL | GFC_STD_GNU | GFC_STD_LEGACY. |
|-----------|-------------------------------------------------|
| Option 1  | Standard-warning flag; prints a warning to standard error. Default: GFC_STD_F95_DEL | GFC_STD_LEGACY. |
| Option 2  | If non zero, enable pedantic checking. Default: off. |
| Option 3  | Unused. |
| Option 4  | If non zero, enable backtracing on run-time errors. Default: off. (Default in the compiler: on.) Note: Installs a signal handler and requires command-line initialization using _gfortran_set_args. |
| Option 5  | If non zero, supports signed zeros. Default: enabled. |
| Option 6  | Enables run-time checking. Possible values are (bitwise or-ed): GFC_RTCHECK_BOUNDS (1), GFC_RTCHECK_ARRAY_TEMPS (2), GFC_RTCHECK_RECURSION (4), GFC_RTCHECK_DO (8), GFC_RTCHECK_POINTER (16), GFC_RTCHECK_MEM (32), GFC_RTCHECK_BITS (64). Default: disabled. |
| Option 7  | Unused. |
| Option 8  | Show a warning when invoking STOP and ERROR STOP if a floating-point exception occurred. Possible values are (bitwise or-ed) GFC_FPE_INVALID (1), GFC_FPE_DENORMAL (2), GFC_FPE_ZERO (4), GFC_FPE_OVERFLOW (8), GFC_FPE_UNDERFLOW (16), GFC_FPE_INEXACT (32). Default: None (0). (Default in the compiler: GFC_FPE_INVALID | GFC_FPE_DENORMAL | GFC_FPE_ZERO | GFC_FPE_OVERFLOW | GFC_FPE_UNDERFLOW.) |

Example:

```c
/* Use gfortran 4.9 default options. */
static int options[] = {68, 511, 0, 0, 1, 1, 0, 0, 31};
_gfortran_set_options (9, &options);
```
8.3.3 _gfortran_set_convert — Set endian conversion

```c
void _gfortran_set_convert (int conv)

_gfortran_set_convert set the representation of data for unformatted files.

Parameters conv – Endian conversion, possible values: GFC_CONVERT_NATIVE (0, default),
GFC_CONVERT_SWAP (1), GFC_CONVERT_BIG (2), GFC_CONVERT_LITTLE (3).
```

Example:

```c
int main (int argc, char *argv[])
{
    /* Initialize libgfortran. */
    _gfortran_set_args (argc, argv);
    _gfortran_set_convert (1);
    return 0;
}
```

8.3.4 _gfortran_set_record_marker — Set length of record markers

```c
void _gfortran_set_record_marker (int val)

_gfortran_set_record_marker sets the length of record markers for unformatted files.

Parameters val – Length of the record marker; valid values are 4 and 8. Default is 4.
```

Example:

```c
int main (int argc, char *argv[])
{
    /* Initialize libgfortran. */
    _gfortran_set_args (argc, argv);
    _gfortran_set_record_marker (8);
    return 0;
}
```

8.3.5 _gfortran_set_fpe — Enable floating point exception traps

```c
void _gfortran_set_fpe (int val)

_gfortran_set_fpe enables floating point exception traps for the specified exceptions. On most systems, this
will result in a SIGFPE signal being sent and the program being aborted.

Parameters [0] (option) – IEEE exceptions. Possible values are (bitwise or-ed) zero (0, default) no trapping, GFC_FPE_INVALID (1), GFC_FPE_DENORMAL (2), GFC_FPE_ZERO (4),
GFC_FPE_OVERFLOW (8), GFC_FPE_UNDERFLOW (16), and GFC_FPE_INEXACT (32).
```

Example:

```c
int main (int argc, char *argv[])
{
    /* Initialize libgfortran. */
    _gfortran_set_args (argc, argv);
    /* FPE for invalid operations such as SQRT(-1.0). */
    _gfortran_set_fpe (1);
    return 0;
}
```
8.3.6 _gfortran_set_max_subrecord_length — Set subrecord length

```c
void _gfortran_set_max_subrecord_length (int val)
    _gfortran_set_max_subrecord_length set the maximum length for a subrecord. This option only
    makes sense for testing and debugging of unformatted I/O.

    Parameters val – the maximum length for a subrecord; the maximum permitted value is
    2147483639, which is also the default.
```

Example:

```c
int main (int argc, char *argv[])
{
    /* Initialize libgfortran. */
    _gfortran_set_args (argc, argv);
    _gfortran_set_max_subrecord_length (8);
    return 0;
}
```

8.4 Naming and argument-passing conventions

This section gives an overview about the naming convention of procedures and global variables and about the argument passing conventions used by GNU Fortran. If a C binding has been specified, the naming convention and some of the argument-passing conventions change. If possible, mixed-language and mixed-compiler projects should use the better defined C binding for interoperability. See see Interoperability with C.

8.4.1 Naming conventions

According the Fortran standard, valid Fortran names consist of a letter between A to Z, a to z, digits 0, 1 to 9 and underscores (_) with the restriction that names may only start with a letter. As vendor extension, the dollar sign ($) is additionally permitted with the option -fdollar-ok, but not as first character and only if the target system supports it.

By default, the procedure name is the lower-cased Fortran name with an appended underscore (_); using -fno-underscoring no underscore is appended while -fsecond-underscore appends two underscores. Depending on the target system and the calling convention, the procedure might be additionally dressed; for instance, on 32bit Windows with stdcall, an at-sign @ followed by an integer number is appended. For the changing the calling convention, see see GNU Fortran Compiler Directives.

For common blocks, the same convention is used, i.e. by default an underscore is appended to the lower-cased Fortran name. Blank commons have the name __BLNK__.

For procedures and variables declared in the specification space of a module, the name is formed by __, followed by the lower-cased module name, _MOD__, and the lower-cased Fortran name. Note that no underscore is appended.
8.4.2 Argument passing conventions

Subroutines do not return a value (matching C99's `void`) while functions either return a value as specified in the platform ABI or the result variable is passed as hidden argument to the function and no result is returned. A hidden result variable is used when the result variable is an array or of type `CHARACTER`.

Arguments are passed according to the platform ABI. In particular, complex arguments might not be compatible to a struct with two real components for the real and imaginary part. The argument passing matches the one of C99's `_Complex`. Functions with scalar complex result variables return their value and do not use a by-reference argument. Note that with the `-ff2c` option, the argument passing is modified and no longer completely matches the platform ABI. Some other Fortran compilers use `f2c` semantic by default; this might cause problems with interoperability.

GNU Fortran passes most arguments by reference, i.e. by passing a pointer to the data. Note that the compiler might use a temporary variable into which the actual argument has been copied, if required semantically (copy-in/copy-out). For arguments with `ALLOCATABLE` and `POINTER` attribute (including procedure pointers), a pointer to the pointer is passed such that the pointer address can be modified in the procedure.

For dummy arguments with the `VALUE` attribute: Scalar arguments of the type `INTEGER`, `LOGICAL`, `REAL` and `COMPLEX` are passed by value according to the platform ABI. (As vendor extension and not recommended, using `%VAL()` in the call to a procedure has the same effect.) For `TYPE(C_PTR)` and procedure pointers, the pointer itself is passed such that it can be modified without affecting the caller.

For Boolean (LOGICAL) arguments, please note that GCC expects only the integer value 0 and 1. If a GNU Fortran LOGICAL variable contains another integer value, the result is undefined. As some other Fortran compilers use -1 for `.TRUE.,` extra care has to be taken – such as passing the value as INTEGER. (The same value restriction also applies to other front ends of GCC, e.g. to GCC's C99 compiler for `_Bool` or GCC's Ada compiler for `Boolean`.)

For arguments of `CHARACTER` type, the character length is passed as a hidden argument at the end of the argument list. For deferred-length strings, the value is passed by reference, otherwise by value. The character length has the C type `size_t` (or `INTEGER(kind=C_SIZE_T)` in Fortran). Note that this is different to older versions of the GNU Fortran compiler, where the type of the hidden character length argument was a `C int`. In order to retain compatibility with older versions, one can e.g. for the following Fortran procedure

```fortran
subroutine fstrlen (s, a)
  character(len=*) :: s
  integer :: a
  print*, len(s)
end subroutine fstrlen
```

define the corresponding C prototype as follows:

```c
#include <limits.h>

#if __GNUC__ > 7
    typedef size_t fortran_charlen_t;
#else
    typedef int fortran_charlen_t;
#endif

void fstrlen_ (char*, int*, fortran_charlen_t);
```

In order to avoid such compiler-specific details, for new code it is instead recommended to use the ISO_C_BINDING feature.

Note with C binding, `CHARACTER(len=1)` result variables are returned according to the platform ABI and no hidden length argument is used for dummy arguments; with `VALUE`, those variables are passed by value.

For `OPTIONAL` dummy arguments, an absent argument is denoted by a NULL pointer, except for scalar dummy arguments of type `INTEGER, LOGICAL, REAL` and `COMPLEX` which have the `VALUE` attribute. For those, a hidden Boolean argument (logical(kind=C_bool), value) is used to indicate whether the argument is present.
Arguments which are assumed-shape, assumed-rank or deferred-rank arrays or, with `-fcoarray=lib`, allocatable scalar coarrays use an array descriptor. All other arrays pass the address of the first element of the array. With `-fcoarray=lib`, the token and the offset belonging to nonallocatable coarrays dummy arguments are passed as hidden argument along the character length hidden arguments. The token is an opaque pointer identifying the coarray and the offset is a passed-by-value integer of kind \texttt{C_PTRDIFF\_T}, denoting the byte offset between the base address of the coarray and the passed scalar or first element of the passed array.

The arguments are passed in the following order:

- Result variable, when the function result is passed by reference
- Character length of the function result, if it is of type \texttt{CHARACTER} and no C binding is used
- The arguments in the order in which they appear in the Fortran declaration
- The the present status for optional arguments with value attribute, which are internally passed by value
- The character length and/or coarray token and offset for the first argument which is a \texttt{CHARACTER} or a nonallocatable coarray dummy argument, followed by the hidden arguments of the next dummy argument of such a type

This chapter is about mixed-language interoperability, but also applies if one links Fortran code compiled by different compilers. In most cases, use of the C Binding features of the Fortran 2003 standard is sufficient, and their use is highly recommended.
9.1 Type and enum ABI Documentation

9.1.1 caf_token_t

Typedef of type `void *` on the compiler side. Can be any data type on the library side.

9.1.2 caf_register_t

Indicates which kind of coarray variable should be registered.

```c
typedef enum caf_register_t {  
    CAF_REGTYPE_COARRAY_STATIC,  
    CAF_REGTYPE_COARRAY_ALLOC,   
    CAF_REGTYPE_LOCK_STATIC,     
    CAF_REGTYPE_LOCK_ALLOC,      
    CAF_REGTYPE_CRITICAL,        
    CAF_REGTYPE_EVENT_STATIC,    
    CAF_REGTYPE_EVENT_ALLOC,     
    CAF_REGTYPE_COARRAY_ALLOC_REGISTER_ONLY,  
    CAF_REGTYPE_COARRAY_ALLOC_ALLOCATE_ONLY  
} caf_register_t;
```

The values `CAF_REGTYPE_COARRAY_ALLOC_REGISTER_ONLY` and `CAF_REGTYPE_COARRAY_ALLOC_ALLOCATE_ONLY` are for allocatable components in derived type coarrays only. The first one sets up the token without allocating memory for allocatable component. The latter one only allocates the memory for an allocatable component in a derived type coarray. The token needs to be setup previously by the REGISTER_ONLY. This allows to have allocatable components un-allocated on some images. The status whether an allocatable component is allocated on a remote image can be queried by `_caf_is_present` which used internally by the ALLOCATED intrinsic.
9.1.3 caf_deregister_t

typedef enum caf_deregister_t {
    CAF_DEREGTYPE_COARRAY_DEREGISTER,
    CAF_DEREGTYPE_COARRAY_DEALLOCATE_ONLY
} caf_deregister_t;

Allows to specify the type of deregistration of a coarray object. The
CAF_DEREGTYPE_COARRAY_DEALLOCATE_ONLY flag is only allowed for allocatable components in derived
type coarrays.

9.1.4 caf_reference_t

The structure used for implementing arbitrary reference chains. A CAF_REFERENCE_T allows to specify a component
reference or any kind of array reference of any rank supported by gfortran. For array references all kinds as known by
the compiler/Fortran standard are supported indicated by a MODE.

typedef enum caf_ref_type_t {
    /* Reference a component of a derived type, either regular one or an
     allocatable or pointer type. For regular ones idx in caf_reference_t is
     set to -1. */
    CAF_REF_COMPONENT,
    /* Reference an allocatable array. */
    CAF_REF_ARRAY,
    /* Reference a non-allocatable/non-pointer array. I.e., the coarray object
     has no array descriptor associated and the addressing is done
     completely using the ref. */
    CAF_REF_STATIC_ARRAY
} caf_ref_type_t;

typedef enum caf_array_ref_t {
    /* No array ref. This terminates the array ref. */
    CAF_ARR_REF_NONE = 0,
    /* Reference array elements given by a vector. Only for this mode
     caf_reference_t.u.a.dim[i].v is valid. */
    CAF_ARR_REF_VECTOR,
    /* A full array ref (:). */
    CAF_ARR_REF_FULL,
    /* Reference a range on elements given by start, end and stride. */
    CAF_ARR_REF_RANGE,
    /* Only a single item is referenced given in the start member. */
    CAF_ARR_REF_SINGLE,
    /* An array ref of the kind (i:), where i is an arbitrary valid index in the
     array. The index i is given in the start member. */
    CAF_ARR_REF_OPEN_END,
    /* An array ref of the kind (:i), where the lower bound of the array ref
     is given by the remote side. The index i is given in the end member. */
    CAF_ARR_REF_OPEN_START
} caf_array_ref_t;

/* References to remote components of a derived type. */
typedef struct caf_reference_t {
    /* A pointer to the next ref or NULL. */
    struct caf_reference_t *next;

(continues on next page)
/* The type of the reference. */
/* caf_ref_type_t, replaced by int to allow specification in fortran FE. */
int type;
/* The size of an item referenced in bytes. I.e. in an array ref this is
the factor to advance the array pointer with to get to the next item.
For component refs this gives just the size of the element referenced. */
size_t item_size;
union {
  struct {
    /* The offset (in bytes) of the component in the derived type.
    Unused for allocatable or pointer components. */
    ptrdiff_t offset;
    /* The offset (in bytes) to the caf_token associated with this
    component. NULL, when not allocatable/pointer ref. */
    ptrdiff_t caf_token_offset;
  } c;
  struct {
    /* The mode of the array ref. See CAF_ARR_REF_*.
    caf_array_ref_t, replaced by unsigend char to allow specification in
    fortran FE. */
    unsigned char mode[GFC_MAX_DIMENSIONS];
    /* The type of a static array. Unset for array's with descriptors. */
    int static_array_type;
    /* Subscript refs (s) or vector refs (v). */
    union {
      struct {
        /* The start and end boundary of the ref and the stride. */
        index_type start, end, stride;
      } s;
      struct {
        /* nvec entries of kind giving the elements to reference. */
        void *vector;
        /* The number of entries in vector. */
        size_t nvec;
        /* The integer kind used for the elements in vector. */
        int kind;
      } v;
    } dim[GFC_MAX_DIMENSIONS];
  } a;
  } u;
} caf_reference_t;

The references make up a single linked list of reference operations. The NEXT member links to the next reference or
NULL to indicate the end of the chain. Component and array refs can be arbitrarily mixed as long as they comply to the
Fortran standard.

NOTES The member STATIC_ARRAY_TYPE is used only when the TYPE is CAF_REF_STATIC_ARRAY. The mem-
ber gives the type of the data referenced. Because no array descriptor is available for a descriptor-less array and type
conversion still needs to take place the type is transported here.

At the moment CAF_ARR_REFVECTOR is not implemented in the front end for descriptor-less arrays. The library
caf_single has untested support for it.
9.1.5 caf_team_t

Opaque pointer to represent a team-handle. This type is a stand-in for the future implementation of teams. It is about to change without further notice.

9.2 Function ABI Documentation

9.2.1 _gfortran_caf_init — Initialization function

```c
void _gfortran_caf_init (int *argc, char ***argv)
```

This function is called at startup of the program before the Fortran main program, if the latter has been compiled with `-fcoarray=lib`. It takes as arguments the command-line arguments of the program. It is permitted to pass two NULL pointers as argument; if non-NULL, the library is permitted to modify the arguments.

**Parameters**

- **argc** – intent(inout) An integer pointer with the number of arguments passed to the program or NULL.
- **argv** – intent(inout) A pointer to an array of strings with the command-line arguments or NULL.

**Note:** The function is modelled after the initialization function of the Message Passing Interface (MPI) specification. Due to the way coarray registration works, it might not be the first call to the library. If the main program is not written in Fortran and only a library uses coarrays, it can happen that this function is never called. Therefore, it is recommended that the library does not rely on the passed arguments and whether the call has been done.

9.2.2 _gfortran_caf_finish — Finalization function

```c
void _gfortran_caf_finish (void)
```

This function is called at the end of the Fortran main program, if it has been compiled with the `-fcoarray=lib` option.

**Note:** For non-Fortran programs, it is recommended to call the function at the end of the main program. To ensure that the shutdown is also performed for programs where this function is not explicitly invoked, for instance non-Fortran programs or calls to the system’s exit() function, the library can use a destructor function. Note that programs can also be terminated using the STOP and ERROR STOP statements; those use different library calls.

9.2.3 _gfortran_caf_this_image — Querying the image number

```c
int _gfortran_caf_this_image (int distance)
```

This function returns the current image number, which is a positive number.

**Parameters**

- **distance** – As specified for the `this_image` intrinsic in TS18508. Shall be a non-negative number.
Note: If the Fortran intrinsic `this_image` is invoked without an argument, which is the only permitted form in Fortran 2008, GCC passes 0 as first argument.

9.2.4 `_gfortran_caf_num_images` — Querying the maximal number of images

```c
int _gfortran_caf_num_images(int distance, int failed)
```

This function returns the number of images in the current team, if `distance` is 0 or the number of images in the parent team at the specified distance. If `failed` is -1, the function returns the number of all images at the specified distance; if it is 0, the function returns the number of nonfailed images, and if it is 1, it returns the number of failed images.

**Parameters**

- `distance` – the distance from this image to the ancestor. Shall be positive.
- `failed` – shall be -1, 0, or 1

**Note:** This function follows TS18508. If the num_image intrinsic has no arguments, then the compiler passes `distance=0` and `failed=-1` to the function.

9.2.5 `_gfortran_caf_image_status` — Query the status of an image

```c
int _gfortran_caf_image_status (int image, caf_team_t * team)
```

Get the status of the image given by the id `image` of the team given by `team`. Valid results are zero, for image is ok, `STAT_STOPPED_IMAGE` from the ISO_FORTRAN_ENV module to indicate that the image has been stopped and `STAT_FAILED_IMAGE` also from ISO_FORTRAN_ENV to indicate that the image has executed a `FAIL_IMAGE` statement.

**Parameters**

- `image` – the positive scalar id of the image in the current TEAM.
- `team` – optional; team on which the inquiry is to be performed.

**Note:** This function follows TS18508. Because team-functionality is not yet implemented a null-pointer is passed for the `team` argument at the moment.

9.2.6 `_gfortran_caf_failed_images` — Get an array of the indexes of the failed images

```c
int _gfortran_caf_failed_images (caf_team_t * team, int * kind)
```

Get an array of image indexes in the current `team` that have failed. The array is sorted ascendingly. When `team` is not provided the current team is to be used. When `kind` is provided then the resulting array is of that integer kind else it is of default integer kind. The returns an unallocated size zero array when no images have failed.

**Parameters**

- `team` – optional; team on which the inquiry is to be performed.
- `image` – optional; the kind of the resulting integer array.
9.2.7 _gfortran_caf_stopped_images — Get an array of the indexes of the stopped images

```c
int _gfortran_caf_stopped_images (caf_team_t * team, int * kind)
```

Get an array of image indexes in the current `team` that have stopped. The array is sorted ascendingly. When `team` is not provided the current team is to be used. When `kind` is provided then the resulting array is of that integer kind else it is of default integer kind. The returns an unallocated size zero array when no images have failed.

**Parameters**

- **team** – optional; team on which the inquiry is to be performed.
- **image** – optional; the kind of the resulting integer array.

**Note:** This function follows TS18508. Because team-functionality is not yet implemented a null-pointer is passed for the `team` argument at the moment.

9.2.8 _gfortran_caf_register — Registering coarrays

```c
void caf_register (size_t size, caf_register_t type, caf_token_t *token,
                  gfc_descriptor_t *desc, int *stat, char *errmsg, size_t errmsg_len)
```

Registers memory for a coarray and creates a token to identify the coarray. The routine is called for both coarrays with `SAVE` attribute and using an explicit `ALLOCATE` statement. If an error occurs and `STAT` is a `NULL` pointer, the function shall abort with printing an error message and starting the error termination. If no error occurs and `STAT` is present, it shall be set to zero. Otherwise, it shall be set to a positive value and, if not-`NULL`, `ERRMSG` shall be set to a string describing the failure. The routine shall register the memory provided in the `DATA`-component of the array descriptor `DESC`, when that component is non-`NULL`, else it shall allocate sufficient memory and provide a pointer to it in the `DATA`-component of `DESC`. The array descriptor has rank zero, when a scalar object is to be registered and the array descriptor may be invalid after the call to _gfortran_caf_register. When an array is to be allocated the descriptor persists.

**Parameters**

- **size** – For normal coarrays, the byte size of the coarray to be allocated; for lock types and event types, the number of elements.
- **type** – one of the `caf_register_t` types.
- **token** – intent(out) An opaque pointer identifying the coarray.
- **desc** – intent(inout) The (pseudo) array descriptor.
- **stat** – intent(out) For allocatable coarrays, stores the `STAT`=; may be `NULL`.
- **errmsg** – intent(out) When an error occurs, this will be set to an error message; may be `NULL`.
- **errmsg_len** – the buffer size of `errmsg`.
Nonallocatable coarrays have to be registered prior use from remote images. In order to guarantee this, they have to be registered before the main program. This can be achieved by creating constructor functions. That is what GCC does such that also for nonallocatable coarrays the memory is allocated and no static memory is used. The token permits to identify the coarray; to the processor, the token is a nonaliasing pointer. The library can, for instance, store the base address of the coarray in the token, some handle or a more complicated struct. The library may also store the array descriptor DESC when its rank is non-zero.

For lock types, the value shall only be used for checking the allocation status. Note that for critical blocks, the locking is only required on one image; in the locking statement, the processor shall always pass an image index of one for critical-block lock variables (CAF_REGTYPE_CRITICAL). For lock types and critical-block variables, the initial value shall be unlocked (or, respectively, not in critical section) such as the value false; for event types, the initial state should be no event, e.g. zero.

9.2.9 _gfortran_caf_deregister — Deregistering coarrays

void caf_deregister (caf_token_t *token, caf_deregister_t type, int *stat,
char *errmsg, size_t errmsg_len)

Called to free or deregister the memory of a coarray; the processor calls this function for automatic and explicit deallocation. In case of an error, this function shall fail with an error message, unless the STAT variable is not null. The library is only expected to free memory it allocated itself during a call to _gfortran_caf_register.

Parameters

- **token** – the token to free.
- **type** – the type of action to take for the coarray. A CAF_DEREGTYPE_COARRAY DEALLOCATE_ONLY is allowed only for allocatable or pointer components of derived type coarrays. The action only deallocates the local memory without deleting the token.
- **stat** – intent(out) Stores the STAT=; may be NULL
- **errmsg** – intent(out) When an error occurs, this will be set to an error message; may be NULL
- **errmsg_len** – the buffer size of errmsg.

Note: For nonallocatable coarrays this function is never called. If a cleanup is required, it has to be handled via the finish, stop and error stop functions, and via destructors.

9.2.10 _gfortran_caf_is_present — Query whether an allocatable or pointer component in a derived type coarray is allocated

void _gfortran_caf_is_present (caf_token_t token, int image_index,
gfc_reference_t *ref)

Used to query the coarray library whether an allocatable component in a derived type coarray is allocated on a remote image.

Parameters

- **token** – An opaque pointer identifying the coarray.
- **image_index** – The ID of the remote image; must be a positive number.
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

- **ref** – A chain of references to address the allocatable or pointer component in the derived type coarray. The object reference needs to be a scalar or a full array reference, respectively.

### 9.2.11 `_gfortran_caf_send` — Sending data from a local image to a remote image

```c
void _gfortran_caf_send (caf_token_t token, size_t offset, int image_index, 
gfc_descriptor_t *dest, caf_vector_t *dst_vector, gfc_descriptor_t *src, 
int dst_kind, int src_kind, bool may_require_tmp, int *stat)
```

Called to send a scalar, an array section or a whole array from a local to a remote image identified by the image_index.

**Parameters**

- **token** – intent(in) An opaque pointer identifying the coarray.
- **offset** – intent(in) By which amount of bytes the actual data is shifted compared to the base address of the coarray.
- **image_index** – intent(in) The ID of the remote image; must be a positive number.
- **dest** – intent(in) Array descriptor for the remote image for the bounds and the size. The base_addr shall not be accessed.
- **dst_vector** – intent(in) If not NULL, it contains the vector subscript of the destination array; the values are relative to the dimension triplet of the dest argument.
- **src** – intent(in) Array descriptor of the local array to be transferred to the remote image
- **dst_kind** – intent(in) Kind of the destination argument
- **src_kind** – intent(in) Kind of the source argument
- **may_require_tmp** – intent(in) The variable is `false` when it is known at compile time that the `dest` and `src` either cannot overlap or overlap (fully or partially) such that walking `src` and `dest` in element wise element order (honoring the stride value) will not lead to wrong results. Otherwise, the value is `true`.
- **stat** – intent(out) when non-NULL give the result of the operation, i.e., zero on success and non-zero on error. When NULL and an error occurs, then an error message is printed and the program is terminated.

**Note:** It is permitted to have `image_index` equal the current image; the memory of the send-to and the send-from might (partially) overlap in that case. The implementation has to take care that it handles this case, e.g. using `memmove` which handles (partially) overlapping memory. If `may_require_tmp` is true, the library might additionally create a temporary variable, unless additional checks show that this is not required (e.g. because walking backward is possible or because both arrays are contiguous and `memmove` takes care of overlap issues).

Note that the assignment of a scalar to an array is permitted. In addition, the library has to handle numeric-type conversion and for strings, padding and different character kinds.
9.2.12 _gfortran_caf_get — Getting data from a remote image

```c
void _gfortran_caf_get (caf_token_t token, size_t offset, int image_index,
                        gfc_descriptor_t *src, caf_vector_t *src_vector, gfc_descriptor_t *dest,
                        int src_kind, int dst_kind, bool may_require_tmp, int *stat)
```

Called to get an array section or a whole array from a remote image identified by the `image_index`.

**Parameters**

- **token** — intent(in) An opaque pointer identifying the coarray.
- **offset** — intent(in) By which amount of bytes the actual data is shifted compared to the base address of the coarray.
- **image_index** — intent(in) The ID of the remote image; must be a positive number.
- **dest** — intent(out) Array descriptor of the local array to store the data retrieved from the remote image.
- **src** — intent(in) Array descriptor for the remote image for the bounds and the size. The `base_addr` shall not be accessed.
- **src_vector** — intent(in) If not NULL, it contains the vector subscript of the source array; the values are relative to the dimension triplet of the `src` argument.
- **dst_kind** — intent(in) Kind of the destination argument.
- **src_kind** — intent(in) Kind of the source argument.
- **may_require_tmp** — intent(in) The variable is `false` when it is known at compile time that the `dest` and `src` either cannot overlap or overlap (fully or partially) such that walking `src` and `dest` in element wise element order (honoring the stride value) will not lead to wrong results. Otherwise, the value is `true`.
- **stat** — intent(out) When non-NULL give the result of the operation, i.e., zero on success and non-zero on error. When NULL and an error occurs, then an error message is printed and the program is terminated.

**Note:** It is permitted to have `image_index` equal the current image; the memory of the send-to and the send-from might (partially) overlap in that case. The implementation has to take care that it handles this case, e.g. using `memmove` which handles (partially) overlapping memory. If `may_require_tmp` is true, the library might additionally create a temporary variable, unless additional checks show that this is not required (e.g. because walking backward is possible or because both arrays are contiguous and `memmove` takes care of overlap issues).

Note that the library has to handle numeric-type conversion and for strings, padding and different character kinds.

9.2.13 _gfortran_caf_sendget — Sending data between remote images

```c
void _gfortran_caf_sendget (caf_token_t dst_token, size_t dst_offset,
                           int dst_image_index, gfc_descriptor_t *dest, caf_vector_t *dst_vector,
                           caf_token_t src_token, size_t src_offset, int src_image_index,
                           gfc_descriptor_t *src, caf_vector_t *src_vector, int dst_kind, int src_kind,
                           bool may_require_tmp, int *stat)
```

Called to send a scalar, an array section or a whole array from a remote image identified by the `src_image_index` to a remote image identified by the `dst_image_index`.

**Parameters**
• **dst_token** – intent(in) An opaque pointer identifying the destination coarray.

• **dst_offset** – intent(in) By which amount of bytes the actual data is shifted compared to the base address of the destination coarray.

• **dst_image_index** – intent(in) The ID of the destination remote image; must be a positive number.

• **dest** – intent(in) Array descriptor for the destination remote image for the bounds and the size. The base_addr shall not be accessed.

• **dst_vector** – intent(int) If not NULL, it contains the vector subscript of the destination array; the values are relative to the dimension triplet of the dest argument.

• **src_token** – intent(in) An opaque pointer identifying the source coarray.

• **src_offset** – intent(in) By which amount of bytes the actual data is shifted compared to the base address of the source coarray.

• **src_image_index** – intent(in) The ID of the source remote image; must be a positive number.

• **src** – intent(in) Array descriptor of the local array to be transferred to the remote image.

• **src_vector** – intent(in) Array descriptor of the local array to be transferred to the remote image.

• **dst_kind** – intent(in) Kind of the destination argument

• **src_kind** – intent(in) Kind of the source argument

• **may_require_tmp** – intent(in) The variable is false when it is known at compile time that the dest and src either cannot overlap or overlap (fully or partially) such that walking src and dest in element wise element order (honoring the stride value) will not lead to wrong results. Otherwise, the value is true.

• **stat** – intent(out) when non-NULL give the result of the operation, i.e., zero on success and non-zero on error. When NULL and an error occurs, then an error message is printed and the program is terminated.

**Note:** It is permitted to have the same image index for both **src_image_index** and **dst_image_index**; the memory of the send-to and the send-from might (partially) overlap in that case. The implementation has to take care that it handles this case, e.g., using memmove which handles (partially) overlapping memory. If **may_require_tmp** is true, the library might additionally create a temporary variable, unless additional checks show that this is not required (e.g., because walking backward is possible or because both arrays are contiguous and memmove takes care of overlap issues).

Note that the assignment of a scalar to an array is permitted. In addition, the library has to handle numeric-type conversion and for strings, padding and different character kinds.
9.2.14  _gfortran_caf_send_by_ref — Sending data from a local image to a remote image with enhanced referencing options

`void _gfortran_caf_send_by_ref (caf_token_t token, int image_index, gfc_descriptor_t *src, caf_reference_t *refs, int dst_kind, int src_kind, bool may_require_tmp, bool dst_reallocatable, int *stat, int dst_type)`

Called to send a scalar, an array section or a whole array from a local to a remote image identified by the `image_index`.

**Parameters**

- **token** – `intent(in)` An opaque pointer identifying the coarray.
- **image_index** – `intent(in)` The ID of the remote image; must be a positive number.
- **src** – `intent(in)` Array descriptor of the local array to be transferred to the remote image.
- **refs** – `intent(in)` The references on the remote array to store the data given by `src`. Guaranteed to have at least one entry.
- **dst_kind** – `intent(in)` Kind of the destination argument.
- **src_kind** – `intent(in)` Kind of the source argument.
- **may_require_tmp** – `intent(in)` The variable is `false` when it is known at compile time that the `dest` and `src` either cannot overlap or overlap (fully or partially) such that walking `src` and `dest` in element wise element order (honoring the stride value) will not lead to wrong results. Otherwise, the value is `true`.
- **dst_reallocatable** – `intent(in)` Set when the destination is of allocatable or pointer type and the refs will allow reallocation, i.e., the ref is a full array or component ref.
- **stat** – `intent(out)` When non-`NULL` give the result of the operation, i.e., zero on success and non-zero on error. When `NULL` and an error occurs, then an error message is printed and the program is terminated.
- **dst_type** – `intent(in)` Give the type of the destination. When the destination is not an array, than the precise type, e.g. of a component in a derived type, is not known, but provided here.

**Note:** It is permitted to have `image_index` equal the current image; the memory of the send-to and the send-from might (partially) overlap in that case. The implementation has to take care that it handles this case, e.g. using `memmove` which handles (partially) overlapping memory. If `may_require_tmp` is true, the library might additionally create a temporary variable, unless additional checks show that this is not required (e.g. because walking backward is possible or because both arrays are contiguous and `memmove` takes care of overlap issues).

Note that the assignment of a scalar to an array is permitted. In addition, the library has to handle numeric-type conversion and for strings, padding and different character kinds.

Because of the more complicated references possible some operations may be unsupported by certain libraries. The library is expected to issue a precise error message why the operation is not permitted.
9.2.15 _gfortran_caf_get_by_ref — Getting data from a remote image using enhanced references

```c
void _gfortran_caf_get_by_ref (caf_token_t token, int image_index,
   caf_reference_t *refs, gfc_descriptor_t *dst, int dst_kind, int src_kind,
   bool may_require_tmp, bool dst_reallocatable, int *stat, int src_type)
```

Called to get a scalar, an array section or a whole array from a remote image identified by the `image_index`.

**Parameters**

- **token** – intent(in) An opaque pointer identifying the coarray.
- **image_index** – intent(in) The ID of the remote image; must be a positive number.
- **refs** – intent(in) The references to apply to the remote structure to get the data.
- **dst** – intent(in) Array descriptor of the local array to store the data transferred from the remote image. May be reallocated where needed and when `DST_REALLOCATABLE` allows it.
- **dst_kind** – intent(in) Kind of the destination argument
- **src_kind** – intent(in) Kind of the source argument
- **may_require_tmp** – intent(in) The variable is `false` when it is known at compile time that the `dest` and `src` either cannot overlap or overlap (fully or partially) such that walking `src` and `dest` in element wise element order (honoring the stride value) will not lead to wrong results. Otherwise, the value is `true`.
- **dst_reallocatable** – intent(in) Set when `DST` is of allocatable or pointer type and its refs allow reallocation, i.e., the full array or a component is referenced.
- **stat** – intent(out) When non-NULL give the result of the operation, i.e., zero on success and non-zero on error. When NULL and an error occurs, then an error message is printed and the program is terminated.
- **src_type** – intent(in) Give the type of the source. When the source is not an array, than the precise type, e.g. of a component in a derived type, is not known, but provided here.

**Note:** It is permitted to have `image_index` equal the current image; the memory of the send-to and the send-from might (partially) overlap in that case. The implementation has to take care that it handles this case, e.g. using `memmove` which handles (partially) overlapping memory. If `may_require_tmp` is true, the library might additionally create a temporary variable, unless additional checks show that this is not required (e.g. because walking backward is possible or because both arrays are contiguous and `memmove` takes care of overlap issues).

Note that the library has to handle numeric-type conversion and for strings, padding and different character kinds. Because of the more complicated references possible some operations may be unsupported by certain libraries. The library is expected to issue a precise error message why the operation is not permitted.
9.2.16 _gfortran_caf_sendget_by_ref — Sending data between remote images using enhanced references on both sides

```c
void _gfortran_caf_sendget_by_ref (caf_token_t dst_token, int dst_image_index,
   caf_reference_t *dst_refs, caf_token_t src_token, int src_image_index,
   caf_reference_t *src_refs, int dst_kind, int src_kind, bool may_require_tmp,
   int *dst_stat, int *src_stat, int dst_type, int src_type)
```

Called to send a scalar, an array section or a whole array from a remote image identified by the 
`src_image_index` to a remote image identified by the `dst_image_index`.

**Parameters**

- `dst_token` – intent(in) An opaque pointer identifying the destination coarray.
- `dst_image_index` – intent(in) The ID of the destination remote image; must be a positive
  number.
- `dst_refs` – intent(in) The references on the remote array to store the data given by the
  source. Guaranteed to have at least one entry.
- `src_token` – intent(in) An opaque pointer identifying the source coarray.
- `src_image_index` – intent(in) The ID of the source remote image; must be a positive
  number.
- `src_refs` – intent(in) The references to apply to the remote structure to get the data.
- `dst_kind` – intent(in) Kind of the destination argument
- `src_kind` – intent(in) Kind of the source argument
- `may_require_tmp` – intent(in) The variable is `false` when it is known at compile time
  that the `dest` and `src` either cannot overlap or overlap (fully or partially) such that walking
  `src` and `dest` in element wise element order (honoring the stride value) will not lead to wrong
  results. Otherwise, the value is `true`.
- `dst_stat` – intent(out) when non- NULL give the result of the send-operation, i.e., zero on
  success and non-zero on error. When NULL and an error occurs, then an error message is
  printed and the program is terminated.
- `src_stat` – intent(out) When non- NULL give the result of the get-operation, i.e., zero on
  success and non-zero on error. When NULL and an error occurs, then an error message is
  printed and the program is terminated.
- `dst_type` – intent(in) Give the type of the destination. When the destination is not an array,
  than the precise type, e.g. of a component in a derived type, is not known, but provided here.
- `src_type` – intent(in) Give the type of the source. When the source is not an array, than the
  precise type, e.g. of a component in a derived type, is not known, but provided here.

**Note:** It is permitted to have the same image index for both `src_image_index` and `dst_image_index`
; the memory of the send-to and the send-from might (partially) overlap in that case. The implementation has
 to take care that it handles this case, e.g. using `memmove` which handles (partially) overlapping memory. If
 `may_require_tmp` is true, the library might additionally create a temporary variable, unless additional checks
 show that this is not required (e.g. because walking backward is possible or because both arrays are contiguous and
 `memmove` takes care of overlap issues).

Note that the assignment of a scalar to an array is permitted. In addition, the library has to handle numeric-type
conversion and for strings, padding and different character kinds.
Because of the more complicated references possible some operations may be unsupported by certain libraries. The library is expected to issue a precise error message why the operation is not permitted.

### 9.2.17 _gfortran_caf_lock — Locking a lock variable

```c
void _gfortran_caf_lock (caf_token_t token, size_t index, int image_index,
int *acquired_lock, int *stat, char *errmsg, size_t errmsg_len)
```

Acquire a lock on the given image on a scalar locking variable or for the given array element for an array-valued variable. If the `acquired_lock` is NULL, the function returns after having obtained the lock. If it is non-NULL, then `acquired_lock` is assigned the value true (one) when the lock could be obtained and false (zero) otherwise. Locking a lock variable which has already been locked by the same image is an error.

**Parameters**

- `token` – intent(in) An opaque pointer identifying the coarray.
- `index` – intent(in) Array index; first array index is 0. For scalars, it is always 0.
- `image_index` – intent(in) The ID of the remote image; must be a positive number.
- `acquired_lock` – intent(out) If not NULL, it returns whether lock could be obtained.
- `stat` – intent(out) For allocatable coarrays, stores the STAT=; may be NULL.
- `errmsg` – intent(out) When an error occurs, this will be set to an error message; may be NULL.
- `errmsg_len` – intent(in) the buffer size of errmsg

**Note:** This function is also called for critical blocks; for those, the array index is always zero and the image index is one. Libraries are permitted to use other images for critical-block locking variables.

### 9.2.18 _gfortran_caf_unlock — Unlocking a lock variable

```c
void _gfortran_caf_unlock (caf_token_t token, size_t index, int image_index,
int *stat, char *errmsg, size_t errmsg_len)
```

Release a lock on the given image on a scalar locking variable or for the given array element for an array-valued variable. Unlocking a lock variable which is unlocked or has been locked by a different image is an error.

**Parameters**

- `token` – intent(in) An opaque pointer identifying the coarray.
- `index` – intent(in) Array index; first array index is 0. For scalars, it is always 0.
- `image_index` – intent(in) The ID of the remote image; must be a positive number.
- `stat` – intent(out) For allocatable coarrays, stores the STAT=; may be NULL.
- `errmsg` – intent(out) When an error occurs, this will be set to an error message; may be NULL.
- `errmsg_len` – intent(in) the buffer size of errmsg

**Note:** This function is also called for critical block; for those, the array index is always zero and the image index is one. Libraries are permitted to use other images for critical-block locking variables.
9.2.19 **_gfortran_caf_event_post— Post an event**

```c
void _gfortran_caf_event_post (caf_token_t token, size_t index,
int image_index, int *stat, char *errmsg, size_t errmsg_len)
```

Increment the event count of the specified event variable.

**Parameters**

- `token` – intent(in) An opaque pointer identifying the coarray.
- `index` – intent(in) Array index; first array index is 0. For scalars, it is always 0.
- `image_index` – intent(in) The ID of the remote image; must be a positive number; zero indicates the current image, when accessed noncoindexed.
- `stat` – intent(out) Stores the STAT=; maybe NULL.
- `errmsg` – intent(out) When an error occurs, this will be set to an error message; may be NULL.
- `errmsg_len` – intent(in) the buffer size of errmsg

**Note:** This acts like an atomic add of one to the remote image’s event variable. The statement is an image-control statement but does not imply sync memory. Still, all preceeding push communications of this image to the specified remote image have to be completed before `event_wait` on the remote image returns.

9.2.20 **_gfortran_caf_event_wait— Wait that an event occurred**

```c
void _gfortran_caf_event_wait (caf_token_t token, size_t index,
int until_count, int *stat, char *errmsg, size_t errmsg_len)
```

Wait until the event count has reached at least the specified `until_count`; if so, atomically decrement the event variable by this amount and return.

**Parameters**

- `token` – intent(in) An opaque pointer identifying the coarray.
- `index` – intent(in) Array index; first array index is 0. For scalars, it is always 0.
- `until_count` – intent(in) The number of events which have to be available before the function returns.
- `stat` – intent(out) Stores the STAT=; may be NULL.
- `errmsg` – intent(out) When an error occurs, this will be set to an error message; may be NULL.
- `errmsg_len` – intent(in) the buffer size of errmsg

**Note:** This function only operates on a local coarray. It acts like a loop checking atomically the value of the event variable, breaking if the value is greater or equal the requested number of counts. Before the function returns, the event variable has to be decremented by the requested `until_count` value. A possible implementation would be a busy loop for a certain number of spins (possibly depending on the number of threads relative to the number of available cores) followed by another waiting strategy such as a sleeping wait (possibly with an increasing number of sleep time) or, if possible, a futex wait.
The statement is an image-control statement but does not imply sync memory. Still, all preceding push communications of this image to the specified remote image have to be completed before `event_wait` on the remote image returns.

### 9.2.21 `_gfortran_caf_event_query — Query event count`

```c
void _gfortran_caf_event_query (caf_token_t token, size_t index,
int image_index, int *count, int *stat)
```

Return the event count of the specified event variable.

**Parameters**

- `token` – intent(in) An opaque pointer identifying the coarray.
- `index` – intent(in) Array index; first array index is 0. For scalars, it is always 0.
- `image_index` – intent(in) The ID of the remote image; must be a positive number; zero indicates the current image when accessed noncoindexed.
- `count` – intent(out) The number of events currently posted to the event variable.
- `stat` – intent(out) Stores the STAT=; may be NULL.

**Note:** The typical use is to check the local event variable to only call `event_wait` when the data is available. However, a coindexed variable is permitted; there is no ordering or synchronization implied. It acts like an atomic fetch of the value of the event variable.

### 9.2.22 `_gfortran_caf_sync_all — All-image barrier`

```c
void _gfortran_caf_sync_all (int *stat, char *errmsg, size_t errmsg_len)
```

Synchronization of all images in the current team; the program only continues on a given image after this function has been called on all images of the current team. Additionally, it ensures that all pending data transfers of previous segment have completed.

**Parameters**

- `stat` – intent(out) Stores the status STAT= and may be NULL.
- `errmsg` – intent(out) When an error occurs, this will be set to an error message; may be NULL.
- `errmsg_len` – intent(in) the buffer size of errmsg

### 9.2.23 `_gfortran_caf_sync_images — Barrier for selected images`

```c
void _gfortran_caf_sync_images (int count, int images[], int *stat,
char *errmsg, size_t errmsg_len)
```

Synchronization between the specified images; the program only continues on a given image after this function has been called on all images specified for that image. Note that one image can wait for all other images in the current team (e.g. via `sync images(*)`) while those only wait for that specific image. Additionally, `sync images` ensures that all pending data transfers of previous segments have completed.

**Parameters**
• **count** – intent(in) The number of images which are provided in the next argument. For a zero-sized array, the value is zero. For sync images (*), the value is -1.

• **images** – intent(in) An array with the images provided by the user. If **count** is zero, a NULL pointer is passed.

• **stat** – intent(out) Stores the status STAT= and may be NULL.

• **errmsg** – intent(out) When an error occurs, this will be set to an error message; may be NULL.

• **errmsg_len** – intent(in) The buffer size of errmsg

### 9.2.24 _gfortran_caf_sync_memory — Wait for completion of segment-memory operations

```c
void _gfortran_caf_sync_memory (int *stat, char *errmsg, size_t errmsg_len)
```

Acts as optimization barrier between different segments. It also ensures that all pending memory operations of this image have been completed.

**Parameters**

- **stat** – intent(out) Stores the status STAT= and may be NULL.

- **errmsg** – intent(out) When an error occurs, this will be set to an error message; may be NULL.

- **errmsg_len** – intent(in) The buffer size of errmsg

**Note:** A simple implementation could be __asm__ __volatile__ (":::memory") to prevent code movements.

### 9.2.25 _gfortran_caf_error_stop — Error termination with exit code

```c
void _gfortran_caf_error_stop (int error)
```

Invoked for an ERROR STOP statement which has an integer argument. The function should terminate the program with the specified exit code.

**Parameters**

- **error** – intent(in) The exit status to be used.

### 9.2.26 _gfortran_caf_error_stop_str — Error termination with string

```c
void _gfortran_caf_error_stop (const char *string, size_t len)
```

Invoked for an ERROR STOP statement which has a string as argument. The function should terminate the program with a nonzero-exit code.

**Parameters**

- **string** – intent(in) The error message (not zero terminated)

- **len** – intent(in) The length of the string
9.2.27  _gfortran_caf_fail_image — Mark the image failed and end its execution

void _gfortran_caf_fail_image ()
   Invoked for a FAIL IMAGE statement. The function should terminate the current image.

Note: This function follows TS18508.

9.2.28  _gfortran_caf_atomic_define — Atomic variable assignment

void _gfortran_caf_atomic_define (caf_token_t token, size_t offset, int image_index, void *value, int *stat, int type, int kind)
   Assign atomically a value to an integer or logical variable.

Parameters

- **token** – intent(in) An opaque pointer identifying the coarray.
- **offset** – intent(in) By which amount of bytes the actual data is shifted compared to the base address of the coarray.
- **image_index** – intent(in) The ID of the remote image; must be a positive number; zero indicates the current image when used noncoindexed.
- **value** – intent(in) the value to be assigned, passed by reference
- **stat** – intent(out) Stores the status STAT= and may be NULL.
- **type** – intent(in) The data type, i.e. BT_INTEGER (1) or BT_LOGICAL (2).
- **kind** – intent(in) The kind value (only 4; always int)

9.2.29  _gfortran_caf_atomic_ref — Atomic variable reference

void _gfortran_caf_atomic_ref (caf_token_t token, size_t offset, int image_index, void *value, int *stat, int type, int kind)
   Reference atomically a value of a kind-4 integer or logical variable.

Parameters

- **token** – intent(in) An opaque pointer identifying the coarray.
- **offset** – intent(in) By which amount of bytes the actual data is shifted compared to the base address of the coarray.
- **image_index** – intent(in) The ID of the remote image; must be a positive number; zero indicates the current image when used noncoindexed.
- **value** – intent(out) The variable assigned the atomically referenced variable.
- **stat** – intent(out) Stores the status STAT= and may be NULL.
- **type** – the data type, i.e. BT_INTEGER (1) or BT_LOGICAL (2).
- **kind** – The kind value (only 4; always int)
9.2.30 \_gfortran\_caf\_atomic\_cas — Atomic compare and swap

\begin{verbatim}
void _gfortran_caf_atomic_cas (caf_token_t token, size_t offset,
   int image_index, void *old, void *compare, void *new_val, int *stat, int type,
   int kind)
\end{verbatim}

Atomic compare and swap of a kind-4 integer or logical variable. Assigns atomically the specified value to the atomic variable, if the latter has the value specified by the passed condition value.

Parameters

- **token** – intent(in) An opaque pointer identifying the coarray.
- **offset** – intent(in) By which amount of bytes the actual data is shifted compared to the base address of the coarray.
- **image_index** – intent(in) The ID of the remote image; must be a positive number; zero indicates the current image when used noncoindexed.
- **old** – intent(out) The value which the atomic variable had just before the cas operation.
- **compare** – intent(in) The value used for comparison.
- **new_val** – intent(in) The new value for the atomic variable, assigned to the atomic variable, if compare equals the value of the atomic variable.
- **stat** – intent(out) Stores the status STAT= and maybe NULL.
- **type** – intent(in) the data type, i.e. `BT_INTEGER` (1) or `BT_LOGICAL` (2).
- **kind** – intent(in) The kind value (only 4; always int)

9.2.31 \_gfortran\_caf\_atomic\_op — Atomic operation

\begin{verbatim}
void _gfortran_caf_atomic_op (int op, caf_token_t token, size_t offset,
   int image_index, void *value, void *old, int *stat, int type, int kind)
\end{verbatim}

Apply an operation atomically to an atomic integer or logical variable. After the operation, \texttt{old} contains the value just before the operation, which, respectively, adds (GFC\_CAF\_ATOMIC\_ADD) atomically the value to the atomic integer variable or does a bitwise AND, OR or exclusive OR between the atomic variable and \texttt{value}; the result is then stored in the atomic variable.

Parameters

- **op** – intent(in) the operation to be performed; possible values `GFC\_CAF\_ATOMIC\_ADD` (1), `GFC\_CAF\_ATOMIC\_AND` (2), `GFC\_CAF\_ATOMIC\_OR` (3), `GFC\_CAF\_ATOMIC\_XOR` (4).
- **token** – intent(in) An opaque pointer identifying the coarray.
- **offset** – intent(in) By which amount of bytes the actual data is shifted compared to the base address of the coarray.
- **image_index** – intent(in) The ID of the remote image; must be a positive number; zero indicates the current image when used noncoindexed.
- **old** – intent(out) The value which the atomic variable had just before the atomic operation.
- **val** – intent(in) The new value for the atomic variable, assigned to the atomic variable, if compare equals the value of the atomic variable.
- **stat** – intent(out) Stores the status STAT= and maybe NULL.
- **type** – intent(in) the data type, i.e. `BT_INTEGER` (1) or `BT_LOGICAL` (2)
9.2.32 _gfortran_caf_co_broadcast — Sending data to all images

```c
void _gfortran_caf_co_broadcast (gfc_descriptor_t *a, int source_image,
int *stat, char *errmsg, size_t errmsg_len)
```

Distribute a value from a given image to all other images in the team. Has to be called collectively.

