Using GNU Fortran

For GCC version 9.0.1 (pre-release)

The gfortran team
## Short Contents

1. **Introduction** .......................................................... 1

   **Invoking GNU Fortran**

2. GNU Fortran Command Options .............................. 7

3. Runtime: Influencing runtime behavior with environment variables ........................................ 31

   **Language Reference**

4. Fortran standards status ........................................ 37

5. Compiler Characteristics ......................................... 43

6. Extensions ............................................................. 49

7. Mixed-Language Programming ............................... 69

8. Coarray Programming ............................................. 83

9. Intrinsic Procedures ............................................... 107

10. Intrinsic Modules .................................................. 283

Contributing ............................................................. 289

GNU General Public License .................................... 293

GNU Free Documentation License ............................. 305

Funding Free Software ........................................... 313

Option Index .......................................................... 315

Keyword Index ......................................................... 317
Table of Contents

1 Introduction ............................................. 1
   1.1 About GNU Fortran ..................................... 1
   1.2 GNU Fortran and GCC .................................... 2
   1.3 Preprocessing and conditional compilation .............. 2
   1.4 GNU Fortran and G77 .................................... 3
   1.5 Project Status .......................................... 3
   1.6 Standards ............................................... 4
       1.6.1 Varying Length Character Strings .................. 4

Part I: Invoking GNU Fortran .............................. 5

2 GNU Fortran Command Options ..................... 7
   2.1 Option summary ............................................. 7
   2.2 Options controlling Fortran dialect .................... 8
   2.3 Enable and customize preprocessing ..................... 13
   2.4 Options to request or suppress errors and warnings ..... 15
   2.5 Options for debugging your program or GNU Fortran ..... 20
   2.6 Options for directory search ............................ 21
   2.7 Influencing the linking step ............................. 22
   2.8 Influencing runtime behavior ............................. 22
   2.9 Options for code generation conventions ............... 23
   2.10 Options for interoperability with other languages ..... 29
   2.11 Environment variables affecting gfortran .............. 29

3 Runtime: Influencing runtime behavior with
   environment variables ................................. 31
   3.1 TMPDIR—Directory for scratch files .................. 31
   3.2 GFORTRAN_STDIN_UNIT—Unit number for standard input .... 31
   3.3 GFORTRAN_STDOUT_UNIT—Unit number for standard output ... 31
   3.4 GFORTRAN_STDERR_UNIT—Unit number for standard error ... 31
   3.5 GFORTRAN_UNBUFFERED_ALL—Do not buffer I/O on all units ... 31
   3.6 GFORTRAN_UNBUFFERED_PRECONNECTED—Do not buffer I/O on
       preconnected units ............................................. 31
   3.7 GFORTRAN_SHOW_LOCUS—Show location for runtime errors ..... 31
   3.8 GFORTRAN_OPTIONAL_PLUS—Print leading + where permitted .... 32
   3.9 GFORTRAN_LIST_SEPARATOR—Separator for list output .......... 32
   3.10 GFORTRAN_CONVERT_UNIT—Set endianness for unformatted I/O
       ......................................................................... 32
   3.11 GFORTRAN_ERROR_BACKTRACE—Show backtrace on run-time errors
       ......................................................................... 33

Part II: Language Reference .......................... 35
4 Fortran standards status ..................... 37
  4.1 Fortran 2003 status .................................. 37
  4.2 Fortran 2008 status .................................. 38
  4.3 Status of Fortran 2018 support .................. 40
    4.3.1 TS 29113 Status (Further Interoperability with C) ........ 40
    4.3.2 TS 18508 Status (Additional Parallel Features) ........ 41

5 Compiler Characteristics ....................... 43
  5.1 KIND Type Parameters ............................... 43
  5.2 Internal representation of LOGICAL variables .......... 43
  5.3 Evaluation of logical expressions .................. 44
  5.4 MAX and MIN intrinsics with REAL NaN arguments ...... 44
  5.5 Thread-safety of the runtime library ............... 44
  5.6 Data consistency and durability .................... 45
  5.7 Files opened without an explicit ACTION= specifier .... 46
  5.8 File operations on symbolic links .................. 46
  5.9 File format of unformatted sequential files .......... 46
  5.10 Asynchronous I/O .................................. 47

6 Extensions ............................................. 49
  6.1 Extensions implemented in GNU Fortran ............... 49
    6.1.1 Old-style kind specifications .................. 49
    6.1.2 Old-style variable initialization ............... 49
    6.1.3 Extensions to namelist .......................... 50
    6.1.4 X format descriptor without count field .......... 51
    6.1.5 Commas in FORMAT specifications ................. 51
    6.1.6 Missing period in FORMAT specifications ........ 51
    6.1.7 I/O item lists .................................. 51
    6.1.8 Q exponent-letter ................................ 51
    6.1.9 BOZ literal constants ............................ 51
    6.1.10 Real array indices ................................ 52
    6.1.11 Unary operators .................................. 52
    6.1.12 Implicitly convert LOGICAL and INTEGER values .... 52
    6.1.13 Hollerith constants support ..................... 53
    6.1.14 Cray pointers .................................... 53
    6.1.15 CONVERT specifier ................................ 55
    6.1.16 OpenMP .......................................... 56
    6.1.17 OpenACC .......................................... 56
    6.1.18 Argument list functions %VAL, %REF and %LOC ...... 57
    6.1.19 Read/Write after EOF marker ..................... 57
    6.1.20 STRUCTURE and RECORD ............................ 57
    6.1.21 UNION and MAP .................................... 60
    6.1.22 Type variants for integer intrinsics ............. 62
    6.1.23 AUTOMATIC and STATIC attributes ............... 63
    6.1.24 Extended math intrinsics ....................... 64
    6.1.25 Form feed as whitespace .......................... 64
    6.1.26 TYPE as an alias for PRINT ..................... 64
6.1.27 %LOC as an rvalue ........................................ 64
6.1.28 .XOR. operator ........................................... 65
6.1.29 Bitwise logical operators ................................ 65
6.1.30 Extended I/O specifiers .................................. 65
6.1.31 Legacy PARAMETER statements ......................... 67
6.1.32 Default exponents ....................................... 67
6.2 Extensions not implemented in GNU Fortran ............. 67
  6.2.1 ENCODE and DECODE statements ...................... 67
  6.2.2 Variable FORMAT expressions ......................... 68
  6.2.3 Alternate complex function syntax ..................... 68
  6.2.4 Volatile COMMON blocks ................................ 68
  6.2.5 OPEN(... NAME=) ...................................... 68

7 Mixed-Language Programming ................................. 69
  7.1 Interoperability with C .................................. 69
    7.1.1 Intrinsic Types ...................................... 69
    7.1.2 Derived Types and struct ............................ 69
    7.1.3 Interoperable Global Variables ...................... 70
    7.1.4 Interoperable Subroutines and Functions ............ 70
    7.1.5 Working with Pointers ................................ 72
    7.1.6 Further Interoperability of Fortran with C .......... 74
  7.2 GNU Fortran Compiler Directives ......................... 75
    7.2.1 ATTRIBUTES directive ................................ 75
    7.2.2 UNROLL directive ..................................... 76
  7.3 Non-Fortran Main Program ................................ 76
    7.3.1 _gfortran_set_args — Save command-line arguments ... 77
    7.3.2 _gfortran_set_options — Set library option flags .... 77
    7.3.3 _gfortran_set_convert — Set endian conversion ....... 78
    7.3.4 _gfortran_set_record_marker — Set length of record markers .................................................. 79
    7.3.5 _gfortran_set_fpe — Enable floating point exception traps ......................................................... 79
    7.3.6 _gfortran_set_max_subrecord_length — Set subrecord length ....................................................... 79
  7.4 Naming and argument-passing conventions ................ 80
    7.4.1 Naming conventions .................................... 80
    7.4.2 Argument passing conventions ........................ 80

8 Coarray Programming ............................................ 83
  8.1 Type and enum ABI Documentation ......................... 83
    8.1.1 caf_token_t ........................................... 83
    8.1.2 caf_register_t ......................................... 83
    8.1.3 caf_deregister_t ....................................... 83
    8.1.4 caf_reference_t ........................................ 83
    8.1.5 caf_team_t .............................................. 85
  8.2 Function ABI Documentation ............................... 85
    8.2.1 _gfortran_caf_init — Initialisation function ....... 86
    8.2.2 _gfortran_caf_finish — Finalization function ...... 86
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.2.3</td>
<td><code>.gfortran_caf_this_image</code> — Querying the image number</td>
</tr>
<tr>
<td>8.2.4</td>
<td><code>.gfortran_caf_num_images</code> — Querying the maximal number of images</td>
</tr>
<tr>
<td>8.2.5</td>
<td><code>.gfortran_caf_image_status</code> — Query the status of an image</td>
</tr>
<tr>
<td>8.2.6</td>
<td><code>.gfortran_caf_failed_images</code> — Get an array of the indexes of the failed images</td>
</tr>
<tr>
<td>8.2.7</td>
<td><code>.gfortran_caf_stopped_images</code> — Get an array of the indexes of the stopped images</td>
</tr>
<tr>
<td>8.2.8</td>
<td><code>.gfortran_caf_register</code> — Registering coarrays</td>
</tr>
<tr>
<td>8.2.9</td>
<td><code>.gfortran_caf_deregister</code> — Deregistering coarrays</td>
</tr>
<tr>
<td>8.2.10</td>
<td><code>.gfortran_caf_is_present</code> — Query whether an allocatable or pointer component in a derived type coarray is allocated</td>
</tr>
<tr>
<td>8.2.11</td>
<td><code>.gfortran_caf_send</code> — Sending data from a local image to a remote image</td>
</tr>
<tr>
<td>8.2.12</td>
<td><code>.gfortran_caf_get</code> — Getting data from a remote image</td>
</tr>
<tr>
<td>8.2.13</td>
<td><code>.gfortran_caf_sendget</code> — Sending data between remote images</td>
</tr>
<tr>
<td>8.2.14</td>
<td><code>.gfortran_caf_send_by_ref</code> — Sending data from a local image to a remote image with enhanced referencing options</td>
</tr>
<tr>
<td>8.2.15</td>
<td><code>.gfortran_caf_get_by_ref</code> — Getting data from a remote image using enhanced references</td>
</tr>
<tr>
<td>8.2.16</td>
<td><code>.gfortran_caf_sendget_by_ref</code> — Sending data between remote images using enhanced references on both sides</td>
</tr>
<tr>
<td>8.2.17</td>
<td><code>.gfortran_caf_lock</code> — Locking a lock variable</td>
</tr>
<tr>
<td>8.2.18</td>
<td><code>.gfortran_caf_lock</code> — Unlocking a lock variable</td>
</tr>
<tr>
<td>8.2.19</td>
<td><code>.gfortran_caf_event_post</code> — Post an event</td>
</tr>
<tr>
<td>8.2.20</td>
<td><code>.gfortran_caf_event_wait</code> — Wait that an event occurred</td>
</tr>
<tr>
<td>8.2.21</td>
<td><code>.gfortran_caf_event_query</code> — Query event count</td>
</tr>
<tr>
<td>8.2.22</td>
<td><code>.gfortran_caf_sync_all</code> — All-image barrier</td>
</tr>
<tr>
<td>8.2.23</td>
<td><code>.gfortran_caf_sync_images</code> — Barrier for selected images</td>
</tr>
<tr>
<td>8.2.24</td>
<td><code>.gfortran_caf_sync_memory</code> — Wait for completion of segment-memory operations</td>
</tr>
<tr>
<td>8.2.25</td>
<td><code>.gfortran_caf_error_stop</code> — Error termination with exit code</td>
</tr>
<tr>
<td>8.2.26</td>
<td><code>.gfortran_caf_error_stop_str</code> — Error termination with string</td>
</tr>
<tr>
<td>8.2.27</td>
<td><code>.gfortran_caf_fail_image</code> — Mark the image failed and end its execution</td>
</tr>
<tr>
<td>8.2.28</td>
<td><code>.gfortran_caf_atomic_define</code> — Atomic variable assignment</td>
</tr>
<tr>
<td>8.2.29</td>
<td><code>.gfortran_caf_atomic_ref</code> — Atomic variable reference</td>
</tr>
</tbody>
</table>
8.2.30 _gfortran_caf_atomic_cas — Atomic compare and swap ........................................................... 102
8.2.31 _gfortran_caf_atomic_op — Atomic operation .......................... 103
8.2.32 _gfortran_caf_co_broadcast — Sending data to all images .................................................... 103
8.2.33 _gfortran_caf_co_max — Collective maximum reduction ........................................................... 104
8.2.34 _gfortran_caf_co_min — Collective minimum reduction ........................................................... 104
8.2.35 _gfortran_caf_co_sum — Collective summing reduction ........................................................... 105
8.2.36 _gfortran_caf_co_reduce — Generic collective reduction ........................................................... 105

9 Intrinsic Procedures ........................................ 107
9.1 Introduction to intrinsic procedures .......................... 107
9.2 ABORT — Abort the program .......................... 107
9.3 ABS — Absolute value ........................................ 108
9.4 ACCESS — Checks file access modes .......................... 109
9.5 ACHAR — Character in ASCII collating sequence .......................... 109
9.6 ACOS — Arccosine function .......................... 110
9.7 ACOSD — Arccosine function, degrees .......................... 111
9.8 ACOSH — Inverse hyperbolic cosine function ......................... 111
9.9 ADJUSTL — Left adjust a string .......................... 112
9.10 ADJUSTR — Right adjust a string .......................... 112
9.11 AIMAG — Imaginary part of complex number .......................... 113
9.12 AINT — Truncate to a whole number .......................... 113
9.13 ALARM — Execute a routine after a given delay .......................... 114
9.14 ALL — All values in MASK along DIM are true ................. 115
9.15 ALLOCATED — Status of an allocatable entity ......................... 116
9.16 AND — Bitwise logical AND .......................... 116
9.17 ANINT — Nearest whole number .......................... 117
9.18 ANY — Any value in MASK along DIM is true ...................... 118
9.19 ASIN — Arccsine function .......................... 119
9.20 ASIND — Arcsine function, degrees .......................... 119
9.21 ASINH — Inverse hyperbolic sine function ......................... 120
9.22 ASSOCIATED — Status of a pointer or pointer/target pair .... 120
9.23 ATAN — Arctangent function .......................... 122
9.24 ATAND — Arctangent function, degrees .......................... 122
9.25 ATAN2 — Arctangent function .......................... 123
9.26 ATAN2D — Arctangent function, degrees .......................... 124
9.27 ATANH — Inverse hyperbolic tangent function ......................... 125
9.28 ATOMIC_ADD — Atomic ADD operation .......................... 125
9.29 ATOMIC_AND — Atomic bitwise AND operation ......................... 126
9.30 ATOMIC_CAS — Atomic compare and swap ......................... 127
9.31 ATOMIC_DEFINE — Setting a variable atomically .......................... 127
9.32 ATOMIC_FETCH_ADD — Atomic ADD operation with prior fetch .......................... 128
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.33</td>
<td>ATOMIC_FETCH_AND — Atomic bitwise AND operation with prior fetch</td>
</tr>
<tr>
<td>9.34</td>
<td>ATOMIC_FETCH_OR — Atomic bitwise OR operation with prior fetch</td>
</tr>
<tr>
<td>9.35</td>
<td>ATOMIC_FETCH_XOR — Atomic bitwise XOR operation with prior fetch</td>
</tr>
<tr>
<td>9.36</td>
<td>ATOMIC_OR — Atomic bitwise OR operation</td>
</tr>
<tr>
<td>9.37</td>
<td>ATOMIC_REF — Obtaining the value of a variable atomically</td>
</tr>
<tr>
<td>9.38</td>
<td>ATOMIC_XOR — Atomic bitwise OR operation</td>
</tr>
<tr>
<td>9.39</td>
<td>BACKTRACE — Show a backtrace</td>
</tr>
<tr>
<td>9.40</td>
<td>BESSEL_J0 — Bessel function of the first kind of order 0</td>
</tr>
<tr>
<td>9.41</td>
<td>BESSEL_J1 — Bessel function of the first kind of order 1</td>
</tr>
<tr>
<td>9.42</td>
<td>BESSEL_JN — Bessel function of the first kind</td>
</tr>
<tr>
<td>9.43</td>
<td>BESSEL_Y0 — Bessel function of the second kind of order 0</td>
</tr>
<tr>
<td>9.44</td>
<td>BESSEL_Y1 — Bessel function of the second kind of order 1</td>
</tr>
<tr>
<td>9.45</td>
<td>BESSEL_YN — Bessel function of the second kind</td>
</tr>
<tr>
<td>9.46</td>
<td>BGE — Bitwise greater than or equal to</td>
</tr>
<tr>
<td>9.47</td>
<td>BGT — Bitwise greater than</td>
</tr>
<tr>
<td>9.48</td>
<td>BIT_SIZE — Bit size inquiry function</td>
</tr>
<tr>
<td>9.49</td>
<td>BLE — Bitwise less than or equal to</td>
</tr>
<tr>
<td>9.50</td>
<td>BLT — Bitwise less than</td>
</tr>
<tr>
<td>9.51</td>
<td>BTTEST — Bit test function</td>
</tr>
<tr>
<td>9.52</td>
<td>C_ASSOCIATED — Status of a C pointer</td>
</tr>
<tr>
<td>9.53</td>
<td>C_F_POINTER — Convert C into Fortran pointer</td>
</tr>
<tr>
<td>9.54</td>
<td>C_F_PROCPOINTER — Convert C into Fortran procedure pointer</td>
</tr>
<tr>
<td>9.55</td>
<td>C_FUNLOC — Obtain the C address of a procedure</td>
</tr>
<tr>
<td>9.56</td>
<td>C_LOC — Obtain the C address of an object</td>
</tr>
<tr>
<td>9.57</td>
<td>C_SIZEOF — Size in bytes of an expression</td>
</tr>
<tr>
<td>9.58</td>
<td>CEILING — Integer ceiling function</td>
</tr>
<tr>
<td>9.59</td>
<td>CHAR — Character conversion function</td>
</tr>
<tr>
<td>9.60</td>
<td>CHDIR — Change working directory</td>
</tr>
<tr>
<td>9.61</td>
<td>CHMOD — Change access permissions of files</td>
</tr>
<tr>
<td>9.62</td>
<td>CMPLX — Complex conversion function</td>
</tr>
<tr>
<td>9.63</td>
<td>CO_BROADCAST — Copy a value to all images the current set of images</td>
</tr>
<tr>
<td>9.64</td>
<td>CO_MAX — Maximal value on the current set of images</td>
</tr>
<tr>
<td>9.65</td>
<td>CO_MIN — Minimal value on the current set of images</td>
</tr>
<tr>
<td>9.66</td>
<td>CO_REDUCE — Reduction of values on the current set of images</td>
</tr>
<tr>
<td>9.67</td>
<td>CO_SUM — Sum of values on the current set of images</td>
</tr>
<tr>
<td>9.68</td>
<td>COMMAND_ARGUMENT_COUNT — Get number of command line arguments</td>
</tr>
<tr>
<td>9.69</td>
<td>COMPILER_OPTIONS — Options passed to the compiler</td>
</tr>
<tr>
<td>9.70</td>
<td>COMPILER_VERSION — Compiler version string</td>
</tr>
<tr>
<td>9.71</td>
<td>COMPLEX — Complex conversion function</td>
</tr>
<tr>
<td>9.72</td>
<td>CONJG — Complex conjugate function</td>
</tr>
<tr>
<td>9.73</td>
<td>COS — Cosine function</td>
</tr>
</tbody>
</table>
9.74  COSD — Cosine function, degrees ........................................ 155
9.75  COSH — Hyperbolic cosine function ................................. 156
9.76  COTAN — Cotangent function .......................................... 156
9.77  COTAND — Cotangent function, degrees ............................ 157
9.78  COUNT — Count function ................................................. 158
9.79  CPU_TIME — CPU elapsed time in seconds .......................... 158
9.80  CSHIFT — Circular shift elements of an array ...................... 159
9.81  CTIME — Convert a time into a string ............................... 160
9.82  DATE_AND_TIME — Date and time subroutine ........................ 161
9.83  DBLE — Double conversion function ................................. 162
9.84  DCMPLX — Double complex conversion function .................. 162
9.85  DIGITS — Significant binary digits function ....................... 163
9.86  DIM — Positive difference .............................................. 163
9.87  DOT_PRODUCT — Dot product function ............................... 164
9.88  DPROD — Double product function .................................... 165
9.89  DREAL — Double real part function .................................. 165
9.90  DSHIFTL — Combined left shift ....................................... 166
9.91  DSHIFTR — Combined right shift ...................................... 166
9.92  DTIME — Execution time subroutine (or function) .............. 167
9.93  EOSHIFT — End-off shift elements of an array .................... 168
9.94  EPSILON — Epsilon function ............................................. 169
9.95  ERF — Error function .................................................... 169
9.96  ERFC — Error function .................................................... 170
9.97  ERFC_SCALED — Error function ....................................... 170
9.98  ETIME — Execution time subroutine (or function) .............. 171
9.99  EVENT_QUERY — Query whether a coarray event has occurred ................................................................. 172
9.100 EXECUTE_COMMAND_LINE — Execute a shell command ....... 172
9.101 EXIT — Exit the program with status ................................ 173
9.102 EXP — Exponential function ............................................. 174
9.103 EXPONENT — Exponent function ...................................... 174
9.104 EXTENDS_TYPE_OF — Query dynamic type for extension .... 175
9.105 FDATE — Get the current time as a string ......................... 175
9.106 FGET — Read a single character in stream mode from stdin ................................................................. 176
9.107 FGETC — Read a single character in stream mode .............. 177
9.108 FINDLOC — Search an array for a value ............................ 178
9.109 FLOOR — Integer floor function ...................................... 178
9.110 FLUSH — Flush I/O unit(s) .............................................. 179
9.111 FNUM — File number function ........................................ 180
9.112 FPUT — Write a single character in stream mode to stdout .. 180
9.113 FPUTC — Write a single character in stream mode .............. 181
9.114 FRACTION — Fractional part of the model representation .... 182
9.115 FREE — Frees memory ..................................................... 182
9.116 FSEEK — Low level file positioning subroutine .................. 183
9.117 FSTAT — Get file status ................................................ 184
9.118 FTELL — Current stream position .................................... 184
9.119 GAMMA — Gamma function ............................................. 185
9.120 GERROR — Get last system error message ............... 186
9.121 GETARG — Get command line arguments ............... 186
9.122 GET_COMMAND — Get the entire command line ......... 187
9.123 GET_COMMAND_ARGUMENT — Get command line arguments ... 187
9.124 GETCWD — Get current working directory ............... 188
9.125 GETENV — Get an environmental variable ............. 189
9.126 GET_ENVIRONMENT_VARIABLE — Get an environmental variable .............................................................. 190
9.127 GETGID — Group ID function .................................. 190
9.128 GETLOG — Get login name .................................. 191
9.129 GETPID — Process ID function ................................ 191
9.130 GETUID — User ID function ................................ 192
9.131 GMTIME — Convert time to GMT info .................... 192
9.132 HOSTNM — Get system host name .......................... 193
9.133 HUGE — Largest number of a kind ....................... 193
9.134 IACHAR — Code in ASCII collating sequence ............. 194
9.135 IALL — Bitwise AND of array elements ................. 195
9.136 IAND — Bitwise logical and ................................ 195
9.137 IANY — Bitwise OR of array elements .................... 196
9.138 IARGC — Get the number of command line arguments .... 197
9.139 IBCLR — Clear bit ........................................... 197
9.140 IBITS — Bit extraction ....................................... 198
9.141 IBSET — Set bit ............................................... 199
9.142 IDATE — Get current local time subroutine (day/month/year) .............................................................. 200
9.143 ICHAR — Character-to-integer conversion function ...... 199
9.144 IDATE — Get current local time subroutine (day/month/year) .............................................................. 200
9.145 IEOR — Bitwise logical exclusive or ....................... 201
9.146 IERROR — Get the last system error number .......... 201
9.147 IMAGE_INDEX — Function that converts a cosubscript to an image index ................................................. 202
9.148 INDEX — Position of a substring within a string ....... 202
9.149 INT — Convert to integer type ............................... 203
9.150 INT2 — Convert to 16-bit integer type .................... 204
9.151 INT8 — Convert to 64-bit integer type .................... 204
9.152 IOR — Bitwise OR of .......................... 204
9.153 IPARITY — Bitwise XOR of array elements ............... 205
9.154 IRAND — Integer pseudo-random number ................. 206
9.155 IS_CONTIGUOUS — Test whether an array is contiguous .... 206
9.156 IS_ENDOFFILE — Test for end-of-file value ............. 207
9.157 IS_ENDOFRECORD — Test for end-of-record value ...... 208
9.158 ISATTY — Whether a unit is a terminal device .......... 208
9.159 ITIME — Get current local time subroutine (hour/minutes/seconds) .............................................................. 210
9.160 IARNG — Send a signal to a process ...................... 211
| 9.164  | KIND — Kind of an entity | 211 |
| 9.165  | LBOUND — Lower dimension bounds of an array | 212 |
| 9.166  | LCOBOUND — Lower codimension bounds of an array | 212 |
| 9.167  | LEADZ — Number of leading zero bits of an integer | 213 |
| 9.168  | LEN — Length of a character entity | 213 |
| 9.169  | LEN_TRIM — Length of a character entity without trailing blank characters | 214 |
| 9.170  | LGE — Lexical greater than or equal | 214 |
| 9.171  | LGT — Lexical greater than | 215 |
| 9.172  | LINK — Create a hard link | 216 |
| 9.173  | LLE — Lexical less than or equal | 216 |
| 9.174  | LLT — Lexical less than | 217 |
| 9.175  | LNB蔺NK — Index of the last non-blank character in a string | 217 |
| 9.176  | LOC — Returns the address of a variable | 218 |
| 9.177  | LOG — Natural logarithm function | 218 |
| 9.178  | LOG10 — Base 10 logarithm function | 219 |
| 9.179  | LOG_GAMMA — Logarithm of the Gamma function | 219 |
| 9.180  | LOGICAL — Convert to logical type | 220 |
| 9.181  | LONG — Convert to integer type | 220 |
| 9.182  | LSHIFT — Left shift bits | 221 |
| 9.183  | LSTAT — Get file status | 221 |
| 9.184  | LTIME — Convert time to local time info | 222 |
| 9.185  | MALLOC — Allocate dynamic memory | 223 |
| 9.186  | MASKL — Left justified mask | 223 |
| 9.187  | MASKR — Right justified mask | 224 |
| 9.188  | MATMUL — matrix multiplication | 224 |
| 9.189  | MAX — Maximum value of an argument list | 225 |
| 9.190  | MAXEXPONENT — Maximum exponent of a real kind | 225 |
| 9.191  | MAXLOC — Location of the maximum value within an array | 226 |
| 9.192  | MAXVAL — Maximum value of an array | 227 |
| 9.193  | MCLOCK — Time function | 227 |
| 9.194  | MCLOCK8 — Time function (64-bit) | 228 |
| 9.195  | MERGE — Merge variables | 228 |
| 9.196  | MERGE_BITS — Merge of bits under mask | 229 |
| 9.197  | MIN — Minimum value of an argument list | 229 |
| 9.198  | MINEEXPONENT — Minimum exponent of a real kind | 230 |
| 9.199  | MINLOC — Location of the minimum value within an array | 230 |
| 9.200  | MINVAL — Minimum value of an array | 231 |
| 9.201  | MOD — Remainder function | 232 |
| 9.202  | MODULO — Modulo function | 233 |
| 9.203  | MOVE_ALLOC — Move allocation from one object to another | 233 |
| 9.204  | MVBITS — Move bits from one integer to another | 234 |
| 9.205  | NEAREST — Nearest representable number | 235 |
| 9.206  | NEW_LINE — New line character | 235 |
| 9.207  | NINT — Nearest whole number | 236 |
| 9.208  | NORM2 — Euclidean vector norms | 236 |
| 9.209  | NOT — Logical negation | 237 |
9.210  NULL — Function that returns an disassociated pointer
9.211  NUM_IMAGES — Function that returns the number of images

9.212  OR — Bitwise logical OR
9.213  PACK — Pack an array into an array of rank one
9.214  PARITY — Reduction with exclusive OR
9.215  PERROR — Print system error message
9.216  POPCNT — Number of bits set
9.217  POPPAR — Parity of the number of bits set
9.218  PRECISION — Decimal precision of a real kind
9.219  PRESENT — Determine whether an optional dummy argument is specified
9.220  PRODUCT — Product of array elements
9.221  RADIIX — Base of a model number
9.222  RAN — Real pseudo-random number
9.223  RAND — Real pseudo-random number
9.224  RANDOM_INIT — Initialize a pseudo-random number generator
9.225  RANDOM_NUMBER — Pseudo-random number
9.226  RANDOM_SEED — Initialize a pseudo-random number sequence
9.227  RANGE — Decimal exponent range
9.228  RANK — Rank of a data object
9.229  REAL — Convert to real type
9.230  RENAME — Rename a file
9.231  REPEAT — Repeated string concatenation
9.232  RESHAPE — Function to reshape an array
9.233  RRSPACING — Reciprocal of the relative spacing
9.234  RSHIFT — Right shift bits
9.235  SAME_TYPE_AS — Query dynamic types for equality
9.236  SCALE — Scale a real value
9.237  SCAN — Scan a string for the presence of a set of characters
9.238  SECONDS — Time function
9.239  SECOND — CPU time function
9.240  SELECTED_CHAR_KIND — Choose character kind
9.241  SELECTED_INT_KIND — Choose integer kind
9.242  SELECTED_REAL_KIND — Choose real kind
9.243  SET_EXPONENT — Set the exponent of the model
9.244  SHAPE — Determine the shape of an array
9.245  SHIFTA — Right shift with fill
9.246  SHIFTL — Left shift
9.247  SHIFTR — Right shift
9.248  SIGN — Sign copying function
9.249  SIGNAL — Signal handling subroutine (or function)
9.250  SIN — Sine function
9.251  SIND — Sine function, degrees
9.252  SINH — Hyperbolic sine function
10 Intrinsic Modules ................................................................. 283

10.1 ISO_FORTRAN_ENV ...................................................... 283
10.2 ISO_C_BINDING .......................................................... 285
10.3 IEEE modules: IEEE_EXCEPTIONS, IEEE_ARITHMETIC, and
     IEEE_FEATURES .......................................................... 286
10.4 OpenMP Modules OMP_LIB and OMP_LIB_KINDS ..................... 287
10.5 OpenACC Module OPENACC ............................................. 287
1 Introduction

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.

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.

1.1 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.

• 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 GNU Fortran Compiler

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.

1.2 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 preprocessing. Source files with the lower case ‘.fpp’ extension are also run through preprocessing.

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; see Section “Introduction” in Using the GNU Compiler Collection (GCC). The two manuals together provide a complete reference for the GNU Fortran compiler.

1.3 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 preprocess included files, use the equivalent preprocessor statement `#include`.

If GNU Fortran invokes the preprocessor, `__GFORTRAN__` is defined. The macros `__GNUC__`, `__GNUC_MINOR__` and `__GNUC_PATCHLEVEL__` can be used to determine the version of the compiler. See Section “Overview” in *The C Preprocessor* for details.

GNU Fortran supports a number of INTEGER and REAL kind types in addition to the kind types required by the Fortran standard. The availability of any given kind type is architecture dependent. The following pre-defined preprocessor macros can be used to conditional include code for these additional kind types: `__GFC_INTEGER_1__`, `__GFC_INTEGER_2__`, `__GFC_INTEGER_8__`, `__GFC_INTEGER_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).

### 1.4 GNU Fortran and G77

The GNU Fortran compiler is the successor to g77, the Fortran 77 front end 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.

### 1.5 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. Note that this is an experimental feature, incomplete, and subject to change in future versions of GCC. See https://gcc.gnu.org/wiki/OpenACC for more information.
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.

1.6 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 Section 4.1 [Fortran 2003 status], page 37, Section 4.2 [Fortran 2008 status], page 38 and Section 4.3 [Fortran 2018 status], page 40 sections of the documentation.

Additionally, the GNU Fortran compilers supports the OpenMP specification (version 4.0 and most of the features of the 4.5 version, http://openmp.org/wp/openmp-specifications/). There also is initial support for the OpenACC specification (targeting version 2.0, http://www.openacc.org/). Note that this is an experimental feature, incomplete, and subject to change in future versions of GCC. See https://gcc.gnu.org/wiki/OpenACC for more information.

1.6.1 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=::).)
Part I: Invoking GNU Fortran
Chapter 2: GNU Fortran Command Options

2 GNU Fortran Command Options

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

See Section “GCC Command Options” in Using the GNU Compiler Collection (GCC), 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.

2.1 Option summary

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 Section 2.2 [Options controlling Fortran dialect], page 8.

-ffall-intrinsics -fbackslash -fcray-pointer -fd-lines-as-code
-fd-lines-as-comments
-fdec -fdec-structure -fdec-intrinsic-ints -fdec-static -fdec-math
-fdec-include -fdefault-double-8 -fdefault-integer-8 -fdefault-real-8
-fdefault-real-10 -fdefault-real-16 -fdollar-ok -ffixed-line-length-n
-ffixed-line-length-none -fpad-source -ffree-form -ffree-line-length-n
-ffree-line-length-none -fimplicit-none -finteger-4-integer-8
-fmax-identifier-length -fmodule-private -ffixed-form -ffree-line-length-n
-ffixed-form -fno-range-check
-fopenacc -fopenmp -freal-4-real-10 -freal-4-real-16 -freal-4-real-8
-freal-8-real-10 -freal-8-real-16 -freal-8-real-4 -std=std
-ftest-forall-temp

Preprocessing Options

See Section 2.3 [Enable and customize preprocessing], page 13.

-Usmacro -cpp -dD -dM -dU -fworking-directory -imultilib dir
-iprefix file -iquote -isysroot dir -isystem dir -nocpp -nostdinc
-undef

Error and Warning Options

See Section 2.4 [Options to request or suppress errors and warnings], page 15.

-Waliasing -Wall -Wampersand -Wargument-mismatch -Warray-bounds -Wc-binding
-type -Wcharacter-truncation -Wconversion
-Wdo-subscript -Wfunctor-elimination -Wimplicit-interface
-Wimplicit-procedure -Wintrinsics-shadow -Wuse-without-only -Wintrinsics-std
-Wline-truncation -Wno-align-commons -Wno-tabs -Wrealeq-q-constant
-Wsurprising -Wuninitialized -Wuninitialized-parameter -Wrealloc-lhs
-Wrealloc-lhs-all -Wfrontend-loop-interchange -Wtarget-lifetime
-Wmax-errors=n -fsyntax-only -pedantic -pedantic-errors

Debugging Options

See Section 2.5 [Options for debugging your program or GNU Fortran], page 20.
-fbacktrace -fdump-fortran-optimized -fdump-fortran-original
-fdump-parse-tree -ffpe-trap=list -ffpe-summary=list

Directory Options
See Section 2.6 [Options for directory search], page 21.
-ldir -Ldir -fintrinsic-modules-path=dir

Link Options
See Section 2.7 [Options for influencing the linking step], page 22.
-static-libgfortran

Runtime Options
See Section 2.8 [Options for influencing runtime behavior], page 22.
-fconvert=conversion -fmax-subrecord-length=length
-frecord-marker=length -fsign-zero

Interoperability Options
See Section 2.10 [Options for interoperability], page 29.
-fc-prototypes

Code Generation Options
See Section 2.9 [Options for code generation conventions], page 23.
-faggressive-function-elimination -fblas-matmul-limit=n
-fbounds-check -fcheck-array-temporaries
-fcheck=<all|array|temps|bounds|do|mem|pointer|recursion>
-fcoarray=<none|single|lib> -fexternal-blas -ff2c -ffrontend-loop-interchange
-ffrontend-optimize
-finit-character=n -finit-integer=n -finit-local-zero
-finit-derived -finit-logical=<true|false> -finit-real=<zero|inf|-inf|nan|snan>
-finline-matmul-limit=n
-fmax-array-constructor=n -fmax-stack-var-size=n -fno-align-commons
-fno-automatic -fno-protect-parens -fno-underscoring
-fsecond-underscore -fpack-derived -frealloc-lhs -frecursive
-frepack-arrays -fshort-enums -fstack-arrays

2.2 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.
-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-structure’ ‘-fdec-intrinsic-ints’ ‘-fdec-static’ ‘-fdec-math’
If ‘-fd-lines-as-code’/‘-fd-lines-as-comments’ are unset, then ‘-fdec’ also sets ‘-fd-lines-as-comments’.

-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, HIAND, 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.

-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, \", and \0 to the ASCII characters alert, backspace, form feed, newline, carriage return, horizontal tab, vertical tab, backslash, and NUL, respectively. Additionally, \xnn, \umnnt and \Unnnnnnnnn
(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 \( \backslash \) are unexpanded.

\[-f\text{module-private}\]
Set the default accessibility of module entities to \texttt{PRIVATE}. Use-associated entities will not be accessible unless they are explicitly declared as \texttt{PUBLIC}.

\[-f\text{fixed-line-length-}n\]
Set column after which characters are ignored in typical fixed-form lines in the source file, and, unless \texttt{-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’.

\[-f\text{no-pad-source}\]
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.

\[-f\text{free-line-length-}n\]
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’.

\[-f\text{max-identifier-length=}n\]
Specify the maximum allowed identifier length. Typical values are 31 (Fortran 95) and 63 (Fortran 2003 and Fortran 2008).

\[-f\text{implicit-none}\]
Specify that no implicit typing is allowed, unless overridden by explicit \texttt{IMPLICIT} statements. This is the equivalent of adding \texttt{implicit none} to the start of every procedure.

\[-f\text{cray-pointer}\]
Enable the Cray pointer extension, which provides C-like pointer functionality.

\[-f\text{openacc}\]
Enable the OpenACC extensions. This includes OpenACC \texttt{!$acc} directives in free form and \texttt{c$acc}, \texttt{*$acc} and \texttt{!$acc} directives in fixed form, \texttt{!$} conditional compilation sentinels in free form and \texttt{c$}, \texttt{*$} and \texttt{!$} sentinels in fixed form, and when linking arranges for the OpenACC runtime library to be linked in.
Note that this is an experimental feature, incomplete, and subject to change in future versions of GCC. See https://gcc.gnu.org/wiki/OpenACC for more information.

-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’.

-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.
-ffdefault-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 `-ffdefault-real-8', `-ffdefault-real-10', and `-ffdefault-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. All other real kind types are unaffected by this option. 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 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.

-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.
2.3 Enable and customize preprocessing

Preprocessor related options. See section Section 1.3 [Preprocessing and conditional compilation], page 2 for more detailed information on preprocessing in gfortran.

- cpp
  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’.

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

-ipv prefix
Specify prefix as the prefix for subsequent ‘-ivwithprefix’ 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.
-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.

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

-D name=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'rename(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.

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

2.4 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.

- **-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**
  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 ‘-Waliasing’, ‘-Wamperands’, ‘-Wconversion’, ‘-Wsurprising’, ‘-Wc-binding-type’, ‘-Wintrinsics-std’, ‘-Wtabs’, ‘-Wintrinsics-shadow’, ‘-Wline-truncation’, ‘-Wtarget-lifetime’, ‘-Winteger-division’, ‘-Wreal-q-constant’, ‘-Wunused’ and ‘-Wundefined-do-loop’.

- **-Waliasing**
  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 bar
end interface
```
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.

-Wargument-mismatch
Warn about type, rank, and other mismatches between formal parameters and
actual arguments to functions and subroutines. These warnings are recom-
mended and thus enabled by default.

-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`.

-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
Enable warning for loop interchanges performed by the `-ffrontend-loop-interchange` option.
-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.

-implicit-procedure
Warn if a procedure is called that has neither an explicit interface nor has been declared as \texttt{EXTERNAL}.

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

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

-real-q-constant
Produce a warning if a real-literal-constant contains a \texttt{q} exponent-letter.

-surprising
Produce a warning when “suspicious” 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 \texttt{CASE} that can never be matched as its lower value is greater than its upper value.
- A LOGICAL SELECT construct has three \texttt{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 \texttt{'-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. \texttt{'-Wtabs'} will cause a warning to be issued if a tab is encountered. Note, \texttt{'-Wtabs'} is active for \texttt{'-pedantic'}, \texttt{'-std=f95'}, \texttt{'-std=f2003'}, \texttt{'-std=f2008'}, \texttt{'-std=f2018'} and \texttt{'-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 \texttt{'-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 ‘-Wunused-parameter’, gfortran’s implementation of this option does not warn about unused dummy arguments (see ‘-Wunused-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 ‘-Wno-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. (:,:,:)) 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'.

-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 Section “Options to Request or Suppress Errors and Warnings” in Using the GNU Compiler Collection (GCC), 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.

2.5 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.

-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. 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.

```bash
-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.)

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

```bash
-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 Section “Options for Debugging Your Program or GCC” in *Using the GNU Compiler Collection (GCC)*, for more information on debugging options.

### 2.6 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.

```bash
-I dir
```

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 Section “Options for Directory Search” in *Using the GNU Compiler Collection (GCC)*, 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.

2.7 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.

2.8 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'.

2.9 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.

-ff2c
Generate code designed to be compatible with code generated by g77 and f2c. The calling conventions used by g77 (originally implemented in f2c) require functions that return type default REAL to actually return the C type double, and functions that return type 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 REAL functions return the C type float, and COMPLEX functions return the GNU C type complex. Additionally, this option implies the ‘-fsecond-underscore’ option, unless ‘-fno-second-underscore’ is explicitly requested.

This does not affect the generation of code that interfaces with the libgfortran library.

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

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

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

With ‘-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.

Caution: The default behavior of GNU Fortran is incompatible with f2c and g77, please use the ‘-ff2c’ option if you want object files compiled with GNU Fortran to be compatible with object code created with these tools.
Use of ‘-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 ‘-funderscoring’, and assuming that j() and max_count() are external functions while my_var and lvar are local variables, a statement like

\[ I = j(\) + \text{MAX\_COUNT} (\text{MY\_VAR}, \text{LVAR}) \]

is implemented as something akin to:

\[ i = j_{}() + \text{max\_count\_}(&\text{my\_var\_}, &\text{lvar}); \]

With ‘-fno-underscoring’, the same statement is implemented as:

\[ i = j() + \text{max\_count}(&\text{my\_var}, &\text{lvar}); \]

Use of ‘-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 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-tamps’ 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.

‘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 typespec.

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
```
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.

`-fbounds-check`
Deprecated alias for ‘-fcheck=bounds’.
---
-fcheck-array-temporaries
  Deprecated alias for ‘-fcheck=array-temps’.

-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.

  ```fortran
  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
  ``````

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

  The default value for n is 65535.

-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 ‘-frecursive’ to allow for recursive procedures which do not have a RECURSIVE attribute or for parallel programs. Use ‘-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 n is 32768.

-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 accesses 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.

-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’).

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 ‘-Ofast’ 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’.

-faggressive-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.
Chapter 2: GNU Fortran Command Options

-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 TRIM(a) with a(1: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 Section “Options for Code Generation Conventions” in Using the GNU Compiler Collection (GCC), for information on more options offered by the GBE shared by gfortran, gcc, and other GNU compilers.

2.10 Options for interoperability with other languages

-fc-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:

```bash
$ gfortran -fc-prototypes -fsyntax-only foo.f90 > foo.h
```

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

2.11 Environment variables affecting gfortran

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 Section “Environment Variables Affecting GCC” in Using the GNU Compiler Collection (GCC), for information on environment variables.

See Chapter 3 [Runtime], page 31, for environment variables that affect the run-time behavior of programs compiled with GNU Fortran.
3 Runtime: Influencing runtime behavior with environment variables

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

### 3.1 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.

1. The environment variable `TMPDIR`, if it exists.
2. 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.
3. The `P_tmpdir` macro if it is defined, otherwise the directory ‘/tmp’.

### 3.2 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.

### 3.3 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.

### 3.4 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.

### 3.5 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.

### 3.6 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.

### 3.7 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.
3.8 **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.

3.9 **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

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

when `a.out` is the compiled Fortran program that you want to run. Default is a single space.

3.10 **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**:

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

Example code for **csh**:
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 Section 6.1.15 [CONVERT specifier], page 55, for an alternative way to specify the data representation for unformatted files. See Section 2.8 [Runtime Options], page 22, 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.

3.11 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.
Part II: Language Reference
4 Fortran standards status

4.1 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 PASS attribute.
- Procedures which are bound to a derived type (type-bound procedures) including PASS, PROCEDURE and 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 ("CLASS") for derived types and unlimited polymorphism ("CLASS(*)") – including SAME_TYPE_AS, EXTENDS_TYPE_OF and 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-constructer initialization are available.
- The ASSOCIATE construct.
- Interoperability with C including enumerations,
- In structure constructors the components with default values may be omitted.
- Extensions to the ALLOCATE statement, allowing for a type-specification with type parameter and for allocation and initialization from a SOURCE= expression; ALLOCATE and DEALLOCATE optionally return an error message string via 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 MOVE_ALLOC.
- The PRIVATE and 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 INTENT may be specified which affect the association status not the value of the pointer target.
- Intrinsics command_argument_count, get_command, get_command_argument, and get_environment_variable.
- 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 (/ 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.

4.2 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 the final draft (FDIS) can be downloaded free of charge from 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 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 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.
- Support for the \texttt{IMPURE} attribute for procedures, which allows for \texttt{ELEMENTAL} procedures without the restrictions of \texttt{PURE}.
- Null pointers (including \texttt{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 \texttt{TARGET} attribute can be used as actual argument to \texttt{POINTER} dummies with \texttt{INTENT(IN)}.
- Pointers including procedure pointers and those in a derived type (pointer components) can now be initialized by a target instead of only by \texttt{NULL}.
- The \texttt{EXIT} statement (with construct-name) can be now be used to leave not only the \texttt{DO} but also the \texttt{ASSOCIATE}, \texttt{BLOCK}, \texttt{IF}, \texttt{SELECT CASE} and \texttt{SELECT TYPE} constructs.
- Internal procedures can now be used as actual argument.
- Minor features: obsolesce diagnostics for \texttt{ENTRY} with \texttt{`-std=f2008`}; a line may start with a semicolon; for internal and module procedures \texttt{END} can be used instead of \texttt{END SUBROUTINE} and \texttt{END FUNCTION}; \texttt{SELECTED_REAL_KIND} now also takes a \texttt{RADIX} argument; intrinsic types are supported for \texttt{TYPE(intrinsic-type-spec)}; multiple type-bound procedures can be declared in a single \texttt{PROCEDURE} statement; implied-shape arrays are supported for named constants (\texttt{PARAMETER}).

4.3 Status of Fortran 2018 support

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

4.3.1 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 Section 7.1.6 [Further Interoperability of Fortran with C], page 74.

- The \texttt{OPTIONAL} attribute is allowed for dummy arguments of \texttt{BIND(C)} procedures.
- The \texttt{RANK} intrinsic is supported.
- GNU Fortran’s implementation for variables with \texttt{ASYNCHRONOUS} attribute is compatible with TS 29113.
- Assumed types (\texttt{TYPE(*)}).
- Assumed-rank (\texttt{DIMENSION(...)}).
- \texttt{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 \texttt{ISO_Fortran_binding.h} is can be found in
  \texttt{-prefix/lib/gcc/$target/$version}.
4.3.2 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`)
5 Compiler Characteristics

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.

5.1 KIND Type Parameters

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

- **INTEGER** 1, 2, 4, 8*, 16*, default: 4**
- **LOGICAL** 1, 2, 4, 8*, 16*, default: 4**
- **REAL** 4, 8, 10*, 16*, default: 4***
- **COMPLEX** 4, 8, 10*, 16*, default: 4***
- **DOUBLE PRECISION** 4, 8, 10*, 16*, default: 8***
- **CHARACTER** 1, 4, default: 1

* not available on all systems
** unless ‘-fdefault-integer-8’ is used
*** unless ‘-fdefault-real-8’ is used (see Section 2.2 [Fortran Dialect Options], page 8)

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 Section 9.240 [SELECTED_CHAR_KIND], page 254, Section 9.241 [SELECTED_INT_KIND], page 255 and Section 9.242 [SELECTED_REAL_KIND], page 256 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 Section 10.1 [ISO_FORTRAN_ENV], page 283 module. For C interoperability, the kind parameters of the Section 10.2 [ISO_C_BINDING], page 285 module should be used.

5.2 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 LOGICAL(KIND=\texttt{N}) variable is represented as an INTEGER(KIND=\texttt{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 Section 7.4.2 [Argument passing conventions], page 80 and Section 7.1 [Interoperability with C], page 69.
5.3 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 .AND. or .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 `-Wfunction-elimination` flag.