**Parameters**

- `a` – intent(inout) An array descriptor with the data to be broadcasted (on `source_image`) or to be received (other images).
- `source_image` – intent(in) The ID of the image from which the data should be broadcasted.
- `stat` – intent(out) Stores the status STAT= and may be NULL.
- `errmsg` – intent(out) When an error occurs, this will be set to an error message; may be NULL.
- `errmsg_len` – intent(in) the buffer size of `errmsg`.

9.2.33 _gfortran_caf_co_max — Collective maximum reduction

```c
void _gfortran_caf_co_max (gfc_descriptor_t *a, int result_image, int *stat,
char *errmsg, int a_len, size_t errmsg_len)
```

Calculates for each array element of the variable `a` the maximum value for that element in the current team; if `result_image` has the value 0, the result shall be stored on all images, otherwise, only on the specified image. This function operates on numeric values and character strings.

**Parameters**

- `a` – intent(inout) An array descriptor for the data to be processed. On the destination image(s) the result overwrites the old content.
- `result_image` – intent(in) The ID of the image to which the reduced value should be copied to; if zero, it has to be copied to all images.
- `stat` – intent(out) Stores the status STAT= and may be NULL.
- `errmsg` – intent(out) When an error occurs, this will be set to an error message; may be NULL.
- `a_len` – intent(in) the string length of argument `a`
- `errmsg_len` – intent(in) the buffer size of `errmsg`

**Note:** If `result_image` is nonzero, the data in the array descriptor `a` on all images except of the specified one become undefined; hence, the library may make use of this.
9.2.34 _gfortran_caf_co_min — Collective minimum reduction

```c
void _gfortran_caf_co_min (gfc_descriptor_t *a, int result_image, int *stat,
char *errmsg, int a_len, size_t errmsg_len)
```

Calculates for each array element of the variable `a` the minimum value for that element in the current team; if `result_image` has the value 0, the result shall be stored on all images, otherwise, only on the specified image. This function operates on numeric values and character strings.

**Parameters**

- `a` – intent(inout) An array descriptor for the data to be processed. On the destination image(s) the result overwrites the old content.
- `result_image` – intent(in) The ID of the image to which the reduced value should be copied to; if zero, it has to be copied to all images.
- `stat` – intent(out) Stores the status `STAT= and maybe NULL.``
- `errmsg` – intent(out) When an error occurs, this will be set to an error message; may be NULL.
- `a_len` – intent(in) the string length of argument `a`
- `errmsg_len` – intent(in) the buffer size of `errmsg`

**Note:** If `result_image` is nonzero, the data in the array descriptor `a` on all images except of the specified one become undefined; hence, the library may make use of this.

9.2.35 _gfortran_caf_co_sum — Collective summing reduction

```c
void _gfortran_caf_co_sum (gfc_descriptor_t *a, int result_image, int *stat,
char *errmsg, size_t errmsg_len)
```

Calculates for each array element of the variable `a` the sum of all values for that element in the current team; if `result_image` has the value 0, the result shall be stored on all images, otherwise, only on the specified image. This function operates on numeric values only.

**Parameters**

- `a` – intent(inout) An array descriptor with the data to be processed. On the destination image(s) the result overwrites the old content.
- `result_image` – intent(in) The ID of the image to which the reduced value should be copied to; if zero, it has to be copied to all images.
- `stat` – intent(out) Stores the status `STAT= and maybe NULL.``
- `errmsg` – intent(out) When an error occurs, this will be set to an error message; may be NULL.
- `errmsg_len` – intent(in) the buffer size of `errmsg`

**Note:** If `result_image` is nonzero, the data in the array descriptor `a` on all images except of the specified one become undefined; hence, the library may make use of this.
9.2.36 _gfortran_caf_co_reduce — Generic collective reduction

```c
void _gfortran_caf_co_reduce (gfc_descriptor_t *a, void * (*opr) (void *, void *), int opr_flags, int result_image, int *stat, char *errmsg, int a_len, size_t errmsg_len)
```

Calculates for each array element of the variable `a` the reduction value for that element in the current team; if `result_image` has the value 0, the result shall be stored on all images, otherwise, only on the specified image. The `opr` is a pure function doing a mathematically commutative and associative operation.

**Parameters**

- **a** – `intent(inout)` An array descriptor with the data to be processed. On the destination image(s) the result overwrites the old content.
- **opr** – `intent(in)` Function pointer to the reduction function
- **opr_flags** – `intent(in)` Flags regarding the reduction function
- **result_image** – `intent(in)` The ID of the image to which the reduced value should be copied to; if zero, it has to be copied to all images.
- **stat** – `intent(out)` Stores the status `STAT=` and maybe `NULL`.
- **errmsg** – `intent(out)` When an error occurs, this will be set to an error message; may be `NULL`.
- **a_len** – `intent(in)` the string length of argument `a`
- **errmsg_len** – `intent(in)` the buffer size of `errmsg`

**Note:** If `result_image` is nonzero, the data in the array descriptor `a` on all images except of the specified one become undefined; hence, the library may make use of this.

For character arguments, the result is passed as first argument, followed by the result string length, next come the two string arguments, followed by the two hidden string length arguments. With C binding, there are no hidden arguments and by-reference passing and either only a single character is passed or an array descriptor.

Some basic guidelines for editing this document:

1. The intrinsic procedures are to be listed in alphabetical order.
2. The generic name is to be used.
3. The specific names are included in the function index and in a table at the end of the node (See ABS entry).
4. Try to maintain the same style for each entry.
10.1 Introduction to intrinsic procedures

The intrinsic procedures provided by GNU Fortran include all of the intrinsic procedures required by the Fortran 95 standard, a set of intrinsic procedures for backwards compatibility with G77, and a selection of intrinsic procedures from the Fortran 2003 and Fortran 2008 standards. Any conflict between a description here and a description in either the Fortran 95 standard, the Fortran 2003 standard or the Fortran 2008 standard is unintentional, and the standard(s) should be considered authoritative.

The enumeration of the KIND type parameter is processor defined in the Fortran 95 standard. GNU Fortran defines the default integer type and default real type by INTEGER(KIND=4) and REAL(KIND=4), respectively. The standard mandates that both data types shall have another kind, which have more precision. On typical target architectures supported by gfortran, this kind type parameter is KIND=8. Hence, REAL(KIND=8) and DOUBLE PRECISION are equivalent. In the description of generic intrinsic procedures, the kind type parameter will be specified by KIND=*, and in the description of specific names for an intrinsic procedure the kind type parameter will be explicitly given (e.g., REAL(KIND=4) or REAL(KIND=8)). Finally, for brevity the optional KIND= syntax will be omitted.

Many of the intrinsic procedures take one or more optional arguments. This document follows the convention used in the Fortran 95 standard, and denotes such arguments by square brackets.

GNU Fortran offers the -std=f95 and -std=gnu options, which can be used to restrict the set of intrinsic procedures to a given standard. By default, gfortran sets the -std=gnu option, and so all intrinsic procedures described here are accepted. There is one caveat. For a select group of intrinsic procedures, g77 implemented both a function and a subroutine. Both classes have been implemented in gfortran for backwards compatibility with g77. It is noted here that these functions and subroutines cannot be intermixed in a given subprogram. In the descriptions that follow, the applicable standard for each intrinsic procedure is noted.

10.2 ABORT — Abort the program

**ABORT ()**

ABORT causes immediate termination of the program. On operating systems that support a core dump, ABORT will produce a core dump. It will also print a backtrace, unless -fno-backtrace is given.

**Returns** Does not return.

**Standard:** GNU extension

**Class:** Subroutine

**Syntax:**
CALL ABORT

Example:

```fortran
program test_abort
  integer :: i = 1, j = 2
  if (i /= j) call abort
end program test_abort
```

See also: EXIT, KILL, BACKTRACE

## 10.3 ABS — Absolute value

ABS(A) computes the absolute value of A.

**Parameters** A – The type of the argument shall be an INTEGER, REAL, or COMPLEX.

**Returns** The return value is of the same type and kind as the argument except the return value is REAL for a COMPLEX argument.

**Standard**: Fortran 77 and later, has overloads that are GNU extensions

**Class**: Elemental function

**Syntax**: 

```
RESULT = ABS(A)
```

Example:

```fortran
program test_abs
  integer :: i = -1
  real :: x = -1.e0
  complex :: z = (-1.e0, 0.e0)
  i = abs(i)
  x = abs(x)
  x = abs(z)
end program test_abs
```

**Specific names**:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS(A)</td>
<td>REAL(4) A</td>
<td>REAL(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>CABS(A)</td>
<td>COMPLEX(4) A</td>
<td>REAL(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>DABS(A)</td>
<td>REAL(8) A</td>
<td>REAL(8)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>IABS(A)</td>
<td>INTEGER(4) A</td>
<td>INTEGER(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>BABSI</td>
<td>INTEGER(1) A</td>
<td>INTEGER(1)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>I1ABS(A)</td>
<td>INTEGER(2) A</td>
<td>INTEGER(2)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>J1ABS(A)</td>
<td>INTEGER(4) A</td>
<td>INTEGER(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>K1ABS(A)</td>
<td>INTEGER(8) A</td>
<td>INTEGER(8)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>ZABS</td>
<td>COMPLEX(8) A</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>CDABS(A)</td>
<td>COMPLEX(8) A</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>
10.4 ACCESS — Checks file access modes

ACCESS (NAME, MODE)

ACCESS (NAME, MODE) checks whether the file NAME exists, is readable, writable or executable. Except for the executable check, ACCESS can be replaced by Fortran 95’s INQUIRE.

Parameters

• NAME – Scalar CHARACTER of default kind with the file name. Trailing blank are ignored unless the character achar (0) is present, then all characters up to and excluding achar (0) are used as file name.

• MODE – Scalar CHARACTER of default kind with the file access mode, may be any concatenation of "r" (readable), "w" (writable) and "x" (executable), or " " to check for existence.

Returns Returns a scalar INTEGER, which is 0 if the file is accessible in the given mode; otherwise or if an invalid argument has been given for MODE the value 1 is returned.

Standard: GNU extension

Class: Inquiry function

Syntax:

RESULT = ACCESS(NAME, MODE)

Example:

```fortran
program access_test
  implicit none
  character(len=*) parameter :: file = 'test.dat'
  character(len=*) parameter :: file2 = 'test.dat ' // achar(0)
  if (access(file,' ') == 0) print *, trim(file), ' is exists'
  if (access(file,'r') == 0) print *, trim(file), ' is readable'
  if (access(file,'w') == 0) print *, trim(file), ' is writable'
  if (access(file,'x') == 0) print *, trim(file), ' is executable'
  if (access(file2,'rwx') == 0) &
    print *, trim(file2), ' is readable, writable and executable'
end program access_test
```

10.5 ACHAR — Character in ASCII collating sequence

ACHAR (I)

ACHAR (I) returns the character located at position I in the ASCII collating sequence.

Parameters

• I – The type shall be INTEGER.

• KIND – (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

Returns The return value is of type CHARACTER with a length of one. If the KIND argument is present, the return value is of the specified kind and of the default kind otherwise.

Standard: Fortran 77 and later, with KIND argument Fortran 2003 and later
**Class:** Elemental function

**Syntax:**

```fortran
RESULT = ACHAR(I [, KIND])
```

**Example:**

```fortran
program test_achar
  character c
  c = achar(32)
end program test_achar
```

**Note:** See ICHAR for a discussion of converting between numerical values and formatted string representations.

**See also:** CHAR, IACHAR, ICHAR

### 10.6 ACOS — Arccosine function

**ACOS (X)**

ACOS (X) computes the arccosine of X (inverse of COS (X)).

**Parameters X** — The type shall either be REAL with a magnitude that is less than or equal to one - or the type shall be COMPLEX.

**Returns** The return value is of the same type and kind as X. The real part of the result is in radians and lies in the range 0 leq Re acos(x) leq pi.

**Standard:** Fortran 77 and later, for a complex argument Fortran 2008 or later

**Class:** Elemental function

**Syntax:**

```fortran
RESULT = ACOS(X)
```

**Example:**

```fortran
program test_acos
  real(8) :: x = 0.866_8
  x = acos(x)
end program test_acos
```

**Specific names:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACOS (X)</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>DACOS (X)</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

**See also:** Inverse function: COS Degrees function: ACOSD
10.7 ACOSD — Arccosine function, degrees

ACOSD (X)
ACOSD (X) computes the arccosine of X in degrees (inverse of COSD (X)).

Parameters X – The type shall either be REAL with a magnitude that is less than or equal to one - or the type shall be COMPLEX.

Returns The return value is of the same type and kind as X. The real part of the result is in degrees and lies in the range 0 leq Re acos(x) leq 180.

Standard: GNU extension, enabled with -fdec-math
Class: Elemental function
Syntax:

RESULT = ACOSD (X)

Example:

```
program test_acosd
  real (8) :: x = 0.866_8
  x = acosd(x)
end program test_acosd
```

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACOSD(X)</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>DACOSD(X)</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

See also: Inverse function: COSD Radians function: ACOS

10.8 ACOSH — Inverse hyperbolic cosine function

ACOSH (X)
ACOSH (X) computes the inverse hyperbolic cosine of X.

Parameters X – The type shall be REAL or COMPLEX.

Returns The return value has the same type and kind as X. If X is complex, the imaginary part of the result is in radians and lies between 0 leq Im acosh(x) leq pi.

Standard: Fortran 2008 and later
Class: Elemental function
Syntax:

RESULT = ACOSH (X)

Example:
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

```fortran
PROGRAM test_acosh
  REAL(8), DIMENSION(3) :: x = (/ 1.0, 2.0, 3.0 /)
  WRITE (*,*) ACOSH(x)
END PROGRAM
```

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>DACOSH(X)</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

See also: Inverse function: COSH

10.9 ADJUSTL — Left adjust a string

**ADJUSTL**(STRING)

ADJUSTL(STRING) will left adjust a string by removing leading spaces. Spaces are inserted at the end of the string as needed.

Parameters **STRING** – The type shall be CHARACTER.

Returns The return value is of type CHARACTER and of the same kind as STRING where leading spaces are removed and the same number of spaces are inserted on the end of STRING.

Standard: Fortran 90 and later

Class: Elemental function

Syntax:

```
RESULT = ADJUSTL(STRING)
```

Example:

```
program test_adjustl
  character(len=20) :: str = ' gfortran'
  str = adjustl(str)
  print *, str
end program test_adjustl
```

See also: ADJUSTR, TRIM

10.10 ADJUSTR — Right adjust a string

**ADJUSTR**(STRING)

ADJUSTR(STRING) will right adjust a string by removing trailing spaces. Spaces are inserted at the start of the string as needed.

Parameters **STR** – The type shall be CHARACTER.

Returns The return value is of type CHARACTER and of the same kind as STRING where trailing spaces are removed and the same number of spaces are inserted at the start of STRING.
**Standard:** Fortran 90 and later

**Class:** Elemental function

**Syntax:**

\[
\text{RESULT} = \text{ADJUSTR(STRING)}
\]

**Example:**

```fortran
program test_adjustr
    character(len=20) :: str = 'gfortran'
    str = adjustr(str)
    print *, str
end program test_adjustr
```

**See also:** ADJUSTL, TRIM

### 10.11 AIMAG — Imaginary part of complex number

**AIMAG(Z)**

AIMAG(Z) yields the imaginary part of complex argument Z. The IMAG(Z) and IMAGPART(Z) intrinsic functions are provided for compatibility with g77, and their use in new code is strongly discouraged.

**Parameters** Z – The type of the argument shall be COMPLEX.

**Returns** The return value is of type REAL with the kind type parameter of the argument.

**Standard:** Fortran 77 and later, has overloads that are GNU extensions

**Class:** Elemental function

**Syntax:**

\[
\text{RESULT} = \text{AIMAG(Z)}
\]

**Example:**

```fortran
program test_aimag
    complex(4) z4
    complex(8) z8
    z4 = cmplx(1.e0_4, 0.e0_4)
    z8 = cmplx(0.e0_8, 1.e0_8)
    print *, aimag(z4), dimag(z8)
end program test_aimag
```

**Specific names:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIMAG(Z)</td>
<td>COMPLEX Z</td>
<td>REAL</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>DIMAG(Z)</td>
<td>COMPLEX(8) Z</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>IMAG(Z)</td>
<td>COMPLEX Z</td>
<td>REAL</td>
<td>GNU extension</td>
</tr>
<tr>
<td>IMAGPART(Z)</td>
<td>COMPLEX Z</td>
<td>REAL</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>
10.12 AINT — Truncate to a whole number

**AINT**(A[,, KIND])

AINT(A [, KIND]) truncates its argument to a whole number.

**Parameters**

- A — The type of the argument shall be REAL.
- KIND — (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

**Returns** The return value is of type REAL with the kind type parameter of the argument if the optional KIND is absent; otherwise, the kind type parameter will be given by KIND. If the magnitude of X is less than one, AINT(X) returns zero. If the magnitude is equal to or greater than one then it returns the largest whole number that does not exceed its magnitude. The sign is the same as the sign of X.

**Standard**: Fortran 77 and later

**Class**: Elemental function

**Syntax**:

\[
\text{RESULT} = \text{AINT}(A [, \text{KIND}])
\]

**Example**:

```fortran
program test_aint
  real(4) x4
  real(8) x8
  x4 = 1.234E0_4
  x8 = 4.321_8
  print *, aint(x4), dint(x8)
  x8 = aint(x4, 8)
end program test_aint
```

**Specific names**:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>AINT(A)</td>
<td>REAL(4) A</td>
<td>REAL(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>DINT(A)</td>
<td>REAL(8) A</td>
<td>REAL(8)</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

10.13 ALARM — Execute a routine after a given delay

**ALARM**(SECONDS, HANDLER[,, STATUS])

ALARM(SECONDS, HANDLER [, STATUS]) causes external subroutine HANDLER to be executed after a delay of SECONDS by using alarm(2) to set up a signal and signal(2) to catch it. If STATUS is supplied, it will be returned with the number of seconds remaining until any previously scheduled alarm was due to be delivered, or zero if there was no previously scheduled alarm.

**Parameters**

- SECONDS — The type of the argument shall be a scalar INTEGER. It is INTENT(IN).
• **HANDLER** – Signal handler (*INTEGER FUNCTION* or *SUBROUTINE*) or dummy/global *INTEGER* scalar. The scalar values may be either *SIG_IGN*=1 to ignore the alarm generated or *SIG_DFL*=0 to set the default action. It is *INTENT(IN)*.

• **STATUS** – (Optional) *STATUS* shall be a scalar variable of the default *INTEGER* kind. It is *INTENT(OUT)*.

**Standard:** GNU extension

**Class:** Subroutine

**Syntax:**

```fortran
CALL ALARM(SECONDS, HANDLER [, STATUS])
```

**Example:**

```fortran
program test_alarm
   external handler_print
   integer i
   call alarm (3, handler_print, i)
   print *, i
   call sleep(10)
end program test_alarm
```

This will cause the external routine `handler_print` to be called after 3 seconds.

### 10.14 ALL — All values in MASK along DIM are true

**ALL**(MASK[, DIM])

*ALL(MASK[, DIM]*) determines if all the values are true in *MASK* in the array along dimension *DIM*.

**Parameters**

- **MASK** – The type of the argument shall be *LOGICAL* and it shall not be scalar.

- **DIM** – (Optional) *DIM* shall be a scalar integer with a value that lies between one and the rank of *MASK*.

**Returns** *ALL(MASK)* returns a scalar value of type *LOGICAL* where the kind type parameter is the same as the kind type parameter of *MASK*. If *DIM* is present, then *ALL(MASK, DIM)* returns an array with the rank of *MASK* minus 1. The shape is determined from the shape of *MASK* where the *DIM* dimension is elided.

**Standard:** Fortran 90 and later

**Class:** Transformational function

**Syntax:**

```fortran
RESULT = ALL(MASK [, DIM])
```

**Example:**
program test_all
  logical l
  l = all((/.true., .true., .true./))
print *, l
call section
contains
  subroutine section
    integer a(2,3), b(2,3)
    a = 1
    b = 1
    b(2,2) = 2
print *, all(a .eq. b, 1)
print *, all(a .eq. b, 2)
end subroutine section
end program test_all

10.15 ALLOCATED — Status of an allocatable entity

ALLOCATED (ARRAY)

ALLOCATED(ARRAY) and ALLOCATED(SCALAR) check the allocation status of ARRAY and SCALAR, respectively.

Parameters

• ARRAY – The argument shall be an ALLOCATABLE array.
• SCALAR – The argument shall be an ALLOCATABLE scalar.

Returns The return value is a scalar LOGICAL with the default logical kind type parameter. If the argument is allocated, then the result is .TRUE.; otherwise, it returns .FALSE.

Standard: Fortran 90 and later. Note, the SCALAR= keyword and allocatable scalar entities are available in Fortran 2003 and later.

Class: Inquiry function

Syntax:

RESULT = ALLOCATED(ARRAY)
RESULT = ALLOCATED(SCALAR)

Example:

program test_allocated
  integer :: i = 4
  real(4), allocatable :: x(:)
  if (.not. allocated(x)) allocate(x(i))
end program test_allocated
10.16 AND — Bitwise logical AND

AND ()
Bitwise logical AND.

Parameters

- **I** – The type shall be either a scalar INTEGER type or a scalar LOGICAL type or a boz-literal-constant.
- **J** – The type shall be the same as the type of I or a boz-literal-constant. I and J shall not both be boz-literal-constants. If either I or J is a boz-literal-constant, then the other argument must be a scalar INTEGER.

Returns The return type is either a scalar INTEGER or a scalar LOGICAL. If the kind type parameters differ, then the smaller kind type is implicitly converted to larger kind, and the return has the larger kind. A boz-literal-constant is converted to an INTEGER with the kind type parameter of the other argument as-if a call to INT occurred.

*Standard*: GNU extension

*Class*: Function

*Syntax*:

```fortran
RESULT = AND(I, J)
```

*Example*:

```fortran
PROGRAM test_and
LOGICAL :: T = .TRUE., F = .FALSE.
INTEGER :: a, b
DATA a / Z'F' /, b / Z'3' /
WRITE (*,*) AND(T, T), AND(T, F), AND(F, T), AND(F, F)
WRITE (*,*) AND(a, b)
END PROGRAM
```

*See also*: Fortran 95 elemental function: IAND

10.17 ANINT — Nearest whole number

ANINT(A[, KIND])

ANINT(A [, KIND]) rounds its argument to the nearest whole number.

Parameters

- **A** – The type of the argument shall be REAL.
- **KIND** – (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

Returns The return value is of type real with the kind type parameter of the argument if the optional KIND is absent; otherwise, the kind type parameter will be given by KIND. If A is greater than zero, ANINT(A) returns AINT(X+0.5). If A is less than or equal to zero then it returns AINT(X–0.5).
**Standard**: Fortran 77 and later

**Class**: Elemental function

**Syntax**:

```
RESULT = ANINT(A [, KIND])
```

**Example**:

```
program test_anint
    real(4) x4
    real(8) x8
    x4 = 1.234E0_4
    x8 = 4.321_8
    print *, anint(x4), dnint(x8)
    x8 = anint(x4, 8)
end program test_anint
```

**Specific names**:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANINT(A)</td>
<td>REAL(4) A</td>
<td>REAL(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>DNINT(A)</td>
<td>REAL(8) A</td>
<td>REAL(8)</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

### 10.18 ANY — Any value in MASK along DIM is true

**ANY**(MASK[, DIM])

ANY(MASK [, DIM]) determines if any of the values in the logical array MASK along dimension DIM are .TRUE..

**Parameters**

- **MASK** – The type of the argument shall be LOGICAL and it shall not be scalar.
- **DIM** – (Optional) DIM shall be a scalar integer with a value that lies between one and the rank of MASK.

**Returns** ANY(MASK) returns a scalar value of type LOGICAL where the kind type parameter is the same as the kind type parameter of MASK. If DIM is present, then ANY(MASK, DIM) returns an array with the rank of MASK minus 1. The shape is determined from the shape of MASK where the DIM dimension is elided.

**Standard**: Fortran 90 and later

**Class**: Transformational function

**Syntax**:

```
RESULT = ANY(MASK [, DIM])
```

**Example**:
program test_any
  logical l
  l = any((/true., .true., .true./))
print *, l
call section
contains
  subroutine section
    integer a(2,3), b(2,3)
    a = 1
    b = 1
    b(2,2) = 2
    print *, any(a .eq. b, 1)
    print *, any(a .eq. b, 2)
  end subroutine section
end subroutine section
end program test_any

10.19 ASIN — Arcsine function

ASIN(X)

ASIN(X) computes the arcsine of its X (inverse of SIN(X)).

Parameters X – The type shall be either REAL and a magnitude that is less than or equal to one - or be COMPLEX.

Returns The return value is of the same type and kind as X. The real part of the result is in radians and lies in the range -pi/2 leq Re asin(x) leq pi/2.

Standard: Fortran 77 and later, for a complex argument Fortran 2008 or later

Class: Elemental function

Syntax:

RESULT = ASIN(X)

Example:

program test_asin
  real(8) :: x = 0.866_8
  x = asin(x)
end program test_asin

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIN(X)</td>
<td>REAL(4) X</td>
<td>REAL(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>DASIN(X)</td>
<td>REAL(8) X</td>
<td>REAL(8)</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

See also: Inverse function: SIN Degrees function: ASIND
10.20 ASIND — Arcsine function, degrees

ASIND(X)

ASIND(X) computes the arcsine of its X in degrees (inverse of SIND(X)).

**Parameters** X – The type shall be either REAL and a magnitude that is less than or equal to one - or be COMPLEX.

**Returns** The return value is of the same type and kind as X. The real part of the result is in degrees and lies in the range -90 leq Re asin(x) leq 90.

**Standard**: GNU extension, enabled with `-fdec-math`.

**Class**: Elemental function

**Syntax**:

\[
\text{RESULT} = \text{ASIND}(X)
\]

**Example**:

```fortran
program test_asind
   real(8) :: x = 0.866_8
   x = asind(x)
end program test_asind
```

**Specific names**:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIND(X)</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>DASIND(X)</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

**See also**: Inverse function: SIND Radians function: ASIN

10.21 ASINH — Inverse hyperbolic sine function

ASINH(X)

ASINH(X) computes the inverse hyperbolic sine of X.

**Parameters** X – The type shall be REAL or COMPLEX.

**Returns** The return value is of the same type and kind as X. If X is complex, the imaginary part of the result is in radians and lies between -pi/2 leq Im asinh(x) leq pi/2.

**Standard**: Fortran 2008 and later

**Class**: Elemental function

**Syntax**:

\[
\text{RESULT} = \text{ASINH}(X)
\]

**Example**:
Program test_asinh

REAL (8), DIMENSION (3) :: x = (/ -1.0, 0.0, 1.0 /)
WRITE (*,*) ASINH(x)

END PROGRAM

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>DASINH(X)</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
<td>GNU extension.</td>
</tr>
</tbody>
</table>

See also: Inverse function: SINH

10.22 ASSOCIATED — Status of a pointer or pointer/target pair

ASSOCIATED (POINTER[, TARGET])

ASSOCIATED(POINTER [, TARGET]) determines the status of the pointer POINTER or if POINTER is associated with the target TARGET.

Parameters

- POINTER – POINTER shall have the POINTER attribute and it can be of any type.
- TARGET – (Optional) TARGET shall be a pointer or a target. It must have the same type, kind type parameter, and array rank as POINTER.

Returns

ASSOCIATED(POINTER) returns a scalar value of type LOGICAL(4). There are several cases:

Standard: Fortran 90 and later

Class: Inquiry function

Syntax:

RESULT = ASSOCIATED(POINTER [, TARGET])

Example:

program test_associated
  implicit none
  real, target :: tgt(2) = (/1., 2./)
  real, pointer :: ptr(:)
  ptr => tgt
  if (associated(ptr) .eqv. .false.) call abort
  if (associated(ptr,tgt) .eqv. .false.) call abort
end program test_associated

See also: NULL
10.23 ATAN — Arctangent function

ATAN (X)

ATAN (X) computes the arctangent of X.

Parameters

• X – The type shall be REAL or COMPLEX; if Y is present, X shall be REAL.
• Y – The type and kind type parameter shall be the same as X.

Returns The return value is of the same type and kind as X. If Y is present, the result is identical to ATAN2(Y, X). Otherwise, it is the arccus tangent of X, where the real part of the result is in radians and lies in the range \(-\pi/2 \leq \text{Re atan}(x) \leq \pi/2\).

Standard: Fortran 77 and later, for a complex argument and for two arguments Fortran 2008 or later

Class: Elemental function

Syntax:

\[
\text{RESULT} = \text{ATAN}(X) \\
\text{RESULT} = \text{ATAN}(Y, X)
\]

Example:

```
program test_atan
    real (8) :: x = 2.866_8
    x = atan(x)
end program test_atan
```

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATAN(X)</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>DATAN(X)</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

See also: Inverse function: TAN Degrees function: ATAND

10.24 ATAND — Arctangent function, degrees

ATAND (X)

ATAND (X) computes the arctangent of X in degrees (inverse of TAND).

Parameters

• X – The type shall be REAL or COMPLEX; if Y is present, X shall be REAL.
• Y – The type and kind type parameter shall be the same as X.

Returns The return value is of the same type and kind as X. If Y is present, the result is identical to ATAND2(Y, X). Otherwise, it is the arcus tangent of X, where the real part of the result is in degrees and lies in the range \(-90 \leq \text{Re atand}(x) \leq 90\).

Standard: GNU extension, enabled with \(-fdec-math.\)
**Class**: Elemental function

**Syntax**:

\[
\text{RESULT} = \text{ATAND} (X) \\
\text{RESULT} = \text{ATAND} (Y, X)
\]

**Example**:

```fortran
program test_atand
    real(8) :: x = 2.866_8
    x = atand(x)
end program test_atand
```

**Specific names**:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATAND (X)</td>
<td>REAL(4) X</td>
<td>REAL(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>DATAND (X)</td>
<td>REAL(8) X</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

**See also**: Inverse function: TAND, Radians function: ATAN

### 10.25 ATAN2 — Arctangent function

\( \text{ATAN2}(Y, X) \)

\( \text{ATAN2}(Y, X) \) computes the principal value of the argument function of the complex number \( X + i \ Y \). This function can be used to transform from Cartesian into polar coordinates and allows to determine the angle in the correct quadrant.

**Parameters**

- \( Y \) – The type shall be REAL.
- \( X \) – The type and kind type parameter shall be the same as \( Y \). If \( Y \) is zero, then \( X \) must be nonzero.

**Returns** The return value has the same type and kind type parameter as \( Y \). It is the principal value of the complex number \( X + i \ Y \). If \( X \) is nonzero, then it lies in the range \(-\pi \leq \text{atan}(x) \leq \pi\). The sign is positive if \( Y \) is positive. If \( Y \) is zero, then the return value is zero if \( X \) is strictly positive, \( \pi \) if \( X \) is negative and \( Y \) is positive zero (or the processor does not handle signed zeros), and \(-\pi \) if \( X \) is negative and \( Y \) is negative zero. Finally, if \( X \) is zero, then the magnitude of the result is \( \pi/2 \).

**Standard**: Fortran 77 and later

**Class**: Elemental function

**Syntax**:

\[
\text{RESULT} = \text{ATAN2} (Y, X)
\]

**Example**:

10.25. ATAN2 — Arctangent function
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

```fortran
program test_atan2
    real(4) :: x = 1.e0_4, y = 0.5e0_4
    x = atan2(y,x)
end program test_atan2
```

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATAN2(X, Y)</td>
<td>REAL(4) X, Y</td>
<td>REAL(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>DATAN2(X, Y)</td>
<td>REAL(8) X, Y</td>
<td>REAL(8)</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

See also: Alias: ATAN Degrees function: ATAN2D

## 10.26 ATAN2D — Arctangent function, degrees

ATAN2D(Y, X) computes the principal value of the argument function of the complex number X + i Y in degrees. This function can be used to transform from Cartesian into polar coordinates and allows to determine the angle in the correct quadrant.

**Parameters**

- **Y** – The type shall be REAL.
- **X** – The type and kind type parameter shall be the same as Y. If Y is zero, then X must be nonzero.

**Returns** The return value has the same type and kind type parameter as Y. It is the principal value of the complex number X + i Y. If X is nonzero, then it lies in the range -180 le atan(x) leq 180. The sign is positive if Y is positive. If Y is zero, then the return value is zero if X is strictly positive, 180 if X is negative and Y is positive zero (or the processor does not handle signed zeros), and -180 if X is negative and Y is negative zero. Finally, if X is zero, then the magnitude of the result is 90.

**Standard**: GNU extension, enabled with `-fdec-math`.

**Class**: Elemental function

**Syntax**:

```fortran
RESULT = ATAN2D(Y, X)
```

**Example**:

```fortran
program test_atan2d
    real(4) :: x = 1.e0_4, y = 0.5e0_4
    x = atan2d(y,x)
end program test_atan2d
```

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATAN2D(X, Y)</td>
<td>REAL(4) X, Y</td>
<td>REAL(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>DATAN2D(X, Y)</td>
<td>REAL(8) X, Y</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>
10.27 ATANH — Inverse hyperbolic tangent function

ATANH(X)
ATANH(X) computes the inverse hyperbolic tangent of X.

**Parameters**
- X – The type shall be REAL or COMPLEX.

**Returns**
The return value has same type and kind as X. If X is complex, the imaginary part of the result is in radians and lies between -pi/2 leq Im atanh(x) leq pi/2.

**Standard**: Fortran 2008 and later

**Class**: Elemental function

**Syntax**:

```
RESULT = ATANH(X)
```

**Example**:

```
PROGRAM test_atanh
  REAL, DIMENSION(3) :: x = (/ -1.0, 0.0, 1.0 /)
  WRITE (*,*) ATANH(x)
END PROGRAM
```

**Specific names**:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATANH(X)</td>
<td>REAL(8)</td>
<td>X</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

10.28 ATOMIC_ADD — Atomic ADD operation

ATOMIC_ADD(ATOM, VALUE)
ATOMIC_ADD(ATOM, VALUE) atomically adds the value of VALUE to the variable ATOM. When STAT is present and the invocation was successful, it is assigned the value 0. If it is present and the invocation has failed, it is assigned a positive value; in particular, for a coindexed ATOM, if the remote image has stopped, it is assigned the value of ISO_FORTRAN_ENV’s STAT_STOPPED_IMAGE and if the remote image has failed, the value STAT_FAILED_IMAGE.

**Parameters**
- ATOM – Scalar coarray or coindexed variable of integer type with ATOMIC_INT_KIND kind.
- VALUE – Scalar of the same type as ATOM. If the kind is different, the value is converted to the kind of ATOM.
- STAT – (optional) Scalar default-kind integer variable.

**Standard**: TS 18508 or later
**Class**: Atomic subroutine

**Syntax**:

```fortran
CALL ATOMIC_ADD (ATOM, VALUE [, STAT])
```

**Example**:

```fortran
program atomic
  use iso_fortran_env
  integer(atomic_int_kind) :: atom[*]
  call atomic_add (atom[1], this_image())
end program atomic
```

**See also**: ATOMIC_DEFINE, ATOMIC_FETCH_ADD, ISO_FORTRAN_ENV, ATOMIC_AND, ATOMIC_OR, ATOMIC_XOR

### 10.29 ATOMIC_AND — Atomic bitwise AND operation

**ATOMIC_AND(ATOM, VALUE)**

ATOMIC_AND(ATOM, VALUE) atomically defines ATOM with the bitwise AND between the values of ATOM and VALUE. When STAT is present and the invocation was successful, it is assigned the value 0. If it is present and the invocation has failed, it is assigned a positive value; in particular, for a coindexed ATOM, if the remote image has stopped, it is assigned the value of ISO_FORTRAN_ENV’s STAT_STOPPED_IMAGE and if the remote image has failed, the value STAT_FAILED_IMAGE.

**Parameters**

- **ATOM** – Scalar coarray or coindexed variable of integer type with ATOMIC_INT_KIND kind.
- **VALUE** – Scalar of the same type as ATOM. If the kind is different, the value is converted to the kind of ATOM.
- **STAT** – (optional) Scalar default-kind integer variable.

**Standard**: TS 18508 or later

**Class**: Atomic subroutine

**Syntax**:

```fortran
CALL ATOMIC_AND (ATOM, VALUE [, STAT])
```

**Example**:

```fortran
program atomic
  use iso_fortran_env
  integer(atomic_int_kind) :: atom[*]
  call atomic_and (atom[1], int('10100011101', 2))
end program atomic
```

**See also**: ATOMIC_DEFINE, ATOMIC_FETCH_AND, ISO_FORTRAN_ENV, ATOMIC_ADD, ATOMIC_OR, ATOMIC_XOR
10.30 ATOMIC_CAS — Atomic compare and swap

**ATOMIC_CAS()**

ATOMIC_CAS compares the variable ATOM with the value of COMPARE; if the value is the same, ATOM is set to the value of NEW. Additionally, OLD is set to the value of ATOM that was used for the comparison. When STAT is present and the invocation was successful, it is assigned the value 0. If it is present and the invocation has failed, it is assigned a positive value; in particular, for a coindexed ATOM, if the remote image has stopped, it is assigned the value of ISO_FORTRAN_ENV’s STAT_STOPPED_IMAGE and if the remote image has failed, the value STAT_FAILED_IMAGE.

**Parameters**

- **ATOM** – Scalar coarray or coindexed variable of either integer type with ATOMIC_INT_KIND kind or logical type with ATOMIC_LOGICAL_KIND kind.
- **OLD** – Scalar of the same type and kind as ATOM.
- **COMPARE** – Scalar variable of the same type and kind as ATOM.
- **NEW** – Scalar variable of the same type as ATOM. If kind is different, the value is converted to the kind of ATOM.
- **STAT** – (optional) Scalar default-kind integer variable.

**Standard:** TS 18508 or later

**Class:** Atomic subroutine

**Syntax:**

```
call atomic_cas (atom, prev, .false., .true.))
```

**Example:**

```fortran
program atomic
use iso_fortran_env
logical(atomic_logical_kind) :: atom[*], prev
call atomic_cas (atom[1], prev, .false., .true.))
end program atomic
```

**See also:** ATOMIC_DEFINE, ATOMIC_REF, ISO_FORTRAN_ENV

10.31 ATOMIC_DEFINE — Setting a variable atomically

**ATOMIC_DEFINE(ATOM, VALUE)**

ATOMIC_DEFINE(ATOM, VALUE) defines the variable ATOM with the value VALUE atomically. When STAT is present and the invocation was successful, it is assigned the value 0. If it is present and the invocation has failed, it is assigned a positive value; in particular, for a coindexed ATOM, if the remote image has stopped, it is assigned the value of ISO_FORTRAN_ENV’s STAT_STOPPED_IMAGE and if the remote image has failed, the value STAT_FAILED_IMAGE.

**Parameters**

- **ATOM** – Scalar coarray or coindexed variable of either integer type with ATOMIC_INT_KIND kind or logical type with ATOMIC_LOGICAL_KIND kind.

**See also:** ATOMIC_DEFINE, ATOMIC_REF, ISO_FORTRAN_ENV
• **VALUE** – Scalar of the same type as `ATOM`. If the kind is different, the value is converted to the kind of `ATOM`.

• **STAT** – (optional) Scalar default-kind integer variable.

**Standard**: Fortran 2008 and later; with `STAT`, TS 18508 or later

**Class**: Atomic subroutine

**Syntax**:

```fortran
CALL ATOMIC_DEFINE (ATOM, VALUE [, STAT])
```

**Example**:

```fortran
program atomic
  use iso_fortran_env
  integer(atomic_int_kind) :: atom[*]
  call atomic_define (atom[1], this_image())
end program atomic
```

**See also**: ATOMIC_REF, ATOMIC_CAS, ISO_FORTRAN_ENV, ATOMIC_ADD, ATOMIC_AND, ATOMIC_OR, ATOMIC_XOR

### 10.32 ATOMIC_FETCH_ADD — Atomic ADD operation with prior fetch

**ATOMIC_FETCH_ADD** `(ATOM, VALUE, OLD)`

`ATOMIC_FETCH_ADD` stores the value of `ATOM` in `OLD` and adds the value of `VALUE` to the variable `ATOM`. When `STAT` is present and the invocation was successful, it is assigned the value 0. If it is present and the invocation has failed, it is assigned a positive value; in particular, for a coindexed `ATOM`, if the remote image has stopped, it is assigned the value of ISO_FORTRAN_ENV’s `STAT_STOPPED_IMAGE` and if the remote image has failed, the value `STAT_FAILED_IMAGE`.

**Parameters**

- **ATOM** – Scalar coarray or coindexed variable of integer type with `ATOMIC_INT_KIND` kind.
- **VALUE** – Scalar of the same type as `ATOM`. If the kind is different, the value is converted to the kind of `ATOM`.
- **OLD** – Scalar of the same type and kind as `ATOM`.
- **STAT** – (optional) Scalar default-kind integer variable.

**Standard**: TS 18508 or later

**Class**: Atomic subroutine

**Syntax**:

```fortran
CALL ATOMIC_FETCH_ADD (ATOM, VALUE, old [, STAT])
```

**Example**:
program atomic
use iso_fortran_env
integer(atomic_int_kind) :: atom[*], old
call atomic_add (atom[1], this_image(), old)
end program atomic

See also: ATOMIC_DEFINE, ATOMIC_ADD, ISO_FORTRAN_ENV, ATOMIC_FETCH_AND, ATOMIC_FETCH_OR, ATOMIC_FETCH_XOR

10.33 ATOMIC_FETCH_AND — Atomic bitwise AND operation with prior fetch

ATOMIC_FETCH_AND()

ATOMIC_AND(ATOM, VALUE) atomically stores the value of ATOM in OLD and defines ATOM with the bitwise AND between the values of ATOM and VALUE. When STAT is present and the invocation was successful, it is assigned the value 0. If it is present and the invocation has failed, it is assigned a positive value; in particular, for a coindexed ATOM, if the remote image has stopped, it is assigned the value of ISO_FORTRAN_ENV’s STAT_STOPPED_IMAGE and if the remote image has failed, the value STAT_FAILED_IMAGE.

Parameters

- ATOM – Scalar coarray or coindexed variable of integer type with ATOMIC_INT_KIND kind.
- VALUE – Scalar of the same type as ATOM. If the kind is different, the value is converted to the kind of ATOM.
- OLD – Scalar of the same type and kind as ATOM.
- STAT – (optional) Scalar default-kind integer variable.

Standard: TS 18508 or later

Class: Atomic subroutine

Syntax:

CALL ATOMIC_FETCH_AND (ATOM, VALUE, OLD [, STAT])

Example:

program atomic
use iso_fortran_env
integer(atomic_int_kind) :: atom[*], old
call atomic_fetch_and (atom[1], int('10100011101'), old)
end program atomic

See also: ATOMIC_DEFINE, ATOMIC_AND, ISO_FORTRAN_ENV, ATOMIC_FETCH_ADD, ATOMIC_FETCH_OR, ATOMIC_FETCH_XOR
10.34 ATOMIC_FETCH_OR — Atomic bitwise OR operation with prior fetch

ATOMIC_FETCH_OR()

ATOMIC_FETCH_OR(ATOM, VALUE) atomically stores the value of ATOM in OLD and defines ATOM with the bitwise OR between the values of ATOM and VALUE. When STAT is present and the invocation was successful, it is assigned the value 0. If it is present and the invocation has failed, it is assigned a positive value; in particular, for a coindexed ATOM, if the remote image has stopped, it is assigned the value of ISO_FORTRAN_ENV’s STAT_STOPPED_IMAGE and if the remote image has failed, the value STAT_FAILED_IMAGE.

Parameters

- **ATOM** – Scalar coarray or coindexed variable of integer type with ATOMIC_INT_KIND kind.
- **VALUE** – Scalar of the same type as ATOM. If the kind is different, the value is converted to the kind of ATOM.
- **OLD** – Scalar of the same type and kind as ATOM.
- **STAT** – (optional) Scalar default-kind integer variable.

Standard: TS 18508 or later

Class: Atomic subroutine

Syntax:

```
CALL ATOMIC_FETCH_OR (ATOM, VALUE, OLD [, STAT])
```

Example:

```fortran
program atomic
  use iso_fortran_env
  integer(atomic_int_kind) :: atom[*], old
  call atomic_fetch_or (atom[1], int(b'10100011101'), old)
end program atomic
```

See also: ATOMIC_DEFINE, ATOMIC_OR, ISO_FORTRAN_ENV, ATOMIC_FETCH_ADD, ATOMIC_FETCH_AND, ATOMIC_FETCH_XOR

10.35 ATOMIC_FETCH_XOR — Atomic bitwise XOR operation with prior fetch

ATOMIC_FETCH_XOR()

ATOMIC_FETCH_XOR(ATOM, VALUE) atomically stores the value of ATOM in OLD and defines ATOM with the bitwise XOR between the values of ATOM and VALUE. When STAT is present and the invocation was successful, it is assigned the value 0. If it is present and the invocation has failed, it is assigned a positive value; in particular, for a coindexed ATOM, if the remote image has stopped, it is assigned the value of ISO_FORTRAN_ENV’s STAT_STOPPED_IMAGE and if the remote image has failed, the value STAT_FAILED_IMAGE.

Parameters

- **ATOM** – Scalar coarray or coindexed variable of integer type with ATOMIC_INT_KIND kind.
• **VALUE** – Scalar of the same type as \textit{ATOM}. If the kind is different, the value is converted to the kind of \textit{ATOM}.

• **OLD** – Scalar of the same type and kind as \textit{ATOM}.

• **STAT** – (optional) Scalar default-kind integer variable.

\textbf{Standard}: TS 18508 or later

\textbf{Class}: Atomic subroutine

\textbf{Syntax}:

\begin{verbatim}
CALL ATOMIC_FETCH_XOR (ATOM, VALUE, OLD [, STAT])
\end{verbatim}

\textbf{Example}:

```
program atomic
    use iso_fortran_env
    integer(atomic_int_kind) :: atom[*], old
    call atomic_fetch_xor (atom[1], int(b'10100011101'), old)
end program atomic
```

\textbf{See also}: ATOMIC_DEFINE, ATOMIC_XOR, ISO_FORTRAN_ENV, ATOMIC_FETCH_ADD, ATOMIC_FETCH_AND, ATOMIC_FETCH_OR

10.36 \textbf{ATOMIC\_OR} — Atomic bitwise OR operation

\textbf{ATOMIC\_OR} (ATOM, VALUE)

\texttt{ATOMIC\_OR(ATOM, VALUE)} atomically defines \textit{ATOM} with the bitwise \texttt{AND} between the values of \textit{ATOM} and \textit{VALUE}. When \textit{STAT} is present and the invocation was successful, it is assigned the value 0. If it is present and the invocation has failed, it is assigned a positive value; in particular, for a coindexed \textit{ATOM}, if the remote image has stopped, it is assigned the value of \texttt{ISO\_FORTRAN\_ENV\_STAT\_STOPPED\_IMAGE} and if the remote image has failed, the value \texttt{STAT\_FAILED\_IMAGE}.

\textbf{Parameters}

• \texttt{ATOM} – Scalar coarray or coindexed variable of integer type with \texttt{ATOMIC\_INT\__KIND} kind.

• \texttt{VALUE} – Scalar of the same type as \textit{ATOM}. If the kind is different, the value is converted to the kind of \textit{ATOM}.

• \texttt{STAT} – (optional) Scalar default-kind integer variable.

\textbf{Standard}: TS 18508 or later

\textbf{Class}: Atomic subroutine

\textbf{Syntax}:

\begin{verbatim}
CALL ATOMIC\_OR (ATOM, VALUE [, STAT])
\end{verbatim}

\textbf{Example}:
program atomic
use iso_fortran_env
integer(atomic_int_kind) :: atom[*]
call atomic_or (atom[1], int(b'10100011101'))
end program atomic

See also: ATOMIC_DEFINE, ATOMIC_FETCH_OR, ISO_FORTRAN_ENV, ATOMIC_ADD,
ATOMIC_OR, ATOMIC_XOR

10.37 ATOMIC_REF — Obtaining the value of a variable atomically

ATOMIC_REF()
ATOMIC_DEFINE (ATOM, VALUE) atomically assigns the value of the variable ATOM to VALUE. When STAT
is present and the invocation was successful, it is assigned the value 0. If it is present and the invocation has failed,
it is assigned a positive value; in particular, for a coindexed ATOM, if the remote image has stopped, it is assigned
the value of ISO_FORTRAN_ENV’s STAT_STOPPED_IMAGE and if the remote image has failed, the value
STAT_FAILED_IMAGE.

Parameters

- VALUE – Scalar of the same type as ATOM. If the kind is different, the value is converted to
  the kind of ATOM.
- ATOM – Scalar coarray or coindexed variable of either integer type with ATOMIC_INT_KIND
  kind or logical type with ATOMIC_LOGICAL_KIND kind.
- STAT – (optional) Scalar default-kind integer variable.

Standard: Fortran 2008 and later; with STAT, TS 18508 or later

Class: Atomic subroutine

Syntax:

CALL ATOMIC_REF(VALUE, ATOM [, STAT])

Example:

program atomic
use iso_fortran_env
logical(atomic_logical_kind) :: atom[*]
logical :: val
call atomic_ref (atom, .false.)
! ...
call atomic_ref (atom, val)
if (val) then
  print *, "Obtained"
end if
end program atomic

See also: ATOMIC_DEFINE, ATOMIC_CAS, ISO_FORTRAN_ENV, ATOMIC_FETCH_ADD,
ATOMIC_FETCH_AND, ATOMIC_FETCH_OR, ATOMIC_FETCH_XOR
10.38 ATOMIC_XOR — Atomic bitwise OR operation

ATOMIC_XOR()

ATOMIC_AND (ATOM, VALUE) atomically defines ATOM with the bitwise XOR between the values of ATOM and VALUE. When STAT is present and the invocation was successful, it is assigned the value 0. If it is present and the invocation has failed, it is assigned a positive value; in particular, for a coindexed ATOM, if the remote image has stopped, it is assigned the value of ISO_FORTRAN_ENV’s STAT_STOPPED_IMAGE and if the remote image has failed, the value STAT_FAILED_IMAGE.