5.4 MAX and MIN intrinsics with REAL NaN arguments

The Fortran standard does not specify what the result of the MAX and 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 MAX or MIN, e.g. with the IEEE_IS_NAN function from the intrinsic module IEEE_ARITHMETIC.

5.5 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 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 getenv function is used, which need not be thread-safe. Similarly, the getenv function is used to implement the GET_ENVIRONMENT_VARIABLE and 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 EXECUTE_COMMAND_LINE and SYSTEM intrinsics are implemented with the system function, which need not be thread-safe. It is the responsibility of the user to ensure that 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 strtod and snprintf. In order to work correctly in locale-aware programs that set the locale using setlocale, the locale is reset to the default “C” locale while executing a formatted READ or WRITE statement. On targets supporting the POSIX 2008 per-thread locale functions (e.g. newlocale, uselocale, freelocale), these are used and thus the global locale set using 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.
5.6 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:

```fortran
! 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.

5.7 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:

1. Attempt to open the file with ACTION='READWRITE'
2. If that fails, try to open with ACTION='READ'
3. If that fails, try to open with ACTION='WRITE'
4. If that fails, generate an error

5.8 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.

5.9 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 compability only) is in effect.

The representation of the record markers is that of unformatted files given with the ‘-fconvert’ option, the Section 6.1.15 [CONVERT specifier], page 55 in an open statement or the Section 3.10 [GFORTRAN_CONVERT_UNIT], page 32 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:

```fortran
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
```

5.10 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.
6 Extensions

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.

6.1 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’.

6.1.1 Old-style kind specifications

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

\[ \text{TYPESPEC} \times, y, z \]

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

\[ \text{TYPESPEC}(k) \times, y, z \]

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

\[
\begin{align*}
\text{INTEGER, PARAMETER :: } & \text{dbl} = \text{KIND}(1.0d0) \\
\text{REAL(KIND=dbl) :: } & x
\end{align*}
\]

6.1.2 Old-style variable initialization

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

\[
\begin{align*}
\text{INTEGER } & i/1/, j/2/ \\
\text{REAL } & x(2,2) /3*0., 1./
\end{align*}
\]

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 \text{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 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.

6.1.3 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 ,
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.
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.

### 6.1.4 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)
```

### 6.1.5 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.

```
PRINT 10, 2, 3
10 FORMAT ('FOO='I1' BAR='I2)
```

### 6.1.6 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')
```

### 6.1.7 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.

### 6.1.8 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.

### 6.1.9 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`, were the prefix is either b, o or z, quote is either ’ or ” and the digits are for binary 0 or 1, for octal between 0 and 7, and for hexadecimal between 0 and F. (Example: `b’01011101’`.)

Up to Fortran 95, BOZ literals were only allowed to initialize integer variables in DATA statements. Since Fortran 2003 BOZ literals are also allowed as argument of `REAL`, `DBLE`,...
INT and CMPLX; the result is the same as if the integer BOZ literal had been converted by TRANSFER to, respectively, real, double precision, integer or complex. As GNU Fortran extension the intrinsic procedures FLOAT, DFLOAT, COMPLEX and DCMPLX are treated alike.

As an extension, GNU Fortran allows hexadecimal BOZ literal constants to be specified using the $X$ prefix, in addition to the standard $Z$ prefix. The BOZ literal can also be specified by adding a suffix to the string, for example, $Z'ABC'$ and $'ABC'Z$ are equivalent.

Furthermore, GNU Fortran allows using BOZ literal constants outside DATA statements and the four intrinsic functions allowed by Fortran 2003. In DATA statements, in direct assignments, where the right-hand side only contains a BOZ literal constant, and for old-style initializers of the form integer i /o'0173'/, the constant is transferred as if TRANSFER had been used; for COMPLEX numbers, only the real part is initialized unless CMPLX is used. In all other cases, the BOZ literal constant is converted to an INTEGER value with the largest decimal representation. This value is then converted numerically to the type and kind of the variable in question. (For instance, real :: r = b'0000001' + 1 initializes r with 2.0.) As different compilers implement the extension differently, one should be careful when doing bitwise initialization of non-integer variables.

Note that initializing an INTEGER variable with a statement such as DATA i/Z'FFFFFFFF'/ will give an integer overflow error rather than the desired result of $-1$ when i is a 32-bit integer on a system that supports 64-bit integers. The ‘-fno-range-check’ option can be used as a workaround for legacy code that initializes integers in this manner.

6.1.10 Real array indices

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

6.1.11 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.

\[ x = y * -z \]

6.1.12 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 :: i
i = 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.
6.1.13 Hollerith constants support

GNU Fortran supports Hollerith constants in assignments, function arguments, and DATA and ASSIGN statements. 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) or LOGICAL variable. The constant will be padded 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 /16Habcdefghijklmnop, 16Hqrstuvwxyz012345/
x(1) = 16HABCDEFGHIJKLMNOP
call foo (4h abc)
```

Invalid Hollerith constants examples:

```
integer*4 a
a = 8H12345678 ! Valid, but the Hollerith constant will be truncated.
a = 0H ! At least one character is needed.
```

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 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
```

6.1.14 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:

```fortran
real target(10)
real pointee(10)
pseudo (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:

```fortran
real ar(10)
pseudo (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 Section 9.185 [MALLOC], page 223).

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

```fortran
integer target(10)
integer iarr(10)
pseudo (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:

```fortran
implicit none
external sub
pointer (subptr,subpte)
external subpte
subptr = loc(sub)
call subpte()
[...]
subroutine sub
[...]
end subroutine sub
```

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.

### 6.1.15 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 Section 3.10 [GFORTRAN_CONVERT_UNIT], page 32, 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.
6.1.16 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 Section “libgomp” in GNU 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.

6.1.17 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 v2.0.

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 Section “libgomp” in GNU 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’.

Note that this is an experimental feature, incomplete, and subject to change in future versions of GCC. See https://gcc.gnu.org/wiki/OpenACC for more information.

6.1.18 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.:

\[\begin{align*}
\text{C} & \quad \text{C prototype} \quad \text{void foo}\_\text{ (float }x\text{);} \\
\text{C} & \quad \text{external foo} \\
\text{C} & \quad \text{real*4 } x \\
\text{C} & \quad x = 3.14159 \\
\text{C} & \quad \text{call foo }\left(\text{%VAL }\left(x\right)\right) \\
\text{C} & \quad \text{end}
\end{align*}\]

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.

6.1.19 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.

6.1.20 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'
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:

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

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 STRUCUTRES like derived types with the following rules and exceptions:

- Structures act like derived types with the \texttt{SEQUENCE} attribute. Otherwise they may contain no specifiers.
- Structures may contain a special field with the name \texttt{%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:

```fortran
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:

```fortran
structure /header/
  ! ...
end structure
record /header/ header
```

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

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

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

```fortran
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)
```

...
! 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 Section 6.1.21 [UNION and MAP], page 60.
- Structures support old-style initialization of components, like those described in Section 6.1.2 [Old-style variable initialization], page 49. 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.

### 6.1.21 UNION and MAP

Unions are an old vendor extension which were commonly used with the non-standard Section 6.1.20 [STRUCTURE and RECORD], page 57 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:

```fortran
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:

```
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | Byte offset |
```

| | | | | | | | |
Following is an example mirroring the layout of an Intel x86_64 register:

```plaintext
structure /reg/
  union ! U0 ! rax
    map
      character(16) rx
    end map
  map
    character(8) rh
    union ! U1
      map
        character(8) rl
      end map
    map
      character(8) ex
    end map
    map
      character(4) eh
    union ! U2
      map
        character(4) el
      end map
    map
      character(4) x
    end map
  end union
end structure

record /reg/ a

! After this assignment...
a.rx = 'AAAAAAA.ABBB.C.D'

! The following is true:
a.rx == 'AAAAAAA.ABBB.C.D'
```
6.1.22 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:

- **B** - INTEGER(kind=1)
- **I** - INTEGER(kind=2)
- **J** - INTEGER(kind=4)
- **K** - INTEGER(kind=8)

GNU Fortran supports these with the flag ‘-fdec-intrinsic-ints’. Intrinsics for which prefixed versions are available and in what form are noted in Chapter 9 [Intrinsic Procedures], page 107. The complete list of supported intrinsics is here:

<table>
<thead>
<tr>
<th>Intrinsic</th>
<th>B</th>
<th>I</th>
<th>J</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Section 9.3 [ABS], page 108</td>
<td>BABS</td>
<td>IIABS</td>
<td>JIABS</td>
<td>KIABS</td>
</tr>
<tr>
<td>Section 9.51 [BTEST], page 139</td>
<td>BBTEST</td>
<td>BITEST</td>
<td>BJTEST</td>
<td>BKTEST</td>
</tr>
<tr>
<td>Section 9.137 [IAND], page 195</td>
<td>BIAND</td>
<td>IIAINS</td>
<td>JIAND</td>
<td>KIAND</td>
</tr>
<tr>
<td>Section 9.140 [IBCLR], page 197</td>
<td>BBCLR</td>
<td>IIBCLR</td>
<td>JIBCLR</td>
<td>KIBCLR</td>
</tr>
<tr>
<td>Section 9.141 [IBITS], page 198</td>
<td>BBITS</td>
<td>IIBITS</td>
<td>JIBITS</td>
<td>KIBITS</td>
</tr>
<tr>
<td>Section 9.142 [IBSET], page 199</td>
<td>BBSET</td>
<td>IIBSET</td>
<td>JIBSET</td>
<td>KIBSET</td>
</tr>
<tr>
<td>Section 9.145 [IEOR], page 201</td>
<td>BIEOR</td>
<td>IIEOR</td>
<td>JIEOR</td>
<td>KIEOR</td>
</tr>
<tr>
<td>Section 9.152 [IOR], page 204</td>
<td>BIOR</td>
<td>IIOR</td>
<td>JIOR</td>
<td>KIOR</td>
</tr>
</tbody>
</table>
6.1.23 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:

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

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
endsubroutine

! Compiled with -fno-automatic
subroutine f
  integer a, b, c, d
  automatic :: a
  ! a is automatic; b, c, and d are static
```
6.1.24 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 Chapter 9 [Intrinsic Procedures], page 107 where it is noted that they are extensions and should be avoided whenever possible.

Specifically, `-fdec-math` enables the Section 9.76 [COTAN], page 156 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>Section 9.6 [ACOS], page 110</td>
<td>Section 9.7 [ACOSD], page 111*</td>
</tr>
<tr>
<td>Section 9.19 [ASIN], page 119</td>
<td>Section 9.20 [ASIND], page 119*</td>
</tr>
<tr>
<td>Section 9.23 [ATAN], page 122</td>
<td>Section 9.24 [ATAND], page 122*</td>
</tr>
<tr>
<td>Section 9.25 [ATAN2], page 123</td>
<td>Section 9.26 [ATAN2D], page 124*</td>
</tr>
<tr>
<td>Section 9.73 [COS], page 154</td>
<td>Section 9.74 [COSD], page 155*</td>
</tr>
<tr>
<td>Section 9.76 [COTAN], page 156*</td>
<td>Section 9.77 [COTAND], page 157*</td>
</tr>
<tr>
<td>Section 9.250 [SIN], page 260</td>
<td>Section 9.251 [SIND], page 261*</td>
</tr>
<tr>
<td>Section 9.266 [TAN], page 271</td>
<td>Section 9.267 [TAND], page 271*</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).

6.1.25 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.

6.1.26 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'
```

6.1.27 %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
```
Chapter 6: Extensions 65

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

6.1.28 .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.

6.1.29 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 Section 9.137 [IAND], page 195 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>Section 9.209 [NOT], page 237</td>
<td>complement</td>
</tr>
<tr>
<td>.AND.</td>
<td>Section 9.137 [IAND], page 195</td>
<td>intersection</td>
</tr>
<tr>
<td>.OR.</td>
<td>Section 9.152 [IOR], page 204</td>
<td>union</td>
</tr>
<tr>
<td>.NEQV.</td>
<td>Section 9.145 [IEOR], page 201</td>
<td>exclusive or</td>
</tr>
<tr>
<td>.EQV.</td>
<td>Section 9.209 [NOT], page 237 (Section 9.145 [IEOR], page 201)</td>
<td>complement of exclusive or</td>
</tr>
</tbody>
</table>

6.1.30 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 \texttt{CARRIAGECONTROL} is specified upon opening a unit for formatted writing, the exact \texttt{CARRIAGECONTROL} setting determines what characters to write between output records. The syntax is:

\begin{verbatim}
OPEN( ..., CARRIAGECONTROL=cc)
\end{verbatim}

Where \textit{cc} is a character expression that evaluates to one of the following values:

- \texttt{'LIST'}: One line feed between records (default)
- \texttt{'FORTRAN'}: Legacy interpretation of the first character (see below)
- \texttt{'NONE'}: No separator between records

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>CHAR(0)</td>
<td>Overprinting</td>
<td>(no advance)</td>
</tr>
</tbody>
</table>

\textbf{READONLY} \quad 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 (i.e. \texttt{CLOSE} with \texttt{STATUS="DELETE"}). The syntax is:

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

\textbf{SHARE} \quad 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 \textit{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 \textit{sh}:

<table>
<thead>
<tr>
<th>Explicit form</th>
<th>Short form</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHARE='DENYRW'</td>
<td>NOSHARED</td>
<td>Exclusive (write) lock</td>
</tr>
<tr>
<td>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.
6.1.31 Legacy PARAMETER statements

For compatibility, GNU Fortran supports legacy 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}

6.1.32 Default exponents

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

6.2 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.

6.2.1 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

\begin{verbatim}
INTEGER*1 LINE(80)
REAL A, B, C
c ... Code that sets LINE
DECODE (80, 9000, LINE) A, B, C
9000 FORMAT (1X, 3(F10.5))
\end{verbatim}

with the following:

\begin{verbatim}
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))
\end{verbatim}

Similarly, replace a code fragment like

\begin{verbatim}
INTEGER*1 LINE(80)
REAL A, B, C
c ... Code that sets A, B and C
ENCODE (80, 9000, LINE) A, B, C
9000 FORMAT (1X, 'OUTPUT IS ', 3(F10.5))
\end{verbatim}

with the following:

\begin{verbatim}
CHARACTER(LEN=80) LINE
REAL A, B, C
\end{verbatim}
6.2.2 Variable FORMAT expressions

A variable FORMAT expression is format statement which includes angle brackets enclosing a Fortran expression: `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
c Variable declaration
CHARACTER(LEN=20) FMT
c Other code here...
c WRITE(FMT,* N+1
WRITE(6,"(I" // ADJUSTL(FMT) // ")") INT1
```

6.2.3 Alternate complex function syntax

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

6.2.4 Volatile COMMON blocks

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

6.2.5 OPEN(... NAME=)

Some Fortran compilers, including g77, let the user declare OPEN(... NAME=). This is invalid standard Fortran syntax and is not supported by gfortran. OPEN(... NAME=) should be replaced with OPEN(... FILE=).
7 Mixed-Language Programming

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.

7.1 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+1]).

7.1.1 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 Section 10.2 [ISO_C_BINDING], page 285.

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.)

7.1.2 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

```fortran
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

```c
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 \texttt{sequence} attribute, type parameters, the \texttt{extends} attribute, nor type-bound procedures. Every component must be of interoperable type and kind and may not have the \texttt{pointer} or \texttt{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.

### 7.1.3 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 \texttt{MODULE}, be of interoperable type, and have neither the \texttt{pointer} nor the \texttt{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, \texttt{_MyProject_flags} is the case-sensitive name of the variable as seen from C programs while \texttt{global_flag} is the case-insensitive name as seen from Fortran. If no binding name is specified, as for \texttt{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 \texttt{errno} of the C library as the C standard allows it to be a macro. Use the \texttt{IERRNO} intrinsic (GNU extension) instead.

### 7.1.4 Interoperable Subroutines and Functions

Subroutines and functions have to have the \texttt{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 Section 7.1.6 [Further Interoperability of Fortran with C], page 74.

To pass a variable by value, use the \texttt{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
```
Note that pointer arguments also frequently need the \texttt{VALUE} attribute, see Section 7.1.5 [Working with Pointers], page 72.

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 \textit{string} of \texttt{print\_C} is a length-one assumed-size array; using \texttt{character(len=*)} is not allowed. The example above uses \texttt{c\_char_"Hello World"} to ensure the string literal has the right type; typically the default character kind and \texttt{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 \texttt{strncpy}, whose prototype is

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

The function \texttt{strncpy} copies at most \texttt{n} characters from string \texttt{s2} to \texttt{s1} and returns \texttt{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 Chapter 9 [Intrinsic Procedures], page 107.
7.1.5 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 Section 7.1.6 [Further Interoperability of Fortran with C], page 74.

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
REAL(KIND=C_DOUBLE), INTENT(IN), VALUE :: x
END SUBROUTINE callback
END INTERFACE

CONTAINS

! Define C-bound procedure.
SUBROUTINE get_values (cproc) BIND(C)
USE, INTRINSIC :: ISO_C_BINDING
TYPE(C_FUNPTR), INTENT(IN), VALUE :: cproc
PROCEDURE(callback), POINTER :: proc

! Convert C to Fortran procedure pointer.
CALL C_F_PROCPOINTER (cproc, proc)

! Call it.
CALL proc (1.0_C_DOUBLE)
CALL proc (-42.0_C_DOUBLE)
CALL proc (18.12_C_DOUBLE)
END SUBROUTINE get_values

END MODULE m

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:

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

It can be used as in the following Fortran code:

```
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

! 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

7.1.6 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.

7.2 GNU Fortran Compiler Directives

7.2.1 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 Section “C Extensions” in Using 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:
• DLEXPRT – 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

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.

### 7.2.2 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.

### 7.3 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.
7.3.1 _gfortran_set_args — Save command-line arguments

Description:
_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).

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

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

Example:
int main (int argc, char *argv[])
{
    /* Initialize libgfortran. */
    _gfortran_set_args (argc, argv);
    return 0;
}

7.3.2 _gfortran_set_options — Set library option flags

Description:
_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 Section 2.9 [Code Gen Options], page 23. Please note that not all flags are actually used.

Syntax: void _gfortran_set_options (int num, int options[])

Arguments:
num number of options passed
argv The list of flag values

option flag list:
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_F2008_OBS | GFC_STD_F77 | GFC_STD_F2018 | GFC_STD_F2018_OBS | 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 (16), GFC_RTCHECK_POINTER (32). 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, 0, 0, 31};
_gfortran_set_options (9, &options);
```

### 7.3.3 \_gfortran\_set\_convert — Set endian conversion

**Description:**

\_gfortran\_set\_convert set the representation of data for unformatted files.

**Syntax:**

void \_gfortran\_set\_convert (int conv)

**Arguments:**

- **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;
}
```
7.3.4 _gfortran_set_record_marker — Set length of record markers

Description:

_gfortran_set_record_marker sets the length of record markers for unformatted files.

Syntax: void _gfortran_set_record_marker (int val)

Arguments:

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;
}
```

7.3.5 _gfortran_set_fpe — Enable floating point exception traps

Description:

_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.

Syntax: void _gfortran_set_fpe (int val)

Arguments:

option[0] 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;
}
```

7.3.6 _gfortran_set_max_subrecord_length — Set subrecord length

Description:

_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.

Syntax: void _gfortran_set_max_subrecord_length (int val)
Arguments:

\[ \text{val} \]
the maximum length for a subrecord; the maximum permitted value is 2147483639, which is also the default.

Example:

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

7.4 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 Section 7.1 [Interoperability with C], page 69.

7.4.1 Naming conventions

According the Fortran standard, valid Fortran names consist of a letter between \( A \) to \( Z \), \( a \) to \( z \), digits \( 0 \) 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 Section 7.2 [GNU Fortran Compiler Directives], page 75.

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.

7.4.2 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
#ifdef __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 a 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
Chapter 8: Coarray Programming

8 Coarray Programming

8.1 Type and enum ABI Documentation

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

8.1.2 caf_register_t
Indicates which kind of coarray variable should be registered.

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.

8.1.3 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.

8.1.4 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;

typedef struct caf_reference_t {
    struct { /* The offset (in bytes) of the component in the derived type. 
                Unused for allocatable or pointer components. */
        /* A pointer to the next ref or NULL. */
        struct caf_reference_t *next;
        /* 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;
    };
    struct {
        /* 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
    } type;
} 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 member 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_REF_VECTOR is not implemented in the front end for descriptor-less arrays. The library caf_single has untested support for it.

### 8.1.5 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.

### 8.2 Function ABI Documentation
8.2.1 _gfortran_caf_init — Initialization function

Description:
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.

Syntax: void _gfortran_caf_init (int *argc, char ***argv)

Arguments:
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.

NOTES 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.

8.2.2 _gfortran_caf_finish — Finalization function

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

Syntax: void _gfortran_caf_finish (void)

NOTES 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.

8.2.3 _gfortran_caf_this_image — Querying the image number

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

Syntax: int _gfortran_caf_this_image (int distance)

Arguments:
distance As specified for the this_image intrinsic in TS18508. Shall be a non-negative number.

NOTES 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.
8.2.4 _gfortran_caf_num_images — Querying the maximal number of images

Description:
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.

Syntax:  
int _gfortran_caf_num_images(int distance, int failed)

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

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

8.2.5 _gfortran_caf_image_status — Query the status of an image

Description:
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.

Syntax:  
int _gfortran_caf_image_status (int image, caf_team_t * team)

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

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

8.2.6 _gfortran_caf_failed_images — Get an array of the indexes of the failed images

Description:
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.

Syntax:  
int _gfortran_caf_failed_images (caf_team_t * team, int * kind)

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

NOTES
This function follows TS18508. Because team-functionality is not yet implemented a null-pointer is passed for the team argument at the moment.
8.2.7 _gfortran_caf_stopped_images — Get an array of the indexes of the stopped images

Description:
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.

Syntax: int _gfortran_caf_stopped_images (caf_team_t * team, int * kind)

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

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

8.2.8 _gfortran_caf_register — Registering coarrays

Description:
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.

For CAF_REGTYPE_COARRAY_STATIC and CAF_REGTYPE_COARRAY_ALLOC, the passed size is the byte size requested. For CAF_REGTYPE_LOCK_STATIC, CAF_REGTYPE_LOCK_ALLOC and CAF_REGTYPE_CRITICAL it is the array size or one for a scalar.

When CAF_REGTYPE_COARRAY_ALLOC_REGISTER_ONLY is used, then only a token for an allocatable or pointer component is created. The SIZE parameter is not used then. On the contrary when CAF_REGTYPE_COARRAY_ALLOCALLOCATE_ONLY is specified, then the token needs to be registered by a previous call with regtype CAF_REGTYPE_COARRAY_ALLOC_REGISTER_ONLY and either the memory specified in the DESC’s data-ptr is registered or allocate when the data-ptr is NULL.

Syntax: 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)
Arguments:

sizeFor normal coarrays, the byte size of the coarray to be allocated; for lock types and event types, the number of elements.
typeone of the caf_register_t types.
tokenintent(out) An opaque pointer identifying the coarray.
descintent(inout) The (pseudo) array descriptor.
statintent(out) For allocatable coarrays, stores the STAT=; may be NULL.
errmsgintent(out) When an error occurs, this will be set to an error message; may be NULL.
errmsg_lenthe buffer size of errmsg.

NOTES 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.

8.2.9_gfortran_caf_deregister — Deregistering coarrays

Description:

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.

Syntax:

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

Arguments:

tokenthe token to free.
typethe 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.
statintent(out) Stores the STAT=; may be NULL.
errmsgintent(out) When an error occurs, this will be set to an error message; may be NULL.
\texttt{errmsg\_len} \quad \text{the buffer size of \texttt{errmsg}.}

\textbf{NOTES} \quad \text{For nonalloatable 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.}

\textbf{8.2.10 \_gfortran\_caf\_is\_present --- Query whether an allocatable or pointer component in a derived type coarray is allocated}

\textit{Description:}

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

\textit{Syntax:}

\texttt{void \_gfortran\_caf\_is\_present (caf\_token\_t token, int image\_index, gfc\_reference\_t \*ref)}

\textit{Arguments:}

- \texttt{token} \quad \text{An opaque pointer identifying the coarray.}
- \texttt{image\_index} \quad \text{The ID of the remote image; must be a positive number.}
- \texttt{ref} \quad \text{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.}

\textbf{8.2.11 \_gfortran\_caf\_send --- Sending data from a local image to a remote image}

\textit{Description:}

\text{Called to send a scalar, an array section or a whole array from a local to a remote image identified by the image\_index.}

\textit{Syntax:}

\texttt{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)}

\textit{Arguments:}

- \texttt{token} \quad \text{intent(in) \text{An opaque pointer identifying the coarray.}}
- \texttt{offset} \quad \text{intent(in) \text{By which amount of bytes the actual data is shifted compared to the base address of the coarray.}}
- \texttt{image\_index} \quad \text{intent(in) \text{The ID of the remote image; must be a positive number.}}
- \texttt{dest} \quad \text{intent(in) \text{Array descriptor for the remote image for the bounds and the size. The base\_addr shall not be accessed.}}
- \texttt{dst\_vector} \quad \text{intent(in) \text{If not NULL, it contains the vector subscript of the destination array; the values are relative to the dimension triplet of the dest argument.}}
- \texttt{src} \quad \text{intent(in) \text{Array descriptor of the local array to be transferred to the remote image}}
- \texttt{dst\_kind} \quad \text{intent(in) \text{Kind of the destination argument}}
- \texttt{src\_kind} \quad \text{intent(in) \text{Kind of the source argument}}
may_require_tmp

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

\textbf{stat} \textbf{intent(out)} when non-NULL give the result of the operation,
\texttt{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.

\textbf{NOTES} It is permitted to have \texttt{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 \texttt{memmove}
which handles (partially) overlapping memory. If \texttt{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 \texttt{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.

\section*{8.2.12 \texttt{\_gfortran\_caf\_get} — Getting data from a remote image}

\textbf{Description:}
Called to get an array section or a whole array from a remote, image identified
by the \texttt{image_index}.

\textbf{Syntax:}
\begin{verbatim}
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)
\end{verbatim}

\textbf{Arguments:}
\begin{itemize}
\item \textit{token} \textbf{intent(in)} An opaque pointer identifying the coarray.
\item \textit{offset} \textbf{intent(in)} By which amount of bytes the actual data is shifted
compared to the base address of the coarray.
\item \textit{image_index} \textbf{intent(in)} The ID of the remote image; must be a positive
number.
\item \textit{dest} \textbf{intent(out)} Array descriptor of the local array to store the
data retrieved from the remote image
\item \textit{src} \textbf{intent(in)} Array descriptor for the remote image for the
bounds and the size. The \texttt{base_addr} shall not be accessed.
\item \textit{src\_vector} \textbf{intent(in)} If not NULL, it contains the vector subscript of the
source array; the values are relative to the dimension triplet
of the \texttt{src} argument.
\item \textit{dst\_kind} \textbf{intent(in)} Kind of the destination argument
\item \textit{src\_kind} \textbf{intent(in)} Kind of the source argument
\end{itemize}
may_require_temp

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.

NOTES

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_temp 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.

8.2.13 _gfortran_caf_sendget — Sending data between remote images

Description:

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.

Syntax:

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)

Arguments:

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.
Chapter 8: Coarray Programming

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.

NOTES 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.

8.2.14 _gfortran_caf_send_by_ref — Sending data from a local image to a remote image with enhanced referencing options

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

Syntax:
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)

Arguments:
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.

**dstreallocatable**
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.

**NOTES**
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.

### 8.2.15 _gfortran_caf_get_by_ref — Getting data from a remote image using enhanced references

**Description:**
Called to get a scalar, an array section or a whole array from a remote image identified by the image_index.

**Syntax:**
```
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)
```
Chapter 8: Coarray Programming

Arguments:

- **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.

NOTES

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.

8.2.16 `_gfortran_caf_sendget_by_ref` — Sending data between remote images using enhanced references on both sides

Description:

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`. 
Syntax:

```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)
```

Arguments:

- `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-
- `src_stat` intent(out) When non-
- `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.

NOTES

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 `memcpy`
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.

### 8.2.17 `_gfortran_caf_lock` — Locking a lock variable

**Description:**

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 func-
tion 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.

**Syntax:**

```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)
```

**Arguments:**

- **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) 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

**NOTES**

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.

### 8.2.18 `_gfortran_caf_unlock` — Unlocking a lock variable

**Description:**

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.

**Syntax:**

```c
void _gfortran_caf_unlock (caf_token_t token, size_t index, int
image_index, int *stat, char *errmsg, size_t errmsg_len)
```
Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Intent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>token</td>
<td>intent(in)</td>
<td>An opaque pointer identifying the coarray.</td>
</tr>
<tr>
<td>index</td>
<td>intent(in)</td>
<td>Array index; first array index is 0. For scalars, it is always 0.</td>
</tr>
<tr>
<td>image_index</td>
<td>intent(in)</td>
<td>The ID of the remote image; must be a positive number.</td>
</tr>
<tr>
<td>stat</td>
<td>intent(out)</td>
<td>For allocatable coarrays, stores the STAT=; may be NULL.</td>
</tr>
<tr>
<td>errmsg</td>
<td>intent(out)</td>
<td>When an error occurs, this will be set to an error message; may be NULL.</td>
</tr>
<tr>
<td>errmsg_len</td>
<td>intent(in)</td>
<td>The buffer size of errmsg</td>
</tr>
</tbody>
</table>

NOTES 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.

8.2.19 _gfortran_caf_event_post — Post an event

Description:
Increment the event count of the specified event variable.

Syntax:
void _gfortran_caf_event_post (caf_token_t token, size_t index, int image_index, int *stat, char *errmsg, size_t errmsg_len)

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Intent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>token</td>
<td>intent(in)</td>
<td>An opaque pointer identifying the coarray.</td>
</tr>
<tr>
<td>index</td>
<td>intent(in)</td>
<td>Array index; first array index is 0. For scalars, it is always 0.</td>
</tr>
<tr>
<td>image_index</td>
<td>intent(in)</td>
<td>The ID of the remote image; must be a positive number; zero indicates the current image, when accessed noncoindexed.</td>
</tr>
<tr>
<td>stat</td>
<td>intent(out)</td>
<td>Stores the STAT=; may be NULL.</td>
</tr>
<tr>
<td>errmsg</td>
<td>intent(out)</td>
<td>When an error occurs, this will be set to an error message; may be NULL.</td>
</tr>
<tr>
<td>errmsg_len</td>
<td>intent(in)</td>
<td>The buffer size of errmsg</td>
</tr>
</tbody>
</table>

NOTES 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 preceding push communications of this image to the specified remote image have to be completed before event_wait on the remote image returns.

8.2.20 _gfortran_caf_event_wait — Wait that an event occurred

Description:
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.

Syntax:
void _gfortran_caf_event_wait (caf_token_t token, size_t index, int until_count, int *stat, char *errmsg, size_t errmsg_len)

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Intent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>token</td>
<td>intent(in)</td>
<td>An opaque pointer identifying the coarray.</td>
</tr>
</tbody>
</table>
Chapter 8: Coarray Programming

8.2.21 _gfortran_caf_event_query — Query event count

Description:
Return the event count of the specified event variable.

Syntax:
void _gfortran_caf_event_query (caf_token_t token, size_t index, int image_index, int *count, int *stat)

Arguments:
- 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.

NOTES
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.

8.2.22 _gfortran_caf_sync_all — All-image barrier

Description:
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 cur-
rent team. Additionally, it ensures that all pending data transfers of previous segment have completed.

Syntax:  
void _gfortran_caf_sync_all (int *stat, char *errmsg, size_t errmsg_len)

Arguments:

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

8.2.23 _gfortran_caf_sync_images — Barrier for selected images

Description:
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.

Syntax:  
void _gfortran_caf_sync_images (int count, int images[], int *stat, char *errmsg, size_t errmsg_len)

Arguments:

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

8.2.24 _gfortran_caf_sync_memory — Wait for completion of segment-memory operations

Description:
Acts as optimization barrier between different segments. It also ensures that all pending memory operations of this image have been completed.

Syntax:  
void _gfortran_caf_sync_memory (int *stat, char *errmsg, size_t errmsg_len)

Arguments:

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

\_\texttt{asm} \_\texttt{volatile} ("\texttt{:::"memory"}) to prevent code movements.

8.2.25 _gfortran_caf_error_stop — Error termination with exit code

Description:
Invoked for an \texttt{ERROR STOP} statement which has an integer argument. The function should terminate the program with the specified exit code.

Syntax: void \_gfortran\_caf\_error\_stop (int error)

Arguments:
- \texttt{error} intent(in) The exit status to be used.

8.2.26 _gfortran_caf_error_stop_str — Error termination with string

Description:
Invoked for an \texttt{ERROR STOP} statement which has a string as argument. The function should terminate the program with a nonzero-exit code.

Syntax: void \_gfortran\_caf\_error\_stop (const char \*string, size_t len)

Arguments:
- \texttt{string} intent(in) The error message (not zero terminated)
- \texttt{len} intent(in) The length of the string

8.2.27 _gfortran_caf_fail_image — Mark the image failed and end its execution

Description:
Invoked for an \texttt{FAIL IMAGE} statement. The function should terminate the current image.

Syntax: void \_gfortran\_caf\_fail\_image ()

NOTES This function follows TS18508.

8.2.28 _gfortran_caf_atomic_define — Atomic variable assignment

Description:
Assign atomically a value to an integer or logical variable.

Syntax: void \_gfortran\_caf\_atomic\_define (caf\_token\_t token, size\_t offset, int image\_index, void \*value, int \*stat, int type, int kind)

Arguments:
- \texttt{token} intent(in) An opaque pointer identifying the coarray.
- \texttt{offset} intent(in) By which amount of bytes the actual data is shifted compared to the base address of the coarray.
- \texttt{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)

8.2.29 _gfortran_caf_atomic_ref — Atomic variable reference

Description:
Reference atomically a value of a kind-4 integer or logical variable.

Syntax: void _gfortran_caf_atomic_ref (caf_token_t token, size_t offset, int image_index, void *value, int *stat, int type, int kind)

Arguments:

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)

8.2.30 _gfortran_caf_atomic_cas — Atomic compare and swap

Description:
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.

Syntax: 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)

Arguments:

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 comparision.
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 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)

8.2.31 _gfortran_caf_atomic_op — Atomic operation

Description:
Apply an operation atomically to an atomic integer or logical variable. After the operation, 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 value; the result is then stored in the atomic variable.

Syntax: 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)

Arguments:
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 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)

8.2.32 _gfortran_caf_co_broadcast — Sending data to all images

Description:
Distribute a value from a given image to all other images in the team. Has to be called collectively.

Syntax: void _gfortran_caf_co_broadcast (gfc_descriptor_t *a, int source_image, int *stat, char *errmsg, size_t errmsg_len)
Arguments:
- `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.

8.2.33 _gfortran_caf_co_max — Collective maximum reduction

Description:
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.

Syntax:
```c
void _gfortran_caf_co_max (gfc_descriptor_t *a, int result_image, int *stat, char *errmsg, int a_len, size_t errmsg_len)
```

Arguments:
- `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

NOTES 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.

8.2.34 _gfortran_caf_co_min — Collective minimum reduction

Description:
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.

Syntax:
```c
void _gfortran_caf_co_min (gfc_descriptor_t *a, int result_image, int *stat, char *errmsg, int a_len, size_t errmsg_len)
```
**Chapter 8: Coarray Programming**

**Arguments:**

- `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`

**NOTES** 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.

### 8.2.35 _gfortran_caf_co_sum — Collective summing reduction

**Description:**
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.

**Syntax:**

```c
void _gfortran_caf_co_sum (gfc_descriptor_t *a, int result_image, int *stat, char *errmsg, size_t errmsg_len)
```

**Arguments:**

- `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 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`

**NOTES** 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.

### 8.2.36 _gfortran_caf_co_reduce — Generic collective reduction

**Description:**
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.
The opr_flags denote the following; the values are bitwise ored. GFC_CAF_BYREF (1) if the result should be returned by reference; GFC_CAF_HIDDENLEN (2) whether the result and argument string lengths shall be specified as hidden arguments; GFC_CAF_ARG_VALUE (4) whether the arguments shall be passed by value, GFC_CAF_ARG_DESC (8) whether the arguments shall be passed by descriptor.

**Syntax:**
```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)
```

**Arguments:**
- `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 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`

**NOTES**
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.
9 Intrinsic Procedures

9.1 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.

9.2 ABORT — Abort the program

Description:

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.

Standard: GNU extension

Class: Subroutine

Syntax: CALL ABORT

Return value:

Does not return.

Example:
program test_abort
  integer :: i = 1, j = 2
  if (i /= j) call abort
end program test_abort

See also: Section 9.101 [EXIT], page 173, Section 9.163 [KILL], page 211, Section 9.39 [BACKTRACE], page 133

9.3 ABS — Absolute value

Description:
  ABS(A) computes the absolute value of A.

Standard: Fortran 77 and later, has overloads that are GNU extensions

Class: Elemental function

Syntax: RESULT = ABS(A)

Arguments:
  A The type of the argument shall be an INTEGER, REAL, or
      COMPLEX.

Return value:
  The return value is of the same type and kind as the argument except the return
  value is REAL for a COMPLEX argument.

Example:

  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>BABS(A)</td>
<td>INTEGER(1) A</td>
<td>INTEGER(1)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>IIABS(A)</td>
<td>INTEGER(2) A</td>
<td>INTEGER(2)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>JIABS(A)</td>
<td>INTEGER(4) A</td>
<td>INTEGER(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>KIABS(A)</td>
<td>INTEGER(8) A</td>
<td>INTEGER(8)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>ZABS(A)</td>
<td>COMPLEX(8) A</td>
<td>COMPLEX(8)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>CDABS(A)</td>
<td>COMPLEX(8) A</td>
<td>COMPLEX(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>
9.4 ACCESS — Checks file access modes

Description:
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.

Standard: GNU extension
Class: Inquiry function
Syntax: RESULT = ACCESS(NAME, MODE)

Arguments:

NAME Scalar CHARACTER of default kind with the file name. Tailing 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.

Return value:
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.

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
```

Specific names:
See also:

9.5 ACHAR — Character in ASCII collating sequence

Description:
ACHAR(I) returns the character located at position I in the ASCII collating sequence.

Standard: Fortran 77 and later, with KIND argument Fortran 2003 and later
Class: Elemental function
Syntax: RESULT = ACHAR(I [, KIND])
Arguments:

- **I**
  - The type shall be **INTEGER**.

- **KIND**
  - (Optional) An **INTEGER** initialization expression indicating the kind parameter of the result.

Return value:

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.

Example:

```fortran
program test_achar
  character c
  c = achar(32)
end program test_achar
```

Note: See Section 9.143 [ICHAR], page 199 for a discussion of converting between numerical values and formatted string representations.

See also: Section 9.59 [CHAR], page 145, Section 9.135 [IACHAR], page 194, Section 9.143 [ICHAR], page 199

### 9.6 ACOS — Arccosine function

Description:

ACOS(X) computes the arccosine of X (inverse of COS(X)).

Standard:

- Fortran 77 and later, for a complex argument Fortran 2008 or later

Class:

- Elemental function

Syntax:

```fortran
RESULT = ACOS(X)
```

Arguments:

- **X**
  - The type shall either be **REAL** with a magnitude that is less than or equal to one - or the type shall be **COMPLEX**.

Return value:

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 $0 \leq \Re (\acos(x)) \leq \pi$.

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) X</td>
<td>REAL(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>DACOS(X)</td>
<td>REAL(8) X</td>
<td>REAL(8)</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

See also: Inverse function: Section 9.73 [COS], page 154 Degrees function: Section 9.7 [ACOSD], page 111
9.7 ACOSD — Arccosine function, degrees

Description:

ACOSD(X) computes the arccosine of X in degrees (inverse of COSD(X)).

This function is for compatibility only and should be avoided in favor of standard constructs wherever possible.

Standard: GNU Extension, enabled with "-fdec-math"

Class: Elemental function

Syntax: RESULT = ACOSD(X)

Arguments:

X The type shall either be REAL with a magnitude that is less than or equal to one - or the type shall be COMPLEX.

Return value:

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 ≤ ℜacos(x) ≤ 180.

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: Section 9.74 [COSD], page 155 Radians function: Section 9.6 [ACOS], page 110

9.8 ACOSH — Inverse hyperbolic cosine function

Description:

ACOSH(X) computes the inverse hyperbolic cosine of X.

Standard: Fortran 2008 and later

Class: Elemental function

Syntax: RESULT = ACOSH(X)

Arguments:

X The type shall be REAL or COMPLEX.

Return value:

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 ≤ ℑacosh(x) ≤ π.

Example:
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: Section 9.75 [COSH], page 156

9.9 ADJUSTL — Left adjust a string

Description:

ADJUSTL(STRING) will left adjust a string by removing leading spaces. Spaces are inserted at the end of the string as needed.

Standard: Fortran 90 and later

Class: Elemental function

Syntax:

RESULT = ADJUSTL(STRING)

Arguments:

STRING The type shall be CHARACTER.

Return value:

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.

Example:

program test_adjustl
  character(len=20) :: str = ' gfortran'
  str = adjustl(str)
  print *, str
end program test_adjustl

See also: Section 9.10 [ADJUSTR], page 112, Section 9.276 [TRIM], page 276

9.10 ADJUSTR — Right adjust a string

Description:

ADJUSTR(STRING) will right adjust a string by removing trailing spaces. Spaces are inserted at the start of the string as needed.

Standard: Fortran 95 and later

Class: Elemental function

Syntax:

RESULT = ADJUSTR(STRING)

Arguments:

STR The type shall be CHARACTER.
Return value:
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.

Example:

```fortran
program test_adjustr
    character(len=20) :: str = 'gfortran'
    str = adjustr(str)
    print *, str
end program test_adjustr
```

See also: Section 9.9 [ADJUSTL], page 112, Section 9.276 [TRIM], page 276

9.11 AIMAG — Imaginary part of complex number

Description:
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.

Standard: Fortran 77 and later, has overloads that are GNU extensions

Class: Elemental function

Syntax: RESULT = AIMAG(Z)

Arguments:
Z The type of the argument shall be COMPLEX.

Return value:
The return value is of type REAL with the kind type parameter of the argument.

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>GNU extension</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>

9.12 AINT — Truncate to a whole number

Description:
AINT(A [, KIND]) truncates its argument to a whole number.

Standard: Fortran 77 and later
Class: Elemental function

Syntax: \[ \text{RESULT} = \text{AINT}(\text{A} [, \text{KIND}]) \]

Arguments:

- \text{A} \quad \text{The type of the argument shall be REAL.}
- \text{KIND} \quad \text{(Optional) An INTEGER initialization expression indicating the kind parameter of the result.}

Return value:

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, \text{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.

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>

9.13 ALARM — Execute a routine after a given delay

Description:

\text{ALARM(SECONDS, HANDLER [, STATUS])} causes external subroutine \text{HANDLER} to be executed after a delay of \text{SECONDS} by using \text{alarm(2)} to set up a signal and \text{signal(2)} to catch it. If \text{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.

Standard: GNU extension

Class: Subroutine

Syntax: \[ \text{CALL ALARM(SECONDS, HANDLER [, STATUS])} \]

Arguments:

- \text{SECONDS} \quad \text{The type of the argument shall be a scalar INTEGER. It is INTENT(IN).}
- \text{HANDLER} \quad \text{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).

Example:

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.

9.14 ALL — All values in MASK along DIM are true

Description:

ALL(MASK [, DIM]) determines if all the values are true in MASK in the array along dimension DIM.

Standard:  Fortran 95 and later

Class:  Transformational function

Syntax:  RESULT = ALL(MASK [, DIM])

Arguments:

- 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.

Return value:

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.

(A)  ALL(MASK) is true if all elements of MASK are true. It also is true if MASK has zero size; otherwise, it is false.

(B)  If the rank of MASK is one, then ALL(MASK,DIM) is equivalent to ALL(MASK). If the rank is greater than one, then ALL(MASK,DIM) is determined by applying ALL to the array sections.

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

9.15 ALLOCATED — Status of an allocatable entity

Description:

ALLOCATED(ARRAY) and ALLOCATED(SCALAR) check the allocation status of ARRAY and SCALAR, respectively.

Standard: Fortran 95 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)

Arguments:

ARRAY The argument shall be an ALLOCATABLE array.

SCALAR The argument shall be an ALLOCATABLE scalar.

Return value:

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.

Example:

program test_allocated
integer :: i = 4
real(4), allocatable :: x(:)
if (.not. allocated(x)) allocate(x(i))
end program test_allocated

9.16 AND — Bitwise logical AND

Description:

Bitwise logical AND.

This intrinsic routine is provided for backwards compatibility with GNU Fortran 77. For integer arguments, programmers should consider the use of the Section 9.137 [IAND], page 195 intrinsic defined by the Fortran standard.

Standard: GNU extension

Class: Function

Syntax: RESULT = AND(I, J)

Arguments:

I The type shall be either a scalar INTEGER type or a scalar LOGICAL type or a xz-literal-constant.
Chapter 9: Intrinsic Procedures

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.

Return value:
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 Section 9.149 [INT], page 203 occurred.

Example:
```
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: Section 9.137 [IAND], page 195

9.17 ANINT — Nearest whole number

Description:
ANINT(\( A \) [, KIND]) rounds its argument to the nearest whole number.

Standard: Fortran 77 and later

Class: Elemental function

Syntax:
RESULT = ANINT(A [, KIND])

Arguments:
- \( A \): The type of the argument shall be REAL.
- \( KIND \) (Optional): An INTEGER initialization expression indicating the kind parameter of the result.

Return value:
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(\( A + 0.5 \)). If \( A \) is less than or equal to zero then it returns AINT(\( A - 0.5 \)).

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>AINT(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>

9.18 ANY — Any value in MASK along DIM is true

Description:

ANY(MASK [, DIM]) determines if any of the values in the logical array MASK along dimension DIM are .TRUE..

Standard: Fortran 95 and later

Class: Transformational function

Syntax: RESULT = ANY(MASK [, DIM])

Arguments:

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.

Return value:

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.

(A) ANY(MASK) is true if any element of MASK is true; otherwise, it is false. It also is false if MASK has zero size.

(B) If the rank of MASK is one, then ANY(MASK,DIM) is equivalent to ANY(MASK). If the rank is greater than one, then ANY(MASK,DIM) is determined by applying ANY to the array sections.

Example:

```fortran
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 *, a.eq. b, 1
      print *, any(a .eq. b, 2)
    end subroutine section
  end program test_any
```
9.19 ASIN — Arcsine function

Description:
ASIN(X) computes the arcsine of its X (inverse of SIN(X)).

Standard: Fortran 77 and later, for a complex argument Fortran 2008 or later

Class: Elemental function

Syntax: RESULT = ASIN(X)

Arguments:
X The type shall be either REAL and a magnitude that is less than or equal to one - or be COMPLEX.

Return value:
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 \(-\pi/2 \leq \Re \text{asin}(x) \leq \pi/2\).

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)</td>
<td>REAL(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>DASIN(X)</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

See also: Inverse function: Section 9.250 [SIN], page 260 Degrees function: Section 9.20 [ASIND], page 119

9.20 ASIND — Arcsine function, degrees

Description:
ASIND(X) computes the arcsine of its X in degrees (inverse of SIND(X)).
This function is for compatibility only and should be avoided in favor of standard constructs wherever possible.

Standard: GNU Extension, enabled with ‘-fdec-math’.

Class: Elemental function

Syntax: RESULT = ASIND(X)

Arguments:
X The type shall be either REAL and a magnitude that is less than or equal to one - or be COMPLEX.

Return value:
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 \(-90 \leq \Re \text{asin}(x) \leq 90\).

Example:
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>X</td>
<td>REAL(4) GNU Extension</td>
</tr>
<tr>
<td>DASIND(X)</td>
<td>REAL(8)</td>
<td>X</td>
<td>REAL(8) GNU Extension</td>
</tr>
</tbody>
</table>

See also: Inverse function: Section 9.251 [SIND], page 261 Radians function: Section 9.19 [ASIN], page 119

9.21 ASINH — Inverse hyperbolic sine function

Description:

ASINH(X) computes the inverse hyperbolic sine of X.

Standard: Fortran 2008 and later

Class: Elemental function

Syntax: RESULT = ASINH(X)

Arguments:

X The type shall be REAL or COMPLEX.

Return value:

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 $-\pi/2 \leq \Im \text{asinh}(x) \leq \pi/2$.