Parameters

• ATOM – Scalar coarray or coindexed variable of integer type with ATOMIC_INT_KIND kind.
• VALUE – Scalar of the same type as ATOM. If the kind is different, the value is converted to the kind of ATOM.
• STAT – (optional) Scalar default-kind integer variable.

Standard: TS 18508 or later

Class: Atomic subroutine

Syntax:

CALL ATOMIC_XOR (ATOM, VALUE [, STAT])

Example:

program atomic
  use iso_fortran_env
  integer(atomic_int_kind) :: atom[*]
  call atomic_xor (atom[1], int('10100011101'))
end program atomic

See also: ATOMIC_DEFINE, ATOMIC_FETCH_XOR, ISO_FORTRAN_ENV, ATOMIC_ADD, ATOMIC_OR, ATOMIC_XOR

10.39 BACKTRACE — Show a backtrace

BACKTRACE()

BACKTRACE shows a backtrace at an arbitrary place in user code. Program execution continues normally afterwards. The backtrace information is printed to the unit corresponding to ERROR_UNIT in ISO_FORTRAN_ENV.

Standard: GNU extension

Class: Subroutine

Syntax:

CALL BACKTRACE

Arguments: None

See also: ABORT
10.40 BESSEL_J0 — Bessel function of the first kind of order 0

**BESSEL_J0(X)**

**BESSEL_J0(X)** computes the Bessel function of the first kind of order 0 of X. This function is available under the name **BESJ0** as a GNU extension.

**Parameters** X – The type shall be **REAL**.

**Returns** The return value is of type **REAL** and lies in the range - 0.4027… leq Bessel (0,x) leq 1. It has the same kind as X.

**Standard**: Fortran 2008 and later

**Class**: Elemental function

**Syntax**:

```fortran
RESULT = BESSEL_J0(X)
```

**Example**:

```fortran
program test_besj0
  real(8) :: x = 0.0_8
  x = bessel_j0(x)
end program test_besj0
```

**Specific names**:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBESJ0(X)</td>
<td>REAL(8) X</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

10.41 BESSEL_J1 — Bessel function of the first kind of order 1

**BESSEL_J1(X)**

**BESSEL_J1(X)** computes the Bessel function of the first kind of order 1 of X. This function is available under the name **BESJ1** as a GNU extension.

**Parameters** X – The type shall be **REAL**.

**Returns** The return value is of type **REAL** and lies in the range - 0.5818… leq Bessel (0,x) leq 0.5818. It has the same kind as X.

**Standard**: Fortran 2008

**Class**: Elemental function

**Syntax**:

```fortran
RESULT = BESSEL_J1(X)
```

**Example**:
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

```fortran
program test_besj1
  real(8) :: x = 1.0_8
  x = bessel_j1(x)
end program test_besj1
```

**Specific names:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBESJ1(X)</td>
<td>REAL(8) X</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

### 10.42 BESSEL_JN — Bessel function of the first kind

**BESSEL_JN** \((N, X)\)

BESSEL_JN \((N, X)\) computes the Bessel function of the first kind of order \(N\) of \(X\). This function is available under the name BESJN as a GNU extension. If \(N\) and \(X\) are arrays, their ranks and shapes shall conform.

**Parameters**

- \(N\) – Shall be a scalar or an array of type INTEGER.
- \(N1\) – Shall be a non-negative scalar of type INTEGER.
- \(N2\) – Shall be a non-negative scalar of type INTEGER.
- \(X\) – Shall be a scalar or an array of type REAL; for BESSEL_JN \((N1, N2, X)\) it shall be scalar.

**Returns** The return value is a scalar of type REAL. It has the same kind as \(X\).

**Standard:** Fortran 2008 and later, negative \(N\) is allowed as GNU extension

**Class:** Elemental function, except for the transformational function BESSEL_JN \((N1, N2, X)\)

**Syntax:**

```fortran
RESULT = BESSEL_JN(N, X)
RESULT = BESSEL_JN(N1, N2, X)
```

**Note:** The transformational function uses a recurrence algorithm which might, for some values of \(X\), lead to different results than calls to the elemental function.

**Example:**

```fortran
program test_besjn
  real(8) :: x = 1.0_8
  x = bessel_jn(5, x)
end program test_besjn
```

**Specific names:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBESJN(N, X)</td>
<td>INTEGER N REAL(8) X</td>
<td>REAL(8)</td>
<td></td>
</tr>
</tbody>
</table>

10.42. BESSEL_JN — Bessel function of the first kind 135
10.43 BESSEL_Y0 — Bessel function of the second kind of order 0

BESSEL_Y0(X)
BESSEL_Y0(X) computes the Bessel function of the second kind of order 0 of X. This function is available under the name BESY0 as a GNU extension.

Parameters X – The type shall be REAL.

Returns The return value is of type REAL. It has the same kind as X.

Standard: Fortran 2008 and later

Class: Elemental function

Syntax:

RESULT = BESSEL_Y0(X)

Example:

```fortran
program test_besy0
    real(8) :: x = 0.0_8
    x = bessel_y0(x)
end program test_besy0
```

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBESY0(X)</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

10.44 BESSEL_Y1 — Bessel function of the second kind of order 1

BESSEL_Y1(X)
BESSEL_Y1(X) computes the Bessel function of the second kind of order 1 of X. This function is available under the name BESY1 as a GNU extension.

Parameters X – The type shall be REAL.

Returns The return value is of type REAL. It has the same kind as X.

Standard: Fortran 2008 and later

Class: Elemental function

Syntax:

RESULT = BESSEL_Y1(X)

Example:

```fortran
program test_besy1
    real(8) :: x = 1.0_8
    x = bessel_y1(x)
end program test_besy1
```
Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBESY1(X)</td>
<td>REAL(8) X</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

10.45 BESSEL_YN — Bessel function of the second kind

BESSEL_YN(N, X)

BESSEL_YN(N, X) computes the Bessel function of the second kind of order N of X. This function is available under the name BESYN as a GNU extension. If N and X are arrays, their ranks and shapes shall conform.

Parameters

- N — Shall be a scalar or an array of type INTEGER.
- N1 — Shall be a non-negative scalar of type INTEGER.
- N2 — Shall be a non-negative scalar of type INTEGER.
- X — Shall be a scalar or an array of type REAL; for BESSEL_YN(N1, N2, X) it shall be scalar.

Returns The return value is a scalar of type REAL. It has the same kind as X.

Standard: Fortran 2008 and later, negative N is allowed as GNU extension

Class: Elemental function, except for the transformational function BESSEL_YN(N1, N2, X)

Syntax:

```fortran
RESULT = BESSEL_YN(N, X)
RESULT = BESSEL_YN(N1, N2, X)
```

Note: The transformational function uses a recurrence algorithm which might, for some values of X, lead to different results than calls to the elemental function.

Example:

```fortran
program test_besyn
  real(8) :: x = 1.0_8
  x = bessel_yn(5, x)
end program test_besyn
```

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBESYN(N,X)</td>
<td>INTEGER N REAL(8) X</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>
10.46 BGE — Bitwise greater than or equal to

BGE ()
Determines whether an integral is a bitwise greater than or equal to another.

Parameters
- I – Shall be of INTEGER type.
- J – Shall be of INTEGER type, and of the same kind as I.

Returns The return value is of type LOGICAL and of the default kind.

Standard: Fortran 2008 and later
Class: Elemental function

Syntax:
RESULT = BGE(I, J)

See also: BGT, BLE, BLT

10.47 BGT — Bitwise greater than

BGT ()
Determines whether an integral is a bitwise greater than another.

Parameters
- I – Shall be of INTEGER type.
- J – Shall be of INTEGER type, and of the same kind as I.

Returns The return value is of type LOGICAL and of the default kind.

Standard: Fortran 2008 and later
Class: Elemental function

Syntax:
RESULT = BGT(I, J)

See also: BGE, BLE, BLT
10.48 BIT_SIZE — Bit size inquiry function

BIT_SIZE(I)

BIT_SIZE(I) returns the number of bits (integer precision plus sign bit) represented by the type of I. The result of BIT_SIZE(I) is independent of the actual value of I.

Parameters
- I – The type shall be INTEGER.

Returns
- The return value is of type INTEGER

Standard: Fortran 90 and later

Class: Inquiry function

Syntax:

\[
\text{RESULT} = \text{BIT\_SIZE}(I)
\]

Example:

```fortran
program test_bit_size
  integer :: i = 123
  integer :: size
  size = bit_size(i)
  print *, size
end program test_bit_size
```

10.49 BLE — Bitwise less than or equal to

BLE(I, J)

Determines whether an integral is a bitwise less than or equal to another.

Parameters

- I – Shall be of INTEGER type.
- J – Shall be of INTEGER type, and of the same kind as I.

Returns
- The return value is of type LOGICAL and of the default kind.

Standard: Fortran 2008 and later

Class: Elemental function

Syntax:

\[
\text{RESULT} = \text{BLE}(I, J)
\]

See also: BGT, BGE, BLT
10.50 BLT — Bitwise less than

BLT ()
Determines whether an integral is a bitwise less than another.

Parameters

• I – Shall be of INTEGER type.
• J – Shall be of INTEGER type, and of the same kind as I.

Returns  The return value is of type LOGICAL and of the default kind.

Standard:  Fortran 2008 and later
Class:  Elemental function

Syntax:

\[
\text{RESULT} = \text{BLT}(I, J)
\]

See also:  BGE, BGT, BLE

10.51 BTEST — Bit test function

BTEST (I, POS)
BTEST(I,POS) returns logical .TRUE. if the bit at POS in I is set. The counting of the bits starts at 0.

Parameters

• I – The type shall be INTEGER.
• POS – The type shall be INTEGER.

Returns  The return value is of type LOGICAL

Standard:  Fortran 90 and later, has overloads that are GNU extensions
Class:  Elemental function

Syntax:

\[
\text{RESULT} = \text{BTEST}(I, \text{POS})
\]

Example:

```
program test_btest
  integer :: i = 32768 + 1024 + 64
  integer :: pos
  logical :: bool
  do pos=0,16
    bool = btest(i, pos)
    print *, pos, bool
  end do
end program test_btest
```
Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTEST(I,POS)</td>
<td>INTEGER I,POS</td>
<td>LOGICAL</td>
<td>Fortran 95 and later</td>
</tr>
<tr>
<td>BBTEST(I,POS)</td>
<td>INTEGER(1) I,POS</td>
<td>LOGICAL(1)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>BITEST(I,POS)</td>
<td>INTEGER(2) I,POS</td>
<td>LOGICAL(2)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>BJTEST(I,POS)</td>
<td>INTEGER(4) I,POS</td>
<td>LOGICAL(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>BKTEST(I,POS)</td>
<td>INTEGER(8) I,POS</td>
<td>LOGICAL(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

10.52 C_ASSOCIATED — Status of a C pointer

C_ASSOCIATED (c_ptr_1[, c_ptr_2])

C_ASSOCIATED(c_ptr_1[, c_ptr_2]) determines the status of the C pointer c_ptr_1 or if c_ptr_1 is associated with the target c_ptr_2.

Parameters

- c_ptr_1 – Scalar of the type C_PTR or C_FUNPTR.
- c_ptr_2 – (Optional) Scalar of the same type as c_ptr_1.

Returns The return value is of type LOGICAL; it is .false. if either c_ptr_1 is a C NULL pointer or if c_ptr_1 and c_ptr_2 point to different addresses.

Standard: Fortran 2003 and later

Class: Inquiry function

Syntax:

RESULT = C_ASSOCIATED(c_ptr_1[, c_ptr_2])

Example:

```fortran
subroutine association_test(a,b)
  use iso_c_binding, only: c_associated, c_loc, c_ptr
  implicit none
  real, pointer :: a
  type(c_ptr) :: b
  if(c_associated(b, c_loc(a))) &
    stop 'b and a do not point to same target'
end subroutine association_test
```

See also: C_LOC, C_FUNLOC
10.53 C_F_POINTER — Convert C into Fortran pointer

C_F_POINTER (CPTR, FPTR[, SHAPE])
C_F_POINTER (CPTR, FPTR[, SHAPE]) assigns the target of the C pointer CPTR to the Fortran pointer FPTR and specifies its shape.

Parameters

- CPTR – scalar of the type C_PTR. It is INTENT (IN).
- FPTR – pointer interoperable with cptr. It is INTENT (OUT).
- SHAPE – (Optional) Rank-one array of type INTEGER with INTENT (IN). It shall be present if and only if fptr is an array. The size must be equal to the rank of fptr.

Standard: Fortran 2003 and later

Class: Subroutine

Syntax:

CALL C_F_POINTER (CPTR, FPTR[, SHAPE])

Example:

```fortran
program main
  use iso_c_binding
  implicit none
  interface
    subroutine my_routine(p) bind(c,name='myC_func')
      import :: c_ptr
      type(c_ptr), intent(out) :: p
    end subroutine
  end interface
  type(c_ptr) :: cptr
  real, pointer :: a(:)
  call my_routine(cptr)
  call c_f_pointer(cptr, a, [12])
end program main
```

See also: C_LOC, C_F_PROCPOINTER

10.54 C_F_PROCPOINTER — Convert C into Fortran procedure pointer

C_F_PROCPOINTER (CPTR, FPTR)
C_F_PROCPOINTER (CPTR, FPTR) Assign the target of the C function pointer CPTR to the Fortran procedure pointer FPTR.

Parameters

- CPTR – scalar of the type C_FUNPTR. It is INTENT (IN).
- FPTR – procedure pointer interoperable with cptr. It is INTENT (OUT).

Standard: Fortran 2003 and later
**Class**: Subroutine

**Syntax**:

```fortran
CALL C_F_PROCPOINTER(cpvt, fptr)
```

**Example**:

```fortran
program main
  use iso_c_binding
  implicit none
  abstract interface
    function func(a)
      import :: c_float
      real(c_float), intent(in) :: a
      real(c_float) :: func
    end function
  end interface
  interface
    function getIterFunc() bind(c,name="getIterFunc")
      import :: c_funptr
      type(c_funptr) :: getIterFunc
    end function
  end interface
  type(c_funptr) :: cfunptr
  procedure(func), pointer :: myFunc
  cfunptr = getIterFunc()
  call c_f_procpointer(cfunptr, myFunc)
end program main
```

**See also**: C_LOC, C_F_POINTER

## 10.55 C_FUNLOC — Obtain the C address of a procedure

**C_FUNLOC** (x)

C_FUNLOC (x) determines the C address of the argument.

**Parameters** x – Interoperable function or pointer to such function.

**Returns** The return value is of type C_FUNPTR and contains the C address of the argument.

**Standard**: Fortran 2003 and later

**Class**: Inquiry function

**Syntax**:

```fortran
RESULT = C_FUNLOC(x)
```

**Example**:

```fortran
module x
  use iso_c_binding
  implicit none
  contains
```

(continues on next page)
See also: C_ASSOCIATED, C_LOC, C_F_POINTER, C_F_PROCPOINTER

10.56 C_LOC — Obtain the C address of an object

\texttt{C\_LOC(X)}

\texttt{C\_LOC(X)} determines the C address of the argument.

\textbf{Parameters} \texttt{X} – Shall have either the POINTER or TARGET attribute. It shall not be a coindexed object. It shall either be a variable with interoperable type and kind type parameters, or be a scalar, nonpolymorphic variable with no length type parameters.

\textbf{Returns} The return value is of type \texttt{C\_PTR} and contains the C address of the argument.

\textbf{Standard}: Fortran 2003 and later

\textbf{Class}: Inquiry function

\textbf{Syntax}:

\texttt{RESULT = C\_LOC(X)}

\textbf{Example}:

\begin{verbatim}
subroutine association_test(a,b)
  use iso_c_binding, only: c_associated, c_loc, c_ptr
  implicit none
  real, pointer :: a
  type(c_ptr) :: b
  if(c_associated(b, c_loc(a))) &
    stop 'b and a do not point to same target'
end subroutine association_test
\end{verbatim}

See also: C_ASSOCIATED, C_FUNLOC, C_F_POINTER, C_F_PROCPOINTER
10.57 C_SIZEOF — Size in bytes of an expression

\texttt{C\_SIZEOF}(X)

\texttt{C\_SIZEOF}(X) calculates the number of bytes of storage the expression \( X \) occupies.

\textbf{Parameters}  \( X \) – The argument shall be an interoperable data entity.

\textbf{Returns} The return value is of type integer and of the system-dependent kind \texttt{C\_SIZE\_T} (from the \texttt{ISO\_C\_BINDING} module). Its value is the number of bytes occupied by the argument. If the argument has the \texttt{POINTER} attribute, the number of bytes of the storage area pointed to is returned. If the argument is of a derived type with \texttt{POINTER} or \texttt{ALLOCATABLE} components, the return value does not account for the sizes of the data pointed to by these components.

\textit{Standard}: Fortran 2008

\textit{Class}: Inquiry function of the module \texttt{ISO\_C\_BINDING}

\textbf{Syntax}:

\begin{verbatim}
N = C\_SIZEOF(X)
\end{verbatim}

\textbf{Example}:

\begin{verbatim}
use iso_c_binding
integer(c_int) :: i
real(c_float) :: r, s(5)
print *, (c_sizeof(s)/c_sizeof(r) == 5)
end
\end{verbatim}

The example will print \texttt{T} unless you are using a platform where default REAL variables are unusually padded.

\textit{See also}: SIZEOF, STORAGE\_SIZE

10.58 CEILING — Integer ceiling function

\texttt{CEILING}(A)

\texttt{CEILING}(A) returns the least integer greater than or equal to \( A \).

\textbf{Parameters}

- \( A \) – The type shall be \texttt{REAL}.

- \texttt{KIND} – (Optional) An \texttt{INTEGER} initialization expression indicating the kind parameter of the result.

\textbf{Returns} The return value is of type \texttt{INTEGER(KIND)} if \texttt{KIND} is present and a default-kind \texttt{INTEGER} otherwise.

\textit{Standard}: Fortran 95 and later

\textit{Class}: Elemental function

\textbf{Syntax}:
RESULT = CEILING(A [, KIND])

Example:

```fortran
program test_ceiling
  real :: x = 63.29
  real :: y = -63.59
  print *, ceiling(x) ! returns 64
  print *, ceiling(y) ! returns -63
end program test_ceiling
```

See also: FLOOR, NINT

10.59 CHAR — Character conversion function

CHAR(I[, KIND])

CHAR(I [, KIND]) returns the character represented by the integer I.

Parameters

- I – The type shall be INTEGER.
- KIND – (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

Returns

The return value is of type CHARACTER(1)

Standard: Fortran 77 and later

Class: Elemental function

Syntax:

RESULT = CHAR(I [, KIND])

Example:

```fortran
program test_char
  integer :: i = 74
  character(1) :: c
  c = char(i)
  print *, i, c ! returns 'J'
end program test_char
```

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(I)</td>
<td>INTEGER</td>
<td>CHARACTER(LEN=1)</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

Note: See ICHAR for a discussion of converting between numerical values and formatted string representations.

See also: ACHAR, IACHAR, ICHAR
10.60 CHDIR — Change working directory

CHDIR()

Change current working directory to a specified path.

Parameters

• NAME – The type shall be CHARACTER of default kind and shall specify a valid path within the file system.

• STATUS – (Optional) INTEGER status flag of the default kind. Returns 0 on success, and a system specific and nonzero error code otherwise.

Standard: GNU extension

Class: Subroutine, function

Syntax:

CALL CHDIR(NAME [, STATUS])

Example:

PROGRAM test_chdir
  CHARACTER(len=255) :: path
  CALL getcwd(path)
  WRITE(*,*) TRIM(path)
  CALL chdir("/tmp")
  CALL getcwd(path)
  WRITE(*,*) TRIM(path)
END PROGRAM

See also: GETCWD

10.61 CHMOD — Change access permissions of files

CHMOD()

CHMOD changes the permissions of a file.

Parameters

• NAME – Scalar CHARACTER of default kind with the file name. Trailing blanks are ignored unless the character achar(0) is present, then all characters up to and excluding achar(0) are used as the file name.

• MODE – Scalar CHARACTER of default kind giving the file permission. MODE uses the same syntax as the chmod utility as defined by the POSIX standard. The argument shall either be a string of a nonnegative octal number or a symbolic mode.

• STATUS – (optional) scalar INTEGER, which is 0 on success and nonzero otherwise.

Returns In either syntax, STATUS is set to 0 on success and nonzero otherwise.

Standard: GNU extension

Class: Subroutine, function
Syntax:

```
CALL CHMOD(NAME, MODE[, STATUS])
STATUS = CHMOD(NAME, MODE)
```

**Example:** CHMOD as subroutine

```
program chmod_test
  implicit none
  integer :: status
  call chmod('test.dat','u+x',status)
  print *, 'Status: ', status
end program chmod_test
```

CHMOD as function:

```
program chmod_test
  implicit none
  integer :: status
  status = chmod('test.dat','u+x')
  print *, 'Status: ', status
end program chmod_test
```

### 10.62 CMPLX — Complex conversion function

**CMPLX**(X[, Y[, KIND]])

CMPLX(X[, Y[, KIND]]) returns a complex number where X is converted to the real component. If Y is present it is converted to the imaginary component. If Y is not present then the imaginary component is set to 0.0. If X is complex then Y must not be present.

**Parameters**

- **X** – The type may be INTEGER, REAL, or COMPLEX.
- **Y** – (Optional; only allowed if X is not COMPLEX.) May be INTEGER or REAL.
- **KIND** – (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

**Returns** The return value is of COMPLEX type, with a kind equal to KIND if it is specified. If KIND is not specified, the result is of the default COMPLEX kind, regardless of the kinds of X and Y.

**Standard:** Fortran 77 and later

**Class:** Elemental function

**Syntax:**

```
RESULT = CMPLX(X [, Y [, KIND]])
```

**Example:**

```
program test_cmplx
  integer :: i = 42
  real :: x = 3.14
  complex :: z
```

(continues on next page)
z = cmplx(i, x)
print *, z, cmplx(x)
end program test_cmplx

See also: COMPLEX

10.63 CO_BROADCAST — Copy a value to all images the current set of images

CO_BROADCAST() copies the value of argument A on the image with image index SOURCE_IMAGE to all images in the current team. A becomes defined as if by intrinsic assignment. If the execution was successful and STAT is present, it is assigned the value zero. If the execution failed, STAT gets assigned a nonzero value and, if present, ERRMSG gets assigned a value describing the occurred error.

Parameters

- **A** – INTENT(INOUT) argument; shall have the same dynamic type and type parameters on all images of the current team. If it is an array, it shall have the same shape on all images.
- **SOURCE_IMAGE** – a scalar integer expression. It shall have the same value on all images and refer to an image of the current team.
- **STAT** – (optional) a scalar integer variable
- **ERRMSG** – (optional) a scalar character variable

Standard: Technical Specification (TS) 18508 or later

Class: Collective subroutine

Syntax:

```fortran
CALL CO_BROADCAST(A, SOURCE_IMAGE [, STAT, ERRMSG])
```

Example:

```fortran
program test
integer :: val(3)
if (this_image() == 1) then
    val = [1, 5, 3]
end if
call co_broadcast (val, source_image=1)
print *, this_image, ":", val
end program test
```

See also: CO_MAX, CO_MIN, CO_SUM, CO_REDUCE
10.64 CO_MAX — Maximal value on the current set of images

CO_MAX()  
CO_MAX determines element-wise the maximal value of \( A \) on all images of the current team. If \( RESULT\_IMAGE \) is present, the maximum values are returned in \( A \) on the specified image only and the value of \( A \) on the other images become undefined. If \( RESULT\_IMAGE \) is not present, the value is returned on all images. If the execution was successful and \( STAT \) is present, it is assigned the value zero. If the execution failed, \( STAT \) gets assigned a nonzero value and, if present, \( ERRMSG \) gets assigned a value describing the occurred error.

Parameters

- \( A \) – shall be an integer, real or character variable, which has the same type and type parameters on all images of the team.
- \( RESULT\_IMAGE \) – (optional) a scalar integer expression; if present, it shall have the same value on all images and refer to an image of the current team.
- \( STAT \) – (optional) a scalar integer variable
- \( ERRMSG \) – (optional) a scalar character variable

Standard: Technical Specification (TS) 18508 or later

Class: Collective subroutine

Syntax:

\[
\text{CALL CO_MAX}(A [, RESULT\_IMAGE, STAT, ERRMSG])
\]

Example:

```fortran
program test
  integer :: val
  val = this_image()
  call co_max(val, result_image=1)
  if (this_image() == 1) then
    write(*,*) "Maximal value", val ! prints num_images()
  end if
end program test
```

See also: CO_MIN, CO_SUM, CO_REDUCE, CO_BROADCAST

10.65 CO_MIN — Minimal value on the current set of images

CO_MIN()  
CO_MIN determines element-wise the minimal value of \( A \) on all images of the current team. If \( RESULT\_IMAGE \) is present, the minimal values are returned in \( A \) on the specified image only and the value of \( A \) on the other images become undefined. If \( RESULT\_IMAGE \) is not present, the value is returned on all images. If the execution was successful and \( STAT \) is present, it is assigned the value zero. If the execution failed, \( STAT \) gets assigned a nonzero value and, if present, \( ERRMSG \) gets assigned a value describing the occurred error.

Parameters

- \( A \) – shall be an integer, real or character variable, which has the same type and type parameters on all images of the team.
• **RESULT_IMAGE** – (optional) a scalar integer expression; if present, it shall have the same value on all images and refer to an image of the current team.

• **STAT** – (optional) a scalar integer variable

• **ERRMSG** – (optional) a scalar character variable

**Standard**: Technical Specification (TS) 18508 or later

**Class**: Collective subroutine

**Syntax**:

```
CALL CO_MIN(A [, RESULT_IMAGE, STAT, ERRMSG])
```

**Example**:

```
program test
  integer :: val
  val = this_image()
  call co_min(val, result_image=1)
  if (this_image() == 1) then
    write(*,*) "Minimal value", val ! prints 1
  end if
end program test
```

**See also**: CO_MAX, CO_SUM, CO_REDUCE, CO_BROADCAST

### 10.66 CO_REDUCE — Reduction of values on the current set of images

**CO_REDUCE** ()

CO_REDUCE determines element-wise the reduction of the value of A on all images of the current team. The pure function passed as **OPERATOR** is used to pairwise reduce the values of A by passing either the value of A of different images or the result values of such a reduction as argument. If A is an array, the deduction is done element wise. If **RESULT_IMAGE** is present, the result values are returned in A on the specified image only and the value of A on the other images become undefined. If **RESULT_IMAGE** is not present, the value is returned on all images. If the execution was successful and **STAT** is present, it is assigned the value zero. If the execution failed, **STAT** gets assigned a nonzero value and, if present, **ERRMSG** gets assigned a value describing the occurred error.

**Parameters**

- **A** – is an INTENT(INOUT) argument and shall be nonpolymorphic. If it is allocatable, it shall be allocated; if it is a pointer, it shall be associated. A shall have the same type and type parameters on all images of the team; if it is an array, it shall have the same shape on all images.

- **OPERATOR** – pure function with two scalar nonallocatable arguments, which shall be nonpolymorphic and have the same type and type parameters as A. The function shall return a nonallocatable scalar of the same type and type parameters as A. The function shall be the same on all images and with regards to the arguments mathematically commutative and associative. Note that **OPERATOR** may not be an elemental function, unless it is an intrinsic function.

- **RESULT_IMAGE** – (optional) a scalar integer expression; if present, it shall have the same value on all images and refer to an image of the current team.
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

- **STAT** – (optional) a scalar integer variable
- **ERRMSG** – (optional) a scalar character variable

**Standard**: Technical Specification (TS) 18508 or later

**Class**: Collective subroutine

**Syntax**:

```fortran
CALL CO_REDUCE(A, OPERATOR, [, RESULT_IMAGE, STAT, ERRMSG])
```

**Example**:

```fortran
program test
  integer :: val
  val = this_image ()
  call co_reduce (val, result_image=1, operator=myprod)
  if (this_image() == 1) then
    write(*,*) "Product value", val ! prints num_images() factorial
  end if
contains
  pure function myprod(a, b)
    integer, value :: a, b
    integer :: myprod
    myprod = a * b
  end function myprod
end program test
```

**Note**: While the rules permit in principle an intrinsic function, none of the intrinsics in the standard fulfill the criteria of having a specific function, which takes two arguments of the same type and returning that type as result.

**See also**: CO_MIN, CO_MAX, CO_SUM, CO_BROADCAST

### 10.67 CO_SUM — Sum of values on the current set of images

**CO_SUM()**

CO_SUM sums up the values of each element of A on all images of the current team. If RESULT_IMAGE is present, the summed-up values are returned in A on the specified image only and the value of A on the other images become undefined. If RESULT_IMAGE is not present, the value is returned on all images. If the execution was successful and STAT is present, it is assigned the value zero. If the execution failed, STAT gets assigned a nonzero value and, if present, ERRMSG gets assigned a value describing the occurred error.

**Parameters**

- **A** – shall be an integer, real or complex variable, which has the same type and type parameters on all images of the team.
- **RESULT_IMAGE** – (optional) a scalar integer expression; if present, it shall have the same value on all images and refer to an image of the current team.
- **STAT** – (optional) a scalar integer variable
- **ERRMSG** – (optional) a scalar character variable

**Standard**: Technical Specification (TS) 18508 or later
**Class:** Collective subroutine

**Syntax:**

```
CALL CO_SUM(A [, RESULT_IMAGE, STAT, ERRMSG])
```

**Example:**

```
program test
    integer :: val
    val = this_image()
    call co_sum(val, result_image=1)
    if (this_image()==1) then
        write(*,*) "The sum is ", val ! prints (n**2 + n)/2, ! with n = num_images()
    end if
end program test
```

**See also:** CO_MAX, CO_MIN, CO_REDUCE, CO_BROADCAST

### 10.68 COMMAND_ARGUMENT_COUNT — Get number of command line arguments

**COMMAND_ARGUMENT_COUNT()**

COMMAND_ARGUMENT_COUNT returns the number of arguments passed on the command line when the containing program was invoked.

**Returns**  The return value is an INTEGER of default kind.

**Standard:** Fortran 2003 and later

**Class:** Inquiry function

**Syntax:**

```
RESULT = COMMAND_ARGUMENT_COUNT()
```

**Example:**

```
program test_command_argument_count
    integer :: count
    count = command_argument_count()
    print *, count
end program test_command_argument_count
```

**See also:** GET_COMMAND, GET_COMMAND_ARGUMENT
10.69 COMPILER_OPTIONS — Options passed to the compiler

COMPILER_OPTIONS()

COMPILER_OPTIONS returns a string with the options used for compiling.

Returns  The return value is a default-kind string with system-dependent length. It contains the compiler flags used to compile the file, which called the COMPILER_OPTIONS intrinsic.

Standard: Fortran 2008

Class: Inquiry function of the module ISO_FORTRAN_ENV

Syntax:

\[
\text{STR} = \text{COMPILER_OPTIONS()}
\]

Arguments: None

Example:

```fortran
use iso_fortran_env
print '(4a)', 'This file was compiled by ', &
    compiler_version(), ' using the options ', &
    compiler_options()
end
```

See also: COMPILER_VERSION, ISO_FORTRAN_ENV

10.70 COMPILER_VERSION — Compiler version string

COMPILER_VERSION()

COMPILER_VERSION returns a string with the name and the version of the compiler.

Returns  The return value is a default-kind string with system-dependent length. It contains the name of the compiler and its version number.

Standard: Fortran 2008

Class: Inquiry function of the module ISO_FORTRAN_ENV

Syntax:

\[
\text{STR} = \text{COMPILER_VERSION()}
\]

Arguments: None

Example:

```fortran
use iso_fortran_env
print '(4a)', 'This file was compiled by ', &
    compiler_version(), ' using the options ', &
    compiler_options()
end
```
See also: COMPILER_OPTIONS, ISO_FORTRAN_ENV

10.71 COMPLEX — Complex conversion function

COMPLEX (X, Y)

COMPLEX (X, Y) returns a complex number where X is converted to the real component and Y is converted to the imaginary component.

Parameters

• X – The type may be INTEGER or REAL.
• Y – The type may be INTEGER or REAL.

Returns If X and Y are both of INTEGER type, then the return value is of default COMPLEX type.

Standard: GNU extension

Class: Elemental function

Syntax:

RESULT = COMPLEX(X, Y)

Example:

```
program test_complex
    integer :: i = 42
    real :: x = 3.14
    print *, complex(i, x)
end program test_complex
```

See also: CMPLX

10.72 CONJG — Complex conjugate function

CONJG (Z)

CONJG (Z) returns the conjugate of Z. If Z is (x, y) then the result is (x, -y)

Parameters Z – The type shall be COMPLEX.

Returns The return value is of type COMPLEX.

Standard: Fortran 77 and later, has an overload that is a GNU extension

Class: Elemental function

Syntax:

\[ Z = \text{CONJG}(Z) \]

Example:
```fortran
program test_conjg
  complex :: z = (2.0, 3.0)
  complex(8) :: dz = (2.71_8, -3.14_8)
  z = conjg(z)
  print *, z
  dz = dconjg(dz)
  print *, dz
end program test_conjg
```

### Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCONJG(Z)</td>
<td>COMPLEX(8) Z</td>
<td>COMPLEX(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

## 10.73 COS — Cosine function

### COS(X)

COS(X) computes the cosine of X.

**Parameters** X – The type shall be REAL or COMPLEX.

**Returns** The return value is of the same type and kind as X. The real part of the result is in radians. If X is of the type REAL, the return value lies in the range -1 leq cos(x) leq 1.

**Standard**: Fortran 77 and later, has overloads that are GNU extensions

**Class**: Elemental function

**Syntax**:

```fortran
RESULT = COS(X)
```

**Example**:

```fortran
program test_cos
  real :: x = 0.0
  x = cos(x)
end program test_cos
```

### Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>COS(X)</td>
<td>REAL(4) X</td>
<td>REAL(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>DCOS(X)</td>
<td>REAL(8) X</td>
<td>REAL(8)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>CCOS(X)</td>
<td>COMPLEX(4) X</td>
<td>COMPLEX(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>ZCOS(X)</td>
<td>COMPLEX(8) X</td>
<td>COMPLEX(8)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>CDCOS(X)</td>
<td>COMPLEX(8) X</td>
<td>COMPLEX(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

**See also**: Inverse function: ACOS Degrees function: COSD
10.74 COSD — Cosine function, degrees

COSD(X)
COSD(X) computes the cosine of X in degrees.

Parameters X – The type shall be REAL or COMPLEX.

Returns The return value is of the same type and kind as X. The real part of the result is in degrees. If X is of the type REAL, the return value lies in the range -1 leq cosd(x) leq 1.

Standard: GNU extension, enabled with \texttt{-fdec-math}.

Class: Elemental function

Syntax:
\begin{verbatim}
RESULT = COSD(X)
\end{verbatim}

Example:
\begin{verbatim}
program test_cosd
  real :: x = 0.0
  x = cosd(x)
end program test_cosd
\end{verbatim}

Specific names:
\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
Name & Argument & Return type & Standard \\
\hline
COSD(X) & REAL(4) X & REAL(4) & GNU extension \\
DCOSD(X) & REAL(8) X & REAL(8) & GNU extension \\
CCOSD(X) & COMPLEX(4) X & COMPLEX(4) & GNU extension \\
ZCOSD(X) & COMPLEX(8) X & COMPLEX(8) & GNU extension \\
CDCOSD(X) & COMPLEX(8) X & COMPLEX(8) & GNU extension \\
\hline
\end{tabular}
\end{center}

See also: Inverse function: ACOSD Radians function: COS

10.75 COSH — Hyperbolic cosine function

COSH(X)
COSH(X) computes the hyperbolic cosine of X.

Parameters X – The type shall be REAL or COMPLEX.

Returns The return value has same type and kind as X. If X is complex, the imaginary part of the result is in radians. If X is REAL, the return value has a lower bound of one, \texttt{cosh(x) geq 1}.

Standard: Fortran 77 and later, for a complex argument Fortran 2008 or later

Class: Elemental function

Syntax:
\begin{verbatim}
X = COSH(X)
\end{verbatim}
Example:

```fortran
program test_cosh
  real(8) :: x = 1.0_8
  x = cosh(x)
end program test_cosh
```

**Specific names:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>COSH(X)</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>DCOSH(X)</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

**See also:** Inverse function: ACOSH

### 10.76 COTAN — Cotangent function

COTAN(X) computes the cotangent of X. Equivalent to COS(X) divided by SIN(X), or 1 / TAN(X).

**Parameters** X – The type shall be REAL or COMPLEX.

**Returns** The return value has same type and kind as X, and its value is in radians.

**Standard:** GNU extension, enabled with `-fdec-math`.

**Class:** Elemental function

**Syntax:**

```
RESULT = COTAN(X)
```

Example:

```fortran
program test_cotan
  real(8) :: x = 0.165_8
  x = cotan(x)
end program test_cotan
```

**Specific names:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>COTAN(X)</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>DCOTAN(X)</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

**See also:** Converse function: TAN Degrees function: COTAND
10.77 COTAND — Cotangent function, degrees

\[ \text{COTAND}(X) \]

COTAND \( (X) \) computes the cotangent of \( X \) in degrees. Equivalent to \( \text{COSD}(X) \) divided by \( \text{SIND}(X) \), or \( 1 / \text{TAND}(X) \).

**Parameters**
- \( X \) – The type shall be REAL or COMPLEX.

**Returns**
- The return value has the same type and kind as \( X \), and its value is in degrees.

**Standard:** GNU extension, enabled with \(-fdec-math\).

This function is for compatibility only and should be avoided in favor of standard constructs wherever possible.

**Class:** Elemental function

**Syntax:**

\[ \text{RESULT} = \text{COTAND}(X) \]

**Example:**

```fortran
program test_cotand
    real(8) :: x = 0.165_8
    x = cotand(x)
end program test_cotand
```

**Specific names:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>COTAND(X)</td>
<td>REAL(4) X</td>
<td>REAL(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>DCOTAND(X)</td>
<td>REAL(8) X</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

**See also:** Converse function: TAND Radians function: COTAN

10.78 COUNT — Count function

\[ \text{COUNT}() \]

Counts the number of \( \text{.TRUE.} \) elements in a logical \( \text{MASK} \), or, if the \( \text{DIM} \) argument is supplied, counts the number of elements along each row of the array in the \( \text{DIM} \) direction. If the array has zero size, or all of the elements of \( \text{MASK} \) are \( \text{.FALSE.} \), then the result is 0.

**Parameters**
- \( \text{MASK} \) – The type shall be LOGICAL.
- \( \text{DIM} \) – (Optional) The type shall be INTEGER.
- \( \text{KIND} \) – (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

**Returns**
- The return value is of type INTEGER and of kind \( \text{KIND} \). If \( \text{KIND} \) is absent, the return value is of default integer kind. If \( \text{DIM} \) is present, the result is an array with a rank one less than the rank of \( \text{ARRAY} \), and a size corresponding to the shape of \( \text{ARRAY} \) with the \( \text{DIM} \) dimension removed.
**Standard**: Fortran 90 and later, with \textit{KIND} argument Fortran 2003 and later

**Class**: Transformational function

**Syntax**: 

```
RESULT = COUNT(MASK [, DIM, KIND])
```

**Example**: 

```fortran
program test_count
   integer, dimension(2,3) :: a, b
   logical, dimension(2,3) :: mask
   a = reshape( (/ 1, 2, 3, 4, 5, 6 /), (/ 2, 3 /))
   b = reshape( (/ 0, 7, 3, 4, 5, 8 /), (/ 2, 3 /))
   print '(3i3)', a(1,:)
   print '(3i3)', a(2,:)
   print '*',
   print '(3i3)', b(1,:)
   print '(3i3)', b(2,:)
   print '*',
   mask = a.ne.b
   print '(3i3)', mask(1,:)
   print '(3i3)', mask(2,:)
   print '*',
   print '(3i3)', count(mask)
   print '*',
   print '(3i3)', count(mask, 1)
   print '*',
   print '(3i3)', count(mask, 2)
end program test_count
```

### 10.79 CPU\_TIME — CPU elapsed time in seconds

**CPU\_TIME()**

Returns a REAL value representing the elapsed CPU time in seconds. This is useful for testing segments of code to determine execution time.

- **Parameters** \textit{TIME} – The type shall be REAL with INTENT(OUT).
- **Returns** None

**Standard**: Fortran 95 and later

**Class**: Subroutine

**Syntax**: 

```
CALL CPU\_TIME\(\text{TIME}\)
```

**Example**: 

```fortran
program test_cpu_time
   real :: start, finish
   call cpu_time(start)
   ! put code to test here
end program test_cpu_time
```

(continues on next page)
call cpu_time(finish)
print '("Time = ",f6.3," seconds.")',finish-start
end program test_cpu_time

See also: SYSTEM_CLOCK, DATE_AND_TIME

10.80 CSHIFT — Circular shift elements of an array

CSHIFT(ARRAY, SHIFT[, DIM])

CSHIFT(ARRAY, SHIFT [, DIM]) performs a circular shift on elements of ARRAY along the dimension of DIM. If DIM is omitted it is taken to be 1. DIM is a scalar of type INTEGER in the range of 1 leq DIM leq n) where n is the rank of ARRAY. If the rank of ARRAY is one, then all elements of ARRAY are shifted by SHIFT places. If rank is greater than one, then all complete rank one sections of ARRAY along the given dimension are shifted. Elements shifted out one end of each rank one section are shifted back in the other end.

Parameters

• ARRAY – Shall be an array of any type.
• SHIFT – The type shall be INTEGER.
• DIM – The type shall be INTEGER.

Returns Returns an array of same type and rank as the ARRAY argument.

Standard: Fortran 90 and later

Class: Transformational function

Syntax:

RESULT = CSHIFT(ARRAY, SHIFT [, DIM])

Example:

program test_cshift
 integer, dimension(3,3) :: a
 a = reshape((/1, 2, 3, 4, 5, 6, 7, 8, 9/), (/3, 3/))
 print '(3i3)', a(1,:)
 print '(3i3)', a(2,:)
 print '(3i3)', a(3,:)
 a = cshift(a, SHIFT=/(1, 2, -1/), DIM=2)
 print *
 print '(3i3)', a(1,:)
 print '(3i3)', a(2,:)
 print '(3i3)', a(3,:)
end program test_cshift
10.81 CTIME — Convert a time into a string

CTIME()

CTIME converts a system time value, such as returned by TIME8, to a string. The output will be of the form Sat Aug 19 18:13:14 1995.

Parameters

- TIME – The type shall be of type INTEGER.
- RESULT – The type shall be of type CHARACTER and of default kind. It is an INTENT(OUT) argument. If the length of this variable is too short for the time and date string to fit completely, it will be blank on procedure return.

Returns

The converted date and time as a string.

Standard: GNU extension

Class: Subroutine, function

Syntax:

CALL CTIME(TIME, RESULT).
RESULT = CTIME(TIME).

Example:

```
program test_ctime
  integer(8) :: i
  character(len=30) :: date
  i = time8()

  ! Do something, main part of the program
  call ctime(i,date)
  print *, 'Program was started on ', date
end program test_ctime
```

See Also: DATE_AND_TIME, GMTIME, LTIME, TIME, TIME8

10.82 DATE_AND_TIME — Date and time subroutine

DATE_AND_TIME(DATE, TIME, ZONE, VALUES)

DATE_AND_TIME(DATE, TIME, ZONE, VALUES) gets the corresponding date and time information from the real-time system clock. DATE is INTENT(OUT) and has form ccymmd. TIME is INTENT(OUT) and has form hhmmss.sss. ZONE is INTENT(OUT) and has form (+)-hhmm, representing the difference with respect to Coordinated Universal Time (UTC). Unavailable time and date parameters return blanks.

Parameters

- DATE – (Optional) The type shall be CHARACTER (LEN=8) or larger, and of default kind.
- TIME – (Optional) The type shall be CHARACTER (LEN=10) or larger, and of default kind.
- ZONE – (Optional) The type shall be CHARACTER (LEN=5) or larger, and of default kind.
- VALUES – (Optional) The type shall be INTEGER(8).
**Returns** None

**Standard**: Fortran 90 and later

**Class**: Subroutine

**Syntax**:

```fortran
CALL DATE_AND_TIME([DATE, TIME, ZONE, VALUES])
```

**Example**:

```fortran
program test_time_and_date
  character(8) :: date
  character(10) :: time
  character(5) :: zone
  integer, dimension(8) :: values
  ! using keyword arguments
  call date_and_time(date, time, zone, values)
  call date_and_time(DATE=date, ZONE=zone)
  call date_and_time(TIME=time)
  call date_and_time(VALUES=values)
  print '(a,2x,a,2x,a)', date, time, zone
  print '(8i5)', values
end program test_time_and_date
```

**See also**: CPU_TIME, SYSTEM_CLOCK

## 10.83 DBLE — Double conversion function

**DBLE (A)**

DBLE (A) Converts A to double precision real type.

**Parameters** A – The type shall be INTEGER, REAL, or COMPLEX.

**Returns** The return value is of type double precision real.

**Standard**: Fortran 77 and later

**Class**: Elemental function

**Syntax**:

```fortran
RESULT = DBLE(A)
```

**Example**:

```fortran
program test_dble
  real :: x = 2.18
  integer :: i = 5
  complex :: z = (2.3, 1.14)
  print *, dble(x), dble(i), dble(z)
end program test_dble
```

**See also**: REAL
10.84 DCMPLX — Double complex conversion function

DCMPLX(X [, Y])

DCMPLX(X [, Y]) returns a double complex number where X is converted to the real component. If Y is present it is converted to the imaginary component. If Y is not present then the imaginary component is set to 0.0. If X is complex then Y must not be present.

Parameters

- X – The type may be INTEGER, REAL, or COMPLEX.
- Y – (Optional if X is not COMPLEX.) May be INTEGER or REAL.

Returns

The return value is of type COMPLEX(8)

Standard: GNU extension

Class: Elemental function

Syntax:

RESULT = DCMPLX(X [, Y])

Example:

```fortran
program test_dcmplx
    integer :: i = 42
    real :: x = 3.14
    complex :: z
    z = cmplx(i, x)
    print *, dcmplx(i)
    print *, dcmplx(x)
    print *, dcmplx(z)
    print *, dcmplx(x,i)
end program test_dcmplx
```

10.85 DIGITS — Significant binary digits function

DIGITS(X)

DIGITS(X) returns the number of significant binary digits of the internal model representation of X. For example, on a system using a 32-bit floating point representation, a default real number would likely return 24.

Parameters

- X – The type may be INTEGER or REAL.

Returns

The return value is of type INTEGER.

Standard: Fortran 90 and later

Class: Inquiry function

Syntax:

RESULT = DIGITS(X)

Example:
**10.86 DIM — Positive difference**

**DIM(X, Y)**  
DIM(X, Y) returns the difference X−Y if the result is positive; otherwise returns zero.

**Parameters**  
- **X** – The type shall be INTEGER or REAL  
- **Y** – The type shall be the same type and kind as X. (As a GNU extension, arguments of different kinds are permitted.)

**Returns**  
The return value is of type INTEGER or REAL. (As a GNU extension, kind is the largest kind of the actual arguments.)

**Standard**: Fortran 77 and later  
**Class**: Elemental function

**Syntax:**  
\[ \text{RESULT} = \text{DIM}(X, Y) \]

**Example:**

```fortran
program test_dim  
integer :: i  
real(8) :: x  
i = dim(4, 15)  
x = dim(4.345_8, 2.111_8)  
print *, i  
print *, x  
end program test_dim
```

**Specific names:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIM(X,Y)</td>
<td>REAL(4) X, Y</td>
<td>REAL(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>IDIM(X,Y)</td>
<td>INTEGER(4) X, Y</td>
<td>INTEGER(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>DDIM(X,Y)</td>
<td>REAL(8) X, Y</td>
<td>REAL(8)</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>
10.87 DOT_PRODUCT — Dot product function

**DOT_PRODUCT (VECTOR A, VECTOR B)**

DOT_PRODUCT (VECTOR A, VECTOR B) computes the dot product multiplication of two vectors VECTOR A and VECTOR B. The two vectors may be either numeric or logical and must be arrays of rank one and of equal size. If the vectors are INTEGER or REAL, the result is SUM (VECTOR A * VECTOR B). If the vectors are COMPLEX, the result is SUM (CONJG (VECTOR A) * VECTOR B). If the vectors are LOGICAL, the result is ANY (VECTOR A .AND. VECTOR B).

Parameters

- **VECTOR A** – The type shall be numeric or LOGICAL, rank 1.
- **VECTOR B** – The type shall be numeric if VECTOR A is of numeric type or LOGICAL if VECTOR A is of type LOGICAL. VECTOR B shall be a rank-one array.

Returns If the arguments are numeric, the return value is a scalar of numeric type, INTEGER, REAL, or COMPLEX. If the arguments are LOGICAL, the return value is .TRUE. or .FALSE..

**Standard**: Fortran 90 and later

**Class**: Transformational function

**Syntax**:

\[
\text{RESULT} = \text{DOT}
\text{PRODUCT(VECTOR A, VECTOR B)}
\]

**Example**:

```fortran
program test_dot_prod
    integer, dimension(3) :: a, b
    a = (/ 1, 2, 3 /)
    b = (/ 4, 5, 6 /)
    print '(3i3)', a
    print *,
    print '(3i3)', b
    print *,
    print *, dot_product(a,b)
end program test_dot_prod
```

10.88 DPROD — Double product function

**DPROD (X, Y)**

DPROD (X, Y) returns the product X * Y.

Parameters

- **X** – The type shall be REAL.
- **Y** – The type shall be REAL.

**Returns** The return value is of type REAL (8).

**Standard**: Fortran 77 and later

**Class**: Elemental function
Syntax:

\[
\text{RESULT} = \text{DPROD}(X, Y)
\]

Example:

```fortran
program test_dprod
  real :: x = 5.2
  real :: y = 2.3
  real(8) :: d
  d = dprod(x, y)
  print *, d
end program test_dprod
```

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPROD(X,Y)</td>
<td>REAL(4) X, Y</td>
<td>REAL(8)</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

10.89 DREAL — Double real part function

DREAL(Z)

DREAL(Z) returns the real part of complex variable Z.

Parameters A – The type shall be COMPLEX(8).

Returns The return value is of type REAL(8).

Standard: GNU extension

Class: Elemental function

Syntax:

\[
\text{RESULT} = \text{DREAL}(A)
\]

Example:

```fortran
program test_dreal
  complex(8) :: z = (1.3_8, 7.2_8)
  print *, dreal(z)
end program test_dreal
```

See also: AIMAG
10.90 DSHIFTL — Combined left shift

DSHIFTL(I, J, SHIFT)

DSHIFTL(I, J, SHIFT) combines bits of I and J. The rightmost SHIFT bits of the result are the leftmost SHIFT bits of J, and the remaining bits are the rightmost bits of I.