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>X</td>
<td>REAL(8) GNU extension</td>
</tr>
</tbody>
</table>

See also: Inverse function: Section 9.252 [SINH], page 262

9.22 ASSOCIATED — Status of a pointer or pointer/target pair

Description:

ASSOCIATED(PINTER [, TARGET]) determines the status of the pointer POINTER or if POINTER is associated with the target TARGET.

Standard: Fortran 95 and later

Class: Inquiry function

Syntax: RESULT = ASSOCIATED(PINTER [, TARGET])
Arguments:

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.

The association status of neither POINTER nor TARGET shall be undefined.

Return value:

ASSOCIATED(POINTER) returns a scalar value of type LOGICAL(4). There are several cases:

(A) When the optional TARGET is not present then

   ASSOCIATED(POINTER) is true if POINTER is associated with a target; otherwise, it returns false.

(B) If TARGET is present and a scalar target, the result is true if

   TARGET is not a zero-sized storage sequence and the target associated with POINTER occupies the same storage units. If POINTER is disassociated, the result is false.

(C) If TARGET is present and an array target, the result is true if

   TARGET and POINTER have the same shape, are not zero-sized arrays, are arrays whose elements are not zero-sized storage sequences, and TARGET and POINTER occupy the same storage units in array element order. As in case (B), the result is false, if POINTER is disassociated.

(D) If TARGET is present and an scalar pointer, the result is true if

   if TARGET is associated with POINTER, the target associated with TARGET are not zero-sized storage sequences and occupy the same storage units. The result is false, if either TARGET or POINTER is disassociated.

(E) If TARGET is present and an array pointer, the result is true if

   target associated with POINTER and the target associated with TARGET have the same shape, are not zero-sized arrays, are arrays whose elements are not zero-sized storage sequences, and TARGET and POINTER occupy the same storage units in array element order. The result is false, if either TARGET or POINTER is disassociated.

Example:

```fortran
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: Section 9.210 [NULL], page 237
9.23 ATAN — Arctangent function

Description:
ATAN(X) computes the arctangent of X.

Standard: Fortran 77 and later, for a complex argument and for two arguments Fortran 2008 or later

Class: Elemental function

Syntax:
RESULT = ATAN(X)
RESULT = ATAN(Y, X)

Arguments:
X The type shall be REAL or COMPLEX; if Y is present, X shall be REAL.
Y shall be of the same type and kind as X.

Return value:
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 the arcus tangent of X, where the real part of the result is in radians and lies in the range \(-\pi/2 \leq \Re \text{atan}(x) \leq \pi/2\).

Example:

```fortran
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) X</td>
<td>REAL(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>DATAN(X)</td>
<td>REAL(8) X</td>
<td>REAL(8)</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

See also: Inverse function: Section 9.266 [TAN], page 271 Degrees function: Section 9.24 [ATAND], page 122

9.24 ATAND — Arctangent function, degrees

Description:
ATAND(X) computes the arctangent of X in degrees (inverse of Section 9.267 [TAND], page 271).

This function is for compatibility only and should be avoided in favor of standard constructs wherever possible.

Standard: GNU Extension, enabled with ‘-fdec-math’.

Class: Elemental function
Syntax:

\[
\begin{align*}
\text{RESULT} &= \text{ATAND}(X) \\
\text{RESULT} &= \text{ATAND}(Y, X)
\end{align*}
\]

Arguments:

\[
\begin{align*}
X & \quad \text{The type shall be REAL or COMPLEX; if } Y \text{ is present, } X \text{ shall be REAL.} \\
Y & \quad \text{shall be of the same type and kind as } X.
\end{align*}
\]

Return value:

The return value is of the same type and kind as \( X \). If \( Y \) is present, the result is identical to \( \text{ATAND2}(Y, X) \). Otherwise, it is the arcus tangent of \( X \), where the real part of the result is in degrees and lies in the range \(-90 \leq \Re \text{atan}(x) \leq 90\).

Example:

```
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: Section 9.267 [TAND], page 271 Radians function: Section 9.23 [ATAN], page 122

9.25 ATAN2 — Arctangent function

Description:

\( \text{ATAN2}(Y, X) \) computes the principal value of the argument function of the complex number \( X + iY \). This function can be used to transform from Cartesian into polar coordinates and allows to determine the angle in the correct quadrant.

Standard: Fortran 77 and later

Class: Elemental function

Syntax: \( \text{RESULT} = \text{ATAN2}(Y, X) \)

Arguments:

\[
\begin{align*}
Y & \quad \text{The type shall be REAL.} \\
X & \quad \text{The type and kind type parameter shall be the same as } Y. \text{ If } Y \text{ is zero, then } X \text{ must be nonzero.}
\end{align*}
\]

Return value:

The return value has the same type and kind type parameter as \( Y \). It is the principal value of the complex number \( X + iY \). 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 \( \pi/2 \).

**Example:**

```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: Section 9.23 [ATAN], page 122 Degrees function: Section 9.26 [ATAN2D], page 124

### 9.26 ATAN2D — Arctangent function, degrees

**Description:**

\( \text{ATAN2D}(Y, X) \) computes the principal value of the argument function of the complex number \( X + iY \) in degrees. This function can be used to transform from Cartesian into polar coordinates and allows to determine the angle in the correct quadrant.

This function is for compatibility only and should be avoided in favor of standard constructs wherever possible.

**Standard:** GNU Extension, enabled with `-fdec-math`.

**Class:** Elemental function

**Syntax:**

\[
\text{RESULT} = \text{ATAN2D}(Y, X)
\]

**Arguments:**

- \( Y \) 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.

**Return value:**

The return value has the same type and kind type parameter as \( Y \). It is the principal value of the complex number \( X + iY \). If \( X \) is nonzero, then it lies in the range \(-180 \leq \text{atan}(x) \leq 180\). 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.

**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>

See also: Alias: Section 9.24 [ATAND], page 122 Radians function: Section 9.25 [ATAN2], page 123

9.27 ATANH — Inverse hyperbolic tangent function

Description:

\texttt{ATANH(X)} computes the inverse hyperbolic tangent of \( X \).

Standard: Fortran 2008 and later

Class: Elemental function

Syntax:

\texttt{RESULT = ATANH(X)}

Arguments:

\( X \quad \text{The type shall be REAL or COMPLEX.} \)

Return value:

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 \(-\pi/2 \leq \mathfrak{I} \text{atanh}(x) \leq \pi/2\).

Example:

\begin{verbatim}
PROGRAM test_atanh
   REAL, DIMENSION(3) :: x = (/ -1.0, 0.0, 1.0 /)
   WRITE (*,*) ATANH(x)
END PROGRAM
\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>DATANH(X)</td>
<td>REAL(8) X</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

See also: Inverse function: Section 9.268 [TANH], page 272

9.28 ATOMIC_ADD — Atomic ADD operation

Description:

\texttt{ATOMIC_ADD(ATOM, VALUE)} atomically adds the value of \( \text{VAR} \) to the variable \( \text{ATOM} \). When \( \text{STAT} \) 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 \( \text{ATOM} \), if the remote image has stopped, it is assigned the value of \( \text{ISO_FORTRAN_ENV} \)'s \( \text{STAT_STOPPED_IMAGE} \) and if the remote image has failed, the value \( \text{STAT_FAILED_IMAGE} \).

Standard: TS 18508 or later

Class: Atomic subroutine

Syntax:

\texttt{CALL ATOMIC_ADD (ATOM, VALUE [, STAT])}
Arguments:

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.

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: Section 9.31 [ATOMIC_DEFINE], page 127, Section 9.32 [ATOMIC_FETCH_ADD], page 128, Section 10.1 [ISO_FORTRAN_ENV], page 283, Section 9.29 [ATOMIC_AND], page 126, Section 9.36 [ATOMIC_OR], page 131, Section 9.38 [ATOMIC_XOR], page 133

9.29 ATOMIC_AND — Atomic bitwise AND operation

Description:

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.

Standard: TS 18508 or later

Class: Atomic subroutine

Syntax: CALL ATOMIC_AND (ATOM, VALUE [, STAT])

Arguments:

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.

Example:

```fortran
program atomic
  use iso_fortran_env
  integer(atomic_int_kind) :: atom[*]
  call atomic_add (atom[1], int(b’10100011101’))
end program atomic
```

See also: Section 9.31 [ATOMIC_DEFINE], page 127, Section 9.33 [ATOMIC_FETCH_AND], page 129, Section 10.1 [ISO_FORTRAN_ENV], page 283, Section 9.28 [ATOMIC_ADD], page 125, Section 9.36 [ATOMIC_OR], page 131, Section 9.38 [ATOMIC_XOR], page 133
9.30 ATOMIC_CAS — Atomic compare and swap

Description:

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.

Standard: TS 18508 or later
Class: Atomic subroutine
Syntax: CALL ATOMIC_CAS (ATOM, OLD, COMPARE, NEW [, STAT])
Arguments:
- 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.

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: Section 9.31 [ATOMIC_DEFINE], page 127, Section 9.37 [ATOMIC_REF], page 132, Section 10.1 [ISO_FORTRAN_ENV], page 283

9.31 ATOMIC_DEFINE — Setting a variable atomically

Description:

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.

Standard: Fortran 2008 and later; with STAT, TS 18508 or later
Class: Atomic subroutine
**Syntax:**

CALL ATOMIC_DEFINE (ATOM, VALUE [, STAT])

**Arguments:**

ATOM 
Scalar coarray or coindexed variable of either integer type with ATOMIC_INT_KIND kind or logical type with ATOMIC_LOGICAL_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.

**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: Section 9.37 [ATOMIC_REF], page 132, Section 9.30 [ATOMIC_CAS], page 127, Section 10.1 [ISO_FORTRAN_ENV], page 283, Section 9.28 [ATOMIC_ADD], page 125, Section 9.29 [ATOMIC_AND], page 126, Section 9.36 [ATOMIC_OR], page 131, Section 9.38 [ATOMIC_XOR], page 133

### 9.32 ATOMIC_FETCH_ADD — Atomic ADD operation with prior fetch

**Description:**

ATOMIC_FETCH_ADD(ATOM, VALUE, OLD) atomically stores the value of ATOM in OLD and adds the value of VAR 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.

**Standard:** TS 18508 or later

**Class:** Atomic subroutine

**Syntax:**

CALL ATOMIC_FETCH_ADD (ATOM, VALUE, OLD [, STAT])

**Arguments:**

ATOM 
Scalar coarray or coindexed variable of integer type with ATOMIC_INT_KIND kind. ATOMIC_LOGICAL_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.

**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: Section 9.31 [ATOMIC_DEFINE], page 127, Section 9.28 [ATOMIC_ADD], page 125, Section 10.1 [ISO_FORTRAN_ENV], page 283, Section 9.33 [ATOMIC_FETCH_AND], page 129, Section 9.34 [ATOMIC_FETCH_OR], page 130, Section 9.35 [ATOMIC_FETCH_XOR], page 130

9.33 ATOMIC_FETCH_AND — Atomic bitwise AND operation with prior fetch

Description:

ATOM 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.

Standard: TS 18508 or later

Class: Atomic subroutine

Syntax: CALL ATOMIC_FETCH_AND (ATOM, VALUE, OLD [, STAT])

Arguments:

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.

Example:

program atomic
    use iso_fortran_env
    integer(atomic_int_kind) :: atom[*], old
    call atomic_fetch_and (atom[1], int(b’10100011101’), old)
end program atomic

See also: Section 9.31 [ATOMIC_DEFINE], page 127, Section 9.29 [ATOMIC_AND], page 126, Section 10.1 [ISO_FORTRAN_ENV], page 283, Section 9.32 [ATOMIC_FETCH_ADD], page 128, Section 9.34 [ATOMIC_FETCH_OR], page 130, Section 9.35 [ATOMIC_FETCH_XOR], page 130
9.34 ATOMIC_FETCH_OR — Atomic bitwise OR operation with prior fetch

Description:

ATOMIC_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.

Standard: TS 18508 or later

Class: Atomic subroutine

Syntax: CALL ATOMIC_FETCH_OR (ATOM, VALUE, OLD [, STAT])

Arguments:

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.

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: Section 9.31 [ATOMIC_DEFINE], page 127, Section 9.36 [ATOMIC_OR], page 131, Section 10.1 [ISO_FORTRAN_ENV], page 283, Section 9.32 [ATOMIC_FETCH_ADD], page 128, Section 9.33 [ATOMIC_FETCH_AND], page 129, Section 9.35 [ATOMIC_FETCH_XOR], page 130

9.35 ATOMIC_FETCH_XOR — Atomic bitwise XOR operation with prior fetch

Description:

ATOMIC_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.

Standard: TS 18508 or later

Class: Atomic subroutine
**Syntax:**

```fortran
CALL ATOMIC_FETCH_XOR (ATOM, VALUE, OLD [, STAT])
```

**Arguments:**

- **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.

**Example:**

```fortran
program atomic
    use iso_fortran_env
    integer(atomic_int_kind) :: atom[*], old
    call atomic_fetch_xor (atom[1], int('10100011101'), old)
end program atomic
```

**See also:**
- Section 9.31 [ATOMIC_DEFINE], page 127
- Section 9.38 [ATOMIC_XOR], page 133
- Section 10.1 [ISO_FORTRAN_ENV], page 283
- Section 9.32 [ATOMIC_FETCH_ADD], page 128
- Section 9.33 [ATOMIC_FETCH_AND], page 129
- Section 9.34 [ATOMIC_FETCH_OR], page 130

### 9.36 ATOMIC_OR — Atomic bitwise OR operation

**Description:**

`ATOMIC_OR(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`.

**Standard:**
- TS 18508 or later

**Class:**
- Atomic subroutine

**Syntax:**

```fortran
CALL ATOMIC_OR (ATOM, VALUE [, , STAT])
```

**Arguments:**

- **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.

**Example:**

```fortran
program atomic
    use iso_fortran_env
    integer(atomic_int_kind) :: atom[*]
    call atomic_or (atom[1], int('10100011101'))
end program atomic
```
9.37 ATOMIC_REF — Obtaining the value of a variable atomically

Description:

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.

Standard: Fortran 2008 and later; with STAT, TS 18508 or later

Class: Atomic subroutine

Syntax: CALL ATOMIC_REF(VALUE, ATOM [, STAT])

Arguments:

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.

Example:

```fortran
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: Section 9.31 [ATOMIC_DEFINE], page 127, Section 9.30 [ATOMIC_CAS], page 127, Section 10.1 [ISO_FORTRAN_ENV], page 283, Section 9.32 [ATOMIC_FETCH_ADD], page 128, Section 9.33 [ATOMIC_FETCH_AND], page 129, Section 9.34 [ATOMIC_FETCH_OR], page 130, Section 9.35 [ATOMIC_FETCH_XOR], page 130
9.38 ATOMIC_XOR — Atomic bitwise OR operation

Description:

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.

Standard: TS 18508 or later
Class: Atomic subroutine
Syntax: CALL ATOMIC_XOR (ATOM, VALUE [, STAT])
Arguments:
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.

Example:

program atomic
use iso_fortran_env
integer(atomic_int_kind) :: atom[*]
call atomic_xor (atom[1], int(b’10100011101’))
end program atomic

See also: Section 9.31 [ATOMIC_DEFINE], page 127, Section 9.35 [ATOMIC_FETCH_XOR], page 130, Section 10.1 [ISO_FORTRAN_ENV], page 283, Section 9.28 [ATOMIC_ADD], page 125, Section 9.36 [ATOMIC_OR], page 131, Section 9.38 [ATOMIC_XOR], page 133

9.39 BACKTRACE — Show a backtrace

Description:

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: Section 9.2 [ABORT], page 107
9.40 **BESSEL_J0 — Bessel function of the first kind of order 0**

*Description:* 
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.

*Standard:* Fortran 2008 and later

*Class:* Elemental function

*Syntax:* 
```
RESULT = BESSEL_J0(X)
```

*Arguments:* 

X  
The type shall be REAL.

*Return value:* 
The return value is of type REAL and lies in the range \(-0.4027... \leq \text{Bessel}(0,x) \leq 1\). It has the same kind as X.

*Example:* 
```
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)</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

9.41 **BESSEL_J1 — Bessel function of the first kind of order 1**

*Description:* 
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.

*Standard:* Fortran 2008

*Class:* Elemental function

*Syntax:* 
```
RESULT = BESSEL_J1(X)
```

*Arguments:* 

X  
The type shall be REAL.

*Return value:* 
The return value is of type REAL and lies in the range \(-0.5818... \leq \text{Bessel}(0,x) \leq 0.5818\). It has the same kind as X.

*Example:* 
```
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)</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>
9.42 BESSEL_JN — Bessel function of the first kind

Description:

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.

BESSEL_JN(N1, N2, X) returns an array with the Bessel functions of the first kind of the orders N1 to N2.

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:

RESULT = BESSEL_JN(N, X)
RESULT = BESSEL_JN(N1, N2, X)

Arguments:

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.

Return value:

The return value is a scalar of type REAL. It has the same kind as 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:

```
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</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
<tr>
<td></td>
<td></td>
<td>REAL(8) X</td>
<td></td>
</tr>
</tbody>
</table>

9.43 BESSEL_Y0 — Bessel function of the second kind of order 0

Description:

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.

Standard: Fortran 2008 and later

Class: Elemental function

Syntax: RESULT = BESSEL_Y0(X)
Arguments:

X  The type shall be REAL.

Return value:

The return value is of type REAL. It has the same kind as 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>

9.44 BESSEL_Y1 — Bessel function of the second kind of order 1

Description:

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.

Standard:  Fortran 2008 and later

Class:     Elemental function

Syntax:    RESULT = BESSEL_Y1(X)

Arguments:

X  The type shall be REAL.

Return value:

The return value is of type REAL. It has the same kind as 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)</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

9.45 BESSEL_YN — Bessel function of the second kind

Description:

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. BESSEL_YN(N1, N2, X) returns an array with the Bessel functions of the first kind of the orders N1 to N2.
**Standard:** Fortran 2008 and later, negative \( N \) is allowed as GNU extension

**Class:** Elemental function, except for the transformational function \texttt{BESSEL\_YN(N1, N2, X)}

**Syntax:**

\[
\begin{align*}
\text{RESULT} &= \text{BESSEL\_YN}(N, X) \\
\text{RESULT} &= \text{BESSEL\_YN}(N1, N2, X)
\end{align*}
\]

**Arguments:**

- \( N \): Shall be a scalar or an array of type \texttt{INTEGER}.
- \( N1 \): Shall be a non-negative scalar of type \texttt{INTEGER}.
- \( N2 \): Shall be a non-negative scalar of type \texttt{INTEGER}.
- \( X \): Shall be a scalar or an array of type \texttt{REAL}; for \texttt{BESSEL\_YN(N1, N2, X)} it shall be scalar.

**Return value:**

The return value is a scalar of type \texttt{REAL}. It has the same kind as \( X \).

**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:**

```
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</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
<tr>
<td></td>
<td>REAL(8)</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

### 9.46 \texttt{BGE} — Bitwise greater than or equal to

**Description:**

Determines whether an integral is a bitwise greater than or equal to another.

**Standard:** Fortran 2008 and later

**Class:** Elemental function

**Syntax:** \( \text{RESULT} = \text{BGE}(I, J) \)

**Arguments:**

- \( I \): Shall be of \texttt{INTEGER} type.
- \( J \): Shall be of \texttt{INTEGER} type, and of the same kind as \( I \).

**Return value:**

The return value is of type \texttt{LOGICAL} and of the default kind.

**See also:** Section 9.47 [BGT], page 138, Section 9.49 [BLE], page 138, Section 9.50 [BLT], page 139
9.47 BGT — Bitwise greater than

**Description:**
Determines whether an integral is a bitwise greater than another.

**Standard:** Fortran 2008 and later

**Class:** Elemental function

**Syntax:**
RESULT = BGT(I, J)

**Arguments:**
- I: Shall be of INTEGER type.
- J: Shall be of INTEGER type, and of the same kind as I.

**Return value:**
The return value is of type LOGICAL and of the default kind.

**See also:** Section 9.46 [BGE], page 137, Section 9.49 [BLE], page 138, Section 9.50 [BLT], page 139

9.48 BIT_SIZE — Bit size inquiry function

**Description:**
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.

**Standard:** Fortran 95 and later

**Class:** Inquiry function

**Syntax:**
RESULT = BIT_SIZE(I)

**Arguments:**
I: The type shall be INTEGER.

**Return value:**
The return value is of type INTEGER.

**Example:**

```fortran
program test_bit_size
  integer :: i = 123
  integer :: size
  size = bit_size(i)
  print *, size
end program test_bit_size
```

9.49 BLE — Bitwise less than or equal to

**Description:**
Determines whether an integral is a bitwise less than or equal to another.

**Standard:** Fortran 2008 and later

**Class:** Elemental function
Syntax: \( \text{RESULT} = \text{BLE}(I, J) \)

Arguments:
- \( I \)  Shall be of \text{INTEGER} type.
- \( J \)  Shall be of \text{INTEGER} type, and of the same kind as \( I \).

Return value:
The return value is of type \text{LOGICAL} and of the default kind.

See also: Section 9.47 \[\text{BGT}\], page 138, Section 9.46 \[\text{BGE}\], page 137, Section 9.50 \[\text{BLT}\], page 139

9.50 BLT — Bitwise less than

Description:
Determine whether an integral is a bitwise less than another.

Standard: Fortran 2008 and later

Class: Elemental function

Syntax: \( \text{RESULT} = \text{BLT}(I, J) \)

Arguments:
- \( I \)  Shall be of \text{INTEGER} type.
- \( J \)  Shall be of \text{INTEGER} type, and of the same kind as \( I \).

Return value:
The return value is of type \text{LOGICAL} and of the default kind.

See also: Section 9.46 \[\text{BGE}\], page 137, Section 9.47 \[\text{BGT}\], page 138, Section 9.49 \[\text{BLE}\], page 138

9.51 BTEST — Bit test function

Description:
\( \text{BTEST}(I, \text{POS}) \) returns logical \text{.TRUE.} if the bit at \text{POS} in \( I \) is set. The counting of the bits starts at 0.

Standard: Fortran 95 and later, has overloads that are GNU extensions

Class: Elemental function

Syntax: \( \text{RESULT} = \text{BTEST}(I, \text{POS}) \)

Arguments:
- \( I \)  The type shall be \text{INTEGER}.
- \( \text{POS} \)  The type shall be \text{INTEGER}.

Return value:
The return value is of type \text{LOGICAL}

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>F95 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>

9.52 C_ASSOCIATED — Status of a C pointer

Description:

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.

Standard: Fortran 2003 and later

Class: Inquiry function

Syntax:

RESULT = C_ASSOCIATED(c_ptr_1[, c_ptr_2])

Arguments:

<table>
<thead>
<tr>
<th>c_ptr_1</th>
<th>Scalar of the type C_PTR or C_FUNPTR.</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_ptr_2</td>
<td>(Optional) Scalar of the same type as c_ptr_1.</td>
</tr>
</tbody>
</table>

Return value:

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.

Example:

```
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: Section 9.56 [C.LOC], page 143, Section 9.55 [C.FUNLOC], page 142
9.53 C_F_POINTER — Convert C into Fortran pointer

Description:

\[ \text{C_F_POINTER}(\text{CPTR}, \text{FPTR}[, \text{SHAPE}]) \] assigns the target of the C pointer \( \text{CPTR} \) to the Fortran pointer \( \text{FPTR} \) and specifies its shape.

Standard: Fortran 2003 and later

Class: Subroutine

Syntax: \[ \text{CALL C_F_POINTER}(\text{CPTR}, \text{FPTR}[, \text{SHAPE}]) \]

Arguments:

- **CPTR**: scalar of the type \( \text{C_PTR} \). It is \text{INTENT(IN)}.
- **FPTR**: pointer interoperable with \( \text{cptr} \). It is \text{INTENT(OUT)}.
- **SHAPE**: (Optional) Rank-one array of type \( \text{INTEGER} \) with \text{INTENT(IN)}. It shall be present if and only if \( \text{fptr} \) is an array. The size must be equal to the rank of \( \text{fpstr} \).

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: Section 9.56 [C_LOC], page 143, Section 9.54 [C_F_PROCPOINTER], page 141

9.54 C_F_PROCPOINTER — Convert C into Fortran procedure pointer

Description:

\[ \text{C_F_PROCPOINTER}(\text{CPTR}, \text{FPTR}) \] Assign the target of the C function pointer \( \text{CPTR} \) to the Fortran procedure pointer \( \text{FPTR} \).

Standard: Fortran 2003 and later

Class: Subroutine

Syntax: \[ \text{CALL C_F_PROCPOINTER}(\text{cptr}, \text{fpstr}) \]

Arguments:

- **CPTR**: scalar of the type \( \text{C_FUNPTR} \). It is \text{INTENT(IN)}.
- **FPTR**: procedure pointer interoperable with \( \text{cptr} \). It is \text{INTENT(OUT)}.

Example:
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 function
  interface
    function getIterFunc() bind(c,name="getIterFunc")
      import :: c_funptr
      type(c_funptr) :: getIterFunc
    end function
  end function
  type(c_funptr) :: cfunptr
  procedure(func), pointer :: myFunc
  cfunptr = getIterFunc()
  call c_f_procpointer(cfunptr, myFunc)
end program main

See also: Section 9.56 [C_LOC], page 143, Section 9.53 [C_F_POINTER], page 141

9.55 C_FUNLOC — Obtain the C address of a procedure

Description:
C_FUNLOC(x) determines the C address of the argument.

Standard: Fortran 2003 and later

Class: Inquiry function

Syntax: RESULT = C_FUNLOC(x)

Arguments:
  x           Interoperable function or pointer to such function.

Return value:
The return value is of type C_FUNPTR and contains the C address of the argument.

Example:

module x
  use iso_c_binding
  implicit none
  contains
    subroutine sub(a) bind(c)
      real(c_float) :: a
      a = sqrt(a)+5.0
    end subroutine sub
  end module x
program main
  use iso_c_binding
  use x
  implicit none
  interface
    subroutine my_routine(p) bind(c,name='myC_func')
import :: c_funptr
  type(c_funptr), intent(in) :: p
end subroutine
end interface

call my_routine(c_funloc(sub))
end program main

See also: Section 9.52 [C_ASSOCIATED], page 140, Section 9.56 [C_LOC], page 143, Section 9.53 [C_F_POINTER], page 141, Section 9.54 [C_F_PROCPOINTER], page 141

9.56 C_LOC — Obtain the C address of an object

Description:
  C_LOC(X) determines the C address of the argument.

Standard: Fortran 2003 and later

Class: Inquiry function

Syntax: RESULT = C_LOC(X)

Arguments:
  X  Shall have either the POINTER or TARGET attribute. It shall
      not be a coindexed object. It shall either be a variable with inter-
      operable type and kind type parameters, or be a scalar, nonpoly-
      morphic variable with no length type parameters.

Return value:
  The return value is of type C_PTR and contains the C address of the argument.

Example:
  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: Section 9.52 [C_ASSOCIATED], page 140, Section 9.55 [C_FUNLOC], page 142, Section 9.53 [C_F_POINTER], page 141, Section 9.54 [C_F_PROCPOINTER], page 141

9.57 C_SIZEOF — Size in bytes of an expression

Description:
  C_SIZEOF(X) calculates the number of bytes of storage the expression X occu-
      -pies.

Standard: Fortran 2008

Class: Inquiry function of the module ISO_C_BINDING
Syntax: \[ N = \text{C\_SIZEOF}(X) \]

Arguments:
\[ X \quad \text{The argument shall be an interoperable data entity.} \]

Return value:
The return value is of type integer and of the system-dependent kind \text{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 \text{POINTER} attribute, the number of bytes of the storage area pointed to is returned. If the argument is of a derived type with \text{POINTER} or \text{ALLOCATABLE} components, the return value does not account for the sizes of the data pointed to by these components.

Example:
```fortran
use iso_c_binding
integer(c_int) :: i
real(c_float) :: r, s(5)
print *, (c_sizeof(s)/c_sizeof(r) == 5)
end
```
The example will print .TRUE. unless you are using a platform where default REAL variables are unusually padded.

See also: Section 9.254 [SIZEOF], page 263, Section 9.261 [STORAGE\_SIZE], page 267

9.58 CEILING — Integer ceiling function

Description:
CEILING(A) returns the least integer greater than or equal to A.

Standard: Fortran 95 and later

Class: Elemental function

Syntax: \[ \text{RESULT} = \text{CEILING}(A [, \text{KIND}]) \]

Arguments:
\[ A \quad \text{The type shall be REAL.} \]
\[ \text{KIND} \quad \text{(Optional) An INTEGER initialization expression indicating the kind parameter of the result.} \]

Return value:
The return value is of type INTEGER(KIND) if \text{KIND} is present and a default-kind INTEGER otherwise.

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: Section 9.109 [FLOOR], page 178, Section 9.207 [NINT], page 236
9.59 CHAR — Character conversion function

Description:  
CHAR(I [, KIND]) returns the character represented by the integer I.

Standard:  Fortran 77 and later  
Class:  Elemental function  
Syntax:  RESULT = CHAR(I [, KIND])  
Arguments:  
I  The type shall be INTEGER.  
KIND  (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

Return value:  
The return value is of type CHARACTER(1)

Example:  
program test_char  
  integer :: i = 74  
  character(1) :: c  
  c = char(i)  
  print *, i, c ! returns 'J'  
end program test_char

Specific names:  
Name  Argument  Return type  Standard  
CHAR(I)  INTEGER I  CHARACTER(LEN=1)  F77 and later

Note:  See Section 9.143 [ICHAR], page 199 for a discussion of converting between numerical values and formatted string representations.

See also:  Section 9.5 [ACHAR], page 109, Section 9.135 [IACHAR], page 194, Section 9.143 [ICHAR], page 199

9.60 CHDIR — Change working directory

Description:  
Change current working directory to a specified path.  
This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

Standard:  GNU extension  
Class:  Subroutine, function  
Syntax:  
CALL CHDIR(NAME [, STATUS])  
STATUS = CHDIR(NAME)  
Arguments:  
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.

**Example:**

```fortran
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:** Section 9.124 [GETCWD], page 188

### 9.61 CHMOD — Change access permissions of files

**Description:**

CHMOD changes the permissions of a file.

This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

**Standard:** GNU extension

**Class:** Subroutine, function

**Syntax:**

```fortran
CALL CHMOD(NAME, MODE[, STATUS])
STATUS = CHMOD(NAME, MODE)
```

**Arguments:**

- **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.

**Return value:**

In either syntax, STATUS is set to 0 on success and nonzero otherwise.

**Example:** CHMOD as subroutine

```fortran
  program chmod_test
    implicit none
    integer :: status
  ```
9.62 **CMPLX** — Complex conversion function

**Description:**

\( \text{CMPLX}(X [, Y [, \text{KIND}]] \) \) 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.

**Standard:** Fortran 77 and later  
**Class:** Elemental function  
**Syntax:** \( \text{RESULT} = \text{CMPLX}(X [, Y [, \text{KIND}]] \) 

**Arguments:**

- \( X \) The type may be \text{INTEGER}, \text{REAL}, or \text{COMPLEX}.
- \( Y \) (Optional; only allowed if \( X \) is not \text{COMPLEX}.) May be \text{INTEGER} or \text{REAL}.
- \( \text{KIND} \) (Optional) An \text{INTEGER} initialization expression indicating the kind parameter of the result.

**Return value:**

The return value is of \text{COMPLEX} type, with a kind equal to \( \text{KIND} \) if it is specified. If \( \text{KIND} \) is not specified, the result is of the default \text{COMPLEX} kind, regardless of the kinds of \( X \) and \( Y \).

**Example:**

```
program test_cmplx
    integer :: i = 42
    real :: x = 3.14
    complex :: z
    z = cmplx(i, x)
    print *, z, cmplx(x)
end program test_cmplx
```

**See also:** Section 9.71 [COMPLEX], page 153

9.63 **CO_BROADCAST** — Copy a value to all images the current set of images

**Description:**

\( \text{CO_BROADCAST} \) copies the value of argument \( A \) on the image with image index \( \text{SOURCE_IMAGE} \) 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.

**Standard:** Technical Specification (TS) 18508 or later

**Class:** Collective subroutine

**Syntax:**

```fortran
CALL CO_BROADCAST(A, SOURCE_IMAGE [, STAT, ERRMSG])
```

**Arguments:**

- **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

**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:** Section 9.64 [CO_MAX], page 148, Section 9.65 [CO_MIN], page 149, Section 9.67 [CO_SUM], page 151, Section 9.66 [CO_REDUCE], page 150

### 9.64 CO_MAX — Maximal value on the current set of images

**Description:**

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.

**Standard:** Technical Specification (TS) 18508 or later

**Class:** Collective subroutine

**Syntax:**

```fortran
CALL CO_MAX(A [, RESULT_IMAGE, STAT, ERRMSG])
```

**Arguments:**

- **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

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: Section 9.65 [CO_MIN], page 149, Section 9.67 [CO_SUM], page 151, Section 9.66 [CO_REDUCE], page 150, Section 9.63 [CO_BROADCAST], page 147

9.65 CO_MIN — Minimal value on the current set of images

Description:

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.

Standard: Technical Specification (TS) 18508 or later

Class: Collective subroutine

Syntax: CALL CO_MIN(A [, RESULT_IMAGE, STAT, ERRMSG])

Arguments:

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

Example:

```fortran
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
```
end if
end program test

See also: Section 9.64 [CO_MAX], page 148, Section 9.67 [CO_SUM], page 151,
Section 9.66 [CO_REDUCE], page 150, Section 9.63 [CO_BROADCAST], page 147

9.66 CO_REDUCE — Reduction of values on the current set of images

Description:

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.

Standard: Technical Specification (TS) 18508 or later

Class: Collective subroutine

Syntax: CALL CO_REDUCE(A, OPERATOR, [, RESULT_IMAGE, STAT, ERRMSG])

Arguments:

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 nonallocat-
able 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 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

Example:
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: Section 9.65 [CO_MIN], page 149, Section 9.64 [CO_MAX], page 148, Section 9.67 [CO_SUM], page 151, Section 9.63 [CO_BROADCAST], page 147

9.67 CO_SUM — Sum of values on the current set of images

Description:
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.

Standard: Technical Specification (TS) 18508 or later

Class: Collective subroutine

Syntax: CALL CO_MIN(A [, RESULT_IMAGE, STAT, ERRMSG])

Arguments:
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

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: Section 9.64 [CO_MAX], page 148, Section 9.65 [CO_MIN], page 149,
Section 9.66 [CO_REDUCE], page 150, Section 9.63 [CO_BROADCAST],
page 147

9.68 COMMAND_ARGUMENT_COUNT — Get number of command line arguments

Description:
COMMAND_ARGUMENT_COUNT returns the number of arguments passed on the com-
mand line when the containing program was invoked.

Standard: Fortran 2003 and later

Class: Inquiry function

Syntax: RESULT = COMMAND_ARGUMENT_COUNT()

Arguments:
None

Return value:
The return value is an INTEGER of default kind.

Example:
program test_command_argument_count
  integer :: count
  count = command_argument_count()
  print *, count
end program test_command_argument_count

See also: Section 9.122 [GET_COMMAND], page 187, Section 9.123
[GET_COMMAND_ARGUMENT], page 187

9.69 COMPILER_OPTIONS — Options passed to the compiler

Description:
COMPILER_OPTIONS returns a string with the options used for compiling.

Standard: Fortran 2008

Class: Inquiry function of the module ISO_FORTRAN_ENV

Syntax: STR = COMPILER_OPTIONS()

Arguments:
None.

Return value:
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.
Example:

```fortran
use iso_fortran_env
print '(4a)', 'This file was compiled by ', &
   compiler_version(), ' using the options ', &
   compiler_options()
end
```

See also: Section 9.70 [COMPILER_VERSION], page 153, Section 10.1 [ISO_FORTRAN_ENV], page 283

### 9.70 COMPILER_VERSION — Compiler version string

**Description:**

COMPILER_VERSION returns a string with the name and the version of the compiler.

**Standard:** Fortran 2008

**Class:** Inquiry function of the module ISO_FORTRAN_ENV

**Syntax:**

```
STR = COMPILER_VERSION()
```

**Arguments:**

None.

**Return value:**

The return value is a default-kind string with system-dependent length. It contains the name of the compiler and its version number.

Example:

```fortran
use iso_fortran_env
print '(4a)', 'This file was compiled by ', &
   compiler_version(), ' using the options ', &
   compiler_options()
end
```

See also: Section 9.69 [COMPILER_OPTIONS], page 152, Section 10.1 [ISO_FORTRAN_ENV], page 283

### 9.71 COMPLEX — Complex conversion function

**Description:**

COMPLEX(X, Y) returns a complex number where X is converted to the real component and Y is converted to the imaginary component.

**Standard:** GNU extension

**Class:** Elemental function

**Syntax:**

```
RESULT = COMPLEX(X, Y)
```

**Arguments:**

- **X** The type may be INTEGER or REAL.
- **Y** The type may be INTEGER or REAL.
Return value:
If \( X \) and \( Y \) are both of \texttt{INTEGER} type, then the return value is of default \texttt{COMPLEX} type.
If \( X \) and \( Y \) are of \texttt{REAL} type, or one is of \texttt{REAL} type and one is of \texttt{INTEGER} type, then the return value is of \texttt{COMPLEX} type with a kind equal to that of the \texttt{REAL} argument with the highest precision.

Example:

```fortran
program test_complex
    integer :: i = 42
    real :: x = 3.14
    print *, complex(i, x)
end program test_complex
```

See also: Section 9.62 [CMPLX], page 147

9.72 CONJG — Complex conjugate function

Description:
\texttt{CONJG(Z)} returns the conjugate of \( Z \). If \( Z \) is \((x, y)\) then the result is \((x, -y)\)

Standard: Fortran 77 and later, has overloads that are GNU extensions

Class: Elemental function

Syntax: \( Z = \text{CONJG}(Z) \)

Arguments:

\( Z \)

The type shall be \texttt{COMPLEX}.

Return value:
The return value is of type \texttt{COMPLEX}.

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>CONJG(Z)</td>
<td>\texttt{COMPLEX} Z</td>
<td>\texttt{COMPLEX}</td>
<td>GNU extension</td>
</tr>
<tr>
<td>DCONJG(Z)</td>
<td>\texttt{COMPLEX(8)} Z</td>
<td>\texttt{COMPLEX(8)}</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

9.73 COS — Cosine function

Description:
\texttt{COS(X)} computes the cosine of \( X \).

Standard: Fortran 77 and later, has overloads that are GNU extensions

Class: Elemental function
Chapter 9: Intrinsic Procedures

Syntax: \[ \text{RESULT} = \cos(X) \]

Arguments:
\[ X \]

The type shall be \texttt{REAL} or \texttt{COMPLEX}.

Return value:
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 \texttt{REAL}, the return value lies in the range \( -1 \leq \cos(x) \leq 1 \).

Example:
```
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>\texttt{REAL(4)} X</td>
<td>\texttt{REAL(4)}</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>( \text{DCOS}(X) )</td>
<td>\texttt{REAL(8)} X</td>
<td>\texttt{REAL(8)}</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>( \text{CCOS}(X) )</td>
<td>\texttt{COMPLEX(4)} X</td>
<td>\texttt{COMPLEX(4)}</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>( \text{ZCOS}(X) )</td>
<td>\texttt{COMPLEX(8)} X</td>
<td>\texttt{COMPLEX(8)}</td>
<td>GNU extension</td>
</tr>
<tr>
<td>( \text{CDCOS}(X) )</td>
<td>\texttt{COMPLEX(8)} X</td>
<td>\texttt{COMPLEX(8)}</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

See also:
Inverse function: Section 9.6 \[\text{ACOS}\], page 110
Degrees function: Section 9.74 \[\text{COSD}\], page 155

9.74 \text{COSD} — Cosine function, degrees

Description:
\( \text{COSD}(X) \) computes the cosine of \( X \) in degrees.

This function is for compatibility only and should be avoided in favor of standard constructs wherever possible.

Standard: GNU Extension, enabled with `\text{-fdec-math}`.

Class: Elemental function

Syntax: \[ \text{RESULT} = \text{COSD}(X) \]

Arguments:
\[ X \]

The type shall be \texttt{REAL} or \texttt{COMPLEX}.

Return value:
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 \texttt{REAL}, the return value lies in the range \( -1 \leq \cosd(x) \leq 1 \).

Example:
```
program test_cosp
  real :: x = 0.0
  x = cosd(x)
end program test_cosp
```
Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>COSD(X)</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
<td>GNU Extension</td>
</tr>
<tr>
<td>DCSOD(X)</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
<td>GNU Extension</td>
</tr>
<tr>
<td>CCOSD(X)</td>
<td>COMPLEX(4) X</td>
<td>COMPLEX(4) X</td>
<td>GNU Extension</td>
</tr>
<tr>
<td>ZCOSD(X)</td>
<td>COMPLEX(8) X</td>
<td>COMPLEX(8) X</td>
<td>GNU extension</td>
</tr>
<tr>
<td>DCOSD(X)</td>
<td>COMPLEX(8) X</td>
<td>COMPLEX(8) X</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

See also: Inverse function: Section 9.7 [ACOSD], page 111

Radians function: Section 9.73 [COS], page 154

### 9.75 COSH — Hyperbolic cosine function

**Description:**
COSH(X) computes the hyperbolic cosine of X.

**Standard:** Fortran 77 and later, for a complex argument Fortran 2008 or later

**Class:** Elemental function

**Syntax:**

```fortran
X = COSH(X)
```

**Arguments:**

| X          | The type shall be REAL or COMPLEX. |

**Return value:**

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, cosh(x) ≥ 1.

**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>DCSOH(X)</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

See also: Inverse function: Section 9.8 [ACOSH], page 111

### 9.76 COTAN — Cotangent function

**Description:**
COTAN(X) computes the cotangent of X. Equivalent to COS(x) divided by SIN(x), or 1 / TAN(x).

This function is for compatibility only and should be avoided in favor of standard constructs wherever possible.

**Standard:** GNU Extension, enabled with `-fdec-math`.

**Class:** Elemental function
Chapter 9: Intrinsic Procedures 157

Syntax: RESULT = COTAN(X)

Arguments:
X The type shall be REAL or COMPLEX.

Return value:
The return value has same type and kind as X, and its value is in radians.

Example:

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>X</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DCOTAN(X)</td>
<td>REAL(8)</td>
<td>X</td>
<td>REAL(8)</td>
</tr>
</tbody>
</table>

See also: Converse function: Section 9.266 [TAN], page 271 Degrees function: Section 9.77 [COTAND], page 157

9.77 COTAND — Cotangent function, degrees

Description:
COTAND(X) computes the cotangent of X in degrees. Equivalent to \( \cos(x) \) divided by \( \sin(x) \), or \( 1 / \tan(x) \).

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: RESULT = COTAND(X)

Arguments:
X The type shall be REAL or COMPLEX.

Return value:
The return value has same type and kind as X, and its value is in degrees.

Example:

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)</td>
<td>X</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DCOTAND(X)</td>
<td>REAL(8)</td>
<td>X</td>
<td>REAL(8)</td>
</tr>
</tbody>
</table>

See also: Converse function: Section 9.267 [TAND], page 271 Radians function: Section 9.76 [COTAN], page 156
9.78 COUNT — Count function

Description:
Counts the number of .TRUE. elements in a logical MASK, or, if the DIM argument is supplied, counts the number of elements along each row of the array in the DIM direction. If the array has zero size, or all of the elements of MASK are .FALSE., then the result is 0.

Standard: Fortran 95 and later, with KIND argument Fortran 2003 and later

Class: Transformational function

Syntax: \[ \text{RESULT} = \text{COUNT}(	ext{MASK} [, \text{DIM}, \text{KIND}]) \]

Arguments:
- \text{MASK}: The type shall be \text{LOGICAL}.
- \text{DIM} (Optional): The type shall be \text{INTEGER}.
- \text{KIND} (Optional): An \text{INTEGER} initialization expression indicating the kind parameter of the result.

Return value:
The return value is of type \text{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.

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 '(',a(1,:),')'
  print '(',a(2,:),')'
  print *
  print '(',b(1,:),')'
  print '(',b(2,:),')'
  print *
  mask = a.ne.b
  print '(',mask(1,:),')'
  print '(',mask(2,:),')'
  print *
  print '(',count(mask),')'
  print *
  print '(',count(mask, 1),')'
  print *
  print '(',count(mask, 2),')'
end program test_count
```

9.79 CPU_TIME — CPU elapsed time in seconds

Description:
Returns a \text{REAL} value representing the elapsed CPU time in seconds. This is useful for testing segments of code to determine execution time.
If a time source is available, time will be reported with microsecond resolution. If no time source is available, \texttt{TIME} is set to \texttt{-1.0}.

Note that \texttt{TIME} may contain a system dependent, arbitrary offset and may not start with \texttt{0.0}. For \texttt{CPU\_TIME}, the absolute value is meaningless, only differences between subsequent calls to this subroutine, as shown in the example below, should be used.

\textit{Standard:} Fortran 95 and later

\textit{Class:} Subroutine

\textit{Syntax:} \texttt{CALL CPU\_TIME(TIME)}

\textit{Arguments:}

\begin{itemize}
  \item \texttt{TIME} \hspace{1cm} The type shall be \texttt{REAL} with \texttt{INTENT(OUT)}.
\end{itemize}

\textit{Return value:}\n
None

\textit{Example:}

\begin{verbatim}
program test_cpu_time
  real :: start, finish
  call cpu_time(start)
  ! put code to test here
  call cpu_time(finish)
  print '("Time = ",f6.3," seconds.")',finish-start
end program test_cpu_time
\end{verbatim}

See also: Section 9.265 [SYSTEM\_CLOCK], page 270, Section 9.82 [DATE\_AND\_TIME], page 161

\section*{9.80 CSHIFT — Circular shift elements of an array}

\textit{Description:}

\texttt{CSHIFT(ARRAY, SHIFT [, DIM])} performs a circular shift on elements of \texttt{ARRAY} along the dimension of \texttt{DIM}. If \texttt{DIM} is omitted it is taken to be \texttt{1}. \texttt{DIM} is a scalar of type \texttt{INTEGER} in the range of \texttt{1 \leq DIM \leq n}) where \texttt{n} is the rank of \texttt{ARRAY}. If the rank of \texttt{ARRAY} is one, then all elements of \texttt{ARRAY} are shifted by \texttt{SHIFT} places. If rank is greater than one, then all complete rank one sections of \texttt{ARRAY} along the given dimension are shifted. Elements shifted out one end of each rank one section are shifted back in the other end.

\textit{Standard:} Fortran 95 and later

\textit{Class:} Transformational function

\textit{Syntax:} \texttt{RESULT = CSHIFT(ARRAY, SHIFT [, DIM])}

\textit{Arguments:}

\begin{itemize}
  \item \texttt{ARRAY} \hspace{1cm} Shall be an array of any type.
  \item \texttt{SHIFT} \hspace{1cm} The type shall be \texttt{INTEGER}.
  \item \texttt{DIM} \hspace{1cm} The type shall be \texttt{INTEGER}.
\end{itemize}

\textit{Return value:}\n
Returns an array of same type and rank as the \texttt{ARRAY} argument.
Example:

```fortran
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
```

9.81 CTIME — Convert a time into a string

**Description:**
CTIME converts a system time value, such as returned by Section 9.271 [TIME8], page 274, to a string. The output will be of the form ‘Sat Aug 19 18:13:14 1995’.

This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

**Standard:** GNU extension

**Class:** Subroutine, function

**Syntax:**
```
CALL CTIME(TIME, RESULT).
RESULT = CTIME(TIME).
```

**Arguments:**

- **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.

**Return value:**
The converted date and time as a string.

**Example:**

```fortran
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:** Section 9.82 [DATE_AND_TIME], page 161, Section 9.131 [GMTIME], page 192, Section 9.184 [LTIME], page 222, Section 9.270 [TIME], page 273, Section 9.271 [TIME8], page 274
9.82 DATE_AND_TIME — Date and time subroutine

Description:

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 re-
spect to Coordinated Universal Time (UTC). Unavailable time and date pa-
rameters return blanks.

VALUES is INTENT(OUT) and provides the following:

<table>
<thead>
<tr>
<th>VALUE(1)</th>
<th>VALUE(2)</th>
<th>VALUE(3)</th>
<th>VALUE(4)</th>
<th>VALUE(5)</th>
<th>VALUE(6)</th>
<th>VALUE(7)</th>
<th>VALUE(8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>The year</td>
<td>The month</td>
<td>The day of the month</td>
<td>Time difference with UTC in minutes</td>
<td>The hour of the day</td>
<td>The minutes of the hour</td>
<td>The seconds of the minute</td>
<td>The milliseconds of the second</td>
</tr>
</tbody>
</table>

Standard: Fortran 95 and later

Class: Subroutine

Syntax: CALL DATE_AND_TIME([DATE, TIME, ZONE, VALUES])

Arguments:

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).