Parameters

- I – Shall be of type INTEGER or a BOZ constant.
- J – Shall be of type INTEGER or a BOZ constant. If both I and J have integer type, then they shall have the same kind type parameter. I and J shall not both be BOZ constants.
- SHIFT – Shall be of type INTEGER. It shall be nonnegative. If I is not a BOZ constant, then SHIFT shall be less than or equal to BIT_SIZE(I); otherwise, SHIFT shall be less than or equal to BIT_SIZE(J).

Returns If either I or J is a BOZ constant, it is first converted as if by the intrinsic function INT to an integer type with the kind type parameter of the other.

Standard: Fortran 2008 and later

Class: Elemental function

Syntax:

```
RESULT = DSHIFTL(I, J, SHIFT)
```

See also: DSHIFTR

10.91 DSHIFTR — Combined right shift

DSHIFTR(I, J, SHIFT)

DSHIFTR(I, J, SHIFT) combines bits of I and J. The leftmost SHIFT bits of the result are the rightmost SHIFT bits of I, and the remaining bits are the leftmost bits of J.

Parameters

- I – Shall be of type INTEGER or a BOZ constant.
- J – Shall be of type INTEGER or a BOZ constant. If both I and J have integer type, then they shall have the same kind type parameter. I and J shall not both be BOZ constants.
- SHIFT – Shall be of type INTEGER. It shall be nonnegative. If I is not a BOZ constant, then SHIFT shall be less than or equal to BIT_SIZE(I); otherwise, SHIFT shall be less than or equal to BIT_SIZE(J).

Returns If either I or J is a BOZ constant, it is first converted as if by the intrinsic function INT to an integer type with the kind type parameter of the other.

Standard: Fortran 2008 and later

Class: Elemental function

Syntax:
RESULT = DSHIFTR(I, J, SHIFT)

See also: DSHIFTL

10.92 DTIME — Execution time subroutine (or function)

DTIME (VALUES, TIME)

DTIME (VALUES, TIME) initially returns the number of seconds of runtime since the start of the process’s execution in TIME. VALUES returns the user and system components of this time in VALUES (1) and VALUES (2) respectively. TIME is equal to VALUES (1) + VALUES (2).

Parameters

• VALUES – The type shall be REAL (4), DIMENSION (2).
• TIME – The type shall be REAL (4).

Returns Elapsed time in seconds since the last invocation or since the start of program execution if not called before.

Standard: GNU extension

Class: Subroutine, function

Syntax:

CALL DTIME(VALUES, TIME).
TIME = DTIME(VALUES), (not recommended).

Example:

program test_dtime
integer(8) :: i, j
real, dimension(2) :: tarray
real :: result
call dtime(tarray, result)
print *, result
print *, tarray(1)
print *, tarray(2)
do i=1,100000000 ! Just a delay
j = i * i - i
end do
call dtime(tarray, result)
print *, result
print *, tarray(1)
print *, tarray(2)
end program test_dtime

See also: CPU_TIME
10.93 EOSHIFT — End-off shift elements of an array

EOSHIFT (ARRAY, SHIFT [, BOUNDARY, DIM])

EOSHIFT (ARRAY, SHIFT [, BOUNDARY, DIM]) performs an end-off shift on elements of ARRAY along the dimension of DIM. If DIM is omitted it is taken to be 1. DIM is a scalar of type INTEGER in the range of 1 leq DIM leq n) where n is the rank of ARRAY. If the rank of ARRAY is one, then all elements of ARRAY are shifted by SHIFT places. If rank is greater than one, then all complete rank one sections of ARRAY along the given dimension are shifted. Elements shifted out one end of each rank one section are dropped. If BOUNDARY is present then the corresponding value of from BOUNDARY is copied back in the other end. If BOUNDARY is not present then the following are copied in depending on the type of ARRAY.

Parameters
- Array – May be any type, not scalar.
- SHIFT – The type shall be INTEGER.
- BOUNDARY – Same type as ARRAY.
- DIM – The type shall be INTEGER.

Returns Returns an array of same type and rank as the ARRAY argument.

Standard: Fortran 90 and later

Class: Transformational function

Syntax:
RESULT = EOSHIFT(ARRAY, SHIFT [, BOUNDARY, DIM])

Example:

```fortran
program test_eoshift
  integer, dimension(3,3) :: a
  a = reshape(/1, 2, 3, 4, 5, 6, 7, 8, 9/), (/ 3, 3 /)
  print '(3i3)', a(1,:)
  print '(3i3)', a(2,:)
  print '(3i3)', a(3,:)
  a = EOSHIFT(a, SHIFT=/1, 2, 1/), BOUNDARY=-5, DIM=2
  print *,
  print '(3i3)', a(1,:)
  print '(3i3)', a(2,:)
  print '(3i3)', a(3,:)
end program test_eoshift
```

10.94 EPSILON — Epsilon function

EPSILON (X)

EPSILON (X) returns the smallest number E of the same kind as X such that 1 + E > 1.

Parameters X – The type shall be REAL.

Returns The return value is of same type as the argument.

Standard: Fortran 90 and later
**Class:** Inquiry function

**Syntax:**

\[
\text{RESULT} = \text{EPSILON}(X)
\]

**Example:**

```fortran
program test_epsilon
  real :: x = 3.143
  real(8) :: y = 2.33
  print *, EPSILON(x)
  print *, EPSILON(y)
end program test_epsilon
```

### 10.95 ERF — Error function

\( \text{ERF}(X) \)

\( \text{ERF}(X) \) computes the error function of \( X \).

**Parameters** \( X \) – The type shall be REAL.

**Returns** The return value is of type REAL, of the same kind as \( X \) and lies in the range \(-1 \leq \text{erf}(x) \leq 1\).

**Standard:** Fortran 2008 and later

**Class:** Elemental function

**Syntax:**

\[
\text{RESULT} = \text{ERF}(X)
\]

**Example:**

```fortran
program test_erf
  real(8) :: x = 0.17_8
  x = erf(x)
end program test_erf
```

**Specific names:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>DERF(X)</td>
<td>REAL(8)</td>
<td>X</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>
10.96 ERFC — Error function

ERFC (X)

ERFC (X) computes the complementary error function of X.

Parameters X – The type shall be REAL.

Returns The return value is of type REAL and of the same kind as X. It lies in the range 0 leq erfc (x) leq 2.

Standard: Fortran 2008 and later

Class: Elemental function

Syntax:

RESULT = ERFC(X)

Example:

program test_erfc
  real(8) :: x = 0.17_8
  x = erfc(x)
end program test_erfc

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>DERFC(X)</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

10.97 ERFC_SCALED — Error function

ERFC_SCALED (X)

ERFC_SCALED (X) computes the exponentially-scaled complementary error function of X.

Parameters X – The type shall be REAL.

Returns The return value is of type REAL and of the same kind as X.

Standard: Fortran 2008 and later

Class: Elemental function

Syntax:

RESULT = ERFC_SCALED(X)

Example:

program test_erfc_scaled
  real(8) :: x = 0.17_8
  x = erfc_scaled(x)
end program test_erfc_scaled
10.98 ETIME — Execution time subroutine (or function)

ETIME (VALUES, TIME)
ETIME (VALUES, TIME) returns the number of seconds of runtime since the start of the process’s execution in TIME. VALUES returns the user and system components of this time in VALUES(1) and VALUES(2) respectively. TIME is equal to VALUES(1) + VALUES(2).

Parameters

• VALUES – The type shall be REAL(4), DIMENSION(2).
• TIME – The type shall be REAL(4).

Returns
Elapsed time in seconds since the start of program execution.

Standard: GNU extension

Class: Subroutine, function

Syntax:

CALL ETIME (VALUES, TIME).
TIME = ETIME (VALUES), (not recommended).

Example:

```
program test_etime
  integer(8) :: i, j
  real, dimension(2) :: tarray
  real :: result
  call ETIME (tarray, result)
  print *, result
  print *, tarray(1)
  print *, tarray(2)
  do i=1,100000000  ! Just a delay
    j = i * i - i
  end do
  call ETIME (tarray, result)
  print *, result
  print *, tarray(1)
  print *, tarray(2)
end program test_etime
```

See also: CPU_TIME

10.99 EVENT_QUERY — Query whether a coarray event has occurred

EVENT_QUERY ()

EVENT_QUERY assigns the number of events to COUNT which have been posted to the EVENT variable and not yet been removed by calling EVENT WAIT. When STAT is present and the invocation was successful, it is assigned the value 0. If it is present and the invocation has failed, it is assigned a positive value and COUNT is assigned the value -1.

Parameters
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

• **EVENT** – (intent(IN)) Scalar of type `EVENT_TYPE`, defined in `ISO_FORTRAN_ENV`; shall not be coindexed.
• **COUNT** – (intent(out)) Scalar integer with at least the precision of default integer.
• **STAT** – (optional) Scalar default-kind integer variable.

**Standard:** TS 18508 or later

**Class:** subroutine

**Syntax:**

```fortran
CALL EVENT_QUERY (EVENT, COUNT [, STAT])
```

**Example:**

```fortran
program atomic
  use iso_fortran_env
  implicit none
  type(event_type) :: event_value_has_been_set[*]
  integer :: cnt
  if (this_image() == 1) then
    call event_query (event_value_has_been_set, cnt)
    if (cnt > 0) write(*,*) "Value has been set"
  elseif (this_image() == 2) then
    event post (event_value_has_been_set[1])
  end if
end program atomic
```

### 10.100 EXECUTE_COMMAND_LINE — Execute a shell command

**EXECUTE_COMMAND_LINE()**

`EXECUTE_COMMAND_LINE` runs a shell command, synchronously or asynchronously.

**Parameters**

- **COMMAND** – Shall be a default CHARACTER scalar.
- **WAIT** – (Optional) Shall be a default LOGICAL scalar.
- **EXITSTAT** – (Optional) Shall be an INTEGER of the default kind.
- **CMDSTAT** – (Optional) Shall be an INTEGER of the default kind.
- **CMDMSG** – (Optional) Shall be a CHARACTER scalar of the default kind.

**Standard:** Fortran 2008 and later

**Class:** Subroutine

**Syntax:**

```fortran
CALL EXECUTE_COMMAND_LINE(COMMAND [, WAIT, EXITSTAT, CMDSTAT, CMDMSG ])
```

**Example:**
program test_exec
    integer :: i
    
    call execute_command_line ("external_prog.exe", exitstat=i)
    print *, "Exit status of external_prog.exe was ", i
    
    call execute_command_line ("reindex_files.exe", wait=.false.)
    print *, "Now reindexing files in the background"
end program test_exec

Note: Because this intrinsic is implemented in terms of the system function call, its behavior with respect to signaling is processor dependent. In particular, on POSIX-compliant systems, the SIGINT and SIGQUIT signals will be ignored, and the SIGCHLD will be blocked. As such, if the parent process is terminated, the child process might not be terminated alongside.

See also: SYSTEM

10.101 EXIT — Exit the program with status.

EXIT ()
EXIT causes immediate termination of the program with status. If status is omitted it returns the canonical success for the system. All Fortran I/O units are closed.

Parameters STATUS — Shall be an INTEGER of the default kind.

Returns STATUS is passed to the parent process on exit.

Standard: GNU extension

Class: Subroutine

Syntax:

CALL EXIT([STATUS])

Example:

program test_exit
    integer :: STATUS = 0
    print *, 'This program is going to exit.'
    call EXIT(STATUS)
end program test_exit

See also: ABORT, KILL
10.102 EXP — Exponential function

EXP (X)
EXP (X) computes the base e exponential of X.

Parameters X – The type shall be REAL or COMPLEX.

Returns The return value has same type and kind as X.

Standard: Fortran 77 and later, has overloads that are GNU extensions

Class: Elemental function

Syntax:

RESULT = EXP (X)

Example:

```
program test_exp
  real :: x = 1.0
  x = exp(x)
end program test_exp
```

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXP (X)</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>DEXP (X)</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>CEXP (X)</td>
<td>COMPLEX(4)</td>
<td>COMPLEX(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>ZEXP (X)</td>
<td>COMPLEX(8)</td>
<td>COMPLEX(8)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>CDEXP (X)</td>
<td>COMPLEX(8)</td>
<td>COMPLEX(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

10.103 EXPONENT — Exponent function

EXPONENT (X)
EXPONENT (X) returns the value of the exponent part of X. If X is zero the value returned is zero.

Parameters X – The type shall be REAL.

Returns The return value is of type default INTEGER.

Standard: Fortran 90 and later

Class: Elemental function

Syntax:

RESULT = EXPONENT (X)

Example:
program test_exponent
  real :: x = 1.0
  integer :: i
  i = exponent(x)
  print *, i
  print *, exponent(0.0)
end program test_exponent

10.104 EXTENDS_TYPE_OF — Query dynamic type for extension

EXTENDS_TYPE_OF ()
Query dynamic type for extension.

Parameters

- A – Shall be an object of extensible declared type or unlimited polymorphic.
- MOLD – Shall be an object of extensible declared type or unlimited polymorphic.

Returns The return value is a scalar of type default logical. It is true if and only if the dynamic type of A is an extension type of the dynamic type of MOLD.

Standard: Fortran 2003 and later
Class: Inquiry function
Syntax:

RESULT = EXTENDS_TYPE_OF(A, MOLD)

See also: SAME_TYPE_AS

10.105 FDATE — Get the current time as a string

FDATE (DATE)
FDATE (DATE) returns the current date (using the same format as CTIME) in DATE. It is equivalent to CALL CTIME(DATE, TIME()).

Parameters DATE – The type shall be of type CHARACTER of the default kind. It is an INTENT (OUT) argument. If the length of this variable is too short for the date and time string to fit completely, it will be blank on procedure return.

Returns The current date and time as a string.

Standard: GNU extension
Class: Subroutine, function
Syntax:

CALL FDATE (DATE).
DATE = FDATE ()

Example:
program test_fdate
    integer(8) :: i, j
    character(len=30) :: date
    call fdate(date)
    print *, 'Program started on ', date
    do i = 1, 100000000 ! Just a delay
        j = i * i - i
    end do
    call fdate(date)
    print *, 'Program ended on ', date
end program test_fdate

See also: DATE_AND_TIME, CTIME

10.106 FGET — Read a single character in stream mode from stdin

FGET ()
A single character in stream mode from stdin by bypassing normal formatted output. Stream I/O should not be mixed with normal record-oriented (formatted or unformatted) I/O on the same unit; the results are unpredictable.

Parameters
- C – The type shall be CHARACTER and of default kind.
- STATUS – (Optional) status flag of type INTEGER. Returns 0 on success, -1 on end-of-file, and a system specific positive error code otherwise.

Standard: GNU extension
Class: Subroutine, function
Syntax:
CALL FGET(C [, STATUS])
STATUS = FGET(C)

Example:

PROGRAM test_fget
    INTEGER, PARAMETER :: strlen = 100
    INTEGER :: status, i = 1
    CHARACTER(len=strlen) :: str = ""
    WRITE (*,*) 'Enter text:'
    DO
        CALL fget(str(i:i), status)
        if (status /= 0 .OR. i > strlen) exit
        i = i + 1
    END DO
    WRITE (*,*) TRIM(str)
END PROGRAM

See also: FGETC, FPUT, FPUTC
10.107 FGETC — Read a single character in stream mode

FGETC()
Read a single character in stream mode by bypassing normal formatted output. Stream I/O should not be mixed with normal record-oriented (formatted or unformatted) I/O on the same unit; the results are unpredictable.

Parameters
- UNIT – The type shall be INTEGER.
- C – The type shall be CHARACTER and of default kind.
- STATUS – (Optional) status flag of type INTEGER. Returns 0 on success, -1 on end-of-file and a system specific positive error code otherwise.

Standard: GNU extension
Class: Subroutine, function

Syntax:

```fortran
CALL FGETC(UNIT, C [, STATUS])
STATUS = FGETC(UNIT, C)
```

Example:

```fortran
PROGRAM test_fgetc
  INTEGER :: fd = 42, status
  CHARACTER :: c

  OPEN(UNIT=fd, FILE="/etc/passwd", ACTION="READ", STATUS = "OLD")
  DO
    CALL fgetc(fd, c, status)
    IF (status /= 0) EXIT
    call fputc(c)
  END DO
  CLOSE(UNIT=fd)
END PROGRAM
```

See also: FGET, FPUT, FPUTC

10.108 FINDLOC — Search an array for a value

FINDLOC()
Determines the location of the element in the array with the value given in the VALUE argument, or, if the DIM argument is supplied, determines the locations of the elements equal to the VALUE argument element along each row of the array in the DIM direction. If MASK is present, only the elements for which MASK is .TRUE. are considered. If more than one element in the array has the value VALUE, the location returned is that of the first such element in array element order if the BACK is not present or if it is .FALSE. If BACK is true, the location returned is that of the last such element. If the array has zero size, or all of the elements of MASK are .FALSE., then the result is an array of zeroes. Similarly, if DIM is supplied and all of the elements of MASK along a given row are zero, the result value for that row is zero.

Parameters
- ARRAY – Shall be an array of intrinsic type.
• **VALUE** – A scalar of intrinsic type which is in type conformance with **ARRAY**.

• **DIM** – (Optional) Shall be a scalar of type **INTEGER**, with a value between one and the rank of **ARRAY**, inclusive. It may not be an optional dummy argument.

• **MASK** – (Optional) Shall be of type **LOGICAL**, and conformable with **ARRAY**.

• **KIND** – (Optional) An **INTEGER** initialization expression indicating the kind parameter of the result.

• **BACK** – (Optional) A scalar of type **LOGICAL**.

**Returns** If **DIM** is absent, the result is a rank-one array with a length equal to the rank of **ARRAY**. If **DIM** is present, the result is an array with a rank one less than the rank of **ARRAY**. and a size corresponding to the size of **ARRAY** with the **DIM** dimension removed. If **DIM** is present and **ARRAY** has a rank of one, the result is a scalar. If the optional argument **KIND** is present, the result is an integer of kind **KIND**, otherwise it is of default kind.

**Standard**: Fortran 2008 and later.

**Class**: Transformational function

**Syntax**:

```
RESULT = FINDLOC(ARRAY, VALUE, DIM [, MASK] [,KIND] [,BACK])
RESULT = FINDLOC(ARRAY, VALUE, [, MASK] [,KIND] [,BACK])
```

**See also**: MAXLOC, MINLOC

### 10.109 FLOOR — Integer floor function

**FLOOR (A)**

**FLOOR (A)** returns the greatest integer less than or equal to **X**.

**Parameters**

• **A** – The type shall be **REAL**.

• **KIND** – (Optional) An **INTEGER** initialization expression indicating the kind parameter of the result.

**Returns** The return value is of type **INTEGER(KIND)** if **KIND** is present and of default-kind **INTEGER** otherwise.

**Standard**: Fortran 95 and later

**Class**: Elemental function

**Syntax**:

```
RESULT = FLOOR(A [, KIND])
```

**Example**:

```
program test_floor
  real :: x = 63.29
  real :: y = -63.59
```

(continues on next page)
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

(continued from previous page)

print *, floor(x) ! returns 63
print *, floor(y) ! returns -64
end program test_floor

See also: CEILING, NINT

10.110 FLUSH — Flush I/O unit(s)

FLUSH()
Flushes Fortran unit(s) currently open for output. Without the optional argument, all units are flushed, otherwise just the unit specified.

Parameters UNIT — (Optional) The type shall be INTEGER.

Standard: GNU extension

Class: Subroutine

Syntax:

CALL FLUSH(UNIT)

Note: Beginning with the Fortran 2003 standard, there is a FLUSH statement that should be preferred over the FLUSH intrinsic.

The FLUSH intrinsic and the Fortran 2003 FLUSH statement have identical effect: they flush the runtime library's I/O buffer so that the data becomes visible to other processes. This does not guarantee that the data is committed to disk.

On POSIX systems, you can request that all data is transferred to the storage device by calling the fsync function, with the POSIX file descriptor of the I/O unit as argument (retrieved with GNU intrinsic FNUM). The following example shows how:

! Declare the interface for POSIX fsync function
interface
  function fsync (fd) bind(c,name="fsync")
    use iso_c_binding, only: c_int
    integer(c_int), value :: fd
    integer(c_int) :: fsync
  end function fsync
end interface

! Variable declaration
integer :: ret

! Opening unit 10
open (10, file="foo")

! ...
! Perform I/O on unit 10
! ...
! Flush and sync

(continues on next page)
flush(10)
ret = fsync(fnum(10))

! Handle possible error
if (ret /= 0) stop "Error calling FSYNC"

10.111 FNUM — File number function

\[
\text{FNUM(UNIT)}
\]

FNUM(UNIT) returns the POSIX file descriptor number corresponding to the open Fortran I/O unit UNIT.

Parameters
UNIT – The type shall be INTEGER.

Returns
The return value is of type INTEGER

Standard: GNU extension
Class: Function
Syntax:
\[
\text{RESULT} = \text{FNUM(UNIT)}
\]

Example:

```
program test_fnum
  integer :: i
  open (unit=10, status = "scratch")
  i = fnum(10)
  print *, i
  close (10)
end program test_fnum
```

10.112 FPUT — Write a single character in stream mode to stdout

\[
\text{FPUT()}
\]

Write a single character in stream mode to stdout by bypassing normal formatted output. Stream I/O should not be mixed with normal record-oriented (formatted or unformatted) I/O on the same unit; the results are unpredictable.

Parameters

- \text{C} – The type shall be CHARACTER and of default kind.
- \text{STATUS} – (Optional) status flag of type INTEGER. Returns 0 on success, -1 on end-of-file and a system specific positive error code otherwise.

Standard: GNU extension
Class: Subroutine, function
Syntax:
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

CALL FPUT(C [, STATUS])
STATUS = FPUT(C)

Example:

PROGRAM test_fput
  CHARACTER(len=10) :: str = "gfortran"
  INTEGER :: i
  DO i = 1, len_trim(str)
    CALL fput(str(i:i))
  END DO
END PROGRAM

See also: FPUT, FGET, FGETC

10.113 FPUTC — Write a single character in stream mode

FPUTC()
Write a single character in stream mode by bypassing normal formatted output. Stream I/O should not be mixed with normal record-oriented (formatted or unformatted) I/O on the same unit; the results are unpredictable.

Parameters

• UNIT – The type shall be INTEGER.
• C – The type shall be CHARACTER and of default kind.
• STATUS – (Optional) status flag of type INTEGER. Returns 0 on success, -1 on end-of-file and a system specific positive error code otherwise.

Standard: GNU extension

Class: Subroutine, function

Syntax:

CALL FPUTC(UNIT, C [, STATUS])
STATUS = FPUTC(UNIT, C)

Example:

PROGRAM test_fputc
  CHARACTER(len=10) :: str = "gfortran"
  INTEGER :: fd = 42, i
  OPEN(UNIT = fd, FILE = "out", ACTION = "WRITE", STATUS="NEW")
  DO i = 1, len_trim(str)
    CALL fputc(fd, str(i:i))
  END DO
  CLOSE(fd)
END PROGRAM

See also: FPUT, FGET, FGETC
10.114 FRACTION — Fractional part of the model representation

FRACTION (X)
FRACTION (X) returns the fractional part of the model representation of X.

Parameters X – The type of the argument shall be a REAL.

Returns The return value is of the same type and kind as the argument. The fractional part of the model representation of X is returned; it is \( X \times \text{RADIX}(X)^{-\text{EXPONENT}(X)} \).

Standard: Fortran 90 and later

Class: Elemental function

Syntax:

\[ Y = \text{FRACTION}(X) \]

Example:

```
program test_fraction
  real :: x
  x = 178.1387e-4
  print *, fraction(x), x * radix(x)**(-exponent(x))
end program test_fraction
```

10.115 FREE — Frees memory

FREE ()
Frees memory previously allocated by MALLOC. The FREE intrinsic is an extension intended to be used with Cray pointers, and is provided in GNU Fortran to allow user to compile legacy code. For new code using Fortran 95 pointers, the memory de-allocation intrinsic is DEALLOCATE.

Parameters PTR – The type shall be INTEGER. It represents the location of the memory that should be de-allocated.

Returns None

Standard: GNU extension

Class: Subroutine

Syntax:

\text{CALL} \ FREE(PTR)

Example: See MALLOC for an example.

See also: MALLOC
10.116 FSEEK — Low level file positioning subroutine

FSEEK()

Moves UNIT to the specified OFFSET. If WHENCE is set to 0, the OFFSET is taken as an absolute value SEEK_SET, if set to 1, OFFSET is taken to be relative to the current position SEEK_CUR, and if set to 2 relative to the end of the file SEEK_END. On error, STATUS is set to a nonzero value. If STATUS the seek fails silently.

Parameters

- UNIT – Shall be a scalar of type INTEGER.
- OFFSET – Shall be a scalar of type INTEGER.
- WHENCE – Shall be a scalar of type INTEGER. Its value shall be either 0, 1 or 2.
- STATUS – (Optional) shall be a scalar of type INTEGER(4).

Standard: GNU extension

Class: Subroutine

Syntax:

CALL FSEEK(UNIT, OFFSET, WHENCE[, STATUS])

Example:

PROGRAM test_fseek
INTEGER, PARAMETER :: SEEK_SET = 0, SEEK_CUR = 1, SEEK_END = 2
INTEGER :: fd, offset, ierr

ierr = 0
offset = 5
fd = 10

OPEN(UNIT=fd, FILE="fseek.test")
CALL FSEEK(fd, offset, SEEK_SET, ierr) ! move to OFFSET
print *, FTELL(fd), ierr

CALL FSEEK(fd, 0, SEEK_END, ierr) ! move to end
print *, FTELL(fd), ierr

CALL FSEEK(fd, 0, SEEK_SET, ierr) ! move to beginning
print *, FTELL(fd), ierr

CLOSE(UNIT=fd)
END PROGRAM

See also: FTELL
10.117 FSTAT — Get file status

FSTAT()

FSTAT is identical to STAT, except that information about an already opened file is obtained.

Parameters

- **UNIT** – An open I/O unit number of type INTEGER.
- **VALUES** – The type shall be INTEGER(4), DIMENSION(13).
- **STATUS** – (Optional) status flag of type INTEGER(4). Returns 0 on success and a system specific error code otherwise.

**Standard**: GNU extension

**Class**: Subroutine, function

**Syntax**:

```
CALL FSTAT(UNIT, VALUES [, STATUS])
STATUS = FSTAT(UNIT, VALUES)
```

**Example**: See STAT for an example.

**See also**: To stat a link: LSTAT To stat a file: STAT

10.118 FTELL — Current stream position

FTELL()

Retrieves the current position within an open file.

Parameters

- **OFFSET** – Shall of type INTEGER.
- **UNIT** – Shall of type INTEGER.

**Returns** In either syntax, OFFSET is set to the current offset of unit number UNIT, or to -1 if the unit is not currently open.

**Standard**: GNU extension

**Class**: Subroutine, function

**Syntax**:

```
CALL FTELL(UNIT, OFFSET)
OFFSET = FTELL(UNIT)
```

**Example**:

```
PROGRAM test_ftell
   INTEGER :: i
   OPEN(10, FILE="temp.dat")
   CALL ftell(10,i)
```

(continues on next page)
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

(continued from previous page)

```fortran
WRITE(*,*) i
END PROGRAM
```

See also: FSEEK

### 10.119 GAMMA — Gamma function

**GAMMA (X)**

GAMMA (X) computes Gamma (Gamma) of X. For positive, integer values of X the Gamma function simplifies to the factorial function Gamma(x)=(x-1)!

**Parameters** X – Shall be of type REAL and neither zero nor a negative integer.

**Returns** The return value is of type REAL of the same kind as X.

**Standard:** Fortran 2008 and later

**Class:** Elemental function

**Syntax:**

```fortran
X = GAMMA(X)
```

**Example:**

```fortran
program test_gamma
    real :: x = 1.0
    x = gamma(x) ! returns 1.0
end program test_gamma
```

**Specific names:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGAMMA(X)</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

See also: Logarithm of the Gamma function: LOG_GAMMA

### 10.120 GERROR — Get last system error message

**GERROR ()**

Returns the system error message corresponding to the last system error. This resembles the functionality of strerror(3) in C.

**Parameters** RESULT – Shall be of type CHARACTER and of default kind.

**Standard:** GNU extension

**Class:** Subroutine

**Syntax:**

```fortran
```
**CALL GERROR(RESULT)**

Example:

```fortran
PROGRAM test_gerror
    CHARACTER(len=100) :: msg
    CALL gerror(msg)
    WRITE(*,*) msg
END PROGRAM
```

*See also:* IERRNO, PERROR

### 10.121 GETARG — Get command line arguments

**GETARG()**

Retrieve the $POS$-th argument that was passed on the command line when the containing program was invoked.

**Parameters**

- $POS$ – Shall be of type INTEGER and not wider than the default integer kind; $POS$ $\geq 0$
- $VALUE$ – Shall be of type CHARACTER and of default kind.

**Returns** After GETARG returns, the $VALUE$ argument holds the $POS$ th command line argument. If $VALUE$ cannot hold the argument, it is truncated to fit the length of $VALUE$. If there are less than $POS$ arguments specified at the command line, $VALUE$ will be filled with blanks. If $POS = 0$, $VALUE$ is set to the name of the program (on systems that support this feature).

**Standard:** GNU extension

**Class:** Subroutine

**Syntax:**

```fortran
CALL GETARG(POS, VALUE)
```

Example:

```fortran
PROGRAM test_getarg
    INTEGER :: i
    CHARACTER(len=32) :: arg

    DO i = 1, iargc()
        CALL getarg(i, arg)
        WRITE(*,*) arg
    END DO
END PROGRAM
```

*See also:* GNU Fortran 77 compatibility function: IARGC Fortran 2003 functions and subroutines: GET_COMMAND, GET_COMMAND_ARGUMENT, COMMAND_ARGUMENT_COUNT

---

**Using GNU Fortran, Release 12.0.1 (experimental 20210414)**

**Chapter 10. Intrinsic Procedures**

188
10.122 GET_COMMAND — Get the entire command line

GET_COMMAND()
Retrieve the entire command line that was used to invoke the program.

Parameters

- **COMMAND** — (Optional) shall be of type CHARACTER and of default kind.
- **LENGTH** — (Optional) Shall be of type INTEGER and of default kind.
- **STATUS** — (Optional) Shall be of type INTEGER and of default kind.

Returns If **COMMAND** is present, stores the entire command line that was used to invoke the program in **COMMAND**. If **LENGTH** is present, it is assigned the length of the command line. If **STATUS** is present, it is assigned 0 upon success of the command, -1 if **COMMAND** is too short to store the command line, or a positive value in case of an error.

Standard: Fortran 2003 and later
Class: Subroutine
Syntax:

```
CALL GET_COMMAND([COMMAND, LENGTH, STATUS])
```

Example:

```fortran
PROGRAM test_get_command
  CHARACTER(len=255) :: cmd
  CALL get_command(cmd)
  WRITE (*,*) TRIM(cmd)
END PROGRAM
```

See also: GET_COMMAND_ARGUMENT, COMMAND_ARGUMENT_COUNT

10.123 GET_COMMAND_ARGUMENT — Get command line arguments

GET_COMMAND_ARGUMENT()
Retrieve the **NUMBER**-th argument that was passed on the command line when the containing program was invoked.

Parameters

- **NUMBER** — Shall be a scalar of type INTEGER and of default kind. **NUMBER** geq 0
- **VALUE** — (Optional) Shall be a scalar of type CHARACTER and of default kind.
- **LENGTH** — (Optional) Shall be a scalar of type INTEGER and of default kind.
- **STATUS** — (Optional) Shall be a scalar of type INTEGER and of default kind.

Returns After **GET_COMMAND_ARGUMENT** returns, the **VALUE** argument holds the **NUMBER**-th command line argument. If **VALUE** cannot hold the argument, it is truncated to fit the length of **VALUE**. If there are less than **NUMBER** arguments specified at the command line, **VALUE** will be filled with blanks. If **NUMBER** = 0, **VALUE** is set to the name of the program (on systems that support this feature). The **LENGTH** argument contains the length of the **NUMBER**-th command.
line argument. If the argument retrieval fails, $STATUS$ is a positive number; if $VALUE$ contains a truncated command line argument, $STATUS$ is -1; and otherwise the $STATUS$ is zero.

**Standard:** Fortran 2003 and later

**Class:** Subroutine

**Syntax:**

```
CALL GET_COMMAND_ARGUMENT( NUMBER [, VALUE, LENGTH, STATUS])
```

**Example:**

```
PROGRAM test_get_command_argument
  INTEGER :: i
  CHARACTER(len=32) :: arg
  i = 0
  DO
    CALL get_command_argument(i, arg)
    IF (LEN_TRIM(arg) == 0) EXIT
    WRITE (*,*) TRIM(arg)
    i = i+1
  END DO
END PROGRAM
```

**See also:** GET_COMMAND, COMMAND_ARGUMENT_COUNT

### 10.124 GETCWD — Get current working directory

**GETCWD()**

Get current working directory.

**Parameters**

- **C** – The type shall be CHARACTER and of default kind.
- **STATUS** – (Optional) status flag. Returns 0 on success, a system specific and nonzero error code otherwise.

**Standard:** GNU extension

**Class:** Subroutine, function

**Syntax:**

```
CALL GETCWD(C [, STATUS])
STATUS = GETCWD(C)
```

**Example:**

```
PROGRAM test_getcwd
  CHARACTER(len=255) :: cwd
  CALL getcwd(cwd)
```

(continues on next page)
WRITE(*,*), TRIM(cwd)
END PROGRAM

See also: CHDIR

10.125 GETENV — Get an environmental variable

GETENV() Get the VALUE of the environmental variable NAME.

Parameters

- NAME – Shall be of type CHARACTER and of default kind.
- VALUE – Shall be of type CHARACTER and of default kind.

Returns Stores the value of NAME in VALUE. If VALUE is not large enough to hold the data, it is truncated. If NAME is not set, VALUE will be filled with blanks.

Standard: GNU extension

Class: Subroutine

Syntax:

CALL GETENV(NAME, VALUE)

Example:

PROGRAM test_getenv
    CHARACTER(len=255) :: homedir
    CALL getenv("HOME", homedir)
    WRITE(*,*), TRIM(homedir)
END PROGRAM

See also: GET_ENVIRONMENT_VARIABLE

10.126 GET_ENVIRONMENT_VARIABLE — Get an environmental variable

GET_ENVIRONMENT_VARIABLE() Get the VALUE of the environmental variable NAME.

Parameters

- NAME – Shall be a scalar of type CHARACTER and of default kind.
- VALUE – (Optional) Shall be a scalar of type CHARACTER and of default kind.
- LENGTH – (Optional) Shall be a scalar of type INTEGER and of default kind.
- STATUS – (Optional) Shall be a scalar of type INTEGER and of default kind.
- TRIM_NAME – (Optional) Shall be a scalar of type LOGICAL and of default kind.
Returns

Stores the value of NAME in VALUE. If VALUE is not large enough to hold the data, it is truncated. If NAME is not set, VALUE will be filled with blanks. Argument LENGTH contains the length needed for storing the environment variable NAME or zero if it is not present. STATUS is -1 if VALUE is present but too short for the environment variable; it is 1 if the environment variable does not exist and 2 if the processor does not support environment variables; in all other cases STATUS is zero. If TRIM_NAME is present with the value .FALSE., the trailing blanks in NAME are significant; otherwise they are not part of the environment variable name.

Standard: Fortran 2003 and later

Class: Subroutine

Syntax:

```
CALL GET_ENVIRONMENT_VARIABLE(NAME[, VALUE, LENGTH, STATUS, TRIM_NAME])
```

Example:

```
PROGRAM test_getenv
  CHARACTER(len=255) :: homedir
  CALL get_environment_variable("HOME", homedir)
  WRITE (*) TRIM(homedir)
END PROGRAM
```

10.127 GETGID — Group ID function

GETGID()

Returns the numerical group ID of the current process.

Returns The return value of GETGID is an INTEGER of the default kind.

Standard: GNU extension

Class: Function

Syntax:

```
RESULT = GETGID()
```

Example: See GETPID for an example.

See also: GETPID, GETUID

10.128 GETLOG — Get login name

GETLOG()

Gets the username under which the program is running.

Parameters C – Shall be of type CHARACTER and of default kind.

Returns Stores the current user name in C. (On systems where POSIX functions geteuid and getpwuid are not available, and the getlogin function is not implemented either, this will return a blank string.)
*Standard*: GNU extension

*Class*: Subroutine

**Syntax:**

```fortran
CALL GETLOG(C)
```

**Example:**

```fortran
PROGRAM TEST_GETLOG
   CHARACTER(32) :: login
   CALL GETLOG(login)
   WRITE(*,*) login
END PROGRAM
```

**See also**: GETUID

### 10.129 GETPID — Process ID function

**GETPID()**

Returns the numerical process identifier of the current process.

**Returns** The return value of GETPID is an INTEGER of the default kind.

*Standard*: GNU extension

*Class*: Function

**Syntax:**

```fortran
RESULT = GETPID()
```

**Example:**

```fortran
program info
   print *, "The current process ID is ", getpid()
   print *, "Your numerical user ID is ", getuid()
   print *, "Your numerical group ID is ", getgid()
end program info
```

**See also**: GETGID, GETUID

### 10.130 GETUID — User ID function

**GETUID()**

Returns the numerical user ID of the current process.

**Returns** The return value of GETUID is an INTEGER of the default kind.

*Standard*: GNU extension

*Class*: Function
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

Syntax:

\[
\text{RESULT} = \text{GETUID}()
\]

Example: See GETPID for an example.

See also: GETPID, GETLOG

10.131 GMTIME — Convert time to GMT info

\text{GMTIME}()

Given a system time value \textit{TIME} (as provided by the TIME intrinsic), fills \textit{VALUES} with values extracted from it appropriate to the UTC time zone (Universal Coordinated Time, also known in some countries as GMT, Greenwich Mean Time), using \texttt{gmtime(3)}.

Parameters

- \textit{TIME} – An INTEGER scalar expression corresponding to a system time, with \texttt{INTENT(IN)}.
- \textit{VALUES} – A default INTEGER array with 9 elements, with \texttt{INTENT(OUT)}.

Returns The elements of \textit{VALUES} are assigned as follows:

\begin{itemize}
  \item \textit{NAME} – Set to the current hostname if it can be obtained, or to a blank string otherwise.
\end{itemize}

Standard: GNU extension

Class: Subroutine

Syntax:

\[
\text{CALL GMTIME(TIME, VALUES)}
\]

See also: DATE_AND_TIME, CTIME, LTIME, TIME, TIME8

10.132 HOSTNM — Get system host name

\text{HOSTNM}()

Retrieves the host name of the system on which the program is running.

Parameters

- \textit{C} – Shall of type CHARACTER and of default kind.
- \textit{STATUS} – (Optional) status flag of type INTEGER. Returns 0 on success, or a system specific error code otherwise.

Returns In either syntax, \textit{NAME} is set to the current hostname if it can be obtained, or to a blank string otherwise.

Standard: GNU extension

Class: Subroutine, function

Syntax:
CALL HOSTNM(C [, STATUS])
STATUS = HOSTNM(NAME)

10.133 HUGE — Largest number of a kind

HUGE (X)
HUGE (X) returns the largest number that is not an infinity in the model of the type of X.

Parameters X – Shall be of type REAL or INTEGER.

Returns The return value is of the same type and kind as X

Standard: Fortran 90 and later
Class: Inquiry function
Syntax:
RESULT = HUGE(X)

Example:

program test_huge_tiny
print *, huge(0), huge(0.0), huge(0.0d0)
print *, tiny(0.0), tiny(0.0d0)
end program test_huge_tiny

10.134 HYPOT — Euclidean distance function

HYPOT (X, Y)
HYPOT (X, Y) is the Euclidean distance function. It is equal to $\sqrt{X^2 + Y^2}$, without undue underflow or overflow.

Parameters

- X – The type shall be REAL.

- Y – The type and kind type parameter shall be the same as X.

Returns The return value has the same type and kind type parameter as X.

Standard: Fortran 2008 and later
Class: Elemental function
Syntax:
RESULT = HYPOT(X, Y)

Example:
program test_hypot
    real(4) :: x = 1.e0_4, y = 0.5e0_4
    x = hypot(x,y)
end program test_hypot

10.135 IACHAR — Code in ASCII collating sequence

IACHAR(C)
IACHAR(C) returns the code for the ASCII character in the first character position of C.

Parameters

• C — Shall be a scalar CHARACTER, with INTENT(IN)

• KIND – (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

Returns  The return value is of type INTEGER and of kind KIND. If KIND is absent, the return value is of default integer kind.

Standard: Fortran 95 and later, with KIND argument Fortran 2003 and later

Class: Elemental function

Syntax:

RESULT = IACHAR(C [, KIND])

Example:

program test_iachar
    integer i
    i = iachar(' ')
end program test_iachar

Note: See ICHAR for a discussion of converting between numerical values and formatted string representations.

See also: ACHAR, CHAR, ICHAR

10.136 IALL — Bitwise AND of array elements

IALL() Reduces with bitwise AND the elements of ARRAY along dimension DIM if the corresponding element in MASK is TRUE.

Parameters

• ARRAY – Shall be an array of type INTEGER

• DIM – (Optional) shall be a scalar of type INTEGER with a value in the range from 1 to n, where n equals the rank of ARRAY.

• MASK – (Optional) shall be of type LOGICAL and either be a scalar or an array of the same shape as ARRAY.
Returns The result is of the same type as `ARRAY`.

**Standard:** Fortran 2008 and later

**Class:** Transformational function

**Syntax:**

```fortran
RESULT = IALL(ARRAY[, MASK])
RESULT = IALL(ARRAY, DIM[, MASK])
```

**Example:**

```fortran
PROGRAM test_iall
  INTEGER(1) :: a(2)
  a(1) = b'00100100'
  a(2) = b'01101010'
  ! prints 00100000
  PRINT '(b8.8)', IALL(a)
END PROGRAM
```

**See also:** IANY, IPARITY, IAND

### 10.137 IAND — Bitwise logical and

**IAND ()**

Bitwise logical AND.

**Parameters**

- **I** – The type shall be `INTEGER` or a boz-literal-constant.
- **J** – The type shall be `INTEGER` with the same kind type parameter as `I` or a boz-literal-constant. `I` and `J` shall not both be boz-literal-constants.

**Returns** The return type is `INTEGER` with the kind type parameter of the arguments. A boz-literal-constant is converted to an `INTEGER` with the kind type parameter of the other argument as-if a call to INT occurred.

**Standard:** Fortran 90 and later, with boz-literal-constant Fortran 2008 and later, has overloads that are GNU extensions

**Class:** Elemental function

**Syntax:**

```fortran
RESULT = IAND(I, J)
```

**Example:**

```fortran
PROGRAM test_iand
  INTEGER :: a, b
  DATA a / Z'F' /, b / Z'3' /
```
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

WRITE (*,*) IAND(a, b)
END PROGRAM

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAND(A)</td>
<td>INTEGER A</td>
<td>INTEGER</td>
<td>Fortran 90 and later</td>
</tr>
<tr>
<td>BIAND(A)</td>
<td>INTEGER(1) A</td>
<td>INTEGER(1)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>IIAND(A)</td>
<td>INTEGER(2) A</td>
<td>INTEGER(2)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>JIAND(A)</td>
<td>INTEGER(4) A</td>
<td>INTEGER(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>KIAND(A)</td>
<td>INTEGER(8) A</td>
<td>INTEGER(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

See also: IOR, IEOR, IBITS, IBSET, IBCLR, NOT

10.138 IANY — Bitwise OR of array elements

IANY ()
Reduces with bitwise OR (inclusive or) the elements of ARRAY along dimension DIM if the corresponding element in MASK is TRUE.

Parameters

- ARRAY – Shall be an array of type INTEGER
- DIM – (Optional) shall be a scalar of type INTEGER with a value in the range from 1 to n, where n equals the rank of ARRAY.
- MASK – (Optional) shall be of type LOGICAL and either be a scalar or an array of the same shape as ARRAY.

Returns The result is of the same type as ARRAY.

Standard: Fortran 2008 and later
Class: Transformational function

Syntax:

RESULT = IANY(ARRAY[, MASK])
RESULT = IANY(ARRAY, DIM[, MASK])

Example:

PROGRAM test_iany
  INTEGER(1) :: a(2)
  a(1) = b'00100100'
  a(2) = b'01101010'
  ! prints 01101110
  PRINT '(b8.8)', IANY(a)
END PROGRAM

See also: IPARITY, IALL, IOR
10.139 IARGC — Get the number of command line arguments

IARGC()
IARGC returns the number of arguments passed on the command line when the containing program was invoked.

Returns The number of command line arguments, type INTEGER(4).

Standard: GNU extension
Class: Function
Syntax:

RESULT = IARGC()

Arguments: None
Example: See GETARG
See also: GNU Fortran 77 compatibility subroutine: GETARG Fortran 2003 functions and subroutines: GET_COMMAND, GET_COMMAND_ARGUMENT, COMMAND_ARGUMENT_COUNT

10.140 IBCLR — Clear bit

IBCLR()
IBCLR returns the value of I with the bit at position POS set to zero.

Parameters
• I – The type shall be INTEGER.
• POS – The type shall be INTEGER.

Returns The return value is of type INTEGER and of the same kind as I.

Standard: Fortran 90 and later, has overloads that are GNU extensions
Class: Elemental function
Syntax:

RESULT = IBCLR(I, POS)

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBCLR(A)</td>
<td>INTEGER A</td>
<td>INTEGER</td>
<td>Fortran 90 and later</td>
</tr>
<tr>
<td>BBCLR(A)</td>
<td>INTEGER(1) A</td>
<td>INTEGER(1)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>IIBCLR(A)</td>
<td>INTEGER(2) A</td>
<td>INTEGER(2)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>JIBCLR(A)</td>
<td>INTEGER(4) A</td>
<td>INTEGER(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>KIBCLR(A)</td>
<td>INTEGER(8) A</td>
<td>INTEGER(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

See also: IBITS, IBSET, IAND, IOR, IEOR, MVBITS
10.141 IBITS — Bit extraction

IBITS(

IBITS extracts a field of length LEN from I, starting from bit position POS and extending left for LEN bits. The result is right-justified and the remaining bits are zeroed. The value of POS+LEN must be less than or equal to the value BIT_SIZE(I).

Parameters

- I – The type shall be INTEGER.
- POS – The type shall be INTEGER.
- LEN – The type shall be INTEGER.

Returns The return value is of type INTEGER and of the same kind as I.

Standard: Fortran 90 and later, has overloads that are GNU extensions

Class: Elemental function

Syntax:

RESULT = IBITS(I, POS, LEN)

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBITS(A)</td>
<td>INTEGER A</td>
<td>INTEGER</td>
<td>Fortran 90 and later</td>
</tr>
<tr>
<td>BBITS(A)</td>
<td>INTEGER(1) A</td>
<td>INTEGER(1)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>JIBITS(A)</td>
<td>INTEGER(2) A</td>
<td>INTEGER(2)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>KIBITS(A)</td>
<td>INTEGER(4) A</td>
<td>INTEGER(4)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

See also: BIT_SIZE, IBCLR, IBSET, IAND, IOR, IEOR

10.142 IBSET — Set bit

IBSET(

IBSET returns the value of I with the bit at position POS set to one.

Parameters

- I – The type shall be INTEGER.
- POS – The type shall be INTEGER.

Returns The return value is of type INTEGER and of the same kind as I.

Standard: Fortran 90 and later, has overloads that are GNU extensions

Class: Elemental function

Syntax:
\textbf{RESULT} = IBSET(I, POS)

\textbf{Specific names:}

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBSET(A)</td>
<td>INTEGER A</td>
<td>INTEGER</td>
<td>Fortran 90 and later</td>
</tr>
<tr>
<td>BBSET(A)</td>
<td>INTEGER(1) A</td>
<td>INTEGER(1)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>IBSET(A)</td>
<td>INTEGER(2) A</td>
<td>INTEGER(2)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>JIBSET(A)</td>
<td>INTEGER(4) A</td>
<td>INTEGER(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>KIBSET(A)</td>
<td>INTEGER(8) A</td>
<td>INTEGER(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

\textbf{See also:} IBCLR, IBITS, IAND, IOR, IEOR, MVBITS

\textbf{10.143 ICHAR — Character-to-integer conversion function}

\texttt{ICHAR(C)} returns the code for the character in the first character position of \texttt{C} in the system’s native character set. The correspondence between characters and their codes is not necessarily the same across different GNU Fortran implementations.

\textbf{Parameters}

- \texttt{C} – Shall be a scalar CHARACTER, with \texttt{INTENT(IN)}
- \texttt{KIND} – (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

\textbf{Returns} The return value is of type INTEGER and of kind \texttt{KIND}. If \texttt{KIND} is absent, the return value is of default integer kind.

\textbf{Standard:} Fortran 77 and later, with \texttt{KIND} argument Fortran 2003 and later

\textbf{Class:} Elemental function

\textbf{Syntax:}

\begin{verbatim}
RESULT = ICHAR(C [, KIND])
\end{verbatim}

\textbf{Example:}

\begin{verbatim}
program test_ichar
  integer i
  i = ichar(' ')
end program test_ichar
\end{verbatim}

\textbf{Specific names:}

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICHAR(C)</td>
<td>CHARACTER</td>
<td>INTEGER(4)</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

\textbf{Note:} No intrinsic exists to convert between a numeric value and a formatted character string representation – for instance, given the CHARACTER value '154', obtaining an INTEGER or REAL value with the value 154, or vice versa. Instead, this functionality is provided by internal-file I/O, as in the following example:
program read_val
    integer value
    character(len=10) string, string2
    string = '154'

    ! Convert a string to a numeric value
    read (string,'(I10)') value
    print *, value

    ! Convert a value to a formatted string
    write (string2,'(I10)') value
    print *, string2
end program read_val

See also: ACHAR, CHAR, IACHAR

10.144 IDATE — Get current local time subroutine (day/month/year)

IDATE(VALUE)

IDATE(VALUE) Fills VALUE with the numerical values at the current local time. The day (in the range 1-31), month (in the range 1-12), and year appear in elements 1, 2, and 3 of VALUE, respectively. The year has four significant digits.

Parameters VALUE – The type shall be INTEGER, DIMENSION(3) and the kind shall be the default integer kind.

Returns Does not return anything.

Standard: GNU extension

Class: Subroutine

Syntax:

CALL IDATE(VALUE)

Example:

program test_idate
    integer, dimension(3) :: tarray
    call idate(tarray)
    print *, tarray(1)
    print *, tarray(2)
    print *, tarray(3)
end program test_idate

See also: DATE_AND_TIME
## 10.145 IEOR — Bitwise logical exclusive or

**IEOR()**
IEOR returns the bitwise Boolean exclusive-OR of `I` and `J`.

**Parameters**
- `I` – The type shall be `INTEGER` or a boz-literal-constant.
- `J` – The type shall be `INTEGER` with the same kind type parameter as `I` or a boz-literal-constant. `I` and `J` shall not both be boz-literal-constants.