Return value: None

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: Section 9.79 [CPU_TIME], page 158, Section 9.265 [SYSTEM_CLOCK], page 270

9.83 DBLE — Double conversion function

Description:
DBLE(A) Converts A to double precision real type.

Standard: Fortran 77 and later

Class: Elemental function

Syntax: RESULT = DBLE(A)

Arguments:
A The type shall be INTEGER, REAL, or COMPLEX.

Return value: The return value is of type double precision real.

Example:

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: Section 9.229 [REAL], page 248

9.84 DCMPLX — Double complex conversion function

Description:
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.

Standard: GNU extension

Class: Elemental function

Syntax: RESULT = DCMPLX(X [, Y])

Arguments:
X The type may be INTEGER, REAL, or COMPLEX.
Y (Optional if X is not COMPLEX.) May be INTEGER or REAL.

Return value: The return value is of type COMPLEX(8)

Example:

program test_dcmplx
  integer :: i = 42
  real :: x = 3.14
  complex :: z
Chapter 9: Intrinsic Procedures

9.85 DIGITS — Significant binary digits function

**Description:**
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.

**Standard:** Fortran 95 and later

**Class:** Inquiry function

**Syntax:**
RESULT = DIGITS(X)

**Arguments:**

X                      The type may be INTEGER or REAL.

**Return value:**

The return value is of type INTEGER.

**Example:**

```fortran
program test_digits
  integer :: i = 12345
  real :: x = 3.143
  real(8) :: y = 2.33
  print *, digits(i)
  print *, digits(x)
  print *, digits(y)
end program test_digits
```

9.86 DIM — Positive difference

**Description:**

DIM(X, Y) returns the difference X-Y if the result is positive; otherwise returns zero.

**Standard:** Fortran 77 and later

**Class:** Elemental function

**Syntax:**
RESULT = DIM(X, Y)

**Arguments:**

X                      The type shall be INTEGER or REAL

Y                      The type shall be the same type and kind as X.

**Return value:**

The return value is of type INTEGER or REAL.

**Example:**

```fortran
z = cmplx(i, x)
p = print *, dcmplx(i)
p = print *, dcmplx(x)
p = print *, dcmplx(z)
p = print *, dcmplx(x,i)
end program test_dcmplx
```
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>

9.87 DOT_PRODUCT — Dot product function

Description:

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).

Standard: Fortran 95 and later

Class: Transformational function

Syntax: RESULT = DOT_PRODUCT(VECTOR_A, VECTOR_B)

Arguments:

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.

Return value:

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..

Example:

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
9.88 DPROD — Double product function

Description:
DPROD(X, Y) returns the product X*Y.

Standard: Fortran 77 and later
Class: Elemental function
Syntax: RESULT = DPROD(X, Y)

Arguments:
X The type shall be REAL.
Y The type shall be REAL.

Return value:
The return value is of type REAL(8).

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)</td>
<td>REAL(8)</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

9.89 DREAL — Double real part function

Description:
DREAL(Z) returns the real part of complex variable Z.

Standard: GNU extension
Class: Elemental function
Syntax: RESULT = DREAL(A)

Arguments:
A The type shall be COMPLEX(8).

Return value:
The return value is of type REAL(8).

Example:
```fortran
program test_dreal
   complex(8) :: z = (1.3_8, 7.2_8)
   print *, dreal(z)
end program test_dreal
```

See also: Section 9.11 [AIMAG], page 113
9.90 DSHIFTL — Combined left shift

Description:

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.

Standard: Fortran 2008 and later

Class: Elemental function

Syntax: RESULT = DSHIFTL(I, J, SHIFT)

Arguments:

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).

Return value:

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.

See also: Section 9.91 [DSHIFTR], page 166

9.91 DSHIFTR — Combined right shift

Description:

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.

Standard: Fortran 2008 and later

Class: Elemental function

Syntax: RESULT = DSHIFTR(I, J, SHIFT)

Arguments:

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).

Return value:

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.
9.92 DTIME — Execution time subroutine (or function)

Description:

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).

Subsequent invocations of DTIME return values accumulated since the previous invocation.

On some systems, the underlying timings are represented using types with sufficiently small limits that overflows (wrap around) are possible, such as 32-bit types. Therefore, the values returned by this intrinsic might be, or become, negative, or numerically less than previous values, during a single run of the compiled program.

Please note, that this implementation is thread safe if used within OpenMP directives, i.e., its state will be consistent while called from multiple threads. However, if DTIME is called from multiple threads, the result is still the time since the last invocation. This may not give the intended results. If possible, use CPU_TIME instead.

This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

VALUES and TIME are INTENT(OUT) and provide the following:

VALUES(1): User time in seconds.
VALUES(2): System time in seconds.
TIME: Run time since start in seconds.

Standard: GNU extension

Class: Subroutine, function

Syntax:

CALL DTIME(VALUES, TIME).
TIME = DTIME(VALUES), (not recommended).

Arguments:

VALUES The type shall be REAL(4), DIMENSION(2).
TIME The type shall be REAL(4).

Return value:

Elapsed time in seconds since the last invocation or since the start of program execution if not called before.

Example:

```fortran
program test_dtime
  integer(8) :: i, j
  real, dimension(2) :: tarray
  real :: result
  call dtime(tarray, result)
end program test_dtime
```
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: Section 9.79 [CPU_TIME], page 158

9.93 EOSHIFT — End-off shift elements of an array

Description:

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 \( 1 \leq \text{DIM} \leq n \) 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.

<table>
<thead>
<tr>
<th>Array Type</th>
<th>Boundary Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric</td>
<td>0 of the type and kind of ARRAY.</td>
</tr>
<tr>
<td>Logical</td>
<td>.FALSE.</td>
</tr>
<tr>
<td>Character(len)</td>
<td>len blanks.</td>
</tr>
</tbody>
</table>

Standard: Fortran 95 and later

Class: Transformational function

Syntax: \( \text{RESULT} = \text{EOSHIFT}(\text{ARRAY, SHIFT [, BOUNDARY, DIM]} \) \)

Arguments:

- ARRAY May be any type, not scalar.
- SHIFT The type shall be INTEGER.
- BOUNDARY Same type as ARRAY.
- DIM The type shall be INTEGER.

Return value:

Returns an array of same type and rank as the ARRAY argument.

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,:)
end program test_eoshift
```
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

9.94 EPSILON — Epsilon function

Description:
EPSILON(X) returns the smallest number $E$ of the same kind as $X$ such that $1 + E > 1$.

Standard: Fortran 95 and later
Class: Inquiry function
Syntax: RESULT = EPSILON(X)
Arguments:
X The type shall be REAL.
Return value:
The return value is of same type as the argument.
Example:

program test_epsilon
  real :: x = 3.143
  real(8) :: y = 2.33
  print *, EPSILON(x)
  print *, EPSILON(y)
end program test_epsilon

9.95 ERF — Error function

Description:
ERF(X) computes the error function of $X$.

Standard: Fortran 2008 and later
Class: Elemental function
Syntax: RESULT = ERF(X)
Arguments:
X The type shall be REAL.
Return value:
The return value is of type REAL, of the same kind as $X$ and lies in the range $-1 \leq erf(x) \leq 1$.
Example:

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) X</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

9.96 ERFC — Error function

Description:
ERFC(X) computes the complementary error function of X.

Standard: Fortran 2008 and later
Class: Elemental function
Syntax: RESULT = ERFC(X)

Arguments:
X The type shall be REAL.

Return value:
The return value is of type REAL and of the same kind as X. It lies in the range $0 \leq \text{erfc}(x) \leq 2$.

Example:

```fortran
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) X</td>
<td>REAL(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

9.97 ERFC_SCALED — Error function

Description:
ERFC_SCALED(X) computes the exponentially-scaled complementary error function of X.

Standard: Fortran 2008 and later
Class: Elemental function
Syntax: RESULT = ERFC_SCALED(X)

Arguments:
X The type shall be REAL.

Return value:
The return value is of type REAL and of the same kind as X.

Example:

```fortran
program test_erfc_scaled
    real(8) :: x = 0.17_8
    x = erfc_scaled(x)
end program test_erfc_scaled
```
9.98 ETIME — Execution time subroutine (or function)

Description:
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).

On some systems, the underlying timings are represented using types with sufficiently small limits that overflows (wrap around) are possible, such as 32-bit types. Therefore, the values returned by this intrinsic might be, or become, negative, or numerically less than previous values, during a single run of the compiled program.

This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

VALUES and TIME are INTENT(OUT) and provide the following:
VALUES(1): User time in seconds.
VALUES(2): System time in seconds.
TIME: Run time since start in seconds.

Standard: GNU extension
Class: Subroutine, function
Syntax:
CALL ETIME(VALUES, TIME).
TIME = ETIME(VALUES), (not recommended).

Arguments:
VALUES The type shall be REAL(4), DIMENSION(2).
TIME The type shall be REAL(4).

Return value:
Elapsed time in seconds since the start of program execution.

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: Section 9.79 [CPU_TIME], page 158
9.99 EVENT_QUERY — Query whether a coarray event has occurred

Description:

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.

Standard: TS 18508 or later

Class: subroutine

Syntax: CALL EVENT_QUERY (EVENT, COUNT [, STAT])

Arguments:

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.

Example:

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

9.100 EXECUTE_COMMAND_LINE — Execute a shell command

Description:

EXECUTE_COMMAND_LINE runs a shell command, synchronously or asynchronously.

The COMMAND argument is passed to the shell and executed, using the C library’s system call. (The shell is sh on Unix systems, and cmd.exe on Windows.) If WAIT is present and has the value false, the execution of the command is asynchronous if the system supports it; otherwise, the command is executed synchronously.

The three last arguments allow the user to get status information. After synchronous execution, EXITSTAT contains the integer exit code of the command, as returned by system. CMDSTAT is set to zero if the command line was executed (whatever its exit status was). CMDMSG is assigned an error message if an error has occurred.
Note that the `system` function need not be thread-safe. It is the responsibility of the user to ensure that `system` is not called concurrently.

**Standard:** Fortran 2008 and later  
**Class:** Subroutine  
**Syntax:** `CALL EXECUTE_COMMAND_LINE(COMMAND [, WAIT, EXITSTAT, CMDSTAT, CMDMSG ])`

**Arguments:**
- `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 an `CHARACTER` scalar of the default kind.

**Example:**

```fortran
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:** Section 9.264 [SYSTEM], page 269

### 9.101 EXIT — Exit the program with status.

**Description:**

`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.

**Standard:** GNU extension  
**Class:** Subroutine  
**Syntax:** `CALL EXIT([STATUS])`

**Arguments:**
- `STATUS` Shall be an `INTEGER` of the default kind.

**Return value:**

`STATUS` is passed to the parent process on exit.
Example:

```fortran
program test_exit
  integer :: STATUS = 0
  print *, 'This program is going to exit.'
  call EXIT(STATUS)
end program test_exit
```

See also: Section 9.2 [ABORT], page 107, Section 9.163 [KILL], page 211

9.102 EXP — Exponential function

**Description:**

\( \text{EXP}(X) \) computes the base \( e \) exponential of \( X \).

**Standard:** Fortran 77 and later, has overloads that are GNU extensions

**Class:** Elemental function

**Syntax:**

\[
\text{RESULT} = \text{EXP}(X)
\]

**Arguments:**

\( X \)

The type shall be \( \text{REAL} \) or \( \text{COMPLEX} \).

**Return value:**

The return value has same type and kind as \( X \).

**Example:**

```fortran
program test_exp
  real :: x = 1.0
  x = exp(x)
end program test_exp
```

**Specific names:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument Type</th>
<th>Return Type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXP(X)</td>
<td>REAL(4) X</td>
<td>REAL(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>DEXP(X)</td>
<td>REAL(8) X</td>
<td>REAL(8)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>CEXP(X)</td>
<td>COMPLEX(4) X</td>
<td>COMPLEX(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>ZEXP(X)</td>
<td>COMPLEX(8) X</td>
<td>COMPLEX(8)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>CDEXP(X)</td>
<td>COMPLEX(8) X</td>
<td>COMPLEX(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

9.103 EXPONENT — Exponent function

**Description:**

\( \text{EXPONENT}(X) \) returns the value of the exponent part of \( X \). If \( X \) is zero the value returned is zero.

**Standard:** Fortran 95 and later

**Class:** Elemental function

**Syntax:**

\[
\text{RESULT} = \text{EXPONENT}(X)
\]

**Arguments:**

\( X \)

The type shall be \( \text{REAL} \).

**Return value:**

The return value is of type default \( \text{INTEGER} \).
Example:

```fortran
program test_exponent
  real :: x = 1.0
  integer :: i
  i = exponent(x)
  print *, i
  print *, exponent(0.0)
end program test_exponent
```

9.104 EXTENDS_TYPE_OF — Query dynamic type for extension

*Description:* Query dynamic type for extension.

*Standard:* Fortran 2003 and later

*Class:* Inquiry function

*Syntax:* `RESULT = EXTENDS_TYPE_OF(A, MOLD)`

*Arguments:*

- **A** Shall be an object of extensible declared type or unlimited polymorphic.
- **MOLD** Shall be an object of extensible declared type or unlimited polymorphic.

*Return value:* 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`.

*See also:* Section 9.235 [SAME_TYPE_AS], page 252

9.105 FDATE — Get the current time as a string

*Description:* `FDATE(DATE)` returns the current date (using the same format as Section 9.81 [CTIME], page 160) in `DATE`. It is equivalent to `CALL CTIME(DATE, TIME())`.

This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

*Standard:* GNU extension

*Class:* Subroutine, function

*Syntax:*

```fortran
CALL FDATE(DATE).
DATE = FDATE().
```

*Arguments:*

- **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.
Return value:
The current date and time as a string.

Example:

```
program test_fdate
  integer(8) :: i, j
  character(len=30) :: date
  call fdate(date)
  print *, 'Program started on ', date
  do i = 1, 10000000 ! Just a delay
    j = i * i - i
  end do
  call fdate(date)
  print *, 'Program ended on ', date
end program test_fdate
```

See also: Section 9.82 [DATE_AND_TIME], page 161, Section 9.81 [CTIME], page 160

9.106 FGET — Read a single character in stream mode from stdin

Description:
Read 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.

This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

Note that the FGET intrinsic is provided for backwards compatibility with g77.

GNU Fortran provides the Fortran 2003 Stream facility. Programmers should consider the use of new stream IO feature in new code for future portability.

See also Section 4.1 [Fortran 2003 status], page 37.

Standard: GNU extension

Class: Subroutine, function

Syntax:
```
CALL FGET(C [, STATUS])
STATUS = FGET(C)
```

Arguments:
- **C**
  - (Optional) status flag of type INTEGER. Returns 0 on success, -1 on end-of-file, and a system specific positive error code otherwise.

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: Section 9.107 [FGETC], page 177, Section 9.112 [FPUT], page 180, Section 9.113 [FPUTC], page 181

9.107 FGETC — Read a single character in stream mode

Description:
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.
This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.
Note that the FGET intrinsic is provided for backwards compatibility with g77.
GNU Fortran provides the Fortran 2003 Stream facility. Programmers should consider the use of new stream IO feature in new code for future portability.
See also Section 4.1 [Fortran 2003 status], page 37.

Standard: GNU extension
Class: Subroutine, function
Syntax:
CALL FGETC(UNIT, C [, STATUS])
STATUS = FGETC(UNIT, C)

Arguments:
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.

Example:
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: Section 9.106 [FGET], page 176, Section 9.112 [FPUT], page 180, Section 9.113 [FPUTC], page 181
9.108 FINDLOC — Search an array for a value

Description:
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 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 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.

Standard: Fortran 2008 and later.
Class: Transformational function
Syntax:
RESULT = FINDLOC(ARRAY, VALUE, DIM [, MASK] [,KIND] [,BACK])
RESULT = FINDLOC(ARRAY, VALUE, [, MASK] [,KIND] [,BACK])

Arguments:
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.
KIND (Optional) An INTEGER initialization expression indicating the kind parameter of the result.
BACK (Optional) A scalar of type LOGICAL.

Return value:
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.

See also: Section 9.191 [MAXLOC], page 226, Section 9.199 [MINLOC], page 230

9.109 FLOOR — Integer floor function

Description:
FLOOR(A) returns the greatest integer less than or equal to X.

Standard: Fortran 95 and later
Class: Elemental function
Syntax:  \[
\text{RESULT} = \text{FLOOR}(A [, \text{KIND}])
\]

Arguments:

- \(A\)  
The type shall be \text{REAL}.
- \text{KIND}  
(Optional) An \text{INTEGER} initialization expression indicating the kind parameter of the result.

Return value:

The return value is of type \text{INTEGER(KIND)} if \text{KIND} is present and of default-kind \text{INTEGER} otherwise.

Example:

```fortran
program test_floor
    real :: x = 63.29
    real :: y = -63.59
    print *, floor(x) ! returns 63
    print *, floor(y) ! returns -64
end program test_floor
```

See also:  Section 9.58 [CEILING], page 144, Section 9.207 [NINT], page 236

9.110 \textbf{FLUSH — Flush I/O unit(s)}

Description:

Flushes Fortran unit(s) currently open for output. Without the optional argument, all units are flushed, otherwise just the unit specified.

Standard:  GNU extension

Class:  Subroutine

Syntax:  CALL FLUSH(UNIT)

Arguments:

- \text{UNIT}  
(Optional) The type shall be \text{INTEGER}.

Note:  Beginning with the Fortran 2003 standard, there is a \text{FLUSH} statement that should be preferred over the \text{FLUSH} intrinsic.

The \text{FLUSH} intrinsic and the Fortran 2003 \text{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 \text{fsync} function, with the POSIX file descriptor of the I/O unit as argument (retrieved with GNU intrinsic \text{FNUM}). The following example shows how:

```fortran
! 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"

9.111 FNUM — File number function

Description:
FNUM(UNIT) returns the POSIX file descriptor number corresponding to the open Fortran I/O unit UNIT.

Standard: GNU extension
Class: Function
Syntax: RESULT = FNUM(UNIT)
Arguments:
UNIT The type shall be INTEGER.

Return value:
The return value is of type INTEGER

Example:

program test_fnum
integer :: i
open (unit=10, status = "scratch")
i = fnum(10)
print *, i
close (10)
end program test_fnum

9.112 FPUT — Write a single character in stream mode to stdout

Description:
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.
This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.
Note that the FGET intrinsic is provided for backwards compatibility with g77. GNU Fortran provides the Fortran 2003 Stream facility. Programmers should
consider the use of new stream IO feature in new code for future portability.
See also Section 4.1 [Fortran 2003 status], page 37.

*Standard:* GNU extension

*Class:* Subroutine, function

*Syntax:*

```fortran
CALL FPUT(C [, STATUS])
STATUS = FPUT(C)
```

*Arguments:*

- **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.

*Example:*

```fortran
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:* Section 9.113 [FPUTC], page 181, Section 9.106 [FGET], page 176, Section 9.107 [FGETC], page 177

### 9.113 **FPUTC** — Write a single character in stream mode

*Description:*

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.

This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

Note that the FGET intrinsic is provided for backwards compatibility with g77. GNU Fortran provides the Fortran 2003 Stream facility. Programmers should consider the use of new stream IO feature in new code for future portability.

See also Section 4.1 [Fortran 2003 status], page 37.

*Standard:* GNU extension

*Class:* Subroutine, function

*Syntax:*

```fortran
CALL FPUTC(UNIT, C [, STATUS])
STATUS = FPUTC(UNIT, C)
```

*Arguments:*

- **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.

Example:

```fortran
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: Section 9.112 [FPUT], page 180, Section 9.106 [FGET], page 176, Section 9.107 [FGETC], page 177

### 9.114 FRACTION — Fractional part of the model representation

**Description:**

FRACTION(X) returns the fractional part of the model representation of X.

**Standard:** Fortran 95 and later

**Class:** Elemental function

**Syntax:**

```fortran
Y = FRACTION(X)
```

**Arguments:**

- **X**
  - The type of the argument shall be a REAL.

**Return value:**

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 \( X \times \text{RADIX}(X)^{-\text{EXPONENT}(X)} \).

**Example:**

```fortran
program test_fraction
    real :: x
    x = 178.1387e-4
    print *, fraction(x), x * radix(x)**(-exponent(x))
end program test_fraction
```

### 9.115 FREE — Frees memory

**Description:**

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.

**Standard:** GNU extension
Class: Subroutine

Syntax: CALL FREE(PTR)

Arguments:
  PTR  The type shall be INTEGER. It represents the location of the memory that should be de-allocated.

Return value:
None

Example: See MALLOC for an example.

See also: Section 9.185 [MALLOC], page 223

9.116 FSEEK — Low level file positioning subroutine

Description:
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.

This intrinsic routine is not fully backwards compatible with g77. In g77, the FSEEK takes a statement label instead of a STATUS variable. If FSEEK is used in old code, change

CALL FSEEK(UNIT, OFFSET, WHENCE, *label)

to

INTEGER :: status
CALL FSEEK(UNIT, OFFSET, WHENCE, status)
IF (status /= 0) GOTO label

Please note that GNU Fortran provides the Fortran 2003 Stream facility. Programmers should consider the use of new stream IO feature in new code for future portability. See also Section 4.1 [Fortran 2003 status], page 37.

Standard: GNU extension

Class: Subroutine

Syntax: CALL FSEEK(UNIT, OFFSET, WHENCE[, STATUS])

Arguments:
  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).

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:  Section 9.118 [FTELL], page 184

9.117 FSTAT — Get file status

Description:
FSTAT is identical to Section 9.260 [STAT], page 266, except that information
about an already opened file is obtained.
The elements in VALUES are the same as described by Section 9.260 [STAT],
page 266.
This intrinsic is provided in both subroutine and function forms; however, only
one form can be used in any given program unit.

Standard: GNU extension
Class: Subroutine, function
Syntax:
CALL FSTAT(UNIT, VALUES [, STATUS])
STATUS = FSTAT(UNIT, VALUES)

Arguments:
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 suc-
cess and a system specific error code otherwise.

Example: See Section 9.260 [STAT], page 266 for an example.
See also: To stat a link: Section 9.183 [LSTAT], page 221, to stat a file: Section 9.260
[STAT], page 266

9.118 FTELL — Current stream position

Description:
Retrieves the current position within an open file.
This intrinsic is provided in both subroutine and function forms; however, only
one form can be used in any given program unit.
**Chapter 9: Intrinsic Procedures**

*Standard:* GNU extension

*Class:* Subroutine, function

*Syntax:*

```fortran
CALL FTELL(UNIT, OFFSET)
OFFSET = FTELL(UNIT)
```

*Arguments:*

- `OFFSET` Shall of type INTEGER.
- `UNIT` Shall of type INTEGER.

*Return value:*

In either syntax, `OFFSET` is set to the current offset of unit number `UNIT`, or to −1 if the unit is not currently open.

*Example:*

```fortran
PROGRAM test_ftell
  INTEGER :: i
  OPEN(10, FILE="temp.dat")
  CALL ftell(10, i)
  WRITE(*,*) i
END PROGRAM
```

*See also:* Section 9.116 [FSEEK], page 183

### 9.119 GAMMA — Gamma function

*Description:*

`GAMMA(X)` computes Gamma (Γ) of `X`. For positive, integer values of `X` the Gamma function simplifies to the factorial function \( \Gamma(x) = (x - 1)! \).

\[
\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} \, dt
\]

*Standard:* Fortran 2008 and later

*Class:* Elemental function

*Syntax:* `X = GAMMA(X)`

*Arguments:*

- `X` Shall be of type REAL and neither zero nor a negative integer.

*Return value:*

The return value is of type REAL of the same kind as `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>GAMMA(X)</td>
<td>REAL(4) X</td>
<td>REAL(4)</td>
<td>GNU Extension</td>
</tr>
<tr>
<td>DGAMMA(X)</td>
<td>REAL(8) X</td>
<td>REAL(8)</td>
<td>GNU Extension</td>
</tr>
</tbody>
</table>
See also: Logarithm of the Gamma function: Section 9.179 [LOG_GAMMA], page 219

9.120 GERROR — Get last system error message

Description:
Returns the system error message corresponding to the last system error. This resembles the functionality of strerror(3) in C.