**Returns** The return type is `INTEGER` with the kind type parameter of the arguments. A boz-literal-constant is converted to an `INTEGER` with the kind type parameter of the other argument as-if a call to INT occurred.

**Standard**: Fortran 90 and later, with boz-literal-constant Fortran 2008 and later, has overloads that are GNU extensions

**Class**: Elemental function

**Syntax**:

```
RESULT = IEOR(I, J)
```

**Specific names**:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEOR(A)</td>
<td>INTEGER A</td>
<td>INTEGER</td>
<td>Fortran 90 and later</td>
</tr>
<tr>
<td>BIEOR(A)</td>
<td>INTEGER(1) A</td>
<td>INTEGER(1)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>JIEOR(A)</td>
<td>INTEGER(2) A</td>
<td>INTEGER(2)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>JIEOR(A)</td>
<td>INTEGER(4) A</td>
<td>INTEGER(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>KIEOR(A)</td>
<td>INTEGER(8) A</td>
<td>INTEGER(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

**See also**: IOR, IAND, IBITS, IBSET, IBCLR, NOT

## 10.146 IERRNO — Get the last system error number

**IERRNO()**
Returns the last system error number, as given by the C `errno` variable.

**Returns** The return value is of type `INTEGER` and of the default integer kind.

**Standard**: GNU extension

**Class**: Function

**Syntax**:

```
RESULT = IERRNO()
```

**Arguments**: None

**See also**: PERROR
10.147 **IMAGE_INDEX — Function that converts a cosubscript to an image index**

**IMAGE_INDEX ( )**

Returns the image index belonging to a cosubscript.

**Parameters**

- **COARRAY** – Coarray of any type.
- **SUB** – default integer rank-1 array of a size equal to the corank of COARRAY.

**Returns** Scalar default integer with the value of the image index which corresponds to the cosubscripts. For invalid cosubscripts the result is zero.

**Standard**: Fortran 2008 and later  
**Class**: Inquiry function.

**Syntax:**

\[
\text{RESULT} = \text{IMAGE_INDEX}(\text{COARRAY}, \text{SUB})
\]

**Example:**

```
INTEGER :: array[2, -1:4, 8, *]  
! Writes 28 (or 0 if there are fewer than 28 images)  
WRITE (*, *) IMAGE_INDEX (array, [2, 0, 3, 1])
```

**See also**: THIS_IMAGE, NUM_IMAGES

10.148 **INDEX — Position of a substring within a string**

**INDEX ( )**

Returns the position of the start of the first occurrence of string SUBSTRING as a substring in STRING, counting from one. If SUBSTRING is not present in STRING, zero is returned. If the BACK argument is present and true, the return value is the start of the last occurrence rather than the first.

**Parameters**

- **STRING** – Shall be a scalar CHARACTER, with INTENT (IN)  
- **SUBSTRING** – Shall be a scalar CHARACTER, with INTENT (IN)  
- **BACK** – (Optional) Shall be a scalar LOGICAL, with INTENT (IN)  
- **KIND** – (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

**Returns** The return value is of type INTEGER and of kind KIND. If KIND is absent, the return value is of default integer kind.

**Standard**: Fortran 77 and later, with KIND argument Fortran 2003 and later  
**Class**: Elemental function

**Syntax:**
RESULT = INDEX(STRING, SUBSTRING [, BACK [, KIND]])

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDEX(STRING, SUBSTRING)</td>
<td>CHARACTER</td>
<td>INTEGER(4)</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

See also: SCAN, VERIFY

10.149 INT — Convert to integer type

INT ()
Convert to integer type

Parameters

- A – Shall be of type INTEGER, REAL, or COMPLEX or a boz-literal-constant.
- KIND – (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

Returns These functions return a INTEGER variable or array under the following rules:

Standard: Fortran 77 and later, with boz-literal-constant Fortran 2008 and later.

Class: Elemental function

Syntax:

RESULT = INT(A [, KIND])

Example:

```fortran
program test_int
    integer :: i = 42
    complex :: z = (-3.7, 1.0)
    print *, int(i)
    print *, int(z), int(z, 8)
end program
```

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT(A)</td>
<td>REAL(4) A</td>
<td>INTEGER</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>IFIX(A)</td>
<td>REAL(4) A</td>
<td>INTEGER</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>IDINT(A)</td>
<td>REAL(8) A</td>
<td>INTEGER</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>
10.150 INT2 — Convert to 16-bit integer type

INT2 ()
Convert to a KIND=2 integer type. This is equivalent to the standard INT intrinsic with an optional argument of KIND=2, and is only included for backwards compatibility.

Parameters A – Shall be of type INTEGER, REAL, or COMPLEX.

Returns The return value is a INTEGER(2) variable.

Standard: GNU extension
Class: Elemental function
Syntax:

RESULT = INT2(A)

See also: INT, INT8, LONG

10.151 INT8 — Convert to 64-bit integer type

INT8 ()
Convert to a KIND=8 integer type. This is equivalent to the standard INT intrinsic with an optional argument of KIND=8, and is only included for backwards compatibility.

Parameters A – Shall be of type INTEGER, REAL, or COMPLEX.

Returns The return value is a INTEGER(8) variable.

Standard: GNU extension
Class: Elemental function
Syntax:

RESULT = INT8(A)

See also: INT, INT2, LONG

10.152 IOR — Bitwise logical or

IOR ()
IOR returns the bitwise Boolean inclusive-OR of I and J.

Parameters

- I – The type shall be INTEGER or a boz-literal-constant.
- J – The type shall be INTEGER with the same kind type parameter as I or a boz-literal-constant. I and J shall not both be boz-literal-constants.
Returns  The return type is INTEGER with the kind type parameter of the arguments. A boz-literal-constant is converted to an INTEGER with the kind type parameter of the other argument as-if a call to INT occurred.

Standard:  Fortran 90 and later, with boz-literal-constant Fortran 2008 and later, has overloads that are GNU extensions

Class:  Elemental function

Syntax:

\[
\text{RESULT} = \text{IOR}(I, J)
\]

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOR(A)</td>
<td>INTEGER A</td>
<td>INTEGER</td>
<td>Fortran 90 and later</td>
</tr>
<tr>
<td>BIOR(A)</td>
<td>INTEGER(1) A</td>
<td>INTEGER(1)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>IIOR(A)</td>
<td>INTEGER(2) A</td>
<td>INTEGER(2)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>JIOR(A)</td>
<td>INTEGER(4) A</td>
<td>INTEGER(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>KIOR(A)</td>
<td>INTEGER(8) A</td>
<td>INTEGER(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

See also: IEEOR, IAND, IBITS, IBSET, IBCLR, NOT

10.153 IPARIITY — Bitwise XOR of array elements

IPARIITY ()

Reduces with bitwise XOR (exclusive or) the elements of ARRAY along dimension DIM if the corresponding element in MASK is TRUE.

Parameters

- ARRAY – Shall be an array of type INTEGER
- DIM – (Optional) shall be a scalar of type INTEGER with a value in the range from 1 to n, where n equals the rank of ARRAY.
- MASK – (Optional) shall be of type LOGICAL and either be a scalar or an array of the same shape as ARRAY.

Returns  The result is of the same type as ARRAY.

Standard:  Fortran 2008 and later

Class:  Transformational function

Syntax:

\[
\text{RESULT} = \text{IPARIITY}([\text{ARRAY}], \text{DIM}, \text{MASK})
\]

Example:
PROGRAM test_iparity
    INTEGER(1) :: a(2)
    a(1) = int(b'00100100', 1)
    a(2) = int(b'01101010', 1)
    ! prints 01001110
    PRINT '(b8.8)', IPARITY(a)
END PROGRAM

See also: IANY, IALL, IEOR, PARITY

10.154 IRAND — Integer pseudo-random number

IRAND(FLAG)
IRAND(FLAG) returns a pseudo-random number from a uniform distribution between 0 and a system-dependent limit (which is in most cases 2147483647). If FLAG is 0, the next number in the current sequence is returned; if FLAG is 1, the generator is restarted by CALL SRAND(0); if FLAG has any other value, it is used as a new seed with SRAND.

Parameters  I – Shall be a scalar INTEGER of kind 4.

Returns  The return value is of INTEGER(kind=4) type.

Standard: GNU extension

Class: Function

Syntax:

RESULT = IRAND(I)

Example:

program test_irand
    integer, parameter :: seed = 86456
    call srand(seed)
    print *, irand(), irand(), irand(), irand()
    print *, irand(seed), irand(), irand(), irand()
end program test_irand

10.155 IS_CONTIGUOUS — Test whether an array is contiguous

IS_CONTIGUOUS()
IS_CONTIGUOUS tests whether an array is contiguous.

Parameters  ARRAY – Shall be an array of any type.

Returns  Returns a LOGICAL of the default kind, which .TRUE. if ARRAY is contiguous and false otherwise.

Standard: Fortran 2008 and later
Class: Inquiry function

Syntax:

RESULT = IS_CONTIGUOUS(ARRAY)

Example:

```
program test
    integer :: a(10)
    a = [1,2,3,4,5,6,7,8,9,10]
    call sub (a) ! every element, is contiguous
    call sub (a(:2)) ! every other element, is noncontiguous
contains
    subroutine sub (x)
        integer :: x(:)
        if (is_contiguous (x)) then
            write (*,*) 'X is contiguous'
        else
            write (*,*) 'X is not contiguous'
        end if
    end subroutine sub
end program test
```

10.156 IS_IOSTAT_END — Test for end-of-file value

IS_IOSTAT_END()

IS_IOSTAT_END() tests whether an variable has the value of the I/O status 'end of file'. The function is equivalent to comparing the variable with the IOSTAT_END parameter of the intrinsic module ISO_FORTRAN_ENV.

Parameters  I – Shall be of the type INTEGER.

Returns Returns a LOGICAL of the default kind, which .TRUE. if I has the value which indicates an end of file condition for IOSTAT= specifiers, and is .FALSE. otherwise.

Standard: Fortran 2003 and later

Class: Elemental function

Syntax:

RESULT = IS_IOSTAT_END(I)

Example:

```
PROGRAM iostat
    IMPLICIT NONE
    INTEGER :: stat, i
    OPEN(88, FILE='test.dat')
    READ(88, *, IOSTAT=stat) i
    IF(IS_IOSTAT_END(stat)) STOP 'END OF FILE'
END PROGRAM
```

10.156. IS_IOSTAT_END — Test for end-of-file value 209
10.157 IS_IOSTAT_EOR — Test for end-of-record value

**IS_IOSTAT_EOR()**

IS_IOSTAT_EOR tests whether an variable has the value of the I/O status ‘end of record’. The function is equivalent to comparing the variable with the IOSTAT_EOR parameter of the intrinsic module ISO_FORTRAN_ENV.

**Parameters**

\( I \) – Shall be of the type INTEGER.

**Returns**

Returns a LOGICAL of the default kind, which .TRUE. if \( I \) has the value which indicates an end of file condition for IOSTAT= specifiers, and is .FALSE. otherwise.

**Standard**: Fortran 2003 and later

**Class**: Elemental function

**Syntax**:

\[
\text{RESULT} = \text{IS_IOSTAT_EOR}(I)
\]

**Example**:

```
PROGRAM iostat
  IMPLICIT NONE
  INTEGER :: stat, i(50)
  OPEN(88, FILE='test.dat', FORM='UNFORMATTED')
  READ(88, IOSTAT=stat) i
  IF(IS_IOSTAT_EOR(stat)) STOP 'END OF RECORD'
END PROGRAM
```

10.158 ISATTY — Whether a unit is a terminal device.

**ISATTY()**

Determine whether a unit is connected to a terminal device.

**Parameters**

UNIT – Shall be a scalar INTEGER.

**Returns**

Returns .TRUE. if the UNIT is connected to a terminal device, .FALSE. otherwise.

**Standard**: GNU extension

**Class**: Function

**Syntax**:

\[
\text{RESULT} = \text{ISATTY}(UNIT)
\]

**Example**:

```
PROGRAM test_isatty
  INTEGER(kind=1) :: unit
  DO unit = 1, 10
    write(*,*) isatty(unit=unit)
  END DO
END PROGRAM
```

**See also**: TTYNAM
10.159 ISHFT — Shift bits

**ISHFT()**

ISHFT returns a value corresponding to \( I \) with all of the bits shifted \( SHIFT \) places. A value of \( SHIFT \) greater than zero corresponds to a left shift, a value of zero corresponds to no shift, and a value less than zero corresponds to a right shift. If the absolute value of \( SHIFT \) is greater than \( \text{BIT\_SIZE}(I) \), the value is undefined. Bits shifted out from the left end or right end are lost; zeros are shifted in from the opposite end.

**Parameters**

- \( I \) – The type shall be \texttt{INTEGER}.
- \( SHIFT \) – The type shall be \texttt{INTEGER}.

**Returns** The return value is of type \texttt{INTEGER} and of the same kind as \( I \).

**Standard:** Fortran 90 and later, has overloads that are GNU extensions

**Class:** Elemental function

**Syntax:**

\[
\text{RESULT} = \text{ISHFT}(I, \text{SHIFT})
\]

**Specific names:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISHFT(A)</td>
<td>INTEGER A</td>
<td>INTEGER</td>
<td>Fortran 90 and later</td>
</tr>
<tr>
<td>BSHFT(A)</td>
<td>INTEGER(1) A</td>
<td>INTEGER(1)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>IISHFT(A)</td>
<td>INTEGER(2) A</td>
<td>INTEGER(2)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>JISHFT(A)</td>
<td>INTEGER(4) A</td>
<td>INTEGER(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>KISHFT(A)</td>
<td>INTEGER(8) A</td>
<td>INTEGER(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

**See also:** ISHFTC

10.160 ISHFTC — Shift bits circularly

**ISHFTC()**

ISHFTC returns a value corresponding to \( I \) with the rightmost \( SIZE \) bits shifted circularly \( SHIFT \) places; that is, bits shifted out one end are shifted into the opposite end. A value of \( SHIFT \) greater than zero corresponds to a left shift, a value of zero corresponds to no shift, and a value less than zero corresponds to a right shift. The absolute value of \( SHIFT \) must be less than \( SIZE \). If the \( SIZE \) argument is omitted, it is taken to be equivalent to \( \text{BIT\_SIZE}(I) \).

**Parameters**

- \( I \) – The type shall be \texttt{INTEGER}.
- \( SHIFT \) – The type shall be \texttt{INTEGER}.
- \( SIZE \) – (Optional) The type shall be \texttt{INTEGER}; the value must be greater than zero and less than or equal to \( \text{BIT\_SIZE}(I) \).

**Returns** The return value is of type \texttt{INTEGER} and of the same kind as \( I \).
**Standard:** Fortran 90 and later, has overloads that are GNU extensions

**Class:** Elemental function

**Syntax:**

```fortran
RESULT = ISHFTC(I, SHIFT [, SIZE])
```

**Specific names:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISHFTC(A)</td>
<td>INTEGER A</td>
<td>INTEGER</td>
<td>Fortran 90 and later</td>
</tr>
<tr>
<td>BISHFTC(A)</td>
<td>INTEGER(1) A</td>
<td>INTEGER(1)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>IISHFTC(A)</td>
<td>INTEGER(2) A</td>
<td>INTEGER(2)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>JISHFTC(A)</td>
<td>INTEGER(4) A</td>
<td>INTEGER(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>KISHFTC(A)</td>
<td>INTEGER(8) A</td>
<td>INTEGER(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

**See also:** ISHFT

### 10.161 ISNAN — Test for a NaN

**ISNAN()**

The **ISNAN** function tests whether a floating-point value is an IEEE Not-a-Number (NaN).

**Parameters**

- **X** — Variable of the type REAL.

**Returns**

Returns a default-kind LOGICAL. The returned value is **TRUE** if **X** is a NaN and **FALSE** otherwise.

**Standard:** GNU extension

**Class:** Elemental function

**Syntax:**

```fortran
ISNAN(X)
```

**Example:**

```fortran
program test_nan
  implicit none
  real :: x
  x = -1.0
  x = sqrt(x)
  if (isnan(x)) stop '"x" is a NaN'
end program test_nan
```
10.162 ITIME — Get current local time subroutine (hour/minutes/seconds)

ITIME (VALUES)

ITIME(VALUES) Fills VALUES with the numerical values at the current local time. The hour (in the range 1-24), minute (in the range 1-60), and seconds (in the range 1-60) appear in elements 1, 2, and 3 of VALUES, respectively.

Parameters VALUES – The type shall be INTEGER, DIMENSION(3) and the kind shall be the default integer kind.

Returns Does not return anything.

Standard: GNU extension

Class: Subroutine

Syntax:

CALL ITIME(VALUES)

Example:

program test_itime
    integer, dimension(3) :: tarray
    call itime(tarray)
    print *, tarray(1)
    print *, tarray(2)
    print *, tarray(3)
end program test_itime

See also: DATE_AND_TIME

10.163 KILL — Send a signal to a process

KILL()

Sends the signal specified by SIG to the process PID. See kill(2).

Parameters

• PID – Shall be a scalar INTEGER with INTENT(IN).
• SIG – Shall be a scalar INTEGER with INTENT(IN).
• STATUS – [Subroutine](Optional) Shall be a scalar INTEGER. Returns 0 on success; otherwise a system-specific error code is returned.
• STATUS – [Function] The kind type parameter is that of pid. Returns 0 on success; otherwise a system-specific error code is returned.

Standard: GNU extension

Class: Subroutine, function
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

Syntax:

```fortran
CALL KILL(PID, SIG [, STATUS])
STATUS = KILL(PID, SIG)
```

See also: ABORT, EXIT

10.164 KIND — Kind of an entity

KIND(X)

KIND(X) returns the kind value of the entity X.

Parameters

- X – Shall be of type LOGICAL, INTEGER, REAL, COMPLEX or CHARACTER. It may be scalar or array valued.

Returns

The return value is a scalar of type INTEGER and of the default integer kind.

Standard: Fortran 95 and later

Class: Inquiry function

Syntax:

```fortran
K = KIND(X)
```

Example:

```fortran
program test_kind
  integer, parameter :: kc = kind(' ')
  integer, parameter :: kl = kind(.true.)
  print *, "The default character kind is ", kc
  print *, "The default logical kind is ", kl
end program test_kind
```

10.165 LBOUND — Lower dimension bounds of an array

LBOUND()

Returns the lower bounds of an array, or a single lower bound along the DIM dimension.

Parameters

- ARRAY – Shall be an array, of any type.
- DIM – (Optional) Shall be a scalar INTEGER.
- KIND – (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

Returns

The return value is of type INTEGER and of kind KIND. If KIND is absent, the return value is of default integer kind. If DIM is absent, the result is an array of the lower bounds of ARRAY. If DIM is present, the result is a scalar corresponding to the lower bound of the array along that dimension. If ARRAY is an expression rather than a whole array or array structure component, or if it has a zero extent along the relevant dimension, the lower bound is taken to be 1.
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

**Standard**: Fortran 90 and later, with KIND argument Fortran 2003 and later  
**Class**: Inquiry function

**Syntax**:

```fortran
RESULT = LBOUND(ARRAY [, DIM [, KIND]])
```

**See also**: UBOUND, LCOBOUND

### 10.166 LCOBOUND — Lower codimension bounds of an array

LCOBOUND ()

Returns the lower bounds of a coarray, or a single lower cobound along the DIM codimension.

**Parameters**

- ARRAY – Shall be a coarray, of any type.
- DIM – (Optional) Shall be a scalar INTEGER.
- KIND – (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

**Returns**

The return value is of type INTEGER and of kind KIND. If KIND is absent, the return value is of default integer kind. If DIM is absent, the result is an array of the lower cobounds of COARRAY. If DIM is present, the result is a scalar corresponding to the lower cobound of the array along that codimension.

**Standard**: Fortran 2008 and later  
**Class**: Inquiry function

**Syntax**:

```fortran
RESULT = LCOBOUND(COARRAY [, DIM [, KIND]])
```

**See also**: UCOBOUND, LBOUND

### 10.167 LEADZ — Number of leading zero bits of an integer

LEADZ ()

LEADZ returns the number of leading zero bits of an integer.

**Parameters**

- I – Shall be of type INTEGER.

**Returns**

The type of the return value is the default INTEGER. If all the bits of I are zero, the result value is BIT_SIZE(I).

**Standard**: Fortran 2008 and later  
**Class**: Elemental function

**Syntax**:
RESULT = LEADZ(I)

Example:

```fortran
PROGRAM test_leadz
  WRITE (*,*) BIT_SIZE(1)  ! prints 32
  WRITE (*,*) LEADZ(1)    ! prints 31
END PROGRAM
```

See also: BIT_SIZE, TRAILZ, POPCNT, POPPAR

10.168 LEN — Length of a character entity

LEN()

Returns the length of a character string. If STRING is an array, the length of an element of STRING is returned. Note that STRING need not be defined when this intrinsic is invoked, since only the length, not the content, of STRING is needed.

Parameters

- **STRING** – Shall be a scalar or array of type CHARACTER, with INTENT(IN)
- **KIND** – (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

Returns The return value is of type INTEGER and of kind KIND. If KIND is absent, the return value is of default integer kind.

Standard: Fortran 77 and later, with KIND argument Fortran 2003 and later

Class: Inquiry function

Syntax:

```
L = LEN(STRING [, KIND])
```

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEN(STRING)</td>
<td>CHARACTER</td>
<td>INTEGER</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

See also: LEN_TRIM, ADJUSTL, ADJUSTR
10.169 LEN_TRIM — Length of a character entity without trailing blank characters

LEN_TRIM()

Returns the length of a character string, ignoring any trailing blanks.

Parameters

• STRING – Shall be a scalar of type CHARACTER, with INTENT(IN)
• KIND – (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

Returns The return value is of type INTEGER and of kind KIND. If KIND is absent, the return value is of default integer kind.

Standard: Fortran 90 and later, with KIND argument Fortran 2003 and later

Class: Elemental function

Syntax:

RESULT = LEN_TRIM(STRING [, KIND])

See also: LEN, ADJUSTL, ADJUSTR

10.170 LGE — Lexical greater than or equal

LGE()

Determines whether one string is lexically greater than or equal to another string, where the two strings are interpreted as containing ASCII character codes. If the String A and String B are not the same length, the shorter is compared as if spaces were appended to it to form a value that has the same length as the longer.

Parameters

• STRING_A – Shall be of default CHARACTER type.
• STRING_B – Shall be of default CHARACTER type.

Returns Returns .TRUE. if STRING_A >= STRING_B, and .FALSE. otherwise, based on the ASCII ordering.

Standard: Fortran 77 and later

Class: Elemental function

Syntax:

RESULT = LGE(STRING_A, STRING_B)

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGE(STRING_A, STRING_B)</td>
<td>CHARACTER</td>
<td>LOGICAL</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>
See also: LGT, LLE, LLT

10.171 LGT — Lexical greater than

LGT ()
Determines whether one string is lexically greater than another string, where the two strings are interpreted as containing ASCII character codes. If the String A and String B are not the same length, the shorter is compared as if spaces were appended to it to form a value that has the same length as the longer.

Parameters

• STRING_A – Shall be of default CHARACTER type.
• STRING_B – Shall be of default CHARACTER type.

Returns Returns .TRUE. if STRING_A > STRING_B, and .FALSE. otherwise, based on the ASCII ordering.

Standard: Fortran 77 and later
Class: Elemental function
Syntax:

```fortran
RESULT = LGT(STRING_A, STRING_B)
```

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGT(STRING_A, STRING_B)</td>
<td>CHARACTER</td>
<td>LOGICAL</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

See also: LGE, LLE, LLT

10.172 LINK — Create a hard link

LINK ()
Makes a (hard) link from file PATH1 to PATH2. A null character (CHAR(0)) can be used to mark the end of the names in PATH1 and PATH2; otherwise, trailing blanks in the file names are ignored. If the STATUS argument is supplied, it contains 0 on success or a nonzero error code upon return; see link(2).

Parameters

• PATH1 – Shall be of default CHARACTER type.
• PATH2 – Shall be of default CHARACTER type.
• STATUS – (Optional) Shall be of default INTEGER type.

Standard: GNU extension
Class: Subroutine, function
Syntax:
CALL LINK(PATH1, PATH2 [, STATUS])
STATUS = LINK(PATH1, PATH2)

See also: SYMLNK, UNLINK

10.173 LLE — Lexical less than or equal

LLE ()
Determines whether one string is lexically less than or equal to another string, where the two strings are interpreted
as containing ASCII character codes. If the String A and String B are not the same length, the shorter is compared
as if spaces were appended to it to form a value that has the same length as the longer.

Parameters
- STRING_A – Shall be of default CHARACTER type.
- STRING_B – Shall be of default CHARACTER type.

Returns Returns .TRUE. if STRING_A <= STRING_B, and .FALSE. otherwise, based on the
ASCII ordering.

Standard: Fortran 77 and later
Class: Elemental function
Syntax:

RESULT = LLE(STRING_A, STRING_B)

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLE(STRING_A, STRING_B)</td>
<td>CHARACTER</td>
<td>LOGICAL</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

See also: LGE, LGT, LLT

10.174 LLT — Lexical less than

LLT ()
Determines whether one string is lexically less than another string, where the two strings are interpreted as con-
taining ASCII character codes. If the String A and String B are not the same length, the shorter is compared as if
spaces were appended to it to form a value that has the same length as the longer.

Parameters
- STRING_A – Shall be of default CHARACTER type.
- STRING_B – Shall be of default CHARACTER type.

Returns Returns .TRUE. if STRING_A < STRING_B, and .FALSE. otherwise, based on the
ASCII ordering.

Standard: Fortran 77 and later
Class: Elemental function

Syntax:

```
RESULT = LLT(STRING_A, STRING_B)
```

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLT(STRING_A, STRING_B)</td>
<td>CHARACTER</td>
<td>LOGICAL</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

See also: LGE, LGT, LLE

10.175 LNBLNK — Index of the last non-blank character in a string

LNBLNK ()

Returns the length of a character string, ignoring any trailing blanks. This is identical to the standard LEN_TRIM intrinsic, and is only included for backwards compatibility.

Parameters

- **STRING** – Shall be a scalar of type CHARACTER, with INTENT(IN)

Returns

- The return value is of INTEGER(kind=4) type.

Standard: GNU extension

Class: Elemental function

Syntax:

```
RESULT = LNBLNK (STRING)
```

See also: INDEX intrinsic, LEN_TRIM

10.176 LOC — Returns the address of a variable

LOC (X)

LOC (X) returns the address of \( X \) as an integer.

Parameters

- **X** – Variable of any type.

Returns

- The return value is of type INTEGER, with a KIND corresponding to the size (in bytes) of a memory address on the target machine.

Standard: GNU extension

Class: Inquiry function

Syntax:

```
RESULT = LOC (X)
```

Example:
```fortran
program test_loc
  integer :: i
  real :: r
  i = loc(r)
  print *, i
end program test_loc
```

### 10.177 LOG — Natural logarithm function

**LOG**(*X*)

LOG(*X*) computes the natural logarithm of *X*, i.e. the logarithm to the base e.

**Parameters** *X* — The type shall be REAL or COMPLEX.

**Returns** The return value is of type REAL or COMPLEX. The kind type parameter is the same as *X*. If *X* is COMPLEX, the imaginary part omega is in the range -pi < omega leq pi.

**Standard**: Fortran 77 and later, has GNU extensions

**Class**: Elemental function

**Syntax**:

```
RESULT = LOG(X)
```

**Example**:

```fortran
program test_log
  real(8) :: x = 2.7182818284590451_8
  complex :: z = (1.0, 2.0)
  x = log(x)  ! will yield (approximately) 1
  z = log(z)
end program test_log
```

**Specific names**:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALOG(X)</td>
<td>REAL(4) X</td>
<td>REAL(4)</td>
<td>Fortran 77 or later</td>
</tr>
<tr>
<td>DLOG(X)</td>
<td>REAL(8) X</td>
<td>REAL(8)</td>
<td>Fortran 77 or later</td>
</tr>
<tr>
<td>CLOG(X)</td>
<td>COMPLEX(4) X</td>
<td>COMPLEX(4)</td>
<td>Fortran 77 or later</td>
</tr>
<tr>
<td>ZLOG(X)</td>
<td>COMPLEX(8) X</td>
<td>COMPLEX(8)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>CDLOG(X)</td>
<td>COMPLEX(8) X</td>
<td>COMPLEX(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>
10.178 LOG10 — Base 10 logarithm function

LOG10(X)

LOG10(X) computes the base 10 logarithm of X.

Parameters X – The type shall be REAL.

Returns The return value is of type REAL or COMPLEX. The kind type parameter is the same as X.

Standard: Fortran 77 and later

Class: Elemental function

Syntax:

RESULT = LOG10(X)

Example:

```
program test_log10
    real(8) :: x = 10.0_8
    x = log10(x)
end program test_log10
```

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALOG10(X)</td>
<td>REAL(4)</td>
<td>X</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DLOG10(X)</td>
<td>REAL(8)</td>
<td>X</td>
<td>REAL(8)</td>
</tr>
</tbody>
</table>

10.179 LOG_GAMMA — Logarithm of the Gamma function

LOG_GAMMA(X)

LOG_GAMMA(X) computes the natural logarithm of the absolute value of the Gamma (Gamma) function.

Parameters X – Shall be of type REAL and neither zero nor a negative integer.

Returns The return value is of type REAL of the same kind as X.

Standard: Fortran 2008 and later

Class: Elemental function

Syntax:

X = LOG_GAMMA(X)

Example:

```
program test_log_gamma
    real :: x = 1.0
    x = lgamma(x) ! returns 0.0
end program test_log_gamma
```
Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGAMMA (X)</td>
<td>REAL(4) X</td>
<td>REAL(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>ALGAMRA (X)</td>
<td>REAL(4) X</td>
<td>REAL(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>DLGAMRA (X)</td>
<td>REAL(8) X</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

See also: Gamma function: GAMMA

10.180 LOGICAL — Convert to logical type

LOGICAL()
Converting one kind of LOGICAL variable to another.

Parameters
• L – The type shall be LOGICAL.
• KIND – (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

Returns The return value is a LOGICAL value equal to L, with a kind corresponding to KIND, or of the default logical kind if KIND is not given.

Standard: Fortran 90 and later
Class: Elemental function
Syntax:

RESULT = LOGICAL(L [, KIND])

See also: INT, REAL, CMPLX

10.181 LONG — Convert to integer type

LONG()
Convert to a KIND=4 integer type, which is the same size as a C long integer. This is equivalent to the standard INT intrinsic with an optional argument of KIND=4, and is only included for backwards compatibility.

Parameters A – Shall be of type INTEGER, REAL, or COMPLEX.

Returns The return value is a INTEGER (4) variable.

Standard: GNU extension
Class: Elemental function
Syntax:

RESULT = LONG(A)

See also: INT, INT2, INT8
10.182 LSHIFT — Left shift bits

**LSHIFT()**

LSHIFT returns a value corresponding to \( I \) with all of the bits shifted left by \( SHIFT \) places. \( SHIFT \) shall be nonnegative and less than or equal to \( \text{BIT\_SIZE}(I) \), otherwise the result value is undefined. Bits shifted out from the left end are lost; zeros are shifted in from the opposite end.

**Parameters**

- \( I \) – The type shall be INTEGER.
- \( SHIFT \) – The type shall be INTEGER.

**Returns** The return value is of type INTEGER and of the same kind as \( I \).

**Standard**: GNU extension

**Class**: Elemental function

**Syntax**:

```
RESULT = LSHIFT(I, SHIFT)
```

**See also**: ISHFT, ISHFTC, RSHIFT, SHIFTA, SHIFTL, SHIFTR

10.183 LSTAT — Get file status

**LSTAT()**

LSTAT is identical to STAT, except that if path is a symbolic link, then the link itself is statted, not the file that it refers to.

**Parameters**

- \( NAME \) – The type shall be CHARACTER of the default kind, a valid path within the file system.
- \( VALUES \) – The type shall be INTEGER(4), DIMENSION(13).
- \( STATUS \) – (Optional) status flag of type INTEGER(4). Returns 0 on success and a system specific error code otherwise.

**Standard**: GNU extension

**Class**: Subroutine, function

**Syntax**:

```
CALL LSTAT(NAME, VALUES [, STATUS])
STATUS = LSTAT(NAME, VALUES)
```

**Example**: See STAT for an example.

**See also**: To stat an open file: FSTAT To stat a file: STAT
10.184 LTIME — Convert time to local time info

LTIME()
Given a system time value TIME (as provided by the TIME intrinsic), fills VALUES with values extracted from it appropriate to the local time zone using localtime(3).

Parameters

- TIME – An INTEGER scalar expression corresponding to a system time, with INTENT(IN).
- VALUES – A default INTEGER array with 9 elements, with INTENT(OUT).

Returns The elements of VALUES are assigned as follows:

Standard: GNU extension

Class: Subroutine

Syntax:

CALL LTIME(TIME, VALUES)

See also: DATE_AND_TIME, CTIME, GMTIME, TIME, TIME8

10.185 MALLOC — Allocate dynamic memory

MALLOC(SIZE)
MALLOC(SIZE) allocates SIZE bytes of dynamic memory and returns the address of the allocated memory. The MALLOC intrinsic is an extension intended to be used with Cray pointers, and is provided in GNU Fortran to allow the user to compile legacy code. For new code using Fortran 95 pointers, the memory allocation intrinsic is ALLOCATE.

Parameters SIZE – The type shall be INTEGER.

Returns The return value is of type INTEGER(K), with K such that variables of type INTEGER(K) have the same size as C pointers(sizeof(void *)).

Standard: GNU extension

Class: Function

Syntax:

PTR = MALLOC(SIZE)

Example: The following example demonstrates the use of MALLOC and FREE with Cray pointers.

```
program test_malloc
  implicit none
  integer i
  real*8 x(*), z
  pointer(ptr_x,x)

  ptr_x = malloc(20*8)
  do i = 1, 20
    ...
  end do

  free(ptr_x)
end program test_malloc
```
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

(continued from previous page)

```fortran
  x(i) = sqrt(1.0d0 / i)
end do
z = 0
do i = 1, 20
   z = z + x(i)
   print *, z
end do
call free(ptr_x)
end program test_malloc
```

See also: FREE

10.186 MASKL — Left justified mask

**MASKL**([, KIND])

MASKL(I[, KIND]) has its leftmost \( I \) bits set to 1, and the remaining bits set to 0.

Parameters

- **I** – Shall be of type INTEGER.
- **KIND** – Shall be a scalar constant expression of type INTEGER.

Returns The return value is of type INTEGER. If KIND is present, it specifies the kind value of the return type; otherwise, it is of the default integer kind.

Standard: Fortran 2008 and later

Class: Elemental function

Syntax:

```fortran
RESULT = MASKL(I[, KIND])
```

See also: MASKR

10.187 MASKR — Right justified mask

**MASKR**()

MASKR(I[, KIND]) has its rightmost \( I \) bits set to 1, and the remaining bits set to 0.

Parameters

- **I** – Shall be of type INTEGER.
- **KIND** – Shall be a scalar constant expression of type INTEGER.

Returns The return value is of type INTEGER. If KIND is present, it specifies the kind value of the return type; otherwise, it is of the default integer kind.

Standard: Fortran 2008 and later

Class: Elemental function
Syntax:

\[
\text{RESULT} = \text{MASKR}(I[, \text{KIND}])
\]

See also: MASKL

10.188 MATMUL — matrix multiplication

MATMUL()

Performs a matrix multiplication on numeric or logical arguments.

Parameters

- **MATRIX_A** – An array of INTEGER, REAL, COMPLEX, or LOGICAL type, with a rank of one or two.
- **MATRIX_B** – An array of INTEGER, REAL, or COMPLEX type if MATRIX_A is of a numeric type; otherwise, an array of LOGICAL type. The rank shall be one or two, and the first (or only) dimension of MATRIX_B shall be equal to the last (or only) dimension of MATRIX_A. MATRIX_A and MATRIX_B shall not both be rank one arrays.

Returns

The matrix product of MATRIX_A and MATRIX_B. The type and kind of the result follow the usual type and kind promotion rules, as for the * or .AND. operators.

Standard: Fortran 90 and later

Class: Transformational function

Syntax:

\[
\text{RESULT} = \text{MATMUL}(\text{MATRIX}_A, \text{MATRIX}_B)
\]

10.189 MAX — Maximum value of an argument list

MAX()

Returns the argument with the largest (most positive) value.

Parameters

- **A1** – The type shall be INTEGER or REAL.
- \{(A3), ..., (A2),\} – An expression of the same type and kind as A1. (As a GNU extension, arguments of different kinds are permitted.)

Returns

The return value corresponds to the maximum value among the arguments, and has the same type and kind as the first argument.

Standard: Fortran 77 and later

Class: Elemental function

Syntax:

\[
\text{RESULT} = \text{MAX}(A1, A2 [, A3 [, ...]])
\]

10.188. MATMUL — matrix multiplication
Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX0(A1)</td>
<td>INTEGER(4) A1</td>
<td>INTEGER(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>AMAX0(A1)</td>
<td>INTEGER(4) A1</td>
<td>REAL(MAX(X))</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>MAX1(A1)</td>
<td>REAL A1</td>
<td>INT(MAX(X))</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>AMAX1(A1)</td>
<td>REAL(4) A1</td>
<td>REAL(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>DMAX1(A1)</td>
<td>REAL(8) A1</td>
<td>REAL(8)</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

See also: MAXLOC MAXVAL, MIN

10.190 MAXEXPONENT — Maximum exponent of a real kind

MAXEXPONENT (X)

MAXEXPONENT (X) returns the maximum exponent in the model of the type of X.

Parameters X – Shall be of type REAL.

Returns The return value is of type INTEGER and of the default integer kind.

Standard: Fortran 90 and later

Class: Inquiry function

Syntax:

RESULT = MAXEXPONENT(X)

Example:

```
program exponents
  real(kind=4) :: x
  real(kind=8) :: y
  print *, minexponent(x), maxexponent(x)
  print *, minexponent(y), maxexponent(y)
end program exponents
```

10.191 MAXLOC — Location of the maximum value within an array

MAXLOC ()

Determines the location of the element in the array with the maximum value, or, if the DIM argument is supplied, determines the locations of the maximum element along each row of the array in the DIM direction. If MASK is present, only the elements for which MASK is .TRUE. are considered. If more than one element in the array has the maximum value, the location returned is that of the first such element in array element order if the BACK is not present, or is false; if BACK is true, the location returned is that of the last such element. If the array has zero size, or all of the elements of MASK are .FALSE., then the result is an array of zeroes. Similarly, if DIM is supplied and all of the elements of MASK along a given row are zero, the result value for that row is zero.

Parameters

- ARRAY – Shall be an array of type INTEGER or REAL.
• **DIM** – (Optional) Shall be a scalar of type INTEGER, with a value between one and the rank of ARRAY, inclusive. It may not be an optional dummy argument.

• **MASK** – Shall be of type LOGICAL, and conformable with ARRAY.

• **KIND** – (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

• **BACK** – (Optional) A scalar of type LOGICAL.

**Returns**

If **DIM** is absent, the result is a rank-one array with a length equal to the rank of ARRAY. If **DIM** is present, the result is an array with a rank one less than the rank of ARRAY, and a size corresponding to the size of ARRAY with the **DIM** dimension removed. If **DIM** is present and ARRAY has a rank of one, the result is a scalar. If the optional argument **KIND** is present, the result is an integer of kind **KIND**, otherwise it is of default kind.

**Standard**: Fortran 95 and later; ARRAY of CHARACTER and the **KIND** argument are available in Fortran 2003 and later. The **BACK** argument is available in Fortran 2008 and later.

**Class**: Transformational function

**Syntax**:

\[
\text{RESULT} = \text{MAXLOC}(\text{ARRAY}, \text{DIM}[, \text{MASK}][,\text{KIND}][,\text{BACK}])
\]

**See also**: FINDLOC, MAX, MAXVAL

### 10.192 MAXVAL — Maximum value of an array

**MAXVAL()**

Determines the maximum value of the elements in an array value, or, if the **DIM** argument is supplied, determines the maximum value along each row of the array in the **DIM** direction. If **MASK** is present, only the elements for which **MASK** is .TRUE. are considered. If the array has zero size, or all of the elements of **MASK** are .FALSE., then the result is $-\text{HUGE(ARRAY)}$ if **ARRAY** is numeric, or a string of nulls if **ARRAY** is of character type.

**Parameters**

• **ARRAY** – Shall be an array of type INTEGER or REAL.

• **DIM** – (Optional) Shall be a scalar of type INTEGER, with a value between one and the rank of ARRAY, inclusive. It may not be an optional dummy argument.

• **MASK** – (Optional) Shall be of type LOGICAL, and conformable with ARRAY.

**Returns**

If **DIM** is absent, or if **ARRAY** has a rank of one, the result is a scalar. If **DIM** is present, the result is an array with a rank one less than the rank of **ARRAY**, and a size corresponding to the size of **ARRAY** with the **DIM** dimension removed. In all cases, the result is of the same type and kind as **ARRAY**.

**Standard**: Fortran 90 and later

**Class**: Transformational function

**Syntax**:
RESULT = MAXVAL(ARRAY, DIM [, MASK])
RESULT = MAXVAL(ARRAY [, MASK])

See also: MAX, MAXLOC

10.193 MCLOCK — Time function

MCLOCK()
Returns the number of clock ticks since the start of the process, based on the function clock(3) in the C standard library.

Returns The return value is a scalar of type INTEGER(4), equal to the number of clock ticks since the start of the process, or -1 if the system does not support clock(3).

Standard: GNU extension
Class: Function
Syntax:
RESULT = MCLOCK()

See also: CTIME, GMTIME, LTIME, MCLOCK, TIME

10.194 MCLOCK8 — Time function (64-bit)

MCLOCK8()
Returns the number of clock ticks since the start of the process, based on the function clock(3) in the C standard library.

Returns The return value is a scalar of type INTEGER(8), equal to the number of clock ticks since the start of the process, or -1 if the system does not support clock(3).

Standard: GNU extension
Class: Function
Syntax:
RESULT = MCLOCK8()

See also: CTIME, GMTIME, LTIME, MCLOCK, TIME8
10.195 MERGE — Merge variables

**MERGE ()**

Select values from two arrays according to a logical mask. The result is equal to `TSOURCE` if `MASK` is `.TRUE.`., or equal to `FSOURCE` if it is `.FALSE.`.

**Parameters**
- `TSOURCE` – May be of any type.
- `FSOURCE` – Shall be of the same type and type parameters as `TSOURCE`.
- `MASK` – Shall be of type `LOGICAL`.

**Returns** The result is of the same type and type parameters as `TSOURCE`.

**Standard**: Fortran 90 and later

**Class**: Elemental function

**Syntax**:

```fortran
RESULT = MERGE(TSOURCE, FSOURCE, MASK)
```

10.196 MERGE_BITS — Merge of bits under mask

**MERGE_BITS (I, J, MASK)**

`MERGE_BITS (I, J, MASK)` merges the bits of `I` and `J` as determined by the mask. The i-th bit of the result is equal to the i-th bit of `I` if the i-th bit of `MASK` is 1; it is equal to the i-th bit of `J` otherwise.

**Parameters**
- `I` – Shall be of type `INTEGER` or a boz-literal-constant.
- `J` – Shall be of type `INTEGER` with the same kind type parameter as `I` or a boz-literal-constant. `I` and `J` shall not both be boz-literal-constants.
- `MASK` – Shall be of type `INTEGER` or a boz-literal-constant and of the same kind as `I`.

**Returns** The result is of the same type and kind as `I`.

**Standard**: Fortran 2008 and later

**Class**: Elemental function

**Syntax**:

```fortran
RESULT = MERGE_BITS(I, J, MASK)
```
10.197 MIN — Minimum value of an argument list

MIN()

Returns the argument with the smallest (most negative) value.

Parameters

- A1 – The type shall be INTEGER or REAL.
- {A3}, ... (A2), - An expression of the same type and kind as A1. (As a GNU extension, arguments of different kinds are permitted.)

Returns The return value corresponds to the minimum value among the arguments, and has the same type and kind as the first argument.

Standard: Fortran 77 and later

Class: Elemental function

Syntax:

\[
\text{RESULT} = \text{MIN}(A1, A2 [, A3, ...])
\]

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIN0(A1)</td>
<td>INTEGER(4) A1</td>
<td>INTEGER(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>AMIN0(A1)</td>
<td>INTEGER(4) A1</td>
<td>REAL(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>MIN1(A1)</td>
<td>REAL A1</td>
<td>INTEGER(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>AMIN1(A1)</td>
<td>REAL(4) A1</td>
<td>INTEGER(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>DMIN1(A1)</td>
<td>REAL(8) A1</td>
<td>REAL(8)</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

See also: MAX, MINLOC, MINVAL

10.198 MINEXPONENT — Minimum exponent of a real kind

MINEXPONENT(X)

MINEXPONENT(X) returns the minimum exponent in the model of the type of X.

Parameters X – Shall be of type REAL.

Returns The return value is of type INTEGER and of the default integer kind.

Standard: Fortran 90 and later

Class: Inquiry function

Syntax:

\[
\text{RESULT} = \text{MINEXPONENT}(X)
\]

Example: See MAXEXPONENT for an example.
10.199 MINLOC — Location of the minimum value within an array

MINLOC ()
Determines the location of the element in the array with the minimum value, or, if the DIM argument is supplied, determines the locations of the minimum element along each row of the array in the DIM direction. If MASK is present, only the elements for which MASK is .TRUE. are considered. If more than one element in the array has the minimum value, the location returned is that of the first such element in array element order if the BACK is not present, or is false; if BACK is true, the location returned is that of the last such element. If the array has zero size, or all of the elements of MASK are .FALSE., then the result is an array of zeroes. Similarly, if DIM is supplied and all of the elements of MASK along a given row are zero, the result value for that row is zero.

Parameters
- ARRAY – Shall be an array of type INTEGER, REAL or CHARACTER.
- DIM – (Optional) Shall be a scalar of type INTEGER, with a value between one and the rank of ARRAY, inclusive. It may not be an optional dummy argument.
- MASK – Shall be of type LOGICAL, and conformable with ARRAY.
- KIND – (Optional) An INTEGER initialization expression indicating the kind parameter of the result.
- BACK – (Optional) A scalar of type LOGICAL.

Returns If DIM is absent, the result is a rank-one array with a length equal to the rank of ARRAY. If DIM is present, the result is an array with a rank one less than the rank of ARRAY, and a size corresponding to the size of ARRAY with the DIM dimension removed. If DIM is present and ARRAY has a rank of one, the result is a scalar. If the optional argument KIND is present, the result is an integer of kind KIND, otherwise it is of default kind.

Standard: Fortran 90 and later; ARRAY of CHARACTER and the KIND argument are available in Fortran 2003 and later. The BACK argument is available in Fortran 2008 and later.

Class: Transformational function

Syntax:

<table>
<thead>
<tr>
<th>Result</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESULT = MINLOC(ARRAY, DIM [, MASK] [,KIND] [,BACK])</td>
<td></td>
</tr>
<tr>
<td>RESULT = MINLOC(ARRAY [, MASK], [,KIND] [,BACK])</td>
<td></td>
</tr>
</tbody>
</table>

See also: FINDLOC, MIN, MINVAL

10.200 MINVAL — Minimum value of an array

MINVAL ()
Determines the minimum value of the elements in an array value, or, if the DIM argument is supplied, determines the minimum value along each row of the array in the DIM direction. If MASK is present, only the elements for which MASK is .TRUE. are considered. If the array has zero size, or all of the elements of MASK are .FALSE., then the result is HUGE(ARRAY) if ARRAY is numeric, or a string of CHAR(255) characters if ARRAY is of character type.

Parameters
- ARRAY – Shall be an array of type INTEGER or REAL.
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

- **DIM** – (Optional) Shall be a scalar of type `INTEGER`, with a value between one and the rank of `ARRAY`, inclusive. It may not be an optional dummy argument.

- **MASK** – Shall be of type `LOGICAL`, and conformable with `ARRAY`.

**Returns** If `DIM` is absent, or if `ARRAY` has a rank of one, the result is a scalar. If `DIM` is present, the result is an array with a rank one less than the rank of `ARRAY`, and a size corresponding to the size of `ARRAY` with the `DIM` dimension removed. In all cases, the result is of the same type and kind as `ARRAY`.

**Standard**: Fortran 90 and later

**Class**: Transformational function

**Syntax**:

```
RESULT = MINVAL(ARRAY, DIM [, MASK])
RESULT = MINVAL(ARRAY [, MASK])
```

**See also**: MIN, MINLOC

### 10.201 MOD — Remainder function

**MOD** *(A, P)*

**MOD**(A, P) computes the remainder of the division of A by P.

**Parameters**

- **A** – Shall be a scalar of type `INTEGER` or `REAL`.

- **P** – Shall be a scalar of the same type and kind as A and not equal to zero. (As a GNU extension, arguments of different kinds are permitted.)

**Returns** The return value is the result of \( A - (\text{INT}(A/P) \times P) \). The type and kind of the return value is the same as that of the arguments. The returned value has the same sign as A and a magnitude less than the magnitude of P. (As a GNU extension, kind is the largest kind of the actual arguments.)

**Standard**: Fortran 77 and later, has overloads that are GNU extensions

**Class**: Elemental function

**Syntax**:

```
RESULT = MOD(A, P)
```

**Example**:

```fortran
program test_mod
  print *, mod(17, 3)
  print *, mod(17.5, 5.5)
  print *, mod(17.5d0, 5.5)
  print *, mod(17.5, 5.5d0)
  print *, mod(-17, 3)
  print *, mod(-17.5, 5.5)
```

(continues on next page)
Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOD(A,P)</td>
<td>INTEGER A,P</td>
<td>INTEGER</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>AMOD(A,P)</td>
<td>REAL(4) A,P</td>
<td>REAL(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>DMOD(A,P)</td>
<td>REAL(8) A,P</td>
<td>REAL(8)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>BMOD(A,P)</td>
<td>INTEGER(1) A,P</td>
<td>INTEGER(1)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>IMOD(A,P)</td>
<td>INTEGER(2) A,P</td>
<td>INTEGER(2)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>JMOD(A,P)</td>
<td>INTEGER(4) A,P</td>
<td>INTEGER(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>KMOD(A,P)</td>
<td>INTEGER(8) A,P</td>
<td>INTEGER(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

See also: MODULO

10.202 MODULO — Modulo function

MODULO(A, P)

MODULO(A, P) computes the A modulo P.

Parameters

- A – Shall be a scalar of type INTEGER or REAL.
- P – Shall be a scalar of the same type and kind as A. It shall not be zero. (As a GNU extension, arguments of different kinds are permitted.)

Returns The type and kind of the result are those of the arguments. (As a GNU extension, kind is the largest kind of the actual arguments.)

Standard: Fortran 95 and later

Class: Elemental function

Syntax:

```
RESULT = MODULO(A, P)
```

Example:

```
program test_modulo
  print *, modulo(17, 3)
  print *, modulo(17.5, 5.5)
  print *, modulo(-17, 3)
  print *, modulo(-17.5, 5.5)
end program test_modulo
```
print *, modulo(17, -3)
print *, modulo(17.5, -5.5)
end program

See also: MOD

10.203 MOVE_ALLOC — Move allocation from one object to another

MOVE_ALLOC (FROM, TO)

MOVE_ALLOC (FROM, TO) moves the allocation from FROM to TO. FROM will become deallocated in the process.

Parameters

- FROM – ALLOCATABLE, INTENT(INOUT), may be of any type and kind.
- TO – ALLOCATABLE, INTENT(OUT), shall be of the same type, kind and rank as FROM.

Returns None

Standard: Fortran 2003 and later

Class: Pure subroutine

Syntax:

CALL MOVE_ALLOC(FROM, TO)

Example:

program test_move_alloc
  integer, allocatable :: a(:), b(:)
  allocate(a(3))
a = [ 1, 2, 3 ]
call move_alloc(a, b)
print *, allocated(a), allocated(b)
print *, b
end program test_move_alloc

10.204 MVBITS — Move bits from one integer to another

MVBITS()

Moves LEN bits from positions FROMPOS through FROMPOS+LEN-1 of FROM to positions TOPOS through TOPOS+LEN-1 of TO. The portion of argument TO not affected by the movement of bits is unchanged. The values of FROMPOS+LEN-1 and TOPOS+LEN-1 must be less than BIT_SIZE(FROM).

Parameters

- FROM – The type shall be INTEGER.
- FROMPOS – The type shall be INTEGER.
• **LEN** – The type shall be **INTEGER**.
• **TO** – The type shall be **INTEGER**, of the same kind as **FROM**.
• **TOPOS** – The type shall be **INTEGER**.

**Standard:** Fortran 90 and later, has overloads that are GNU extensions

**Class:** Elemental subroutine

**Syntax:**

```fortran
CALL MVBITS(FROM, FROMPOS, LEN, TO, TOPOS)
```

**Specific names:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVBITS(A)</td>
<td>INTEGER A</td>
<td>INTEGER</td>
<td>Fortran 90 and later</td>
</tr>
<tr>
<td>BMVBITS(A)</td>
<td>INTEGER(1) A</td>
<td>INTEGER(1)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>IMVBITS(A)</td>
<td>INTEGER(2) A</td>
<td>INTEGER(2)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>JMVBITS(A)</td>
<td>INTEGER(4) A</td>
<td>INTEGER(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>KMVBITS(A)</td>
<td>INTEGER(8) A</td>
<td>INTEGER(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

**See also:** IBCLR, IBSET, IBITS, IAND, IOR, IEO

### 10.205 NEAREST — Nearest representable number

**NEAREST** *(X, S)*

**NEAREST**(X, S) returns the processor-representable number nearest to X in the direction indicated by the sign of S.

**Parameters**

- **X** – Shall be of type **REAL**.
- **S** – Shall be of type **REAL** and not equal to zero.

**Returns** The return value is of the same type as X. If S is positive, NEAREST returns the processor-representable number greater than X and nearest to it. If S is negative, NEAREST returns the processor-representable number smaller than X and nearest to it.

**Standard:** Fortran 90 and later

**Class:** Elemental function

**Syntax:**

```fortran
RESULT = NEAREST(X, S)
```

**Example:**

```fortran
program test_nearest
real :: x, y
x = nearest(42.0, 1.0)
y = nearest(42.0, -1.0)
```

(continues on next page)
10.206 NEW_LINE — New line character

NEW_LINE(C)
NEW_LINE(C) returns the new-line character.

Parameters C – The argument shall be a scalar or array of the type CHARACTER.

Returns Returns a CHARACTER scalar of length one with the new-line character of the same kind as parameter C.

Standard: Fortran 2003 and later
Class: Inquiry function
Syntax:

RESULT = NEW_LINE(C)

Example:

program newline
  implicit none
  write(*,'(A)') 'This is record 1.'//NEW_LINE('A')//'This is record 2.'
end program newline

10.207 NINT — Nearest whole number

NINT(A)
NINT(A) rounds its argument to the nearest whole number.

Parameters

• A – The type of the argument shall be REAL.
• KIND – (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

Returns Returns A with the fractional portion of its magnitude eliminated by rounding to the nearest whole number and with its sign preserved, converted to an INTEGER of the default kind.

Standard: Fortran 77 and later, with KIND argument Fortran 90 and later
Class: Elemental function
Syntax:

RESULT = NINT(A [, KIND])

Example:
program test_nint
  real(4) x4
  real(8) x8
  x4 = 1.234E0_4
  x8 = 4.321_8
  print *, nint(x4), idnint(x8)
end program test_nint

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return Type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>NINT(A)</td>
<td>REAL(4) A</td>
<td>INTEGER</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>IDNINT(A)</td>
<td>REAL(8) A</td>
<td>INTEGER</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

See also: CEILING, FLOOR

10.208 NORM2 — Euclidean vector norms

NORM2()

Calculates the Euclidean vector norm (L_2 norm) of ARRAY along dimension DIM.

Parameters

- ARRAY – Shall be an array of type REAL
- DIM – (Optional) shall be a scalar of type INTEGER with a value in the range from 1 to n, where n equals the rank of ARRAY.

Returns The result is of the same type as ARRAY.

Standard: Fortran 2008 and later

Class: Transformational function

Syntax:

RESULT = NORM2(ARRAY[, DIM])

Example:

PROGRAM test_sum
  REAL :: x(5) = [ real :: 1, 2, 3, 4, 5 ]
  print *, NORM2(x) ! = sqrt(55.) ~ 7.416
END PROGRAM
10.209 **NOT — Logical negation**

**NOT ()**

NOT returns the bitwise Boolean inverse of \( I \).

**Parameters** \( I \) – The type shall be INTEGER.

**Returns** The return type is INTEGER, of the same kind as the argument.

**Standard**: Fortran 90 and later, has overloads that are GNU extensions

**Class**: Elemental function

**Syntax**:

\[
\text{RESULT} = \text{NOT}(I)
\]

**Specific names**:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT(A)</td>
<td>INTEGER A</td>
<td>INTEGER</td>
<td>Fortran 95 and later</td>
</tr>
<tr>
<td>BNOT(A)</td>
<td>INTEGER(1) A</td>
<td>INTEGER(1)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>INOT(A)</td>
<td>INTEGER(2) A</td>
<td>INTEGER(2)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>JNOT(A)</td>
<td>INTEGER(4) A</td>
<td>INTEGER(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>KNOT(A)</td>
<td>INTEGER(8) A</td>
<td>INTEGER(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

**See also**: IAND, Ieor, IOR, IBITS, IBSET, IBCLR

10.210 **NULL — Function that returns an disassociated pointer**

**NULL ()**

Returns a disassociated pointer.

**Parameters** \( MOLD \) – (Optional) shall be a pointer of any association status and of any type.

**Returns** A disassociated pointer.

**Standard**: Fortran 95 and later

**Class**: Transformational function

**Syntax**:

\[
\text{PTR} \rightarrow \text{NULL}( \text{MOLD} )
\]

**Example**:

\[
\text{REAL, POINTER, DIMENSION}(1) :: VEC \rightarrow \text{NULL}()
\]

**See also**: ASSOCIATED
10.211 NUM_IMAGES — Function that returns the number of images

NUM_IMAGES()

Returns the number of images.

Parameters

- DISTANCE – (optional, intent(in)) Nonnegative scalar integer
- FAILED – (optional, intent(in)) Scalar logical expression

Returns

Scalar default-kind integer. If DISTANCE is not present or has value 0, the number of images in the current team is returned. For values smaller or equal distance to the initial team, it returns the number of images index on the ancestor team which has a distance of DISTANCE from the invoking team. If DISTANCE is larger than the distance to the initial team, the number of images of the initial team is returned. If FAILED is not present the total number of images is returned; if it has the value .TRUE., the number of failed images is returned, otherwise, the number of images which do have not the failed status.

Standard: Fortran 2008 and later. With DISTANCE or FAILED argument, Technical Specification (TS) 18508 or later

Class: Transformational function

Syntax:

RESULT = NUM_IMAGES(DISTANCE, FAILED)

Example:

```
INTEGER :: value[*]
INTEGER :: i
value = THIS_IMAGE()
SYNC ALL
IF (THIS_IMAGE() == 1) THEN
  DO i = 1, NUM.Images()
    WRITE(*, '(2(a,i0))') 'value[, , i] is ', value[i]
  END DO
END IF
```

See also: THIS_IMAGE, IMAGE_INDEX

10.212 OR — Bitwise logical OR

OR()

Bitwise logical OR.

Parameters

- I – The type shall be either a scalar INTEGER type or a scalar LOGICAL type or a boz-literal-constant.
- J – The type shall be the same as the type of I or a boz-literal-constant. I and J shall not both be boz-literal-constants. If either I and J is a boz-literal-constant, then the other argument must be a scalar INTEGER.
Returns: The return type is either a scalar INTEGER or a scalar LOGICAL. If the kind type parameters
differ, then the smaller kind type is implicitly converted to larger kind, and the return has the larger
kind. A boz-literal-constant is converted to an INTEGER with the kind type parameter of the other
argument as-if a call to INT occurred.

Standard: GNU extension

Class: Function

Syntax:

\[
\text{RESULT} = \text{OR}(I, J)
\]

Example:

```fortran
PROGRAM test_or
   LOGICAL :: T = .TRUE., F = .FALSE.
   INTEGER :: a, b
   DATA a / '{'F'}/, b / '{'3'}/
   WRITE (*,*) OR(T, T), OR(T, F), OR(F, T), OR(F, F)
   WRITE (*,*) OR(a, b)
END PROGRAM
```

See also: Fortran 95 elemental function: IOR

10.213 PACK — Pack an array into an array of rank one

PACK()
Stores the elements of \text{ARRAY} in an array of rank one.

Parameters

- \text{ARRAY} – Shall be an array of any type.
- \text{MASK} – Shall be an array of type \text{LOGICAL} and of the same size as \text{ARRAY}. Alternatively, it
  may be a \text{LOGICAL} scalar.
- \text{VECTOR} – (Optional) shall be an array of the same type as \text{ARRAY} and of rank one. If
  present, the number of elements in \text{VECTOR} shall be equal to or greater than the number of
  true elements in \text{MASK}. If \text{MASK} is scalar, the number of elements in \text{VECTOR} shall be equal
to or greater than the number of elements in \text{ARRAY}.

Returns: The result is an array of rank one and the same type as that of \text{ARRAY}. If \text{VECTOR} is present,
the result size is that of \text{VECTOR}, the number of \text{TRUE} values in \text{MASK} otherwise.

Standard: Fortran 90 and later

Class: Transformational function

Syntax:

\[
\text{RESULT} = \text{PACK}(\text{ARRAY}, \text{MASK}[,\text{VECTOR}])
\]

Example: Gathering nonzero elements from an array:
PROGRAM test_pack_1
   INTEGER :: m(6)
   m = (/ 1, 0, 0, 0, 5, 0 /)
   WRITE(*, FMT="(6(I0, ' '))") pack(m, m /= 0) ! "1 5"
END PROGRAM

Gathering nonzero elements from an array and appending elements from VECTOR:

PROGRAM test_pack_2
   INTEGER :: m(4)
   m = (/ 1, 0, 0, 2 /)
   ! The following results in "1 2 3 4"
   WRITE(*, FMT="(4(I0, ' '))") pack(m, m /= 0, (/ 0, 0, 3, 4 /))
END PROGRAM

See also: UNPACK

10.214 PARITY — Reduction with exclusive OR

PARITY()
Calculates the parity, i.e. the reduction using .XOR. of MASK along dimension DIM.

Parameters

• MASK – Shall be an array of type LOGICAL
• DIM – (Optional) shall be a scalar of type INTEGER with a value in the range from 1 to n, where n equals the rank of MASK.

Returns The result is of the same type as MASK.

Standard: Fortran 2008 and later
Class: Transformational function

Syntax:

RESULT = PARITY(MASK[, DIM])

Example:

PROGRAM test_sum
   LOGICAL :: x(2) = [ .true., .false. ]
   print *, PARITY(x) ! prints "T" (true).
END PROGRAM
10.215  PERROR — Print system error message

PERROR()  
Prints (on the C stderr stream) a newline-terminated error message corresponding to the last system error. This is prefixed by STRING, a colon and a space. See perror(3).

Parameters STRING — A scalar of type CHARACTER and of the default kind.

Standard: GNU extension
Class: Subroutine
Syntax:

CALL PERROR(STRING)

See also: IERRNO

10.216  POPCNT — Number of bits set

POPCNT(I)  
POPCNT(I) returns the number of bits set ('1' bits) in the binary representation of I.

Parameters I — Shall be of type INTEGER.

Returns The return value is of type INTEGER and of the default integer kind.

Standard: Fortran 2008 and later
Class: Elemental function
Syntax:

RESULT = POPCNT(I)

Example:

program test_population
   print *, popcnt(127), poppar(127)
   print *, popcnt(huge(0_4)), poppar(huge(0_4))
   print *, popcnt(huge(0_8)), poppar(huge(0_8))
end program test_population

See also: POPPAR, LEADZ, TRAILZ
10.217 POPPAR — Parity of the number of bits set

POPPAR(I)

POPPAR(I) returns parity of the integer I, i.e. the parity of the number of bits set (‘1’ bits) in the binary representation of I. It is equal to 0 if I has an even number of bits set, and 1 for an odd number of ‘1’ bits.

Parameters I – Shall be of type INTEGER.

Returns The return value is of type INTEGER and of the default integer kind.

Standard: Fortran 2008 and later

Class: Elemental function

Syntax:

RESULT = POPPAR(I)

Example:

```
program test_population
  print *, popcnt(127), poppar(127)
  print *, popcnt(huge(0_4)), poppar(huge(0_4))
  print *, popcnt(huge(0_8)), poppar(huge(0_8))
end program test_population
```

See also: POPCNT, LEADZ, TRAILZ

10.218 PRECISION — Decimal precision of a real kind

PRECISION(X)

PRECISION(X) returns the decimal precision in the model of the type of X.

Parameters X – Shall be of type REAL or COMPLEX. It may be scalar or valued.

Returns The return value is of type INTEGER and of the default integer kind.

Standard: Fortran 90 and later

Class: Inquiry function

Syntax:

RESULT = PRECISION(X)

Example:

```
program prec_and_range
  real(kind=4) :: x(2)
  complex(kind=8) :: y
  
  print *, precision(x), range(x)
  print *, precision(y), range(y)
end program prec_and_range
```
**See also:** SELECTED_REAL_KIND, RANGE

### 10.219 PRESENT — Determine whether an optional dummy argument is specified

**PRESENT ()**
Determines whether an optional dummy argument is present.

**Parameters**
- **A** – May be of any type and may be a pointer, scalar or array value, or a dummy procedure. It shall be the name of an optional dummy argument accessible within the current subroutine or function.

**Returns**
Returns either **TRUE** if the optional argument *A* is present, or **FALSE** otherwise.

**Standard:** Fortran 90 and later

**Class:** Inquiry function

**Syntax:**

```fortran
RESULT = PRESENT(A)
```

**Example:**

```fortran
PROGRAM test_present
  WRITE(*,*), f(), f(42) ! "F T"
END PROGRAM
 CONTAINS
  LOGICAL FUNCTION f(x)
    INTEGER, INTENT(IN), OPTIONAL :: x
    f = PRESENT(x)
  END FUNCTION
END PROGRAM
```

### 10.220 PRODUCT — Product of array elements

**PRODUCT ()**
Multiplies the elements of **ARRAY** along dimension **DIM** if the corresponding element in **MASK** is **TRUE**.

**Parameters**

- **ARRAY** – Shall be an array of type **INTEGER**, **REAL** or **COMPLEX**.
- **DIM** – (Optional) shall be a scalar of type **INTEGER** with a value in the range from 1 to *n*, where *n* equals the rank of **ARRAY**.
- **MASK** – (Optional) shall be of type **LOGICAL** and either be a scalar or an array of the same shape as **ARRAY**.

**Returns**
The result is of the same type as **ARRAY**.

**Standard:** Fortran 90 and later

**Class:** Transformational function

**Syntax:**

```fortran
RESULT = PRODUCT(ARRAY [, DIM [, MASK]])
```
RESULT = PRODUCT ARRAY[, MASK])
RESULT = PRODUCT ARRAY, DIM[, MASK])

Example:

PROGRAM test_product
INTEGER :: x(5) = (/ 1, 2, 3, 4, 5 /)
print *, PRODUCT(x) ! all elements, product = 120
print *, PRODUCT(x, MASK=MOD(x, 2)==1) ! odd elements, product = 15
END PROGRAM

See also: SUM

10.221 RADIX — Base of a model number

RADIX(X)
RADIX(X) returns the base of the model representing the entity X.

Parameters X – Shall be of type INTEGER or REAL

Returns The return value is a scalar of type INTEGER and of the default integer kind.

Standard: Fortran 90 and later

Class: Inquiry function

Syntax:

RESULT = RADIX(X)

Example:

program test_radix
print *, "The radix for the default integer kind is", radix(0)
print *, "The radix for the default real kind is", radix(0.0)
end program test_radix

See also: SELECTED_REAL_KIND

10.222 RAN — Real pseudo-random number

RAN()
For compatibility with HP FORTRAN 77/iX, the RAN intrinsic is provided as an alias for RAND. See RAND for complete documentation.

Standard: GNU extension

Class: Function

See also: RAND, RANDOM_NUMBER
10.223 RAND — Real pseudo-random number

RAND(FLAG)
RAND(FLAG) returns a pseudo-random number from a uniform distribution between 0 and 1. If FLAG is 0, the next number in the current sequence is returned; if FLAG is 1, the generator is restarted by CALL SRAND(0); if FLAG has any other value, it is used as a new seed with SRAND.

Parameters
I – Shall be a scalar INTEGER of kind 4.

Returns
The return value is of REAL type and the default kind.

Standard: GNU extension

Class: Function

Syntax:
RESULT = RAND(I)

Example:

```fortran
program test_rand
   integer, parameter :: seed = 86456

   call srand(seed)
   print *, rand(), rand(), rand(), rand()
   print *, rand(seed), rand(), rand(), rand()
end program test_rand
```

See also: SRAND, RANDOM_NUMBER

10.224 RANDOM_INIT — Initialize a pseudo-random number generator

RANDOM_INIT()

Initializes the state of the pseudorandom number generator used by RANDOM_NUMBER.

Parameters

- **REPEATABLE** – Shall be a scalar with a LOGICAL type, and it is INTENT(IN). If it is .true., the seed is set to a processor-dependent value that is the same each time RANDOM_INIT is called from the same image. The term ‘same image’ means a single instance of program execution. The sequence of random numbers is different for repeated execution of the program. If it is .false., the seed is set to a processor-dependent value.

- **IMAGE_DISTINCT** – Shall be a scalar with a LOGICAL type, and it is INTENT(IN). If it is .true., the seed is set to a processor-dependent value that is distinct from the seed set by a call to RANDOM_INIT in another image. If it is .false., the seed is set to a value that does depend which image called RANDOM_INIT.

Standard: Fortran 2018

Class: Subroutine

Syntax:
CALL RANDOM_INIT(REPEATABLE, IMAGE_DISTINCT)

Example:

```
program test_random_seed
  implicit none
  real x(3), y(3)
  call random_init(.true., .true.)
  call random_number(x)
  call random_init(.true., .true.)
  call random_number(y)
  ! x and y are the same sequence
  if (any(x /= y)) call abort
end program test_random_seed
```

See also: RANDOM_NUMBER, RANDOM_SEED

10.225 RANDOM_NUMBER — Pseudo-random number

RANDOM_NUMBER()
Returns a single pseudorandom number or an array of pseudorandom numbers from the uniform distribution over the range 0 leq x < 1.

Parameters

HARVEST – Shall be a scalar or an array of type REAL.

Standard: Fortran 90 and later
Class: Subroutine
Syntax:

```
CALL RANDOM_NUMBER(HARVEST)
```

Example:

```
program test_random_number
  REAL :: r(5,5)
  CALL RANDOM_NUMBER(r)
end program
```

See also: RANDOM_SEED, RANDOM_INIT

10.226 RANDOM_SEED — Initialize a pseudo-random number sequence

RANDOM_SEED()
Restarts or queries the state of the pseudorandom number generator used by RANDOM_NUMBER.

Parameters

- SIZE – (Optional) Shall be a scalar and of type default INTEGER, with INTENT(OUT). It specifies the minimum size of the arrays used with the PUT and GET arguments.
• **PUT** – (Optional) Shall be an array of type default INTEGER and rank one. It is IN-TENT(IN) and the size of the array must be larger than or equal to the number returned by the **SIZE** argument.

• **GET** – (Optional) Shall be an array of type default INTEGER and rank one. It is IN-TENT(OUT) and the size of the array must be larger than or equal to the number returned by the **SIZE** argument.

**Standard**: Fortran 90 and later

**Class**: Subroutine

**Syntax**:

```fortran
CALL RANDOM_SEED([SIZE, PUT, GET])
```

**Example**:

```fortran
program test_random_seed
  implicit none
  integer, allocatable :: seed(:)
  integer :: n

  call random_seed(size = n)
  allocate(seed(n))
  call random_seed(get=seed)
  write (*, *) seed
end program test_random_seed
```

**See also**: RANDOM_NUMBER, RANDOM_INIT

### 10.227 RANGE — Decimal exponent range

**RANGE**(X)

RANGE(X) returns the decimal exponent range in the model of the type of X.

**Parameters** X – Shall be of type INTEGER, REAL or COMPLEX.

**Returns** The return value is of type INTEGER and of the default integer kind.

**Standard**: Fortran 90 and later

**Class**: Inquiry function

**Syntax**:

```fortran
RESULT = RANGE(X)
```

**Example**: See PRECISION for an example.

**See also**: SELECTED_REAL_KIND, PRECISION
10.228 RANK — Rank of a data object

RANK (A)

RANK (A) returns the rank of a scalar or array data object.

Parameters A – can be of any type

Returns The return value is of type INTEGER and of the default integer kind. For arrays, their rank is returned; for scalars zero is returned.

Standard: Technical Specification (TS) 29113

Class: Inquiry function

Syntax:

RESULT = RANK(A)

Example:

program test_rank
  integer :: a
  real, allocatable :: b(:,:)

  print *, rank(a), rank(b) ! Prints: 0 2
end program test_rank

10.229 REAL — Convert to real type

REAL (A[, KIND])

REAL (A[, KIND]) converts its argument A to a real type. The REALPART function is provided for compatibility with g77, and its use is strongly discouraged.

Parameters

• A – Shall be INTEGER, REAL, or COMPLEX.

• KIND – (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

Returns These functions return a REAL variable or array under the following rules:

Standard: Fortran 77 and later, with KIND argument Fortran 90 and later, has GNU extensions

Class: Elemental function

Syntax:

RESULT = REAL (A [, KIND])
RESULT = REALPART (Z)

Example:
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

```fortran
program test_real
  complex :: x = (1.0, 2.0)
  print *, real(x), real(x, 8), realpart(x)
end program test_real
```

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOAT(A)</td>
<td>INTEGER(4)</td>
<td>REAL(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>DFLOAT(A)</td>
<td>INTEGER(4)</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>FLOATI(A)</td>
<td>INTEGER(2)</td>
<td>REAL(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>FLOATJ(A)</td>
<td>INTEGER(4)</td>
<td>REAL(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>FLOATK(A)</td>
<td>INTEGER(4)</td>
<td>REAL(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>SNGL(A)</td>
<td>INTEGER(8)</td>
<td>REAL(4)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

See also: DBLE

10.230 RENAME — Rename a file

RENAME (

 Renames a file from file PATH1 to PATH2. A null character (CHAR(0)) can be used to mark the end of the names in PATH1 and PATH2; otherwise, trailing blanks in the file names are ignored. If the STATUS argument is supplied, it contains 0 on success or a nonzero error code upon return; see rename(2).

Parameters

- **PATH1** – Shall be of default CHARACTER type.
- **PATH2** – Shall be of default CHARACTER type.
- **STATUS** – (Optional) Shall be of default INTEGER type.

Standard: GNU extension

Class: Subroutine, function

Syntax:

```fortran
CALL RENAME(PATH1, PATH2 [, STATUS])
STATUS = RENAME(PATH1, PATH2)
```

See also: LINK
**10.231 REPEAT — Repeated string concatenation**

REPEAT ()
Concatenates *NCOPIES* copies of a string.

Parameters
- **STRING** – Shall be scalar and of type CHARACTER.
- **NCOPIES** – Shall be scalar and of type INTEGER.

Returns A new scalar of type CHARACTER built up from *NCOPIES* copies of **STRING**.

*Standard:* Fortran 90 and later

*Class:* Transformational function

*Syntax:*

\[
\text{RESULT} = \text{REPEAT}(\text{STRING}, \text{NCOPIES})
\]

*Example:*

```fortran
program test_repeat
  write(*,*) repeat("x", 5) ! "xxxxx"
end program
```

**10.232 RESHAPE — Function to reshape an array**

RESHAPE ()
Reshapes **SOURCE** to correspond to **SHAPE**. If necessary, the new array may be padded with elements from **PAD** or permuted as defined by **ORDER**.

Parameters
- **SOURCE** – Shall be an array of any type.
- **SHAPE** – Shall be of type INTEGER and an array of rank one. Its values must be positive or zero.
- **PAD** – (Optional) shall be an array of the same type as **SOURCE**.
- **ORDER** – (Optional) shall be of type INTEGER and an array of the same shape as **SHAPE**. Its values shall be a permutation of the numbers from 1 to n, where n is the size of **SHAPE**. If **ORDER** is absent, the natural ordering shall be assumed.

Returns The result is an array of shape **SHAPE** with the same type as **SOURCE**.

*Standard:* Fortran 90 and later

*Class:* Transformational function

*Syntax:*

\[
\text{RESULT} = \text{RESHAPE}(\text{SOURCE}, \text{SHAPE}[, \text{PAD}, \text{ORDER}])
\]

*Example:*
PROGRAM test_reshape
  INTEGER, DIMENSION(4) :: x
  WRITE(*,*) SHAPE(x) ! prints "4"
  WRITE(*,*) SHAPE(RESHAPE(x, (/2, 2/))) ! prints "2 2"
END PROGRAM

See also: SHAPE

10.233 RRSPACING — Reciprocal of the relative spacing

RRSPACING(X)

RRSPACING(X) returns the reciprocal of the relative spacing of model numbers near X.

Parameters X – Shall be of type REAL.

Returns The return value is of the same type and kind as X. The value returned is equal to ABS(FRACTION(X)) * FLOAT(RADIX(X))**DIGITS(X).

Standard: Fortran 90 and later

Class: Elemental function

Syntax:

RESULT = RRSPACING(X)

See also: SPACING

10.234 RSHIFT — Right shift bits

RSHIFT()

RSHIFT returns a value corresponding to I with all of the bits shifted right by SHIFT places. SHIFT shall be nonnegative and less than or equal to BIT_SIZE(I), otherwise the result value is undefined. Bits shifted out from the right end are lost. The fill is arithmetic: the bits shifted in from the left end are equal to the leftmost bit, which in two’s complement representation is the sign bit.

Parameters

• I – The type shall be INTEGER.
• SHIFT – The type shall be INTEGER.

Returns The return value is of type INTEGER and of the same kind as I.

Standard: GNU extension

Class: Elemental function

Syntax:

RESULT = RSHIFT(I, SHIFT)

See also: ISHFT, ISHFTC, LSHIFT, SHIFTA, SHIFTR, SHIFTL
10.235 SAME_TYPE_AS — Query dynamic types for equality

SAME_TYPE_AS()
Query dynamic types for equality.

Parameters
- A – Shall be an object of extensible declared type or unlimited polymorphic.
- B – Shall be an object of extensible declared type or unlimited polymorphic.

Returns The return value is a scalar of type default logical. It is true if and only if the dynamic type of A is the same as the dynamic type of B.

Standard: Fortran 2003 and later
Class: Inquiry function
Syntax:
RESULT = SAME_TYPE_AS(A, B)

See also: EXTENDS_TYPE_OF

10.236 SCALE — Scale a real value

SCALE(X, I)
SCALE(X, I) returns $X \times \text{RADIX}(X)^{**I}$.

Parameters
- X – The type of the argument shall be a REAL.
- I – The type of the argument shall be a INTEGER.

Returns The return value is of the same type and kind as X. Its value is $X \times \text{RADIX}(X)^{**I}$.

Standard: Fortran 90 and later
Class: Elemental function
Syntax:
RESULT = SCALE(X, I)

Example:
program test_scale
real :: x = 178.1387e-4
integer :: i = 5
print *, scale(x, i), x*radix(x)**i
end program test_scale
10.237 SCAN — Scan a string for the presence of a set of characters

SCAN()  
Scans a STRING for any of the characters in a SET of characters.

Parameters
• STRING – Shall be of type CHARACTER.
• SET – Shall be of type CHARACTER.
• BACK – (Optional) shall be of type LOGICAL.
• KIND – (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

Returns  The return value is of type INTEGER and of kind KIND. If KIND is absent, the return value is of default integer kind.

Standard:  Fortran 90 and later, with KIND argument Fortran 2003 and later
Class:  Elemental function
Syntax:

RESULT = SCAN(STRING, SET[, BACK [, KIND]])

Example:

PROGRAM test_scan
  WRITE(*,*) SCAN("FORTRAN", "AO")  ! 2, found 'O'
  WRITE(*,*) SCAN("FORTRAN", "AO", .TRUE.) ! 6, found 'A'
  WRITE(*,*) SCAN("FORTRAN", "C++")  ! 0, found none
END PROGRAM

See also:  INDEX intrinsic, VERIFY

10.238 SECNDS — Time function

SECNDS(X)  
SECNDS(X) gets the time in seconds from the real-time system clock. X is a reference time, also in seconds. If this is zero, the time in seconds from midnight is returned. This function is non-standard and its use is discouraged.

Parameters
• T – Shall be of type REAL(4).
• X – Shall be of type REAL(4).

Returns  None

Standard:  GNU extension
Class:  Function
Syntax:
RESULT = SECNDS (X)

Example:

```fortran
program test_secnds
    integer :: i
    real(4) :: t1, t2
    print *, secnds (0.0)  ! seconds since midnight
    t1 = secnds (0.0)     ! reference time
    do i = 1, 10000000    ! do something
        end do
    t2 = secnds (t1)      ! elapsed time
    print *, "Something took ", t2, " seconds."
end program test_secnds
```

10.239 SECOND — CPU time function

SECOND()

Returns a REAL(4) value representing the elapsed CPU time in seconds. This provides the same functionality as the standard CPU_TIME intrinsic, and is only included for backwards compatibility.

Parameters TIME – Shall be of type REAL(4).

Returns In either syntax, TIME is set to the process's current runtime in seconds.

Standard: GNU extension

Class: Subroutine, function

Syntax:

```
CALL SECOND(TIME)
TIME = SECOND()
```

See also: CPU_TIME

10.240 SELECTED_CHAR_KIND — Choose character kind

SELECTED_CHAR_KIND(NAME)

SELECTED_CHAR_KIND(NAME) returns the kind value for the character set named NAME, if a character set with such a name is supported, or -1 otherwise. Currently, supported character sets include 'ASCII' and 'DEFAULT', which are equivalent, and 'ISO_10646' (Universal Character Set, UCS-4) which is commonly known as Unicode.

Parameters NAME – Shall be a scalar and of the default character type.

Standard: Fortran 2003 and later

Class: Transformational function

Syntax:
RESULT = SELECTED_CHAR_KIND(NAME)

Example:

```fortran
program character_kind
  use iso_fortran_env
  implicit none
  integer, parameter :: ascii = selected_char_kind("ascii")
  integer, parameter :: ucs4 = selected_char_kind('ISO_10646')
  character(kind=ascii, len=26) :: alphabet
  character(kind=ucs4, len=30) :: hello_world
  alphabet = ascii_"abcdefghijklmnopqrstuvwxyz"
  hello_world = ucs4_"Hello World and Ni Hao -- " &
  // char (int (z'4F60'), ucs4) &
  // char (int (z'597D'), ucs4)
  write (*,*) alphabet
  open (output_unit, encoding='UTF-8')
  write (*,*) trim (hello_world)
end program character_kind
```

10.241 SELECTED_INT_KIND — Choose integer kind

SELECTED_INT_KIND(R)

SELECTED_INT_KIND(R) return the kind value of the smallest integer type that can represent all values ranging from \(-10^R\) (exclusive) to \(10^R\) (exclusive). If there is no integer kind that accommodates this range, SELECTED_INT_KIND returns \(-1\).

Parameters R – Shall be a scalar and of type INTEGER.

Standard: Fortran 90 and later

Class: Transformational function

Syntax:

```
RESULT = SELECTED_INT_KIND(R)
```

Example:

```fortran
program large_integers
  integer, parameter :: k5 = selected_int_kind(5)
  integer, parameter :: k15 = selected_int_kind(15)
  integer(kind=k5) :: i5
  integer(kind=k15) :: i15
  print *, huge(i5), huge(i15)
  ! The following inequalities are always true
  print *, huge(i5) >= 10_k5**5-1
  print *, huge(i15) >= 10_k15**15-1
end program large_integers
```
10.242 SELECTED_REAL_KIND — Choose real kind

SELECTED_REAL_KIND (P, R)

SELECTED_REAL_KIND (P, R) returns the kind value of a real data type with decimal precision of at least \( P \) digits, exponent range of at least \( R \), and with a radix of \( \text{RADIX} \).

Parameters

- \( P \) – (Optional) shall be a scalar and of type INTEGER.
- \( R \) – (Optional) shall be a scalar and of type INTEGER.
- \( \text{RADIX} \) – (Optional) shall be a scalar and of type INTEGER.

Returns

SELECTED_REAL_KIND returns the value of the kind type parameter of a real data type with decimal precision of at least \( P \) digits, a decimal exponent range of at least \( R \), and with the requested \( \text{RADIX} \). If the \( \text{RADIX} \) parameter is absent, real kinds with any radix can be returned. If more than one real data type meet the criteria, the kind of the data type with the smallest decimal precision is returned. If no real data type matches the criteria, the result is standard:

Fortran 90 and later, with \( \text{RADIX} \) Fortran 2008 or later

Class:
Transformational function

Syntax:

\[
\text{RESULT} = \text{SELECTED_REAL_KIND}(\{P, R, \text{RADIX}\})
\]

Example:

```fortran
program real_kinds
  integer, parameter :: p6 = selected_real_kind(6)
  integer, parameter :: p10r100 = selected_real_kind(10, 100)
  integer, parameter :: r400 = selected_real_kind(r=400)
  real(kind=p6) :: x
  real(kind=p10r100) :: y
  real(kind=r400) :: z
  print *, precision(x), range(x)
  print *, precision(y), range(y)
  print *, precision(z), range(z)
end program real_kinds
```

See also:
PRECISION, RANGE, RADIX

10.243 SET_EXPONENT — Set the exponent of the model

SET_EXPONENT (X, I)

SET_EXPONENT (X, I) returns the real number whose fractional part is that of \( X \) and whose exponent part is \( I \).

Parameters

- \( X \) – Shall be of type REAL.
- \( I \) – Shall be of type INTEGER.
Returns: The return value is of the same type and kind as \( X \). The real number whose fractional part is that of \( X \) and whose exponent part if \( I \) is returned; it is \( \text{FRACTION}(X) \times \text{RADIX}(X)^\ast I \).

**Standard:** Fortran 90 and later

**Class:** Elemental function

**Syntax:**

\[
\text{RESULT} = \text{SET\_EXPONENT}(X, I)
\]

**Example:**

```fortran
PROGRAM test_setexp
  REAL :: x = 178.1387e-4
  INTEGER :: i = 17
  PRINT *, SET\_EXPONENT(x, i), FRACTION(x) \ast RADIX(x)**i
END PROGRAM
```

10.244 SHAPE — Determine the shape of an array

**SHAPE()**

Determines the shape of an array.

**Parameters**

- **SOURCE** – Shall be an array or scalar of any type. If \( \text{SOURCE} \) is a pointer it must be associated and allocatable arrays must be allocated.
- **KIND** – (Optional) An \texttt{INTEGER} initialization expression indicating the kind parameter of the result.

**Returns** An \texttt{INTEGER} array of rank one with as many elements as \( \text{SOURCE} \) has dimensions. The elements of the resulting array correspond to the extend of \( \text{SOURCE} \) along the respective dimensions. If \( \text{SOURCE} \) is a scalar, the result is the rank one array of size zero. If \( \text{KIND} \) is absent, the return value has the default integer kind otherwise the specified kind.

**Standard:** Fortran 90 and later, with \( \text{KIND} \) argument Fortran 2003 and later

**Class:** Inquiry function

**Syntax:**

\[
\text{RESULT} = \text{SHAPE}(\text{SOURCE} [, \text{KIND}])
\]

**Example:**

```fortran
PROGRAM test_shape
  INTEGER, DIMENSION(-1:1, -1:2) :: A
  WRITE(*,*) SHAPE(A) ! (/ 3, 4 /)
  WRITE(*,*) SIZE(SHAPE(42)) ! (/ /)
END PROGRAM
```

**See also:** RESHAPE, SIZE
10.245 SHIFTA — Right shift with fill

SHIFTA()

SHIFTA returns a value corresponding to I with all of the bits shifted right by SHIFT places. SHIFT that be nonnegative and less than or equal to BIT_SIZE(I), otherwise the result value is undefined. Bits shifted out from the right end are lost. The fill is arithmetic: the bits shifted in from the left end are equal to the leftmost bit, which in two's complement representation is the sign bit.

Parameters

• I – The type shall be INTEGER.
• SHIFT – The type shall be INTEGER.

Returns The return value is of type INTEGER and of the same kind as I.

Standard: Fortran 2008 and later

Class: Elemental function

Syntax:

\[
\text{RESULT} = \text{SHIFTA}(I, \text{SHIFT})
\]

See also: SHIFTL, SHIFTR

10.246 SHIFTL — Left shift

SHIFTL()

SHIFTL returns a value corresponding to I with all of the bits shifted left by SHIFT places. SHIFT shall be nonnegative and less than or equal to BIT_SIZE(I), otherwise the result value is undefined. Bits shifted out from the left end are lost, and bits shifted in from the right end are set to 0.

Parameters

• I – The type shall be INTEGER.
• SHIFT – The type shall be INTEGER.

Returns The return value is of type INTEGER and of the same kind as I.

Standard: Fortran 2008 and later

Class: Elemental function

Syntax:

\[
\text{RESULT} = \text{SHIFTL}(I, \text{SHIFT})
\]

See also: SHIFTA, SHIFTR
10.247 SHIFTR — Right shift

**SHIFTR()**

SHIFTR returns a value corresponding to $I$ with all of the bits shifted right by $SHIFT$ places. $SHIFT$ shall be nonnegative and less than or equal to BIT_SIZE($I$), otherwise the result value is undefined. Bits shifted out from the right end are lost, and bits shifted in from the left end are set to 0.

**Parameters**

- $I$ – The type shall be INTEGER.
- $SHIFT$ – The type shall be INTEGER.

**Returns** The return value is of type INTEGER and of the same kind as $I$.

**Standard:** Fortran 2008 and later

**Class:** Elemental function

**Syntax:**

```
RESULT = SHIFTR(I, SHIFT)
```

**See also:** SHIFTA, SHIFTL

10.248 SIGN — Sign copying function

**SIGN(A, B)**

SIGN(A, B) returns the value of $A$ with the sign of $B$.

**Parameters**

- $A$ – Shall be of type INTEGER or REAL
- $B$ – Shall be of the same type and kind as $A$.

**Returns** The kind of the return value is that of $A$ and $B$. If $Bge 0$ then the result is $ABS(A)$, else it is $-ABS(A)$.

**Standard:** Fortran 77 and later

**Class:** Elemental function

**Syntax:**

```
RESULT = SIGN(A, B)
```

**Example:**

```fortran
program test_sign
print *, sign(-12,1)
print *, sign(-12,0)
print *, sign(-12,-1)
print *, sign(-12.,1.)
print *, sign(-12.,0.)
```

(continues on next page)
print *, sign(-12.,-1.)
end program test_sign

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGN(A,B)</td>
<td>REAL(4) A, B</td>
<td>REAL(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>ISIGN(A,B)</td>
<td>INTEGER(4) A, B</td>
<td>INTEGER(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>DSIGN(A,B)</td>
<td>REAL(8) A, B</td>
<td>REAL(8)</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

10.249 SIGNAL — Signal handling subroutine (or function)

SIGNAL(NUMBER, HANDLER[, STATUS])

SIGNAL(NUMBER, HANDLER[, STATUS]) causes external subroutine HANDLER to be executed with a single integer argument when signal NUMBER occurs. If HANDLER is an integer, it can be used to turn off handling of signal NUMBER or revert to its default action. See signal(2).

Parameters

- NUMBER – Shall be a scalar integer, with INTENT(IN)
- HANDLER – Signal handler (INTEGER FUNCTION or SUBROUTINE) or dummy/global INTEGER scalar. It is INTENT(IN).
- STATUS – (Optional) STATUS shall be a scalar integer. It has INTENT(OUT).

Returns The SIGNAL function returns the value returned by signal(2).

Standard: GNU extension

Class: Subroutine, function

Syntax:

CALL SIGNAL(NUMBER, HANDLER[, STATUS])
STATUS = SIGNAL(NUMBER, HANDLER)

Example:

program test_signal
  intrinsic signal
  external handler_print
  call signal (12, handler_print)
  call signal (10, 1)
  call sleep (30)
end program test_signal
10.250 SIN — Sine function

\texttt{SIN} (X)

\texttt{SIN} (X) computes the sine of \( X \).

- **Parameters** \( X \) – The type shall be \texttt{REAL} or \texttt{COMPLEX}.
- **Returns** The return value has same type and kind as \( X \).

**Standard**: Fortran 77 and later

**Class**: Elemental function

**Syntax**:

\begin{verbatim}
RESULT = SIN(X)
\end{verbatim}

**Example**:

\begin{verbatim}
program test_sin
  real :: x = 0.0
  x = sin(x)
end program test_sin
\end{verbatim}

**Specific names**:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIN(X)</td>
<td>\texttt{REAL(4)} X</td>
<td>\texttt{REAL(4)}</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>DSIN(X)</td>
<td>\texttt{REAL(8)} X</td>
<td>\texttt{REAL(8)}</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>CSIN(X)</td>
<td>\texttt{COMPLEX(4)} X</td>
<td>\texttt{COMPLEX(4)}</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>ZSIN(X)</td>
<td>\texttt{COMPLEX(8)} X</td>
<td>\texttt{COMPLEX(8)}</td>
<td>GNU extension</td>
</tr>
<tr>
<td>CDSIN(X)</td>
<td>\texttt{COMPLEX(8)} X</td>
<td>\texttt{COMPLEX(8)}</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

**See also**: Inverse function: \texttt{ASIN}, Degrees function: \texttt{SIND}

10.251 SIND — Sine function, degrees

\texttt{SIND} (X)

\texttt{SIND} (X) computes the sine of \( X \) in degrees.

- **Parameters** \( X \) – The type shall be \texttt{REAL} or \texttt{COMPLEX}.
- **Returns** The return value has same type and kind as \( X \), and its value is in degrees.

**Standard**: GNU extension, enabled with \texttt{-fdec-math}.

**Class**: Elemental function

**Syntax**:

\begin{verbatim}
RESULT = SIND(X)
\end{verbatim}

**Example**:
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

**Specific names:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIND</td>
<td>REAL(4) X</td>
<td>REAL(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>DSIND</td>
<td>REAL(8) X</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>CSIND</td>
<td>COMPLEX(4) X</td>
<td>COMPLEX(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>ZSIND</td>
<td>COMPLEX(8) X</td>
<td>COMPLEX(8)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>CDSIND</td>
<td>COMPLEX(8) X</td>
<td>COMPLEX(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

See also: Inverse function: ASIND

Radians function: SIN

### 10.252 SINH — Hyperbolic sine function

**SINH(X)**

SINH(X) computes the hyperbolic sine of X.

- **Parameters** X — The type shall be REAL or COMPLEX.
- **Returns** The return value has same type and kind as X.

**Standard:** Fortran 90 and later, for a complex argument Fortran 2008 or later, has a GNU extension

**Class:** Elemental function

**Syntax:**

\[
\text{RESULT} = \text{SINH}(X)
\]

**Example:**

```fortran
program test_sinh
real :: x = - 1.0_8
x = sinh(x)
end program test_sinh
```

**Specific names:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSINH</td>
<td>REAL(8) X</td>
<td>REAL(8)</td>
<td>Fortran 90 and later</td>
</tr>
</tbody>
</table>

See also: ASINH
10.253 SIZE — Determine the size of an array

SIZE()
Determine the extent of ARRAY along a specified dimension DIM, or the total number of elements in ARRAY if DIM is absent.

Parameters
• ARRAY – Shall be an array of any type. If ARRAY is a pointer it must be associated and allocatable arrays must be allocated.
• DIM – (Optional) shall be a scalar of type INTEGER and its value shall be in the range from 1 to n, where n equals the rank of ARRAY.
• KIND – (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

Returns The return value is of type INTEGER and of kind KIND. If KIND is absent, the return value is of default integer kind.

Standard: Fortran 90 and later, with KIND argument Fortran 2003 and later

Syntax:
RESULT = SIZE(ARRAY[, DIM [, KIND]])

Example:
PROGRAM test_size
   WRITE(*,*) SIZE((/ 1, 2 /)) ! 2
END PROGRAM

See also: SHAPE, RESHAPE

10.254 SIZEOF — Size in bytes of an expression

SIZEOF(X)
SIZEOF(X) calculates the number of bytes of storage the expression X occupies.

Parameters X – The argument shall be of any type, rank or shape.

Returns The return value is of type integer and of the system-dependent kind C_SIZE_T (from the ISO_C_BINDING module). Its value is the number of bytes occupied by the argument. If the argument has the POINTER attribute, the number of bytes of the storage area pointed to is returned. If the argument is of a derived type with POINTER or ALLOCATABLE components, the return value does not account for the sizes of the data pointed to by these components. If the argument is polymorphic, the size according to the dynamic type is returned. The argument may not be a procedure or procedure pointer. Note that the code assumes for arrays that those are contiguous; for contiguous arrays, it returns the storage or an array element multiplied by the size of the array.

Standard: GNU extension

Class: Inquiry function
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

Syntax:

\[ N = \text{SIZEOF}(X) \]

Example:

```fortran
integer :: i
real :: r, s(5)
print *, (sizeof(s)/sizeof(r) == 5)
end
```

The example will print `.TRUE.` unless you are using a platform where default REAL variables are unusually padded.

See also: C_SIZEOF, STORAGE_SIZE

10.255 SLEEP — Sleep for the specified number of seconds

SLEEP ()
Calling this subroutine causes the process to pause for SECONDS seconds.

Parameters SECONDS – The type shall be of default INTEGER.

Standard: GNU extension
Class: Subroutine
Syntax:

```
CALL SLEEP(SECONDS)
```

Example:

```fortran
program test_sleep
    call sleep(5)
end
```

10.256 SPACING — Smallest distance between two numbers of a given type

SPACING ()
Determines the distance between the argument X and the nearest adjacent number of the same type.

Parameters X – Shall be of type REAL.

Returns The result is of the same type as the input argument X.

Standard: Fortran 90 and later
Class: Elemental function
Syntax:
RESULT = SPACING(X)

Example:

```
PROGRAM test_spacing
  INTEGER, PARAMETER :: SGL = SELECTED_REAL_KIND(p=6, r=37)
  INTEGER, PARAMETER :: DBL = SELECTED_REAL_KIND(p=13, r=200)

  WRITE(*,*) spacing(1.0_SGL)  ! "1.1920929E-07" on i686
  WRITE(*,*) spacing(1.0_DBL)  ! "2.220446049250313E-016" on i686
END PROGRAM
```

See also: RRSPACING

10.257 SPREAD — Add a dimension to an array

SPREAD ()
Replicates a SOURCE array NCOPIES times along a specified dimension DIM.

Parameters

- SOURCE – Shall be a scalar or an array of any type and a rank less than seven.
- DIM – Shall be a scalar of type INTEGER with a value in the range from 1 to n+1, where n equals the rank of SOURCE.
- NCOPIES – Shall be a scalar of type INTEGER.

Returns
The result is an array of the same type as SOURCE and has rank n+1 where n equals the rank of SOURCE.

Standard: Fortran 90 and later

Class: Transformational function

Syntax:

```
RESULT = SPREAD(SOURCE, DIM, NCOPIES)
```

Example:

```
PROGRAM test_spread
  INTEGER :: a = 1, b(2) = (/ 1, 2 /)
  WRITE(*,*) SPREAD(a, 1, 2)      ! "1 1"
  WRITE(*,*) SPREAD(b, 1, 2)      ! "1 1 2 2"
END PROGRAM
```

See also: UNPACK
10.258 SQRT — Square-root function

**SQRT (X)**

SQRT (X) computes the square root of X.

**Parameters**  
X – The type shall be REAL or COMPLEX.

**Returns**  
The return value is of type REAL or COMPLEX. The kind type parameter is the same as X.

**Standard:** Fortran 77 and later  
**Class:** Elemental function

**Syntax:**

```fortran
RESULT = SQRT(X)
```

**Example:**

```fortran
program test_sqrt
  real(8) :: x = 2.0_8
  complex :: z = (1.0, 2.0)
  x = sqrt(x)
  z = sqrt(z)
end program test_sqrt
```

**Specific names:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQRT(X)</td>
<td>REAL(4) X</td>
<td>REAL(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>DSQRT(X)</td>
<td>REAL(8) X</td>
<td>REAL(8)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>CSQRT(X)</td>
<td>COMPLEX(4) X</td>
<td>COMPLEX(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>ZSQRT(X)</td>
<td>COMPLEX(8) X</td>
<td>COMPLEX(8)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>CDSQRT(X)</td>
<td>COMPLEX(8) X</td>
<td>COMPLEX(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

10.259 SRAND — Reinitialize the random number generator

**SRAND ()**

SRAND reinitializes the pseudo-random number generator called by RAND and IRAND. The new seed used by the generator is specified by the required argument SEED.

**Parameters**  
SEED – Shall be a scalar INTEGER(kind=4).

**Returns**  
Does not return anything.

**Standard:** GNU extension

**Class:** Subroutine

**Syntax:**

```fortran
CALL SRAND(SEED)
```

**Example:** See RAND and IRAND for examples.
Notes: The Fortran standard specifies the intrinsic subroutines RANDOM_SEED to initialize the pseudo-random number generator and RANDOM_NUMBER to generate pseudo-random numbers. These subroutines should be used in new codes.

Please note that in GNU Fortran, these two sets of intrinsics (RAND, IRAND and SRAND on the one hand, RANDOM_NUMBER and RANDOM_SEED on the other hand) access two independent pseudo-random number generators.

See also: RAND, RANDOM_SEED, RANDOM_NUMBER

10.260 STAT — Get file status

STAT ()

This function returns information about a file. No permissions are required on the file itself, but execute (search) permission is required on all of the directories in path that lead to the file.

Parameters

- NAME – The type shall be CHARACTER, of the default kind and a valid path within the file system.
- VALUES – The type shall be INTEGER(4), DIMENSION(13).
- STATUS – (Optional) status flag of type INTEGER(4). Returns 0 on success and a system specific error code otherwise.

Standard: GNU extension

Class: Subroutine, function

Syntax:

CALL STAT(NAME, VALUES [, STATUS])

Example:

```
PROGRAM test_stat
    INTEGER, DIMENSION(13) :: buff
    INTEGER :: status

    CALL STAT("/etc/passwd", buff, status)

    IF (status == 0) THEN
        WRITE (*, FMT="('Device ID:', T30, I19)") buff(1)
        WRITE (*, FMT="('Inode number:', T30, I19)") buff(2)
        WRITE (*, FMT="('File mode (octal):', T30, O19)") buff(3)
        WRITE (*, FMT="('Number of links:', T30, I19)") buff(4)
        WRITE (*, FMT="('Owner''s uid:', T30, I19)") buff(5)
        WRITE (*, FMT="('Owner''s gid:', T30, I19)") buff(6)
        WRITE (*, FMT="('Device where located:', T30, I19)") buff(7)
        WRITE (*, FMT="('File size:', T30, I19)") buff(8)
        WRITE (*, FMT="('Last access time:', T30, A19)") CTIME(buff(9))
        WRITE (*, FMT="('Last modification time', T30, A19)") CTIME(buff(10))
        WRITE (*, FMT="('Last status change time:', T30, A19)") CTIME(buff(11))
    END IF
```
WRITE (*, FMT="('Preferred block size:', T30, I19)") buff(12)
WRITE (*, FMT="('No. of blocks allocated:', T30, I19)") buff(13)
END IF
END PROGRAM

See also: To stat an open file: FSTAT To stat a link: LSTAT

10.261 STORAGE_SIZE — Storage size in bits

STORAGE_SIZE()

Returns the storage size of argument A in bits.

Parameters

- A – Shall be a scalar or array of any type.
- KIND – (Optional) shall be a scalar integer constant expression.

Standard: Fortran 2008 and later

Class: Inquiry function

Syntax:

\[
\text{RESULT} = \text{STORAGE\_SIZE}(A [, \text{KIND}])
\]

Return Value: The result is a scalar integer with the kind type parameter specified by KIND (or default integer type if KIND is missing). The result value is the size expressed in bits for an element of an array that has the dynamic type and type parameters of A.

See also: C\_SIZEOF, SIZEOF

10.262 SUM — Sum of array elements

SUM()

Adds the elements of ARRAY along dimension DIM if the corresponding element in MASK is TRUE.

Parameters

- ARRAY – Shall be an array of type INTEGER, REAL or COMPLEX.
- DIM – (Optional) shall be a scalar of type INTEGER with a value in the range from 1 to n, where n equals the rank of ARRAY.
- MASK – (Optional) shall be of type LOGICAL and either be a scalar or an array of the same shape as ARRAY.

Returns: The result is of the same type as ARRAY.

Standard: Fortran 90 and later

Class: Transformational function

Syntax:
RESULT = SUM(ARRAY[, MASK])
RESULT = SUM(ARRAY, DIM[, MASK])

Example:

```fortran
PROGRAM test_sum
   INTEGER :: x(5) = (/ 1, 2, 3, 4, 5 /)
   print *, SUM(x) ! all elements, sum = 15
   print *, SUM(x, MASK=MOD(x, 2)==1) ! odd elements, sum = 9
END PROGRAM
```

See also: PRODUCT

10.263 SYMLNK — Create a symbolic link

SYMLNK ()

Makes a symbolic link from file PATH1 to PATH2. A null character (CHAR(0)) can be used to mark the end of the names in PATH1 and PATH2; otherwise, trailing blanks in the file names are ignored. If the STATUS argument is supplied, it contains 0 on success or a nonzero error code upon return; see symlink(2). If the system does not supply symlink(2), ENOSYS is returned.

Parameters

- **PATH1** – Shall be of default CHARACTER type.
- **PATH2** – Shall be of default CHARACTER type.
- **STATUS** – (Optional) Shall be of default INTEGER type.

Standard: GNU extension

Class: Subroutine, function

Syntax:

```fortran
CALL SYMLNK(PATH1, PATH2 [, STATUS])
STATUS = SYMLNK(PATH1, PATH2)
```

See also: LINK, UNLINK

10.264 SYSTEM — Execute a shell command

SYSTEM ()

Passes the command COMMAND to a shell (see system(3)). If argument STATUS is present, it contains the value returned by system(3), which is presumably 0 if the shell command succeeded. Note that which shell is used to invoke the command is system-dependent and environment-dependent.

Parameters

- **COMMAND** – Shall be of default CHARACTER type.
- **STATUS** – (Optional) Shall be of default INTEGER type.

Standard: GNU extension
**Class:** Subroutine, function

**Syntax:**

```
CALL SYSTEM(COMMAND [, STATUS])
STATUS = SYSTEM(COMMAND)
```

**See also:** EXECUTE_COMMAND_LINE, which is part of the Fortran 2008 standard and should considered in new code for future portability.

## 10.265 SYSTEM_CLOCK — Time function

**SYSTEM_CLOCK()**

Determines the `COUNT` of a processor clock since an unspecified time in the past modulo `COUNT_MAX`. `COUNT_RATE` determines the number of clock ticks per second. If the platform supports a monotonic clock, that clock is used and can, depending on the platform clock implementation, provide up to nanosecond resolution. If a monotonic clock is not available, the implementation falls back to a realtime clock.

**Parameters**

- `COUNT` – (Optional) shall be a scalar of type `INTEGER` with `INTENT(OUT)`.
- `COUNT_RATE` – (Optional) shall be a scalar of type `INTEGER` or `REAL`, with `INTENT(OUT)`.
- `COUNT_MAX` – (Optional) shall be a scalar of type `INTEGER` with `INTENT(OUT)`.

**Standard:** Fortran 90 and later

**Class:** Subroutine

**Syntax:**

```
CALL SYSTEM_CLOCK([COUNT, COUNT_RATE, COUNT_MAX])
```

**Example:**

```
PROGRAM test_system_clock
  INTEGER :: count, count_rate, count_max
  CALL SYSTEM_CLOCK(count, count_rate, count_max)
  WRITE(*,*) count, count_rate, count_max
END PROGRAM
```

**See also:** DATE_AND_TIME, CPU_TIME
10.266  TAN — Tangent function

TAN(X)

TAN(X) computes the tangent of X.

Parameters  X — The type shall be REAL or COMPLEX.

Returns  The return value has same type and kind as X, and its value is in radians.

Standard:  Fortran 77 and later, for a complex argument Fortran 2008 or later

Class:  Elemental function

Syntax:

RESULT = TAN(X)

Example:

```fortran
program test_tan
  real(8) :: x = 0.165_8
  x = tan(x)
end program test_tan
```

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAN(X)</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>DTAN(X)</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

See also:  Inverse function: ATAN Degrees function: TAND

10.267  TAND — Tangent function, degrees

TAND(X)

TAND(X) computes the tangent of X in degrees.

Parameters  X — The type shall be REAL or COMPLEX.

Returns  The return value has same type and kind as X, and its value is in degrees.

Standard:  GNU extension, enabled with `-fdec-math`

Class:  Elemental function

Syntax:

RESULT = TAND(X)

Example:

```fortran
program test_tand
  real(8) :: x = 0.165_8
  x = tand(x)
end program test_tand
```
Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAND (X)</td>
<td>REAL(4) X</td>
<td>REAL(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>DTAND (X)</td>
<td>REAL(8) X</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

See also: Inverse function: ATAND Radians function: TAN

10.268 TANH — Hyperbolic tangent function

\( \text{TANH}(X) \)

\( \text{TANH}(X) \) computes the hyperbolic tangent of \( X \).

**Parameters** \( X \) – The type shall be \texttt{REAL} or \texttt{COMPLEX}.

**Returns** The return value has same type and kind as \( X \). If \( X \) is complex, the imaginary part of the result is in radians. If \( X \) is \texttt{REAL}, the return value lies in the range \(-1 \leq \text{tanh}(x) \leq 1\).

**Standard**: Fortran 77 and later, for a complex argument Fortran 2008 or later

**Class**: Elemental function

**Syntax**:

\[ X = \text{TANH}(X) \]

**Example**:

```
program test_tanh
  real(8) :: x = 2.1_8
  x = tanh(x)
end program test_tanh
```

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>TANH(X)</td>
<td>REAL(4) X</td>
<td>REAL(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>DTANH(X)</td>
<td>REAL(8) X</td>
<td>REAL(8)</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

See also: ATANH

10.269 THIS_IMAGE — Function that returns the cosubscript index of this image

\( \text{THIS_IMAGE}() \)

\( \text{THIS_IMAGE}() \) returns the cosubscript for this image.

**Parameters**

- **DISTANCE** – (optional, intent(in)) Nonnegative scalar integer (not permitted together with \texttt{COARRAY}).
• **COARRAY** — Coarray of any type (optional; if *DIM* present, required).

• **DIM** — default integer scalar (optional). If present, *DIM* shall be between one and the corank of **COARRAY**.

**Returns**  Default integer. If **COARRAY** is not present, it is scalar; if **DISTANCE** is not present or has value 0, its value is the image index on the invoking image for the current team, for values smaller or equal distance to the initial team, it returns the image index on the ancestor team which has a distance of **DISTANCE** from the invoking team. If **DISTANCE** is larger than the distance to the initial team, the image index of the initial team is returned. Otherwise when the **COARRAY** is present, if **DIM** is not present, a rank-1 array with corank elements is returned, containing the cosubscripts for **COARRAY** specifying the invoking image. If **DIM** is present, a scalar is returned, with the value of the **DIM** element of **THIS_IMAGE** (**COARRAY**).

**Standard:** Fortran 2008 and later. With **DISTANCE** argument, Technical Specification (TS) 18508 or later

**Class:** Transformational function

**Syntax:**

```
RESULT = THIS_IMAGE()
RESULT = THIS_IMAGE(DISTANCE)
RESULT = THIS_IMAGE(COARRAY [, DIM])
```

**Example:**

```fortran
INTEGER :: value[*]
INTEGER :: i
value = THIS_IMAGE()
SYNC ALL
IF (THIS_IMAGE() == 1) THEN
  DO i = 1, NUM_IMAGES()
    WRITE(*, '(2(a,i0))') 'value[', i, '] is ', value[i]
  END DO
END IF

! Check whether the current image is the initial image
IF (THIS_IMAGE(HUGE(1)) /= THIS_IMAGE())
  error stop "something is rotten here"
```

**See also:** NUM_IMAGES, IMAGE_INDEX

### 10.270 TIME — Time function

**TIME ()**

Returns the current time encoded as an integer (in the manner of the function `time(3)` in the C standard library). This value is suitable for passing to CTIME, GMTIME, and LTIME.

**Returns**  The return value is a scalar of type **INTEGER(4)**.

**Standard:** GNU extension

**Class:** Function

**Syntax:**
RESULT = TIME()

See also: DATE_AND_TIME, CTIME, GMTIME, LTIME, MCLOCK, TIME8

10.271 TIME8 — Time function (64-bit)

TIME8()

Returns the current time encoded as an integer (in the manner of the function time (3) in the C standard library). This value is suitable for passing to CTIME, GMTIME, and LTIME.

Returns The return value is a scalar of type INTEGER(8).

Standard: GNU extension
Class: Function
Syntax:
RESULT = TIME8()

See also: DATE_AND_TIME, CTIME, GMTIME, LTIME, MCLOCK8, TIME

10.272 TINY — Smallest positive number of a real kind

TINY(X)

TINY(X) returns the smallest positive (non zero) number in the model of the type of X.

Parameters X – Shall be of type REAL.

Returns The return value is of the same type and kind as X

Standard: Fortran 90 and later
Class: Inquiry function
Syntax:
RESULT = TINY(X)

Example: See HUGE for an example.
10.273 TRAILZ — Number of trailing zero bits of an integer

TRAILZ ()

TRAILZ returns the number of trailing zero bits of an integer.

Parameters I – Shall be of type INTEGER.

Returns The type of the return value is the default INTEGER. If all the bits of I are zero, the result value is BIT_SIZE(I).

Standard: Fortran 2008 and later

Class: Elemental function

Syntax:

RESULT = TRAILZ(I)

Example:

PROGRAM test_trailz
  WRITE (*,*) TRAILZ(8)  ! prints 3
END PROGRAM

See also: BIT_SIZE, LEADZ, POPPAR, POPCNT

10.274 TRANSFER — Transfer bit patterns

TRANSFER ()

Interprets the bitwise representation of SOURCE in memory as if it is the representation of a variable or array of the same type and type parameters as MOLD.

Parameters

- SOURCE – Shall be a scalar or an array of any type.
- MOLD – Shall be a scalar or an array of any type.
- SIZE – (Optional) shall be a scalar of type INTEGER.

Returns The result has the same type as MOLD, with the bit level representation of SOURCE. If SIZE is present, the result is a one-dimensional array of length SIZE. If SIZE is absent but MOLD is an array (of any size or shape), the result is a one-dimensional array of the minimum length needed to contain the entirety of the bitwise representation of SOURCE. If SIZE is absent and MOLD is a scalar, the result is a scalar.

Standard: Fortran 90 and later

Class: Transformational function

Syntax:

RESULT = TRANSFER(SOURCE, MOLD[, SIZE])

Example:
PROGRAM test_transfer
   integer :: x = 2143289344
   print *, transfer(x, 1.0) ! prints "NaN" on i686
END PROGRAM

10.275 TRANSPOSE — Transpose an array of rank two

TRANSPOSE ()
   Transpose an array of rank two. Element (i, j) of the result has the value MATRIX(j, i), for all i, j.

Parameters MATRIX – Shall be an array of any type and have a rank of two.

Returns The result has the same type as MATRIX, and has shape (/ m, n /) if MATRIX has shape (/ n, m /).

Standard: Fortran 90 and later

Class: Transformational function

Syntax:

RESULT = TRANSPOSE(MATRIX)

10.276 TRIM — Remove trailing blank characters of a string

TRIM ()
   Removes trailing blank characters of a string.

Parameters STRING – Shall be a scalar of type CHARACTER.

Returns A scalar of type CHARACTER which length is that of STRING less the number of trailing blanks.

Standard: Fortran 90 and later

Class: Transformational function

Syntax:

RESULT = TRIM(STRING)

Example:

PROGRAM test_trim
   CHARACTER(len=10), PARAMETER :: s = "GFORTRAN "
   WRITE(*,*), LEN(s), LEN(TRIM(s)) ! "10 8", with/without trailing blanks
END PROGRAM

See also: ADJUSTL, ADJUSTR
10.277 TTYNAM — Get the name of a terminal device.

**TTYNAM ()**
Get the name of a terminal device. For more information, see ttyname(3).

**Parameters**

- **UNIT** – Shall be a scalar INTEGER.
- **NAME** – Shall be of type CHARACTER.

**Standard**: GNU extension

**Class**: Subroutine, function

**Syntax**:

```
CALL TTYNAM(UNIT, NAME)
NAME = TTYNAM(UNIT)
```

**Example**:

```
PROGRAM test_ttynam
  INTEGER :: unit
  DO unit = 1, 10
    IF (isatty(unit=unit)) WRITE(*,*) ttynam(unit)
  END DO
END PROGRAM
```

**See also**: ISATTY

10.278 UBOUND — Upper dimension bounds of an array

**UBOUND ()**
Returns the upper bounds of an array, or a single upper bound along the DIM dimension.

**Parameters**

- **ARRAY** – Shall be an array, of any type.
- **DIM** – (Optional) Shall be a scalar INTEGER.
- **KIND** – (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

**Returns** The return value is of type INTEGER and of kind KIND. If KIND is absent, the return value is of default integer kind. If DIM is absent, the result is an array of the upper bounds of ARRAY. If DIM is present, the result is a scalar corresponding to the upper bound of the array along that dimension. If ARRAY is an expression rather than a whole array or array structure component, or if it has a zero extent along the relevant dimension, the upper bound is taken to be the number of elements along the relevant dimension.

**Standard**: Fortran 90 and later, with KIND argument Fortran 2003 and later

**Class**: Inquiry function

**Syntax**:
RESULT = UBOUND(ARRAY [, DIM [, KIND]])

See also: LBOUND, LCOBOUND

10.279 UCOBOUND — Upper codimension bounds of an array

UCOBOUND ()

Returns the upper bounds of a coarray, or a single upper bound along the \textit{DIM} codimension.

Parameters

- \textbf{ARRAY} – Shall be an coarray, of any type.
- \textbf{DIM} – (Optional) Shall be a scalar \texttt{INTEGER}.
- \textbf{KIND} – (Optional) An \texttt{INTEGER} initialization expression indicating the kind parameter of the result.

Returns

The return value is of type \texttt{INTEGER} and of kind \texttt{KIND}. If \texttt{KIND} is absent, the return value is of default integer kind. If \texttt{DIM} is absent, the result is an array of the lower bounds of \texttt{COARRAY}. If \texttt{DIM} is present, the result is a scalar corresponding to the lower bound of the array along that codimension.

Standard: Fortran 2008 and later

Class: Inquiry function

Syntax:

\texttt{RESULT = UCOBOUND(COARRAY [, DIM [, KIND]])}

See also: LCOBOUND, LBOUND

10.280 UMASK — Set the file creation mask

UMASK ()

Sets the file creation mask to \texttt{MASK}. If called as a function, it returns the old value. If called as a subroutine and argument \texttt{OLD} if it is supplied, it is set to the old value. See \texttt{umask(2)}.

Parameters

- \textbf{MASK} – Shall be a scalar of type \texttt{INTEGER}.
- \textbf{OLD} – (Optional) Shall be a scalar of type \texttt{INTEGER}.

Standard: GNU extension

Class: Subroutine, function

Syntax:

\texttt{CALL UMASK(MASK [, OLD])}
\texttt{OLD = UMASK(MASK)}
10.281 UNLINK — Remove a file from the file system

**UNLINK** ()

Unlinks the file *PATH*. A null character (CHAR(0)) can be used to mark the end of the name in *PATH*; otherwise, trailing blanks in the file name are ignored. If the *STATUS* argument is supplied, it contains 0 on success or a nonzero error code upon return; see *unlink(2)*.

**Parameters**

- **PATH** – Shall be of default CHARACTER type.
- **STATUS** – (Optional) Shall be of default INTEGER type.

**Standard**: GNU extension

**Class**: Subroutine, function

**Syntax**:

```fortran
CALL UNLINK(PATH [, STATUS])
STATUS = UNLINK(PATH)
```

**See also**: LINK, SYMLNK

10.282 UNPACK — Unpack an array of rank one into an array

**UNPACK** ()

Store the elements of *VECTOR* in an array of higher rank.

**Parameters**

- **VECTOR** – Shall be an array of any type and rank one. It shall have at least as many elements as *MASK* has TRUE values.
- **MASK** – Shall be an array of type LOGICAL.
- **FIELD** – Shall be of the same type as *VECTOR* and have the same shape as *MASK*.

**Returns** The resulting array corresponds to *FIELD* with TRUE elements of *MASK* replaced by values from *VECTOR* in array element order.

**Standard**: Fortran 90 and later

**Class**: Transformational function

**Syntax**:

```fortran
RESULT = UNPACK(VECTOR, MASK, FIELD)
```

**Example**:

```fortran
PROGRAM test_unpack
   integer :: vector(2) = (/1,1/)
   logical :: mask(4) = (/ .TRUE., .FALSE., .FALSE., .TRUE. /)
   integer :: field(2,2) = 0, unity(2,2)
```

(continues on next page)
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

(continued from previous page)

! result: unity matrix  
unity = unpack(vector, reshape(mask, (/2,2/)), field)  
END PROGRAM

See also: PACK, SPREAD

10.283 VERIFY — Scan a string for characters not a given set

VERIFY ()

Verifies that all the characters in STRING belong to the set of characters in SET.

Parameters

- STRING – Shall be of type CHARACTER.
- SET – Shall be of type CHARACTER.
- BACK – (Optional) shall be of type LOGICAL.
- KIND – (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

Returns The return value is of type INTEGER and of kind KIND. If KIND is absent, the return value is of default integer kind.

Standard: Fortran 90 and later, with KIND argument Fortran 2003 and later

Class: Elemental function

Syntax:

RESULT = VERIFY(STRING, SET[, BACK [, KIND]])

Example:

PROGRAM test_verify
  WRITE(*,*) VERIFY("FORTRAN", "AQ") ! 1, found 'F'
  WRITE(*,*) VERIFY("FORTRAN", "FOO") ! 3, found 'R'
  WRITE(*,*) VERIFY("FORTRAN", "C++") ! 1, found 'F'
  WRITE(*,*) VERIFY("FORTRAN", "C++", .TRUE.) ! 7, found 'N'
  WRITE(*,*) VERIFY("FORTRAN", "FORTRAN") ! 0' found none
END PROGRAM

See also: SCAN, INDEX intrinsic
10.284 XOR — Bitwise logical exclusive OR

XOR ()
Bitwise logical exclusive or.

Parameters

- \(I\) – The type shall be either a scalar \texttt{INTEGER} type or a scalar \texttt{LOGICAL} type or a boz-literal-constant.
- \(J\) – The type shall be the same as the type of \(I\) or a boz-literal-constant. \(I\) and \(J\) shall not both be boz-literal-constants. If either \(I\) and \(J\) is a boz-literal-constant, then the other argument must be a scalar \texttt{INTEGER}.

Returns The return type is either a scalar \texttt{INTEGER} or a scalar \texttt{LOGICAL}. If the kind type parameters differ, then the smaller kind type is implicitly converted to larger kind, and the return has the larger kind. A boz-literal-constant is converted to an \texttt{INTEGER} with the kind type parameter of the other argument as-if a call to \texttt{INT} occurred.

Standard: GNU extension

Class: Function

Syntax:

\[
\text{RESULT} = \text{XOR}(I, J)
\]

Example:

```fortran
PROGRAM test_xor
    LOGICAL :: T = .TRUE., F = .FALSE.
    INTEGER :: a, b
    DATA a / Z'F' /, b / Z'3' /
    WRITE (*,*) XOR(T, T), XOR(T, F), XOR(F, T), XOR(F, F)
    WRITE (*,*) XOR(a, b)
END PROGRAM
```

See also: Fortran 95 elemental function: IEOR
11.1 OpenACC Module OPENACC

**Standard**: OpenACC Application Programming Interface v2.6

The OpenACC Fortran runtime library routines are provided both in a form of a Fortran 90 module, named OPENACC, and in form of a Fortran include file named openacc_lib.h. The procedures provided by OPENACC can be found in the TopIntroductionlibgompGNU Offloading and Multi Processing Runtime Library manual, the named constants defined in the modules are listed below.

For details refer to the actual OpenACC Application Programming Interface v2.6.

OPENACC provides the scalar default-integer named constant openacc_version with a value of the form yyyy/mm, where yyyy is the year and mm the month of the OpenACC version; for OpenACC v2.6 the value is 201711.

11.2 ISO_FORTRAN_ENV

**Standard**: Fortran 2003 and later, except when otherwise noted

The ISO_FORTRAN_ENV module provides the following scalar default-integer named constants:

**ATOMIC_INT_KIND**: Default-kind integer constant to be used as kind parameter when defining integer variables used in atomic operations. (Fortran 2008 or later.)

**ATOMIC_LOGICAL_KIND**: Default-kind integer constant to be used as kind parameter when defining logical variables used in atomic operations. (Fortran 2008 or later.)

**CHARACTER_KINDS**: Default-kind integer constant array of rank one containing the supported kind parameters of the CHARACTER type. (Fortran 2008 or later.)

**CHARACTER_STORAGE_SIZE**: Size in bits of the character storage unit.

**ERROR_UNIT**: Identifies the preconnected unit used for error reporting.

**FILE_STORAGE_SIZE**: Size in bits of the file-storage unit.

**INPUT_UNIT**: Identifies the preconnected unit identified by the asterisk (*) in READ statement.

**INT8, INT16, INT32, INT64**: Kind type parameters to specify an INTEGER type with a storage size of 16, 32, and 64 bits. It is negative if a target platform does not support the particular kind. (Fortran 2008 or later.)

**INTEGER_KINDS**: Default-kind integer constant array of rank one containing the supported kind parameters of the INTEGER type. (Fortran 2008 or later.)

**IOSTAT_END**: The value assigned to the variable passed to the IOSTAT= specifier of an input/output statement if an end-of-file condition occurred.
**IOSTAT_EOR**: The value assigned to the variable passed to the IOSTAT= specifier of an input/output statement if an end-of-record condition occurred.

**IOSTAT_INQUIRE_INTERNAL_UNIT**: Scalar default-integer constant, used by INQUIRE for the IOSTAT= specifier to denote an internal unit number identifies an internal unit. (Fortran 2008 or later.)

**NUMERIC_STORAGE_SIZE**: The size in bits of the numeric storage unit.

**LOGICAL_KINDS**: Default-kind integer constant array of rank one containing the supported kind parameters of the LOGICAL type. (Fortran 2008 or later.)

**OUTPUT_UNIT**: Identifies the preconnected unit identified by the asterisk (*) in WRITE statement.

**REAL32, REAL64, REAL128**: Kind type parameters to specify a REAL type with a storage size of 32, 64, and 128 bits. It is negative if a target platform does not support the particular kind. (Fortran 2008 or later.)

**REAL_KINDS**: Default-kind integer constant array of rank one containing the supported kind parameters of the REAL type. (Fortran 2008 or later.)

**STAT_LOCKED**: Scalar default-integer constant used as STAT= return value by LOCK to denote that the lock variable is locked by the executing image. (Fortran 2008 or later.)

**STAT_LOCKED_OTHER_IMAGE**: Scalar default-integer constant used as STAT= return value by UNLOCK to denote that the lock variable is locked by another image. (Fortran 2008 or later.)

**STAT_STOPPED_IMAGE**: Positive, scalar default-integer constant used as STAT= return value if the argument in the statement requires synchronisation with an image, which has initiated the termination of the execution. (Fortran 2008 or later.)

**STAT_FAILED_IMAGE**: Positive, scalar default-integer constant used as STAT= return value if the argument in the statement requires communication with an image, which is in the failed state. (TS 18508 or later.)

**STAT_UNLOCKED**: Scalar default-integer constant used as STAT= return value by UNLOCK to denote that the lock variable is unlocked. (Fortran 2008 or later.)

The module provides the following derived type:

**LOCK_TYPE**: Derived type with private components to be use with the LOCK and UNLOCK statement. A variable of its type has to be always declared as coarray and may not appear in a variable-definition context. (Fortran 2008 or later.)

The module also provides the following intrinsic procedures: COMPILER_OPTIONS and COMPILER_VERSION.

### 11.3 ISO_C_BINDING

**Standard**: Fortran 2003 and later, GNU extensions

The following intrinsic procedures are provided by the module; their definition can be found in the section Intrinsic Procedures of this manual.

**C_ASSOCIATEDC_F_POINTERC_F_PROCPOINTERC_FUNLOCC_LOCC_SIZEOF**

The ISO_C_BINDING module provides the following named constants of type default integer, which can be used as KIND type parameters.

In addition to the integer named constants required by the Fortran 2003 standard and C_PTRDIFF_T of TS 29113, GNU Fortran provides as an extension named constants for the 128-bit integer types supported by the C compiler: C_INT128_T, C_INT_LEAST128_T, C_INT_FAST128_T. Furthermore, if __float128 is supported in C, the named constants C_FLOAT128, C_FLOAT128_COMPLEX are defined.
<table>
<thead>
<tr>
<th>Fortran Type</th>
<th>Named constant</th>
<th>C type</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>C_INT</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>C_SHORT</td>
<td>short int</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>C_LONG</td>
<td>long int</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>C_LONG_LONG</td>
<td>long long int</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>C_SIGNED_CHAR</td>
<td>signed char/unsigned char</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>C_SIZE_T</td>
<td>size_t</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>C_INT8_T</td>
<td>int8_t</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>C_INT16_T</td>
<td>int16_t</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>C_INT32_T</td>
<td>int32_t</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>C_INT64_T</td>
<td>int64_t</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>C_INT128_T</td>
<td>int128_t</td>
<td>Ext.</td>
</tr>
<tr>
<td>INTEGER</td>
<td>C_INT_LEAST8_T</td>
<td>int_least8_t</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>C_INT_LEAST16_T</td>
<td>int_least16_t</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>C_INT_LEAST32_T</td>
<td>int_least32_t</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>C_INT_LEAST64_T</td>
<td>int_least64_t</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>C_INT_LEAST128_T</td>
<td>int_least128_t</td>
<td>Ext.</td>
</tr>
<tr>
<td>INTEGER</td>
<td>C_INT_FAST8_T</td>
<td>int_fast8_t</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>C_INT_FAST16_T</td>
<td>int_fast16_t</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>C_INT_FAST32_T</td>
<td>int_fast32_t</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>C_INT_FAST64_T</td>
<td>int_fast64_t</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>C_INT_FAST128_T</td>
<td>int_fast128_t</td>
<td>Ext.</td>
</tr>
<tr>
<td>INTEGER</td>
<td>C_INTPTR_T</td>
<td>intptr_t</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>C_PTRDIFF_T</td>
<td>ptrdiff_t</td>
<td>TS 29113</td>
</tr>
<tr>
<td>REAL</td>
<td>C_FLOAT</td>
<td>float</td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td>C_DOUBLE</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td>C_LONG_DOUBLE</td>
<td>long double</td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td>C_FLOAT128</td>
<td>__float128</td>
<td>Ext.</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>C_FLOAT_COMPLEX</td>
<td>float _Complex</td>
<td></td>
</tr>
<tr>
<td>COMPLEX</td>
<td>C_DOUBLE_COMPLEX</td>
<td>double _Complex</td>
<td></td>
</tr>
<tr>
<td>COMPLEX</td>
<td>C_LONG_DOUBLE_COMPLEX</td>
<td>long double _Complex</td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td>C_FLOAT128_COMPLEX</td>
<td>__float128 _Complex</td>
<td>Ext.</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>C_BOOL</td>
<td>_Bool</td>
<td></td>
</tr>
<tr>
<td>CHARACTER</td>
<td>C_CHAR</td>
<td>char</td>
<td></td>
</tr>
</tbody>
</table>

Additionally, the following parameters of type `CHARACTER(KIND=C_CHAR)` are defined.

<table>
<thead>
<tr>
<th>Name</th>
<th>C definition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_NULL_CHAR</td>
<td>null character</td>
<td>'\0'</td>
</tr>
<tr>
<td>C_ALERT</td>
<td>alert</td>
<td>'\a'</td>
</tr>
<tr>
<td>C_BACKSPACE</td>
<td>backspace</td>
<td>'\b'</td>
</tr>
<tr>
<td>C_FORM_FEED</td>
<td>form feed</td>
<td>'f'</td>
</tr>
<tr>
<td>C_NEW_LINE</td>
<td>new line</td>
<td>'n'</td>
</tr>
<tr>
<td>C_CARRIAGE_RETURN</td>
<td>carriage return</td>
<td>'\r'</td>
</tr>
<tr>
<td>C_HORIZONTAL_TAB</td>
<td>horizontal tab</td>
<td>'\t'</td>
</tr>
<tr>
<td>C_VERTICAL_TAB</td>
<td>vertical tab</td>
<td>'\v'</td>
</tr>
</tbody>
</table>

Moreover, the following two named constants are defined:
Both are equivalent to the value `NULL` in C.

### 11.4 IEEE modules: `IEEE_EXCEPTIONS`, `IEEE_ARITHMETIC`, and `IEEE_FEATURES`

**Standard:** Fortran 2003 and later

The `IEEE_EXCEPTIONS`, `IEEE_ARITHMETIC`, and `IEEE_FEATURES` intrinsic modules provide support for exceptions and IEEE arithmetic, as defined in Fortran 2003 and later standards, and the IEC 60559:1989 standard *(Binary floating-point arithmetic for microprocessor systems)*. These modules are only provided on the following supported platforms:

- i386 and x86_64 processors
- platforms which use the GNU C Library (glibc)
- platforms with support for SysV/386 routines for floating point interface (including Solaris and BSDs)
- platforms with the AIX OS

For full compliance with the Fortran standards, code using the `IEEE_EXCEPTIONS` or `IEEE_ARITHMETIC` modules should be compiled with the following options: `-fno-unsafe-math-optimizations -frounding-math -fsignaling-nans`.

### 11.5 OpenMP Modules OMP_LIB and OMP_LIB_KINDS

The OpenMP Fortran runtime library routines are provided both in a form of two Fortran modules, named `OMP_LIB` and `OMP_LIB_KINDS`, and in a form of a Fortran include file named `omp_lib.h`. The procedures provided by `OMP_LIB` can be found in the *TopIntrouctionlibgompGNU Offloading and Multi Processing Runtime Library manual*, the named constants defined in the modules are listed below.

For details refer to the actual OpenMP Application Program Interface v4.5 and OpenMP Application Program Interface v5.0.

`OMP_LIB_KINDS` provides the following scalar default-integer named constants:

```
omp_allocator_handle_kind
omp_alloc_traits_key_kind
omp_alloc_traits_val_kind
omp_depend_kind
omp_lock_kind
omp_lock_hint_kind
omp_nest_lock_kind
omp_pause_resource_kind
omp_memspace_handle_kind
omp_proc_bind_kind
omp_sched_kind
omp_sync_hint_kind
```
OMP_LIB provides the scalar default-integer named constant `openmp_version` with a value of the form `yyyymm`, where `yyyy` is the year and `mm` the month of the OpenMP version; for OpenMP v4.5 the value is `201511`.

The following derived type:

```fortran
omp_alloctrait
```

The following scalar integer named constants of the kind `omp_sched_kind`:

```fortran
omp_sched_static
omp_sched_dynamic
omp_sched_guided
omp_sched_auto
```

And the following scalar integer named constants of the kind `omp_proc_bind_kind`:

```fortran
omp_proc_bind_false
omp_proc_bind_true
omp_proc_bind_master
omp_proc_bind_close
omp_proc_bind_spread
```

The following scalar integer named constants are of the kind `omp_lock_hint_kind`:

```fortran
omp_lock_hint_none
omp_lock_hint_uncontended
omp_lock_hint_contended
omp_lock_hint_nonspeculative
omp_lock_hint_speculative
omp_sync_hint_none
omp_sync_hint_uncontended
omp_sync_hint_contended
omp_sync_hint_nonspeculative
omp_sync_hint_speculative
```

And the following two scalar integer named constants are of the kind `omp_pause_resource_kind`:

```fortran
omp_pause_soft
omp_pause_hard
```

The following scalar integer named constants are of the kind `omp_alloctrait_key_kind`:

```fortran
omp_atk_sync_hint
omp_atk_alignment
omp_atk_access
omp_atk_pool_size
omp_atk_fallback
omp_atk_fb_data
omp_atk_pinned
omp_atk_partition
```

The following scalar integer named constants are of the kind `omp_alloctrait_val_kind`:

```fortran
omp_atv_default
omp_atv_false
omp_atv_true
omp_atv_contended
omp_atv_uncontended
```
The following scalar integer named constants are of the kind `omp_allocator_handle_kind`:

<table>
<thead>
<tr>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>omp_null_allocator</code></td>
</tr>
<tr>
<td><code>omp_default_mem_alloc</code></td>
</tr>
<tr>
<td><code>omp_large_cap_mem_alloc</code></td>
</tr>
<tr>
<td><code>omp_const_mem_alloc</code></td>
</tr>
<tr>
<td><code>omp_high_bw_mem_alloc</code></td>
</tr>
<tr>
<td><code>omp_low_lat_mem_alloc</code></td>
</tr>
<tr>
<td><code>omp_cgroup_mem_alloc</code></td>
</tr>
<tr>
<td><code>omp_pteam_mem_alloc</code></td>
</tr>
<tr>
<td><code>omp_thread_mem_alloc</code></td>
</tr>
</tbody>
</table>

The following scalar integer named constants are of the kind `omp_memspace_handle_kind`:

<table>
<thead>
<tr>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>omp_default_mem_space</code></td>
</tr>
<tr>
<td><code>omp_large_cap_mem_space</code></td>
</tr>
<tr>
<td><code>omp_const_mem_space</code></td>
</tr>
<tr>
<td><code>omp_high_bw_mem_space</code></td>
</tr>
<tr>
<td><code>omp_low_lat_mem_space</code></td>
</tr>
</tbody>
</table>
Free software is only possible if people contribute to efforts to create it. We're always in need of more people helping out with ideas and comments, writing documentation and contributing code.

If you want to contribute to GNU Fortran, have a look at the long lists of projects you can take on. Some of these projects are small, some of them are large; some are completely orthogonal to the rest of what is happening on GNU Fortran, but others are ‘mainstream’ projects in need of enthusiastic hackers. All of these projects are important! We will eventually get around to the things here, but they are also things doable by someone who is willing and able.

12.1 Contributors to GNU Fortran

Most of the parser was hand-crafted by Andy Vaught, who is also the initiator of the whole project. Thanks Andy! Most of the interface with GCC was written by Paul Brook.

The following individuals have contributed code and/or ideas and significant help to the GNU Fortran project (in alphabetical order):

- Janne Blomqvist
- Steven Bosscher
- Paul Brook
- Tobias Burnus
- François-Xavier Coudert
- Bud Davis
- Jerry DeLisle
- Erik Edelmann
- Bernhard Fischer
- Daniel Franke
- Richard Guenther
- Richard Henderson
- Katherine Holcomb
- Jakub Jelinek
- Niels Kristian Bech Jensen
- Steven Johnson
- Steven G. Kargl
The following people have contributed bug reports, smaller or larger patches, and much needed feedback and encouragement for the GNU Fortran project:

- Bill Clodius
- Dominique d’Humières
- Kate Hedstrom
- Erik Schnetter
- Joost VandeVondele

Many other individuals have helped debug, test and improve the GNU Fortran compiler over the past few years, and we welcome you to do the same! If you already have done so, and you would like to see your name listed in the list above, please contact us.

### 12.2 Projects

**Help build the test suite**  Solicit more code for donation to the test suite: the more extensive the testsuite, the smaller the risk of breaking things in the future! We can keep code private on request.

**Bug hunting/squishing**  Find bugs and write more test cases! Test cases are especially very welcome, because it allows us to concentrate on fixing bugs instead of isolating them. Going through the bugzilla database at [https://gcc.gnu.org/bugzilla/](https://gcc.gnu.org/bugzilla/) to reduce testcases posted there and add more information (for example, for which version does the testcase work, for which versions does it fail?) is also very helpful.
12.3 Proposed Extensions

Here’s a list of proposed extensions for the GNU Fortran compiler, in no particular order. Most of these are necessary to be fully compatible with existing Fortran compilers, but they are not part of the official J3 Fortran 95 standard.

12.3.1 Compiler extensions:

- User-specified alignment rules for structures.
- Automatically extend single precision constants to double.
- Compile code that conserves memory by dynamically allocating common and module storage either on stack or heap.
- Compile flag to generate code for array conformance checking (suggest -CC).
- User control of symbol names (underscores, etc).
- Compile setting for maximum size of stack frame size before spilling parts to static or heap.
- Flag to force local variables into static space.
- Flag to force local variables onto stack.

12.3.2 Environment Options

- Pluggable library modules for random numbers, linear algebra. LA should use BLAS calling conventions.
- Environment variables controlling actions on arithmetic exceptions like overflow, underflow, precision loss—Generate NaN, abort, default. action.
- Set precision for fp units that support it (i387).
- Variable for setting fp rounding mode.
- Variable to fill uninitialized variables with a user-defined bit pattern.
- Environment variable controlling filename that is opened for that unit number.
- Environment variable to clear/trash memory being freed.
- Environment variable to control tracing of allocations and frees.
- Environment variable to display allocated memory at normal program end.
- Environment variable for filename for * IO-unit.
- Environment variable for temporary file directory.
- Environment variable forcing standard output to be line buffered (Unix).
GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007
Copyright (C) 2007 Free Software Foundation, Inc. https://fsf.org/
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

13.1 Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change
the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change
all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the
GNU General Public License for most of our software; it applies also to any other work released this way by its authors.
You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to
make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you
receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs,
and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights.
Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities
to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients
the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you
must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer
you this License giving you legal permission to copy, distribute and/or modify it.

For the developers’ and authors’ protection, the GPL clearly explains that there is no warranty for this free software. For
both users’ and authors’ sake, the GPL requires that modified versions be marked as changed, so that their problems will
not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although
the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users’ freedom to change the
software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely
where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those
products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains
in future versions of the GPL, as needed to protect the freedom of users.
Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

13.2 TERMS AND CONDITIONS

13.2.1 0. Definitions.

“This License” refers to version 3 of the GNU General Public License.

“Copyright” also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

“The Program” refers to any copyrightable work licensed under this License. Each licensee is addressed as “you”. “Licensees” and “recipients” may be individuals or organizations.

To “modify” a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a “modified version” of the earlier work or a work “based on” the earlier work.

A “covered work” means either the unmodified Program or a work based on the Program.

To “propagate” a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To “convey” a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays “Appropriate Legal Notices” to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

13.2.2 1. Source Code.

The “source code” for a work means the preferred form of the work for making modifications to it. “Object code” means any non-source form of a work.

A “Standard Interface” means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However,
it does not include the work’s System Libraries, or general-purpose tools or generally available free programs which are
used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source
includes interface definition files associated with source files for the work, and the source code for shared libraries and
dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication
or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the
Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

13.2.3 2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided
the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program.
The output from running a covered work is covered by this License only if the output, given its content, constitutes a
covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license other-
wise remains in force. You may convey covered works to others for the sole purpose of having them make modifications
exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of
this License in conveying all material for which you do not control copyright. Those thus making or running the covered
works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from
making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not
allowed; section 10 makes it unnecessary.

13.2.4 3. Protecting Users’ Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations
under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting
circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the
extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you
disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work’s users,
your or third parties’ legal rights to forbid circumvention of technological measures.

13.2.5 4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program’s source code as you receive it, in any medium, provided that you
conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that
this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the
absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection
for a fee.
13.2.6 5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to “keep intact all notices”.

c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an “aggregate” if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation’s users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

13.2.7 6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling.
In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

“Installation Information” for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

### 13.2.8 7. Additional Terms.

“Additional permissions” are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
d) Limiting the use for publicity purposes of names of licensors or authors of the material; or
e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

### 13.2. TERMS AND CONDITIONS
All other non-permissive additional terms are considered “further restrictions” within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

### 13.2.9 8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

### 13.2.10 9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

### 13.2.11 10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party’s predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.
13.2.12 11. Patents.

A “contributor” is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor’s “contributor version”.

A contributor’s “essential patent claims” are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, “control” includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor’s essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a “patent license” is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To “grant” such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. “Knowingly relying” means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient’s use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is “discriminatory” if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.


If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13.2. TERMS AND CONDITIONS
13.2.14 13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

13.2.15 14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

13.2.16 15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULDN’T THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

13.2.17 16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
13.2.18 17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

13.3 How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.>
Copyright (C) <year> <name of author>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

<program> Copyright (C) <year> <name of author>
This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an “about box”.

You should also get your employer (if you work as a programmer) or school, if any, to sign a “copyright disclaimer” for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see https://www.gnu.org/licenses/.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read https://www.gnu.org/licenses/why-not-lgpl.html.
GNU FREE DOCUMENTATION LICENSE

Version 1.3, 3 November 2008


Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The Document, below, refers to any such manual or work. Anymember of the public is a licensee, and is addressed as “you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.
A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called Opaque.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

The “publisher” means any person or entity that distributes copies of the Document to the public.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a
complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled “History”, Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled “History” in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the “History” section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled “Acknowledgements” or “Dedications”, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.
You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and...
disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION
You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE
The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING
“Massive Multiauthor Collaboration Site” (or “MMC Site”) means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A “Massive Multiauthor Collaboration” (or “MMC”) contained in the site means any set of copyrightable works thus published on the MMC site.

“CC-BY-SA” means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

“Incorporate” means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is “eligible for relicensing” if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:
FUNDING FREE SOFTWARE

If you want to have more free software a few years from now, it makes sense for you to help encourage people to contribute funds for its development. The most effective approach known is to encourage commercial redistributors to donate.

Users of free software systems can boost the pace of development by encouraging for-a-fee distributors to donate part of their selling price to free software developers—the Free Software Foundation, and others.

The way to convince distributors to do this is to demand it and expect it from them. So when you compare distributors, judge them partly by how much they give to free software development. Show distributors they must compete to be the one who gives the most.

To make this approach work, you must insist on numbers that you can compare, such as, ‘We will donate ten dollars to the Frobnitz project for each disk sold.’ Don’t be satisfied with a vague promise, such as ‘A portion of the profits are donated,’ since it doesn’t give a basis for comparison.

Even a precise fraction ‘of the profits from this disk’ is not very meaningful, since creative accounting and unrelated business decisions can greatly alter what fraction of the sales price counts as profit. If the price you pay is $50, ten percent of the profit is probably less than a dollar; it might be a few cents, or nothing at all.

Some redistributors do development work themselves. This is useful too; but to keep everyone honest, you need to inquire how much they do, and what kind. Some kinds of development make much more long-term difference than others. For example, maintaining a separate version of a program contributes very little; maintaining the standard version of a program for the whole community contributes much. Easy new ports contribute little, since someone else would surely do them; difficult ports such as adding a new CPU to the GNU Compiler Collection contribute more; major new features or packages contribute the most.

By establishing the idea that supporting further development is ‘the proper thing to do’ when distributing free software for a fee, we can assure a steady flow of resources into making more free software.
Non-alphabetical

$, 12
%LOC, 56
%REF, 56
%VAL, 56
&, 17
[...], 36
_gfortran_caf_atomic_cas, 103
_gfortran_caf_atomic_define, 102
_gfortran_caf_atomic_op, 103
_gfortran_caf_atomic_ref, 102
_gfortran_caf_co_broadcast, 104
_gfortran_caf_co_max, 104
_gfortran_caf_co_min, 105
_gfortran_caf_co_reduce, 106
_gfortran_caf_co_sum, 105
_gfortran_caf_deregister, 91
_gfortran_caf_error_stop, 101
_gfortran_caf_error_stop_str, 101
_gfortran_caf_event_post, 99
_gfortran_caf_event_query, 100
_gfortran_caf_event_wait, 99
_gfortran_caf_fail_image, 102
_gfortran_caf_failed_images, 89
_gfortran_caf_finish, 88
_gfortran_caf_get, 93
_gfortran_caf_get_by_ref, 96
_gfortran_caf_image_status, 89
_gfortran_caf_init, 88
_gfortran_caf_is_present, 91
_gfortran_caf_lock, 98
_gfortran_caf_num_images, 89
_gfortran_caf_register, 90
_gfortran_caf_send, 92
_gfortran_caf_send_by_ref, 95
_gfortran_caf_sendget, 93
_gfortran_caf_sendget_by_ref, 97
_gfortran_caf_stopped_images, 90
_gfortran_caf_sync_all, 100
_gfortran_caf_sync_images, 100
_gfortran_caf_sync_memory, 101
_gfortran_caf_this_image, 88
_gfortran_caf_unlock, 98
_gfortran_set_args, 77
_gfortran_set_convert, 80
_gfortran_set_fpe, 80
_gfortran_set_max_subrecord_length, 81
_gfortran_set_options, 78
_gfortran_set_record_marker, 80

A
ABORT, 107
ABORT() built-in function, 107
ABS, 108
ABS() built-in function, 108
absolute value, 108
ACCESS, 109
access mode, 109
ACCESS() built-in function, 109
ACCESS='STREAM' I/O, 36
ACHAR, 109
ACHAR() built-in function, 109
ACOS, 110
ACOS() built-in function, 110
ACOSD, 111
ACOSD() built-in function, 111
ACOSH, 111
ACOSH() built-in function, 111
action, 44
add, 125
add elements, 271
ADD with fetch, 128
adjust left, 112
adjust right, 112
adjust string, 112
ADJUSTL, 112
ADJUSTL()
built-in function, 112
ADJUSTR, 112
ADJUSTR()
   built-in function, 112
AIMAG, 113
AIMAG()
   built-in function, 113
AINT, 114
AINT()
   built-in function, 114
ALARM, 114
ALARM()
   built-in function, 114
ALGAMA, 222
aliasing, 17
alignment of COMMON blocks, 19, 27
ALL, 115
all, 17
all warnings, 17
ALL()
   built-in function, 115
ALLOCATABLE components of derived types, 36
ALLOCATABLE dummy arguments, 36
ALLOCATABLE function results, 36
ALLOCATED, 116
ALLOCATED()
   built-in function, 116
allocation, 116, 236
allow-invalid-boz, 11
ALOG, 221
ALOG10, 222
AMAX0, 227
AMAX1, 227
AMIN0, 232
AMIN1, 232
AMOD, 234
ampersand, 17
AND, 117, 126, 196
AND of array elements, 196
AND with fetch, 129
AND()
   built-in function, 117
ANINT, 117
ANINT()
   built-in function, 117
ANY, 118
ANY()
   built-in function, 118
apply condition, 115, 118
Apredicate=answer, 15
A-predicate=answer, 15
-Apredicate=answer
   command line option, 15
area hyperbolic cosine, 111
area hyperbolic sine, 120
area hyperbolic tangent, 125
argument list functions, 56
arguments, 153, 188, 189, 199
array, 24, 36, 51, 115, 118, 159, 161, 170, 196, 198,
   207, 208, 214, 228, 229, 231, 233, 242, 246,
   253, 260, 266, 268, 271, 279, 280, 282
array temporaries, 17
ASCII, 109, 196
ASCII collating sequence, 109, 196
ASIN, 119
ASIN()  
   built-in function, 119
ASIND, 120
ASIND()
   built-in function, 120
ASINH, 120
ASINH()
   built-in function, 120
assertion, 15
ASSOCIATED, 121
ASSOCIATED()
   built-in function, 121
association status, 121, 141
asynchronous, 45
asynchronous I/O, 45
ATAN, 122
ATAN()
   built-in function, 122
ATAN2, 123
ATAN2()
   built-in function, 123
ATAN2D, 124
ATAN2D()
   built-in function, 124
ATAND, 122
ATAND()
   built-in function, 122
ATANH, 125
ATANH()
   built-in function, 125
Atomic subroutine, 125
ATOMIC_ADD, 125
ATOMIC_ADD()
   built-in function, 125
ATOMIC_AND, 126
ATOMIC_AND()
   built-in function, 126
ATOMIC_CAS()  
   built-in function, 127
ATOMIC_DEFINE, 127

Using GNU Fortran, Release 12.0.1 (experimental 20210414)

ATOMIC_DEFINE()
   built-in function, 127
ATOMIC_FETCH_ADD, 128
ATOMIC_FETCH_ADD()
   built-in function, 128
ATOMIC_FETCH_AND, 129
ATOMIC_FETCH_AND()
   built-in function, 129
ATOMIC_FETCH_OR, 130
ATOMIC_FETCH_OR()
   built-in function, 130
ATOMIC_FETCH_XOR, 130
ATOMIC_FETCH_XOR()
   built-in function, 130
ATOMIC_OR, 131
ATOMIC_OR()
   built-in function, 131
ATOMIC_REF, 132
ATOMIC_REF()
   built-in function, 132
ATOMIC_XOR, 133
ATOMIC_XOR()
   built-in function, 133
Authors, 291
AUTOMATIC, 61

B
BABS, 108
backslash, 12
BACKSPACE, 56
BACKTRACE, 133
backtrace, 21, 133
BACKTRACE()
   built-in function, 133
base, 247
base 10 logarithm function, 222
BBCLR, 199
BBITS, 200
BBSET, 200
BBTEST, 140
BESJ0, 134
BESJ1, 134
BESJN, 135
Bessel function, 134137
BESSEL_J0, 134
BESSEL_J0()
   built-in function, 134
BESSEL_J1, 134
BESSEL_J1()
   built-in function, 134
BESSEL_JN, 135
BESSEL_JN()
   built-in function, 135
BESSEL_Y0, 136
BESSEL_Y0()
   built-in function, 136
BESSEL_Y1, 136
BESSEL_Y1()
   built-in function, 136
BESSEL_YN, 137
BESSEL_YN()
   built-in function, 137
BESY0, 136
BESY1, 136
BESYN, 137
BGE, 138
BGE()
   built-in function, 138
BGT, 138
BGT()
   built-in function, 138
BIAND, 197
BIEOR, 203
binary representation, 244, 245
BIOR, 206
bit intrinsics checking, 24
BIT_SIZE, 139
BIT_SIZE()
   built-in function, 139
BITEST, 140
bits, 139, 140, 196, 198200, 207, 211, 224, 231, 236, 240, 254, 261, 262, 278
bits set, 244
bitwise, 63, 117, 197, 203, 206, 240, 241, 284
bitwise comparison, 138140
bitwise logical and, 117, 197
bitwise logical exclusive or, 203, 284
bitwise logical not, 240
bitwise logical or, 206, 241
BJTEST, 140
BKTEST, 140
BLE, 139
BLE()
   built-in function, 139
BLT, 140
BLT()
   built-in function, 140
BMOD, 234
BMVBITS, 236
BNOT, 240
bounds checking, 24
BOZ literal constants, 50
BSHFT, 211
BSHFTC, 211
BTEST, 140
BTEST()
   built-in function, 140
built-in function
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABORT()</td>
<td>107</td>
</tr>
<tr>
<td>ABS()</td>
<td>108</td>
</tr>
<tr>
<td>ACCESS()</td>
<td>109</td>
</tr>
<tr>
<td>ACHAR()</td>
<td>109</td>
</tr>
<tr>
<td>ACOS()</td>
<td>110</td>
</tr>
<tr>
<td>ACOSD()</td>
<td>111</td>
</tr>
<tr>
<td>ACOSH()</td>
<td>111</td>
</tr>
<tr>
<td>ADJUSTL()</td>
<td>112</td>
</tr>
<tr>
<td>ADJUSTR()</td>
<td>112</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>ACOS()</td>
<td>110</td>
</tr>
<tr>
<td>ACOSD()</td>
<td>111</td>
</tr>
<tr>
<td>ACOSH()</td>
<td>111</td>
</tr>
<tr>
<td>ADJUSTL()</td>
<td>112</td>
</tr>
<tr>
<td>ADJUSTR()</td>
<td>112</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>ACOS()</td>
<td>110</td>
</tr>
<tr>
<td>ACOSD()</td>
<td>111</td>
</tr>
<tr>
<td>ACOSH()</td>
<td>111</td>
</tr>
<tr>
<td>ADJUSTL()</td>
<td>112</td>
</tr>
<tr>
<td>ADJUSTR()</td>
<td>112</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>ACOS()</td>
<td>110</td>
</tr>
<tr>
<td>ACOSD()</td>
<td>111</td>
</tr>
<tr>
<td>ACOSH()</td>
<td>111</td>
</tr>
<tr>
<td>ADJUSTL()</td>
<td>112</td>
</tr>
<tr>
<td>ADJUSTR()</td>
<td>112</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>ACOS()</td>
<td>110</td>
</tr>
<tr>
<td>ACOSD()</td>
<td>111</td>
</tr>
<tr>
<td>ACOSH()</td>
<td>111</td>
</tr>
<tr>
<td>ADJUSTL()</td>
<td>112</td>
</tr>
<tr>
<td>ADJUSTR()</td>
<td>112</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>ACOS()</td>
<td>110</td>
</tr>
<tr>
<td>ACOSD()</td>
<td>111</td>
</tr>
<tr>
<td>ACOSH()</td>
<td>111</td>
</tr>
<tr>
<td>ADJUSTL()</td>
<td>112</td>
</tr>
<tr>
<td>ADJUSTR()</td>
<td>112</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>ACOS()</td>
<td>110</td>
</tr>
<tr>
<td>ACOSD()</td>
<td>111</td>
</tr>
<tr>
<td>ACOSH()</td>
<td>111</td>
</tr>
<tr>
<td>ADJUSTL()</td>
<td>112</td>
</tr>
<tr>
<td>ADJUSTR()</td>
<td>112</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>ACOS()</td>
<td>110</td>
</tr>
<tr>
<td>ACOSD()</td>
<td>111</td>
</tr>
<tr>
<td>ACOSH()</td>
<td>111</td>
</tr>
<tr>
<td>ADJUSTL()</td>
<td>112</td>
</tr>
<tr>
<td>ADJUSTR()</td>
<td>112</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>ACOS()</td>
<td>110</td>
</tr>
<tr>
<td>ACOSD()</td>
<td>111</td>
</tr>
<tr>
<td>ACOSH()</td>
<td>111</td>
</tr>
<tr>
<td>ADJUSTL()</td>
<td>112</td>
</tr>
<tr>
<td>ADJUSTR()</td>
<td>112</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>ACOS()</td>
<td>110</td>
</tr>
<tr>
<td>ACOSD()</td>
<td>111</td>
</tr>
<tr>
<td>ACOSH()</td>
<td>111</td>
</tr>
<tr>
<td>ADJUSTL()</td>
<td>112</td>
</tr>
<tr>
<td>ADJUSTR()</td>
<td>112</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>ACOS()</td>
<td>110</td>
</tr>
<tr>
<td>ACOSD()</td>
<td>111</td>
</tr>
<tr>
<td>ACOSH()</td>
<td>111</td>
</tr>
<tr>
<td>ADJUSTL()</td>
<td>112</td>
</tr>
<tr>
<td>ADJUSTR()</td>
<td>112</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>ACOS()</td>
<td>110</td>
</tr>
<tr>
<td>ACOSD()</td>
<td>111</td>
</tr>
<tr>
<td>ACOSH()</td>
<td>111</td>
</tr>
<tr>
<td>ADJUSTL()</td>
<td>112</td>
</tr>
<tr>
<td>ADJUSTR()</td>
<td>112</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>ACOS()</td>
<td>110</td>
</tr>
<tr>
<td>ACOSD()</td>
<td>111</td>
</tr>
<tr>
<td>ACOSH()</td>
<td>111</td>
</tr>
<tr>
<td>ADJUSTL()</td>
<td>112</td>
</tr>
<tr>
<td>ADJUSTR()</td>
<td>112</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>ACOS()</td>
<td>110</td>
</tr>
<tr>
<td>ACOSD()</td>
<td>111</td>
</tr>
<tr>
<td>ACOSH()</td>
<td>111</td>
</tr>
<tr>
<td>ADJUSTL()</td>
<td>112</td>
</tr>
<tr>
<td>ADJUSTR()</td>
<td>112</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>ACOS()</td>
<td>110</td>
</tr>
<tr>
<td>ACOSD()</td>
<td>111</td>
</tr>
<tr>
<td>ACOSH()</td>
<td>111</td>
</tr>
<tr>
<td>ADJUSTL()</td>
<td>112</td>
</tr>
<tr>
<td>ADJUSTR()</td>
<td>112</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>ACOS()</td>
<td>110</td>
</tr>
<tr>
<td>ACOSD()</td>
<td>111</td>
</tr>
<tr>
<td>ACOSH()</td>
<td>111</td>
</tr>
<tr>
<td>ADJUSTL()</td>
<td>112</td>
</tr>
<tr>
<td>ADJUSTR()</td>
<td>112</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>ACOS()</td>
<td>110</td>
</tr>
<tr>
<td>ACOSD()</td>
<td>111</td>
</tr>
<tr>
<td>ACOSH()</td>
<td>111</td>
</tr>
<tr>
<td>ADJUSTL()</td>
<td>112</td>
</tr>
<tr>
<td>ADJUSTR()</td>
<td>112</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>ACOS()</td>
<td>110</td>
</tr>
<tr>
<td>ACOSD()</td>
<td>111</td>
</tr>
<tr>
<td>ACOSH()</td>
<td>111</td>
</tr>
<tr>
<td>ADJUSTL()</td>
<td>112</td>
</tr>
<tr>
<td>ADJUSTR()</td>
<td>112</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
<tr>
<td>AIMAG()</td>
<td>113</td>
</tr>
</tbody>
</table>
Index 317

FLUSH(), 181
FNUM(), 182
FPUT(), 182
FPUTC(), 183
FRACTION(), 184
FREE(), 184
FSEEK(), 185
FSTAT(), 186
FTELL(), 186
GAMMA(), 187
GERROR(), 187
GET_COMMAND(), 189
GET_COMMAND_ARGUMENT(), 189
GET_ENVIRONMENT_VARIABLE(), 191
GETARG(), 188
GETCWD(), 190
GETENV(), 191
GETGID(), 192
GETLOG(), 192
GETPID(), 193
GETUID(), 193
GETTIME(), 194
HOSTNM(), 194
HUGE(), 195
HYPOT(), 195
IACHAR(), 196
IALL(), 196
IAND(), 197
IARGC(), 199
IBCLR(), 199
IBITS(), 200
IBSET(), 200
ICHAR(), 201
IDATE(), 202
IEOR(), 203
IERRNO(), 203
IMAGE_INDEX(), 204
INDEX(), 204
INT(), 205
INT2(), 206
INT8(), 206
IOR(), 206
IPARITY(), 207
IRAND(), 208
IS_CONTIGUOUS(), 208
IS_IOSTAT_END(), 209
IS_IOSTAT_EOR(), 210
ISATTY(), 210
ISHFT(), 211
ISHFTC(), 211
ISNAN(), 212
ITIME(), 213
KILL(), 213
KIND(), 214
LBOUND(), 214
LCOBOUND(), 215
LEADZ(), 215
LEN(), 216
LEN_TRIM(), 217
LGE(), 217
LGT(), 218
LINK(), 218
LLE(), 219
LRT(), 219
LNBLNK(), 220
LOC(), 220
LOG(), 221
LOG10(), 222
LOG_GAMMA(), 222
LOGICAL(), 223
LONG(), 223
LSHIFT(), 224
LSTAT(), 224
LTIME(), 225
MALLOC(), 225
MASKL(), 226
MASKR(), 226
MATMUL(), 227
MAX(), 227
MAXEXponent(), 228
MAXLOC(), 228
MAXVAL(), 229
MCLOCK(), 230
MCLOCK8(), 230
MERGE(), 231
MERGE_BITS(), 231
MIN(), 232
MINexponent(), 232
MINLOC(), 233
MINVAL(), 233
MOD(), 234
MODULO(), 235
MOVE_ALLOC(), 236
MVBITS(), 236
NEAREST(), 237
NEW_LINE(), 238
NINT(), 238
NORM2(), 239
NOT(), 240
NULL(), 240
NUM_IMAGES(), 241
OR(), 241
PACK(), 242
PARITY(), 243
PError(), 244
POPCNT(), 244
POPpar(), 245
PRECISION(), 245
PRESENT(), 246
PRODUCT(), 246
RADIX(), 247
RAN(), 247
RAND(), 248
RANDOM_INIT(), 248
RANDOM_NUMBER(), 249
RANDOM_SEED(), 249
RANGE(), 250
RANK(), 251
REAL(), 251
RENAME(), 252
REPEAT(), 253
RESHAPE(), 253
RRSPACING(), 254
RSHIFT(), 254
SAME_TYPE_AS(), 255
SCALE(), 255
SCAN(), 256
SECONDS(), 256
SECOND(), 257
SELECTED_CHAR_KIND(), 257
SELECTED_INT_KIND(), 258
SELECTED_REAL_KIND(), 259
SET_EXPONENT(), 259
SHAPE(), 260
SHIFTA(), 261
SHIFTL(), 261
SHIFTR(), 262
SIGN(), 262
SIGNAL(), 263
SIN(), 264
SIND(), 264
SINH(), 265
SIZE(), 266
SIZEOF(), 266
SLEEP(), 267
SPACING(), 267
SPREAD(), 268
SQRT(), 269
SRAND(), 269
STAT(), 270
STORAGE_SIZE(), 271
SUM(), 271
SYMLNK(), 272
SYSTEM(), 272
SYSTEM_CLOCK(), 273
TAN(), 274
TAND(), 274
TANH(), 275
THIS_IMAGE(), 275
TIME(), 276
TIME8(), 277
TINY(), 277
TRAILZ(), 278
TRANSFER(), 278
TRANSPOSE(), 279
TRIM(), 279
TTYNAM(), 280
UBOUND(), 280
UCOBOUND(), 281
UMASK(), 281
UNLINK(), 282
UNPACK(), 282
VERIFY(), 283
XOR(), 284

C

-C
command line option, 15
-C
command line option, 15
C address of pointers, 142
C address of procedures, 143
C association status, 143
C binding type, 17
C pointer, 141
C_ASSOCIATED, 141
C_ASSOCIATED()
built-in function, 141
C_F_POINTER, 142
C_F_POINTER()
built-in function, 142
C_F_PROCPOINTER, 142
C_F_PROCPOINTER(
built-in function, 142
C_FUNLOC, 143
C_FUNLOC()
built-in function, 143
C_LOC, 144
C_LOC()
built-in function, 144
C_SIZEOF, 145
C_SIZEOF()
built-in function, 145
CABS, 108
calling convention, 23
CARRIAGECONTROL, 63
CC, 16
-CC
command line option, 15
CCOS, 156
CCOSD, 157
CDABS, 108
CDCOS, 156
CDCOSD, 157
CDEXP, 176
CDLOG, 221
CDSIN, 264
CDSIND, 264
CDSQRT, 269
CEILING, 145
ceiling, 117, 145
CEILING() built-in function, 145
CEILING()
CEILING() built-in function, 145
CEXP, 176
change access mode, 147
change dimensions, 253
CHAR, 146
CHAR()
CHAR() built-in function, 146
character, 257
character kind, 257
character set, 12
character truncation, 18
CHDIR, 147
CHDIR()
CHDIR() built-in function, 147
checking array temporaries, 24
checking subscripts, 24
CHMOD, 147
CHMOD()
CHMOD() built-in function, 147
clear, 199
clock ticks, 230, 273
CLOG, 221
CMPLX, 148
CMPLX()
CMPLX() built-in function, 148
CO_BROADCAST, 149
CO_BROADCAST()
CO_BROADCAST() built-in function, 149
CO_MAX, 150
CO_MAX()
CO_MAX() built-in function, 150
CO_MIN, 150
CO_MIN()
CO_MIN() built-in function, 150
CO_REDUCE, 151
CO_REDUCE()
CO_REDUCE() built-in function, 151
CO_SUM, 152
CO_SUM()
CO_SUM() built-in function, 152
Coarray, 8893, 95106
cOADJOUT, 204, 215, 241, 275, 281
OADJOUT, 85
cOADJOUT, 24
code generation, 22
collating sequence, 109, 196
Collectives, 149152
combine arrays, 231
combined, 168
command line, 174
command line option
-Apredicate=answer, 15
-A-predicate=answer, 15
-C, 15
-CC, 15
-cpp, 14
-dD, 14
-dI, 14
-dM, 14
-dN, 14
-Dname, 16
-Dname=definition, 16
dU, 14
-aggressive-function-elimination, 27
-falign-commons, 27
-fall-intrinsics, 10
-fallow-argument-mismatch, 11
-fallow-invalid-boz, 11
-fbslash, 12
-fblas-matmul-limit=n, 26
-fbounds-check, 25
-fcheck=<keyword>, 24
-fcheck-array-temporaries, 25
-fcheck=conversion, 22
-fc-prototypes, 28
-fc-prototypes-external, 28
-fcray-pointer, 12
-fdebug-aux-vars, 20
-fdec, 11
-fdec-blank-format-item, 11
-fdec-char-conversions, 11
-fdec-format-defaults, 11
-fdec-include, 11
-fdec-intrinsic-ints, 11
-fdec-math, 11
-fdec-static, 11
-fdec-structure, 11
-fdefault-double-8, 13
-fdefault-integer-8, 13
-fdefault-real-8, 13
-fdefault-real-10, 13
-fdefault-real-16, 13
-fd-lines-as-code, 11
-fd-lines-as-comments, 11
-fdollar-ok, 11
-fdump-fortran-global, 20
-fdump-fortran-optimized, 20
-fdump-fortran-original, 20
-fdump-parse-tree, 20
-fexternal-blas, 26
Using GNU Fortran, Release 12.0.1 (experimental 20210414)
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

command options, 9
COMMAND_ARGUMENT_COUNT, 153
COMMAND_ARGUMENT_COUNT()  
    built-in function, 153
command-line arguments, 153, 188, 189, 199
COMMON, 66
compare and swap, 127
comparison, 217219
compiler, 154
compiler flags inquiry function, 154
COMPILER_OPTIONS, 154
COMPILER_OPTIONS()  
    built-in function, 154
COMPILER_VERSION, 154
COMPILER_VERSION()  
    built-in function, 154
complementary, 172
COMPLEX, 155
complex conjugate, 155
Complex function, 66
complex numbers, 113, 148, 155, 164, 167, 251
COMPLEX()  
    built-in function, 155
concatenate, 253
condition testing, 115, 118
Conditional compilation, 6
conditionally add elements, 271
conditionally count elements, 159
conditionally multiply elements, 246
CONJG, 155
CONJG()  
    built-in function, 155
consistency, 43
constructors, 36
contiguity, 208
Contributing, 291
Contributors, 291
conventions, 22
conversion, 18, 51, 52, 146, 148, 155, 162164, 196, 201, 205, 206, 223, 251
conversion to, 148, 155, 164
conversion to GMT info, 194
conversion to local time info, 225
conversion to string, 162
convert C to Fortran, 142, 144
CONVERT specifier, 54
core, 107
COS, 156
COS()  
    built-in function, 156
COSD, 157
COSD()  
    built-in function, 157
COSH, 157

D
DABS, 108
DACOS, 110
DACOSD, 111
DACOSH, 111
DASIN, 119
DASIND, 120
DASINH, 120
DATAN, 122
DATAN2, 123
DATAN2D, 124
DATAND, 122

Index 321
dummy argument, 19
dump, 107
duplicate dimensions, 268
duplicate elements, 268
durability, 43

E
elapsed, 160, 169, 173, 256, 257
elapsed time, 169, 256, 257
element counting, 159
Elimination of functions with identical argument lists, 28
enable, 14
ENCOD, 65
end of file, 209
end of record, 210
ENUM, 36
ENUM statement, 36
ENUMERATOR, 36
ENUMERATOR statement, 36
environment variable, 29, 31, 191
    CARRIAGECONTROL, 63
    CHARACTER, 41
    COMPLEX, 41
    GFORTRAN_CONVERT_UNIT, 33
    GFORTRAN_ERROR_BACKTRACE, 34
    GFORTRAN_FORMATTED_BUFFER_SIZE, 34
    GFORTRAN_UNBUFFERED_PRECONNECTED, 32
    GFORTRAN_UNFORMATTED_BUFFER_SIZE, 34
    INTEGER, 41
    LOGICAL, 41
    READONLY, 64
    REAL, 41
    SHARE, 64
    TEMP, 31
    TMP, 31
    TMPDIR, 31
EOF, 56
EOSHIFT, 170
EOSHIFT()  
    built-in function, 170
EPSILON, 170
epsilon, 170
EPSILON()  
    built-in function, 170
ERF, 171
ERF()  
    built-in function, 171
ERFC, 172
ERFC()  
    built-in function, 172
ERFC_SCALED, 172
ERFC_SCALED()  
    built-in function, 172
error, 16
error function, 171, 172
error handling, 187, 203, 244
errors, 16
escape characters, 12
ETIME, 173
ETIME()  
    built-in function, 173
Euclidean, 239
Euclidean distance, 195
Euclidean vector norm, 239
EVENT_QUERY, 173
EVENT_QUERY()  
    built-in function, 173
Events, 173
EXECUTE_COMMAND_LINE, 174
EXECUTE_COMMAND_LINE()  
    built-in function, 174
EXIT, 175
EXIT()  
    built-in function, 175
EXP, 176
EXP()  
    built-in function, 176
EXPONENT, 176
exponent, 65, 176
EXPONENT()  
    built-in function, 176
exponential function, 176, 221, 222
exponentially-scaled, 172
expression size, 145, 266
EXTENDS_TYPE_OF, 177
EXTENDS_TYPE_OF()  
    built-in function, 177
extensions, 47, 65
extra, 18
extra warnings, 18
extract, 200

F
f2c calling convention, 23, 24
Factorial function, 187
faggressive-function-elimination, 28
-faggressive-function-elimination command line option, 27
fall-intrinsics, 27
-fall-intrinsics command line option, 27
fallow-argument-mismatch, 11
-fallow-argument-mismatch command line option, 11
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

-fallow-invalid-boz
  command line option, 11
-fbackslash
  command line option, 12
-fblas-matmul-limit, 26
-fblas-matmul-limit=n
  command line option, 26
-fbounds-check, 25
-fbounds-check
  command line option, 25
-fcheck, 24
-fcheck=<keyword>
  command line option, 24
-fcheck-array-temporaries, 25
-fcheck-array-temporaries
  command line option, 25
-fcoarray, 24
-fcoarray=<keyword>
  command line option, 24
-fdecimal-char-conversions, 11
-fdecimal-char-conversions
  command line option, 11
-fdecimal-format-defaults, 11
-fdecimal-format-defaults
  command line option, 11
-fdecimal-intrinsic-ints, 11
-fdecimal-intrinsic-ints
  command line option, 11
-fdefault-double-8, 13
-fdefault-double-8
  command line option, 13
-fdefault-integer-8, 13
-fdefault-integer-8
  command line option, 13
-fdefault-real-8, 13
-fdefault-real-8
  command line option, 13
-fdefault-real-10, 13
-fdefault-real-10
  command line option, 13
-fdefault-real-16, 13
-fdefault-real-16
  command line option, 13
-fdebug-aux-vars, 20
-fdebug-aux-vars
  command line option, 20
-fdec, 11
-fdec
  command line option, 11
-fdec-blank-format-item, 11
-fdec-blank-format-item
  command line option, 11
-fdecimal-char-conversions, 11
-fdecimal-char-conversions
  command line option, 11
-fdefault-real-8, 13
-fdefault-real-8
  command line option, 13
-fdefault-real-10, 13
-fdefault-real-10
  command line option, 13
-fdefault-real-16, 13
-fdefault-real-16
  command line option, 13
-fconvert=conversion, 22
-fconvert=conversion
  command line option, 22
-fc-prototypes, 28
-fc-prototypes-external
  command line option, 28
-fcray-pointer, 12
-fcray-pointer
  command line option, 12
-FDATE, 177
-FDATE()
  built-in function, 177
-fdebug-aux-vars, 20
-fdebug-aux-vars
  command line option, 20
-fdec, 11
-fdec
  command line option, 11
-fdec-blank-format-item, 11
-fdec-blank-format-item
  command line option, 11
-fdefault-real-16, 13
-fdefault-real-16
  command line option, 13
-fdump-parse-tree, 20
-fdump-parse-tree
  command line option, 20
-fexternal-blas, 26
-fexternal-blas
  command line option, 26
-ff2c
  command line option, 23
-ffixed-form, 10
-ffixed-form
  command line option, 10
-ffixed-line-length-n, 12
-ffixed-line-length-n
command line option, 12
-ffpe-summary=list, 21
-ffpe-summary=list
command line option, 21
-ffpe-trap=list, 20
-ffpe-trap=list
command line option, 20
-ffree-form, 10
-ffree-form
command line option, 10
-ffree-line-length-n, 12
-ffree-line-length-n
command line option, 12
-frontend-loop-interchange
command line option, 28
-frontend-optimize
command line option, 28
FGET, 178
FGET() built-in function, 178
FGETC, 179
FGETC() built-in function, 179
file, 44
file creation mask, 281
file format, 10, 12
file number, 182
file operation, 178, 179, 181, 183, 185, 186
file status, 186, 224, 270
file system, 109, 147, 186, 218, 224, 252, 270, 272, 281, 282
fimplicit-none, 12
fimplicit-none command line option, 12
find missing set, 283
find non-blank character, 220
find subset, 256
find substring, 204
FINDLOC, 179
findloc, 179
FINDLOC() built-in function, 179
finit-character, 27
finit-character=n command line option, 27
finit-derived, 27
finit-derived command line option, 27
finit-integer, 27
finit-integer=n command line option, 27
finit-local-zero, 27
finit-local-zero command line option, 27
finit-logical, 27
finit-logical=<true|false> command line option, 27
finit-real, 27
finit-real=<zero|inf|-inf|nan|snan> command line option, 27
fline-arg-packing, 26
fline-arg-packing command line option, 26
fline-matmul-limit, 26
fline-matmul-limit=n command line option, 26
finteger-4-integer-8, 13
finteger-4-integer-8 command line option, 13
fintrinsic-modules-path dir command line option, 21
fintrinsic-modules-pathdir, 22
first kind, 134, 135
fixed, 10, 12
flags inquiry function, 154
FLOAT, 251
FLOATI, 251
floating point, 176, 184, 237, 254, 255, 259, 267
FLOATJ, 251
FLOATK, 251
FLOOR, 180
floor, 114, 180
FLOOR() built-in function, 180
FLUSH, 36, 181
flush, 181
FLUSH statement, 36
FLUSH() built-in function, 181
fmax-array-constructor, 25
fmax-array-constructor=n command line option, 25
fmax-errors=n, 16
fmax-errors=n command line option, 16
fmax-identifier-length=n, 12
fmax-identifier-length=n command line option, 12
fmax-stack-var-size, 25
fmax-stack-var-size=n command line option, 25
fmax-subrecord-length=length, 22
fmax-subrecord-length=length command line option, 22
fmodule-private, 12
fmodule-private command line option, 12
fno-automatic, 22
-fno-automatic
  command line option, 22
fno-backtrace, 21
-fno-backtrace
  command line option, 21
-fno-pad-source
  command line option, 12
fno-protect-parens, 27
-fno-protect-parens
  command line option, 27
-fno-range-check
  command line option, 12
fno-underscoring, 23
-fno-underscoring
  command line option, 23
FNUM, 182
FNUM()
  built-in function, 182
fopenacc, 12
-fopenacc
  command line option, 12
fopenmp, 12
-fopenmp
  command line option, 12
for included files, 21
form feed whitespace, 62
FORMAT, 66
Fortran, 28
Fortran 77, 7
Fortran dialect, 10
fpack-derived, 26
-fpack-derived
  command line option, 26
fpad-source, 12
FPP, 6
fpp, 14
FPUT, 182
FPUT()
  built-in function, 182
FPUTC, 183
FPUTC()
  built-in function, 183
FRACTION, 184
fraction, 184
FRACTION()
  built-in function, 184
frange-check, 13
freal-4-real-8, 13
-freal-4-real-8
  command line option, 13
freal-4-real-10, 13
-freal-4-real-10
  command line option, 13
freal-4-real-16, 13
-freal-4-real-16
  command line option, 13
freal-8-real-4, 13
-freal-8-real-4
  command line option, 13
freal-8-real-10, 13
-freal-8-real-10
  command line option, 13
freal-8-real-16, 13
-freal-8-real-16
  command line option, 13
frealloc-lhs, 27
-freallocate
  command line option, 27
frecord-marker=length, 22
-frecord-marker=length
  command line option, 22
frecursive, 27
-frecursive
  command line option, 27
FREE, 184
free, 10, 12
FREE()
  built-in function, 184
frepack-arrays, 26
-frepack-arrays
  command line option, 26
Front-end optimization, 28
frontend-loop-interchange, 28
frontend-optimize, 28
fsecond-underscore, 24
-fsecond-underscore
  command line option, 24
FSEEK, 185
FSEEK()
  built-in function, 185
fshort-enums, 26, 36
-fshort-enums
  command line option, 26
fsign-zero, 22
-fsign-zero
  command line option, 22
fstack-arrays, 25
-fstack-arrays
  command line option, 25
FSTAT, 186
FSTAT()
  built-in function, 186
fsyntax-only, 17
-fsyntax-only
  command line option, 17
command line option, 25
FTELL, 186
FTELL()
    built-in function, 186
f test-f or-all-temp, 14
-f test-f or-all-temp
    command line option, 14
function elimination, 19
f working-directory, 15
-f working-directory
    command line option, 15

G

-g77, 7
g77 calling convention, 23, 24
GAMMA, 187
Gamma function, 187, 222
GAMMA()
    built-in function, 187
gather elements, 242
GCC, 6
Generating C prototypes from external procedures, 28
Generating C prototypes from Fortran BIND(C) enteties, 28
generic reduction, 151
GERROR, 187
GERROR()
    built-in function, 187
g et, 200
GET_COMMAND, 189
GET_COMMAND()
    built-in function, 189
GET_COMMAND_ARGUMENT, 189
GET_COMMAND_ARGUMENT()
    built-in function, 189
GET_ENVIRONMENT_VARIABLE, 191
GET_ENVIRONMENT_VARIABLE()
    built-in function, 191
GETARG, 188
GETARG()
    built-in function, 188
GETCWD, 190
GETCWD()
    built-in function, 190
GETENV, 191
GETENV()
    built-in function, 191
GETGID, 192
GETGID()
    built-in function, 192
GETLOG, 192
GETLOG()
    built-in function, 192
GETPID, 193
GETPID()
    built-in function, 193
GETUID, 193
GETUID()
    built-in function, 193
gfortran command, 9
GFORTRAN_CONVERT_UNIT, 33
GFORTRAN_ERROR_BACKTRACE, 34
GFORTRAN_FORMATTED_BUFFER_SIZE, 34
GFORTRAN_UNBUFFERED_PRECONNECTED, 32
GFORTRAN_UNFORMATTED_BUFFER_SIZE, 34
GMTIME, 194
GMTIME()
    built-in function, 194
GNU Compiler Collection, 6
GNU Fortran command options, 9
group ID, 192

H

-H
    command line option, 16
hard link, 218
Hollerith constants, 51
host name, 194
HOSTNM, 194
HOSTNM()
    built-in function, 194
HUGE, 195
HUGE()
    built-in function, 195
hyperbolic, 111, 120, 125, 157, 265, 275
hyperbolic cosine, 157
hyperbolic function, 111, 120, 125, 157, 265, 275
hyperbolic sine, 265
hyperbolic tangent, 275
HYPOT, 195
HYPOT()
    built-in function, 195

I

I/O item lists, 50
I/O specifiers, 63
IABS, 108
IACHAR, 196
IACHAR()
    built-in function, 196
IALL, 196
IALL()
    built-in function, 196
IAND, 197
IAND()
    built-in function, 197

Index
INDEX(), 204
INDEX of this image, 275
INDEX (), built-in function, 204
indices of type real, 51
initialization, 248
INOT, 240
input/output, 45
INT, 205
INT (), built-in function, 205
INT2, 206
INT2 (), built-in function, 206
INT8, 206
INT8 ()
   built-in function, 206
integer, 60, 258
integer division, 18
integer kind, 258
Interoperability, 69
INTRINSIC, 36
intrinsic, 19
intrinsic Modules, 285
intrinsic procedures, 107
INTRINSIC statement, 36
intrinsics, 60, 62
intrinsics of other standards, 18
Introduction, 1
inverse, 110, 111, 119, 120, 122125, 176, 221, 222
inverse hyperbolic cosine, 111
inverse hyperbolic sine, 120
inverse hyperbolic tangent, 125
IOMSG= specifier, 36
IOR, 206
IOR ()
   built-in function, 206
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

I
IOSTAT, 209, 210
IPARITY, 207
IPARITY()
  built-in function, 207
iprefix prefix, 15
-iprefix prefix
  command line option, 15
iquote dir, 15
-iquote dir
  command line option, 15
IRAND, 208
IRAND()
  built-in function, 208
IS_CONTIGUOUS()
  built-in function, 208
IS_IOSTAT_END, 209
IS_IOSTAT_END()
  built-in function, 209
IS_IOSTAT_EOR, 208, 210
IS_IOSTAT_EOR()
  built-in function, 210
ISATTY, 210
ISATTY()
  built-in function, 210
ISHFT, 211
ISHFT()
  built-in function, 211
ISHFTC, 211
ISHFTC()
  built-in function, 211
ISIGN, 262
ISNAN, 212
ISNAN()
  built-in function, 212
ISO_FORTRAN_ENV, 36
ISO_FORTRAN_ENV statement, 36
isysroot dir, 15
-isysroot dir
  command line option, 15
isystem dir, 15
-system dir
  command line option, 15
ITIME, 213
ITIME()
  built-in function, 213

J
Jdir, 21
- Jdir
  command line option, 21
JIABS, 108
JIAND, 197
JIANDL, 199
JIBITS, 200
Using GNU Fortran, Release 12.0.1 (experimental 20210414)

lexical comparison of strings, 217
LGAMMA, 222
LGE, 217
LGE() built-in function, 217
LGT, 218
LGT() built-in function, 218
libf2c calling convention, 23, 24
libgfortran initialization, 77, 78, 80, 81
limiting, 16
limits, 195, 277
line truncation, 18
LINK, 218
LINK() built-in function, 218
linking, 22
LLE, 219
LLE() built-in function, 219
LLT, 219
LLT() built-in function, 219
LNBLNK, 220
LNBLNK() built-in function, 220
LOC, 62, 220
LOC() built-in function, 220
location of a variable in memory, 220
location of maximum element, 228
location of minimum element, 233
LOG, 221
LOG() built-in function, 221
LOG10, 222
LOG10() built-in function, 222
LOG_GAMMA, 222
LOG_GAMMA() built-in function, 222
logarithm function, 176, 221
logarithm function with base 10, 222
logarithm of, 222
LOGICAL, 223
logical, 42, 63
logical and, 117, 197
logical exclusive or, 203, 284
logical not, 240
logical or, 206, 241
LOGICAL() built-in function, 223
login name, 192
LONG, 223
LONG() built-in function, 223
loop interchange, 18, 28
lower bound, 214, 215
LSHIFT, 224
LSHIFT() built-in function, 224
LSTAT, 224
LSTAT() built-in function, 224
LTIME, 225
LTIME() built-in function, 225
M
MALLOCS, 225
MALLOCS() built-in function, 225
MAP, 58
mask, 226
MASKL, 226
MASKL() built-in function, 226
MASKR, 226
MASKR() built-in function, 226
math, 62
MATMUL, 227
MATMUL() built-in function, 227
matrix, 227, 279
matrix multiplication, 227
MAX, 42, 227
MAX() built-in function, 227
MAX0, 227
MAX1, 227
MAXEXPONENT, 228
MAXEXPONENT() built-in function, 228
maximal value, 150
maximum exponent, 228
maximum value, 227, 229
MAXLOC, 228
MAXLOC() built-in function, 228
MAXVAL, 229
MAXVAL() built-in function, 229
MCLOCK, 230
MCLOCK() built-in function, 230
MCLOCK8, 230
MCLOCK8()
built-in function, 230
Mdir, 21
memory checking, 24
MERGE, 231
merge, 231
merge arrays, 231
MERGE()
   built-in function, 231
MERGE_BITS, 231
MERGE_BITS()
   built-in function, 231
messages, 16
MIN, 42, 232
MIN()
   built-in function, 232
MIN0, 232
MIN1, 232
MINEXPONENT, 232
MINEXPONENT()
   built-in function, 232
minimal value, 150
minimum exponent, 232
minimum value, 232, 233
MINLOC, 233
MINLOC()
   built-in function, 233
MINVAL, 233
MINVAL()
   built-in function, 233
Mixed-language programming, 69
MOD, 234
MOD()
   built-in function, 234
model representation, 164, 170, 195, 228, 232, 245, 247, 250, 277
module entities, 12
module search path, 21, 22
MODULO, 235
modulo, 235
MODULO()
   built-in function, 235
move, 236, 278
MOVE_ALLOC, 236
MOVE_ALLOC()
   built-in function, 236
moving, 236
moving allocation, 236
multiply array elements, 246
multiply elements, 246
MVBITS, 236
MVBITS()
   built-in function, 236

N
NAME, 66
name and version, 154
Namelist, 48
NaN, 42
natural logarithm function, 221
NEAREST, 237
nearest different, 237
nearest whole number, 238
NEAREST()
   built-in function, 237
negate, 240
negative forms, 9
NEW_LINE, 238
NEW_LINE()
   built-in function, 238
newline, 238
NINT, 238
NINT()
   built-in function, 238
no linemarkers, 16
-nocpp
   command line option, 14
non-standard intrinsics, 18
norm, 239
NORM2, 239
NORM2()
   built-in function, 239
NOSHARED, 63
nostdinc, 15
-nostdinc
   command line option, 15
NOT, 240
not implemented, 65
NOT()
   built-in function, 240
notification, 19
NULL, 240
NULL()
   built-in function, 240
NUM_IMAGES, 241
NUM_IMAGES()
   built-in function, 241
number of, 139, 153, 199, 241
number of elements, 159, 266

O
old-style, 47
open, 44
OpenACC, 12, 55
OpenMP, 12, 55
operators, 51, 63
options, 9, 10, 13, 14, 16, 202
options inquiry function, 154
OR, 131, 198, 241
OR of array elements, 198
OR with fetch, 130
OR()
   built-in function, 241
output, 238
overwrite recursive, 18

P
P, 16
-P
   command line option, 16
PACK, 242
PACK()
   built-in function, 242
packing, 242
PARAMETER, 64
PARITY, 243
Parity, 243
parity, 207, 245
PARITY()
   built-in function, 243
paths, 21, 22
pedantic, 17
   -pedantic
       command line option, 17
pedantic-errors, 17
   -pedantic-errors
       command line option, 17
permutation, 161
FERROR, 244
FERROR()
   built-in function, 244
pointer, 52, 121, 141143, 184, 225, 240
pointer checking, 24
POPCNT, 244
POPCNT()
   built-in function, 244
POPPAR, 245
POPPAR()
   built-in function, 245
position, 185, 186
positive difference, 165
PRECISION, 245
precision, 245
PRECISION()
   built-in function, 245
Preprocessing, 6
preprocessing, 15, 16
preprocessor, 6, 14, 15
PRESENT, 246
PRESENT()
   built-in function, 246
private, 12

procedure pointer, 144
process ID, 193
PRODUCT, 246
product, 166, 227, 246
PRODUCT()
   built-in function, 246
program termination, 107, 175
PROTECTED, 36
PROTECTED statement, 36

Q
Q edit descriptor, 67
Q exponent-letter, 50
q exponent-letter, 18

R
RADIX, 247
radix, 247, 259
RADIX()
   built-in function, 247
RAN, 247
RAN()
   built-in function, 247
RAND, 248
RANDD, 248
RAND()  
   built-in function, 248
random number generation, 208, 247249, 269
RANDOM_INIT, 248
RANDOM_INIT()
   built-in function, 248
RANDOM_NUMBER, 249
RANDOM_NUMBER()
   built-in function, 249
RANDOM_SEED, 249
RANDOM_SEED()
   built-in function, 249
RANGE, 250
range, 250
range checking, 24
RANGE()
   built-in function, 250
RANK, 251
rank, 251
RANK()
   built-in function, 251
read character, 178, 179
READONLY, 63
REAL, 251
real, 259
real kind, 259
real kind type promotion, 13
real number, 176, 184, 237, 254, 255, 259, 267
real part, 167, 251
REAL()
built-in function, 251
Reallocate the LHS in assignments, 19, 27
REALPART, 251
re-association of parenthesized expressions, 27
RECORD, 56
record marker, 44
reduce dimension, 242
Reduction, 243
reference, 132
relative spacing, 254, 267
remainder, 234
remove file, 282
remove trailing whitespace, 279
RENAME, 252
rename file, 252
RENAME() built-in function, 252
repacking arrays, 26
REPEAT, 253
repeat, 253
REPEAT() built-in function, 253
RESHAPE, 253
RESHAPE() built-in function, 253
REWIND, 56
right, 168, 262
right justified, 226
right shift, 168
right with fill, 261
root, 269
rotate, 161
rounding, 114, 117, 145, 180, 238
RRSpacing, 254
RRSpacing() built-in function, 254
RSHIFT, 254
RSHIFT() built-in function, 254
runtime, 22
run-time, 22
run-time checking, 24
S
SAME_TYPE_AS, 255
SAME_TYPE_AS() built-in function, 255
SAVE, 22
SAVE statement, 22
SCALE, 255
scale, 255
SCALE() built-in function, 255
SCAN, 256
SCAN() built-in function, 256
scatter elements, 282
search, 21, 22
search path, 21
search paths, 21
search paths for inclusion, 21
SECNDS, 256
SECNDS() built-in function, 256
SECOND, 257
second kind, 136, 137
SECOND() built-in function, 257
seeding, 249, 269
seeding a random number generator, 249, 269
seek, 185
SELECTED_CHAR_KIND, 257
SELECTED_CHAR_KIND() built-in function, 257
SELECTED_INT_KIND, 258
SELECTED_INT_KIND() built-in function, 258
SELECTED_REAL_KIND, 259
SELECTED_REAL_KIND() built-in function, 259
sequential, 44
set, 200
set exponent, 259
set_args, 77
set_convert, 80
SET_EXPONENT, 259
SET_EXPONENT() built-in function, 259
set_fpe, 80
set_max_subrecord_length, 81
set_options, 78
set_record_marker, 80
SHAPE, 260
shape, 260
SHAPE() built-in function, 260
SHARE, 63
SHARED, 63
shift, 168, 170, 211, 261, 262
shift circular, 211
shift circularly, 161
shift left, 224, 261
shift right, 254, 261, 262
SHIFTA, 261
SHIFTA() built-in function, 261
SHIFTL, 261
SHIFTL()  
    built-in function, 261
SHIFTR, 262
SHIFTR()  
    built-in function, 262
SHORT, 206
SIGN, 262
sign copying, 262
SIGN()  
    built-in function, 262
SIGNAL, 263
signal handling, 263
SIGNAL()  
    built-in function, 263
SIGNIFICANT DIGITS, 164
SIN, 264
SIN()  
    built-in function, 264
SIND, 264
SIND()  
    built-in function, 264
sine, 119, 120, 264, 265
SINH, 265
SINH()  
    built-in function, 265
SIZE, 266
size, 266
size of a variable, 139
size of an expression, 145, 266
SIZE()  
    built-in function, 266
SIZEOF, 266
SIZEOF()  
    built-in function, 266
SLEEP, 267
SLEEP()  
    built-in function, 267
smallest number, 277
SNGL, 251
soft link, 272
SPACING, 267
SPACING()  
    built-in function, 267
SPREAD, 268
SPREAD()  
    built-in function, 268
SQRT, 269
SQRT()  
    built-in function, 269
square-root, 269
SRAND, 269
SRAND()  
    built-in function, 269

Standards, 8
STAT, 270
STAT()  
    built-in function, 270
statement, 22, 36
STATIC, 61
static, 22
static-libgfortran, 22
-static-libgfortran 
    command line option, 22
status, 116, 121, 240
-std=std 
    command line option, 14
std=std option, 14
storage size, 271
STORAGE_SIZE, 271
STORAGE_SIZE()  
    built-in function, 271
STREAM I/O, 36
stream mode, 178, 179, 182, 183
string, 112, 204, 216220, 253, 256, 279, 283
strings, 8
STRUCTURE, 56
structure packing, 26
subrecord, 44
subscript checking, 24
substring position, 204
SUM, 271
sum, 271
sum array elements, 271
sum of values, 152
SUM()  
    built-in function, 271
suppressing, 16
suppressing warnings, 16
suspicious code, 18
symbol names, 12, 23, 24
symbolic link, 44
SYMLNK, 272
SYMLNK()  
    built-in function, 272
syntax checking, 17
SYSTEM, 272
system, 147, 174, 187, 190, 192194, 203, 210, 244, 263,
272, 280
system call, 174, 272
SYSTEM()  
    built-in function, 272
SYSTEM_CLOCK, 273
SYSTEM_CLOCK()  
    built-in function, 273

T

tabs, 19
tabulators, 19
tail-call-workaround, 25
TAN, 274
TAN ()
  built-in function, 274
TAND, 274
TAND ()
  built-in function, 274
tangent, 122125, 274, 275
TAN, 275
TANH ()
  built-in function, 275
TEMP, 31
terminal, 210, 280
terminate program, 107, 175
testing, 140
THIS_IMAGE, 275
THIS_IMAGE ()
  built-in function, 275
threads, 42
thread-safety, 42
TIME, 276
time, 160, 162, 169, 173, 177, 194, 213, 225, 230, 256,
  257, 273, 276, 277
TIME ()
  built-in function, 276
TIME8, 277
TIME8 ()
  built-in function, 277
TINY, 277
TINY ()
  built-in function, 277
TMP, 31
TMPDIR, 31
to character, 52, 146
to complex, 148, 155, 164
to errors, 20
to integer, 51, 196, 201, 205, 206, 223
to logical, 51, 223
to program, 153, 188, 189, 199
to real, 163, 251
to string, 162
TR 15581, 36
trace, 21
TRAILZ, 278
TRAILZ ()
  built-in function, 278
TRANSFER, 278
TRANSFER ()
  built-in function, 278
transforming, 23, 24
transforming symbol names, 23, 24
transmogrify, 253
TRANSPOSE, 279
TRANSPOSE ()
  built-in function, 279
trigonometric function, 110, 111, 119, 120,
  122124, 156159, 264, 274
trigonometric functions, 62
TRIM, 279
TRIM ()
  built-in function, 279
TTYNAM, 280
TTYNAM ()
  built-in function, 280
type alias print, 62
type cast, 278
U
UBOUND, 280
UBOUND ()
  built-in function, 280
UCOBOUND, 281
UCOBOUND ()
  built-in function, 281
UMASK, 281
UMASK ()
  built-in function, 281
Uname, 16
  -Uname
    command line option, 16
unary, 51
  -undef
    command line option, 15
undefine macros, 16
undefined do loop, 19
underflow, 19
underscore, 23, 24
underscores, 23, 24
unformatted, 44
unformatted sequential, 44
UNION, 58
UNLINK, 282
UNLINK ()
  built-in function, 282
UNPACK, 282
UNPACK ()
  built-in function, 282
unpacking, 282
unset, 199
unused, 19
unused dummy argument, 19
unused parameter, 19
upper bound, 280, 281
USE, 36
use statements, 19
user ID, 193
user id, 193

\( V \)
VALUE, 36
value broadcasting, 149
VALUE statement, 36
variable attributes, 61
variable representation, 42
varying length, 8
Varying length character strings, 8
Varying length strings, 8
vector, 166
vector product, 166
VERIFY, 283
VERIFY()
   built-in function, 283
version of the compiler, 154
VOLATILE, 36, 66
VOLATILE statement, 36

\( W \)
Waliasing, 17
-Waliasing
   command line option, 17
Walign-commons, 19
-Walign-commons
   command line option, 19
Wall, 17
-Wall
   command line option, 17
Wampersand, 17
-Wampersand
   command line option, 17
warning, 1618
warnings, 1620
Warray-temporaries, 17
-Warray-temporaries
   command line option, 17
Wc-binding-type, 17
-Wc-binding-type
   command line option, 17
Wcharacter-truncation, 18
-Wcharacter-truncation
   command line option, 17
Wcompare-reals, 19
-Wcompare-reals
   command line option, 19
Wconversion, 18
-Wconversion
   command line option, 18
Wconversion-extra, 18
-Wconversion-extra
   command line option, 18
Wdo-subscript, 20
-Wdo-subscript
   command line option, 19
Werror, 20
-Werror
   command line option, 20
Wextra, 18
-Wextra
   command line option, 18
Wfrontend-loop-interchange, 18
-Wfrontend-loop-interchange
   command line option, 18
Wfunction-elimination, 19
-Wfunction-elimination
   command line option, 19
Wimplicit-interface, 18
-Wimplicit-interface
   command line option, 18
Wimplicit-procedure, 18
-Wimplicit-procedure
   command line option, 18
Winteger-division, 18
-Winteger-division
   command line option, 18
W intrinsic-shadow, 19
-W intrinsic-shadow
   command line option, 19
Wintrinsics-std, 18
-Wintrinsics-std
   command line option, 18
with core dump, 107
without trailing whitespace, 217
Wline-truncation, 18
-Wline-truncation
   command line option, 18
Wno-overwrite-recursive
   command line option, 18
Woverwrite-recursive, 18
Wpedantic, 17
-Wpedantic
   command line option, 17
Wrealloc-lhs, 19
-Wrealloc-lhs
   command line option, 19
Wrealloc-lhs-all, 19
-Wrealloc-lhs-all
   command line option, 19
Wreal-q-constant, 18
-Wreal-q-constant
   command line option, 18
write character, 182, 183
Wsurprising, 18
-Wsurprising
command line option, 18
Wtabs, 19
-Wtabs
  command line option, 18
-Wtarget-lifetime
  command line option, 19
Wtarget-lifetime, 19
Wundefined-do-loop, 19
-Wundefined-do-loop
  command line option, 19
Wunderflow, 19
-Wunderflow
  command line option, 19
Wunused-dummy-argument, 19
-Wunused-dummy-argument
  command line option, 19
Wunused-parameter, 19
-Wunused-parameter
  command line option, 19
Wuse-without-only, 19
-Wuse-without-only
  command line option, 19
Wzerotrip, 19
-Wzerotrip
  command line option, 19

X
XOR, 133, 207, 243, 284
xor, 63
XOR of array elements, 207
XOR reduction, 243
XOR with fetch, 130
XOR()
  built-in function, 284

Z
ZABS, 108
ZCOS, 156
ZCOSD, 157
zero bits, 215, 278
ZEXP, 176
ZLOG, 221
ZSIN, 264
ZSIND, 264
ZSQRT, 269