Standard: GNU extension
Class: Subroutine
Syntax: CALL GERROR(RESULT)
Arguments:
RESULT Shall of type CHARACTER and of default

Example:

PROGRAM test_gerror
   CHARACTER(len=100) :: msg
   CALL gerror(msg)
   WRITE(*,*) msg
END PROGRAM

See also: Section 9.146 [IERRNO], page 201, Section 9.215 [PERROR], page 241

9.121 GETARG — Get command line arguments

Description:
Retrieve the POS-th argument that was passed on the command line when the containing program was invoked.

This intrinsic routine is provided for backwards compatibility with GNU Fortran 77. In new code, programmers should consider the use of the Section 9.123 [GET_COMMAND_ARGUMENT], page 187 intrinsic defined by the Fortran 2003 standard.

Standard: GNU extension
Class: Subroutine
Syntax: CALL GETARG(POS, VALUE)
Arguments:
POS Shall be of type INTEGER and not wider than the default integer kind; POS ≥ 0
VALUE Shall be of type CHARACTER and of default kind.

Return value:
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).
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: Section 9.139 [IARGC], page 197
Fortran 2003 functions and subroutines: Section 9.122 [GET_COMMAND], page 187, Section 9.123 [GET_COMMAND_ARGUMENT], page 187, Section 9.68 [COMMAND_ARGUMENT_COUNT], page 152

### 9.122 GET_COMMAND — Get the entire command line

**Description:**
Retrieve the entire command line that was used to invoke the program.

**Standard:** Fortran 2003 and later

**Class:** Subroutine

**Syntax:**
```fortran
CALL GET_COMMAND([COMMAND, LENGTH, STATUS])
```

**Arguments:**
- `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.

**Return value:**
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.

**Example:**

```fortran
PROGRAM test_get_command
   CHARACTER(len=255) :: cmd
   CALL get_command(cmd)
   WRITE (*,*) TRIM(cmd)
END PROGRAM
```

See also: Section 9.123 [GET_COMMAND_ARGUMENT], page 187, Section 9.68 [COMMAND_ARGUMENT_COUNT], page 152

### 9.123 GET_COMMAND_ARGUMENT — Get command line arguments

**Description:**
Retrieve the `NUMBER`-th argument that was passed on the command line when the containing program was invoked.
Class: Subroutine

Syntax: CALL GET_COMMAND_ARGUMENT(NUMBER [, VALUE, LENGTH, STATUS])

Arguments:

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.

Return value:

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.

Example:

```fortran
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: Section 9.122 [GET_COMMAND], page 187, Section 9.68 [COMMAND_ARGUMENT_COUNT], page 152

9.124 GETCWD — Get current working directory

Description:

Get current working directory.

This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

Standard: GNU extension
Class: Subroutine, function

Syntax:

```fortran
CALL GETCWD(C [, STATUS])
STATUS = GETCWD(C)
```

Arguments:

- **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.

Example:

```fortran
PROGRAM test_getcwd
    CHARACTER(len=255) :: cwd
    CALL getcwd(cwd)
    WRITE(*,*) TRIM(cwd)
END PROGRAM
```

See also: Section 9.60 [CHDIR], page 145

### 9.125 GETENV — Get an environmental variable

**Description:**

Get the `VALUE` of the environmental variable `NAME`.

This intrinsic routine is provided for backwards compatibility with GNU Fortran 77. In new code, programmers should consider the use of the Section 9.126 [GET_ENVIRONMENT_VARIABLE], page 190 intrinsic defined by the Fortran 2003 standard.

Note that `GETENV` need not be thread-safe. It is the responsibility of the user to ensure that the environment is not being updated concurrently with a call to the `GETENV` intrinsic.

**Standard:** GNU extension

**Class:** Subroutine

**Syntax:**

```fortran
CALL GETENV(NAME, VALUE)
```

**Arguments:**

- **NAME**: Shall be of type `CHARACTER` and of default kind.
- **VALUE**: Shall be of type `CHARACTER` and of default kind.

**Return value:**

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.

Example:

```fortran
PROGRAM test_getenv
    CHARACTER(len=255) :: homedir
    CALL getenv("HOME", homedir)
    WRITE (*,*) TRIM(homedir)
END PROGRAM
```

See also: Section 9.126 [GET_ENVIRONMENT_VARIABLE], page 190
9.126 GET_ENVIRONMENT_VARIABLE — Get an environmental variable

Description:
Get the VALUE of the environmental variable NAME.

Note that GET_ENVIRONMENT_VARIABLE need not be thread-safe. It is the responsibility of the user to ensure that the environment is not being updated concurrently with a call to the GET_ENVIRONMENT_VARIABLE intrinsic.

Standard: Fortran 2003 and later
Class: Subroutine
Syntax: CALL GET_ENVIRONMENT_VARIABLE(NAME[, VALUE, LENGTH, STATUS, TRIM_NAME])

Arguments:
- **NAME** (Optional) Shall be a scalar of type CHARACTER and of default kind.
- **VALUE** 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.

Return value:
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.

Example:
```fortran
PROGRAM test_getenv
  CHARACTER(len=255) :: homedir
  CALL get_environment_variable("HOME", homedir)
  WRITE (*,*) TRIM(homedir)
END PROGRAM
```

9.127 GETGID — Group ID function

Description:
Returns the numerical group ID of the current process.

Standard: GNU extension
Class: Function
Chapter 9: Intrinsic Procedures

9.128 GETLOG — Get login name

Description:
Gets the username under which the program is running.

Standard: GNU extension
Class: Subroutine
Syntax: CALL GETLOG(C)
Arguments:
C Shall be of type CHARACTER and of default kind.

Return value:
Stores the current user name in LOGIN. (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.)

Example:
```fortran
PROGRAM TEST_GETLOG
  CHARACTER(32) :: login
  CALL GETLOG(login)
  WRITE(*,*) login
END PROGRAM
```

See also: Section 9.130 [GETUID], page 192

9.129 GETPID — Process ID function

Description:
Returns the numerical process identifier of the current process.

Standard: GNU extension
Class: Function
Syntax: RESULT = GETPID()

Return value:
The return value of GETPID is an INTEGER of the default kind.

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: Section 9.127 [GETGID], page 190, Section 9.130 [GETUID], page 192
9.130 GETUID — User ID function

Description:
Returns the numerical user ID of the current process.

Standard: GNU extension
Class: Function
Syntax: RESULT = GETUID()

Return value:
The return value of GETUID is an INTEGER of the default kind.

Example: See GETPID for an example.
See also: Section 9.129 [GETPID], page 191, Section 9.128 [GETLOG], page 191

9.131 GMTIME — Convert time to GMT info

Description:
Given a system time value TIME (as provided by the Section 9.270 [TIME], page 273 intrinsic), fills 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 gmtime(3).

This intrinsic routine is provided for backwards compatibility with GNU Fortran 77. In new code, programmers should consider the use of the Section 9.82 [DATE_AND_TIME], page 161 intrinsic defined by the Fortran 95 standard.

Standard: GNU extension
Class: Subroutine
Syntax: CALL GMTIME(TIME, VALUES)

Arguments:
TIME An INTEGER scalar expression corresponding to a system time, with INTENT(IN).
VALUES A default INTEGER array with 9 elements, with INTENT(OUT).

Return value:
The elements of VALUES are assigned as follows:
1. Seconds after the minute, range 0–59 or 0–61 to allow for leap seconds
2. Minutes after the hour, range 0–59
3. Hours past midnight, range 0–23
4. Day of month, range 1–31
5. Number of months since January, range 0–11
6. Years since 1900
7. Number of days since Sunday, range 0–6
8. Days since January 1, range 0–365
9. Daylight savings indicator: positive if daylight savings is in effect, zero if not, and negative if the information is not available.
9.132 HOSTNM — Get system host name

Description:
Retrieves the host name of the system on which the program is running.
This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

Standard: GNU extension
Class: Subroutine, function
Syntax:

CALL HOSTNM(C [, STATUS])
STATUS = HOSTNM(NAME)

Arguments:
C Shall of type CHARACTER and of default kind.
STATUS (Optional) status flag of type INTEGER. Returns 0 on success, or a system specific error code otherwise.

Return value:
In either syntax, NAME is set to the current hostname if it can be obtained, or to a blank string otherwise.

9.133 HUGE — Largest number of a kind

Description:
HUGE(X) returns the largest number that is not an infinity in the model of the type of X.

Standard: Fortran 95 and later
Class: Inquiry function
Syntax: RESULT = HUGE(X)

Arguments:
X Shall be of type REAL or INTEGER.

Return value:
The return value is of the same type and kind as 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
9.134 HYPOT — Euclidean distance function

Description:

HYPOT(X,Y) is the Euclidean distance function. It is equal to $\sqrt{X^2 + Y^2}$, without undue underflow or overflow.

Standard: Fortran 2008 and later

Class: Elemental function

Syntax: RESULT = HYPOT(X, Y)

Arguments:

X The type shall be REAL.
Y The type and kind type parameter shall be the same as X.

Return value:

The return value has the same type and kind type parameter as X.

Example:

```
program test_hypot
  real(4) :: x = 1.e0_4, y = 0.5e0_4
  x = hypot(x,y)
end program test_hypot
```

9.135 IACHAR — Code in ASCII collating sequence

Description:

IACHAR(C) returns the code for the ASCII character in the first character position of C.

Standard: Fortran 95 and later, with KIND argument Fortran 2003 and later

Class: Elemental function

Syntax: RESULT = IACHAR(C [, KIND])

Arguments:

C Shall be a scalar CHARACTER, with INTENT(IN)
KIND (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

Return value:

The return value is of type INTEGER and of kind KIND. If KIND is absent, the return value is of default integer kind.

Example:

```
program test_iachar
  integer i
  i = iachar(’ ’)
end program test_iachar
```

Note: See Section 9.143 [ICHAR], page 199 for a discussion of converting between numerical values and formatted string representations.

See also: Section 9.5 [ACHAR], page 109, Section 9.59 [CHAR], page 145, Section 9.143 [ICHAR], page 199
9.136 IALL — Bitwise AND of array elements

Description:
Reduces with bitwise AND the elements of ARRAY along dimension DIM if the corresponding element in MASK is TRUE.

Standard: Fortran 2008 and later
Class: Transformational function
Syntax:
RESULT = IALL(ARRAY[, MASK])
RESULT = IALL(ARRAY, DIM[, MASK])

Arguments:
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.

Return value:
The result is of the same type as ARRAY.
If DIM is absent, a scalar with the bitwise ALL of all elements in ARRAY is returned. Otherwise, an array of rank n-1, where n equals the rank of ARRAY, and a shape similar to that of ARRAY with dimension DIM dropped is returned.

Example:

```fortran
PROGRAM test_iand
  INTEGER(1) :: a(2)
  a(1) = b’00100100’
  a(2) = b’01101010’
  ! prints 00100000
  PRINT ‘(b8.8)’, IALL(a)
END PROGRAM
```

See also: Section 9.138 [IANY], page 196, Section 9.153 [IPARITY], page 205, Section 9.137 [IAND], page 195

9.137 IAND — Bitwise logical and

Description:
Bitwise logical AND.

Standard: Fortran 95 and later, has overloads that are GNU extensions
Class: Elemental function
Syntax: RESULT = IAND(I, J)
Arguments:
I The type shall be INTEGER or a boz-literal-constant.
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.

Return value:
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 Section 9.149 [INT], page 203 occurred.

Example:

```fortran
PROGRAM test_iand
  INTEGER :: a, b
  DATA a / Z'F' /, b / Z'3' /
  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</td>
<td>INTEGER</td>
<td>Fortran 95 and later</td>
</tr>
<tr>
<td>BIAND(A)</td>
<td>INTEGER(1)</td>
<td>INTEGER(1)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>IIAND(A)</td>
<td>INTEGER(2)</td>
<td>INTEGER(2)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>JIAND(A)</td>
<td>INTEGER(4)</td>
<td>INTEGER(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>KIAND(A)</td>
<td>INTEGER(8)</td>
<td>INTEGER(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

See also: Section 9.152 [IOR], page 204, Section 9.145 [IEOR], page 201, Section 9.141 [IBITS], page 198, Section 9.142 [IBSET], page 199, Section 9.140 [IBCLR], page 197, Section 9.209 [NOT], page 237

### 9.138 IANY — Bitwise OR of array elements

Description:
Reduces with bitwise OR (inclusive or) the elements of ARRAY along dimension DIM if the corresponding element in MASK is TRUE.

Standard: Fortran 2008 and later

Class: Transformational function

Syntax:

```fortran
RESULT = IANY(ARRAY[, MASK])
RESULT = IANY(ARRAY, DIM[, MASK])
```

Arguments:

- **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.

Return value:
The result is of the same type as ARRAY.

If DIM is absent, a scalar with the bitwise OR of all elements in ARRAY is returned. Otherwise, an array of rank n-1, where n equals the rank of ARRAY,
and a shape similar to that of ARRAY with dimension DIM dropped is returned.

Example:

```fortran
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: Section 9.153 [IPARITY], page 205, Section 9.136 [IALL], page 195, Section 9.152 [IOR], page 204

9.139 IARGC — Get the number of command line arguments

Description:

IARGC returns the number of arguments passed on the command line when the containing program was invoked.

This intrinsic routine is provided for backwards compatibility with GNU Fortran 77. In new code, programmers should consider the use of the Section 9.68 [COMMAND_ARGUMENT_COUNT], page 152 intrinsic defined by the Fortran 2003 standard.

Standard: GNU extension

Class: Function

Syntax: RESULT = IARGC()

Arguments: None.

Return value: The number of command line arguments, type INTEGER(4).

Example: See Section 9.121 [GETARG], page 186

See also: GNU Fortran 77 compatibility subroutine: Section 9.121 [GETARG], page 186

Fortran 2003 functions and subroutines: Section 9.122 [GET_COMMAND], page 187, Section 9.123 [GET_COMMAND_ARGUMENT], page 187, Section 9.68 [COMMAND_ARGUMENT_COUNT], page 152

9.140 IBCLR — Clear bit

Description:

IBCLR returns the value of I with the bit at position POS set to zero.

Standard: Fortran 95 and later, has overloads that are GNU extensions

Class: Elemental function
Syntax: \[ \text{RESULT} = \text{IBCLR}(I, \text{POS}) \]

Arguments:

1. \( I \) The type shall be \text{INTEGER}.
2. \( \text{POS} \) The type shall be \text{INTEGER}.

Return value:

The return value is of type \text{INTEGER} and of the same kind as \( I \).

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 95 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>JJBCLR(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: Section 9.141 \([\text{IBITS}]\), page 198, Section 9.142 \([\text{IBSET}]\), page 199, Section 9.137 \([\text{IAND}]\), page 195, Section 9.152 \([\text{IOR}]\), page 204, Section 9.145 \([\text{IEOR}]\), page 201, Section 9.204 \([\text{MVBITS}]\), page 234

9.141 IBITS — Bit extraction

Description:

\text{IBITS} extracts a field of length \( LEN \) from \( I \), starting from bit position \( \text{POS} \) and extending left for \( LEN \) bits. The result is right-justified and the remaining bits are zeroed. The value of \( \text{POS} + \text{LEN} \) must be less than or equal to the value \( \text{BIT\_SIZE}(I) \).

Standard: Fortran 95 and later, has overloads that are GNU extensions

Class: Elemental function

Syntax: \[ \text{RESULT} = \text{IBITS}(I, \text{POS}, \text{LEN}) \]

Arguments:

1. \( I \) The type shall be \text{INTEGER}.
2. \( \text{POS} \) The type shall be \text{INTEGER}.
3. \( \text{LEN} \) The type shall be \text{INTEGER}.

Return value:

The return value is of type \text{INTEGER} and of the same kind as \( I \).

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 95 and later</td>
</tr>
<tr>
<td>IBITS(A)</td>
<td>INTEGER(1) A</td>
<td>INTEGER(1)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>IBITS(A)</td>
<td>INTEGER(2) A</td>
<td>INTEGER(2)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>IBITS(A)</td>
<td>INTEGER(4) A</td>
<td>INTEGER(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>IBITS(A)</td>
<td>INTEGER(8) A</td>
<td>INTEGER(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

See also: Section 9.48 \([\text{BIT\_SIZE}]\), page 138, Section 9.140 \([\text{IBCLR}]\), page 197, Section 9.142 \([\text{IBSET}]\), page 199, Section 9.137 \([\text{IAND}]\), page 195, Section 9.152 \([\text{IOR}]\), page 204, Section 9.145 \([\text{IEOR}]\), page 201
9.142 IBSET — Set bit

Description:
IBSET returns the value of I with the bit at position POS set to one.

Standard: Fortran 95 and later, has overloads that are GNU extensions

Class: Elemental function

Syntax: RESULT = IBSET(I, POS)

Arguments:
- I The type shall be INTEGER.
- POS The type shall be INTEGER.

Return value:
The return value is of type INTEGER and of the same kind as I.

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 95 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>IIBSET(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>

See also: Section 9.140 [IBCLR], page 197, Section 9.141 [IBITS], page 198, Section 9.137 [IAND], page 195, Section 9.152 [IOR], page 204, Section 9.145 [IEOR], page 201, Section 9.204 [MVBITS], page 234

9.143 ICHAR — Character-to-integer conversion function

Description:
ICHAR(C) returns the code for the character in the first character position of 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.

Standard: Fortran 95 and later, with KIND argument Fortran 2003 and later

Class: Elemental function

Syntax: RESULT = ICHAR(C [, KIND])

Arguments:
- C Shall be a scalar CHARACTER, with INTENT(IN)
- KIND (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

Return value:
The return value is of type INTEGER and of kind KIND. If KIND is absent, the return value is of default integer kind.

Example:
```fortran
program test_ichar
  integer i
  i = ichar(’ ’)
end program test_ichar
```

**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 C</td>
<td>INTEGER(4)</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

**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:

```fortran
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:** Section 9.5 [ACHAR], page 109, Section 9.59 [CHAR], page 145, Section 9.135 [IACHAR], page 194

### 9.144 IDATE — Get current local time subroutine (day/month/year)

**Description:**

IDATE(VALEUES) Fills VALUES 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 VALUES, respectively. The year has four significant digits.

This intrinsic routine is provided for backwards compatibility with GNU Fortran 77. In new code, programmers should consider the use of the Section 9.82 [DATE_AND_TIME], page 161 intrinsic defined by the Fortran 95 standard.

**Standard:** GNU extension

**Class:** Subroutine

**Syntax:** CALL IDATE(VALEUES)

**Arguments:**

VALUES The type shall be INTEGER, DIMENSION(3) and the kind shall be the default integer kind.

**Return value:**

Does not return anything.
Example:

```fortran
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: Section 9.82 [DATE_AND_TIME], page 161

### 9.145 IEOR — Bitwise logical exclusive or

**Description:**

IEOR returns the bitwise Boolean exclusive-OR of \( I \) and \( J \).

**Standard:**

Fortran 95 and later, has overloads that are GNU extensions

**Class:**

Elemental function

**Syntax:**

RESULT = IEOR(I, J)

**Arguments:**

- \( I \)  
  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.

**Return value:**

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 Section 9.149 [INT], page 203 occurred.

**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 95 and later</td>
</tr>
<tr>
<td>BIEOR(A)</td>
<td>INTEGER(1) A INTEGER(1)</td>
<td>GNU extension</td>
<td></td>
</tr>
<tr>
<td>IIEOR(A)</td>
<td>INTEGER(2) A INTEGER(2)</td>
<td>GNU extension</td>
<td></td>
</tr>
<tr>
<td>JIEOR(A)</td>
<td>INTEGER(4) A INTEGER(4)</td>
<td>GNU extension</td>
<td></td>
</tr>
<tr>
<td>KIEOR(A)</td>
<td>INTEGER(8) A INTEGER(8)</td>
<td>GNU extension</td>
<td></td>
</tr>
</tbody>
</table>

See also: Section 9.152 [IOR], page 204, Section 9.137 [IAND], page 195, Section 9.141 [IBITS], page 198, Section 9.142 [IBSET], page 199, Section 9.140 [IBCLR], page 197, Section 9.209 [NOT], page 237

### 9.146 IERRNO — Get the last system error number

**Description:**

Returns the last system error number, as given by the C \texttt{errno} variable.

**Standard:**

GNU extension

**Class:**

Function

**Syntax:**

RESULT = IERRNO()
Arguments:
None.

Return value:
The return value is of type INTEGER and of the default integer kind.

See also: Section 9.215 [PERROR], page 241

9.147 IMAGE_INDEX — Function that converts a cosubscript to an image index

Description:
Returns the image index belonging to a cosubscript.

Standard: Fortran 2008 and later
Class: Inquiry function.
Syntax: RESULT = IMAGE_INDEX(COARRAY, SUB)
Arguments:
COARRAY Coarray of any type.
SUB default integer rank-1 array of a size equal to the corank of COARRAY.

Return value:
Scalar default integer with the value of the image index which corresponds to the cosubscripts. For invalid cosubscripts the result is zero.

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: Section 9.269 [THIS_IMAGE], page 272, Section 9.211 [NUM_IMAGES], page 238

9.148 INDEX — Position of a substring within a string

Description:
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.

Standard: Fortran 77 and later, with KIND argument Fortran 2003 and later
Class: Elemental function
Syntax: RESULT = INDEX(STRING, SUBSTRING [, BACK [, KIND]])
Arguments:
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.

**Return value:**
The return value is of type INTEGER and of kind KIND. If KIND is absent, the
return value is of default integer 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, CHARACTER)</td>
<td>INTEGER(4)</td>
<td>Fortran 77 and later</td>
<td></td>
</tr>
</tbody>
</table>

See also: Section 9.237 [SCAN], page 253, Section 9.283 [VERIFY], page 280

### 9.149 INT — Convert to integer type

**Description:**
Convert to integer type

**Standard:** Fortran 77 and later

**Class:** Elemental function

**Syntax:**
RESULT = INT(A [, KIND])

**Arguments:**

- **A**  Shall be of type INTEGER, REAL, or COMPLEX.
- **KIND** (Optional) An INTEGER initialization expression indicating
the kind parameter of the result.

**Return value:**
These functions return a INTEGER variable or array under the following rules:

- **A**  If A is of type INTEGER, INT(A) = A
- **B**  If A is of type REAL and |A| < 1, INT(A) equals 0. If |A| ≥ 1, then
INT(A) is the integer whose magnitude is the largest integer that
does not exceed the magnitude of A and whose sign is the same as
the sign of A.
- **C**  If A is of type COMPLEX, rule B is applied to the real part of A.

**Example:**

    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>
9.150 INT2 — Convert to 16-bit integer type

Description:
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.
The SHORT intrinsic is equivalent to INT2.

Standard: GNU extension
Class: Elemental function
Syntax: RESULT = INT2(A)
Arguments:
A Shall be of type INTEGER, REAL, or COMPLEX.

Return value:
The return value is a INTEGER(2) variable.

See also: Section 9.149 [INT], page 203, Section 9.151 [INT8], page 204, Section 9.181 [LONG], page 220

9.151 INT8 — Convert to 64-bit integer type

Description:
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.

Standard: GNU extension
Class: Elemental function
Syntax: RESULT = INT8(A)
Arguments:
A Shall be of type INTEGER, REAL, or COMPLEX.

Return value:
The return value is a INTEGER(8) variable.

See also: Section 9.149 [INT], page 203, Section 9.150 [INT2], page 204, Section 9.181 [LONG], page 220

9.152 IOR — Bitwise logical or

Description:
IOR returns the bitwise Boolean inclusive-OR of I and J.

Standard: Fortran 95 and later, has overloads that are GNU extensions
Class: Elemental function
Syntax: RESULT = IOR(I, J)
Arguments:

\[ I \]
The type shall be \texttt{INTEGER} or a boz-literal-constant.

\[ J \]
The type shall be \texttt{INTEGER} with the same kind type parameter as \texttt{I} or a boz-literal-constant. \texttt{I} and \texttt{J} shall not both be boz-literal-constants.

Return value:

The return type is \texttt{INTEGER} with the kind type parameter of the arguments. A boz-literal-constant is converted to an \texttt{INTEGER} with the kind type parameter of the other argument as-if a call to Section 9.149 [INT], page 203 occurred.

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 95 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: Section 9.145 [IEOR], page 201, Section 9.137 [IAND], page 195, Section 9.141 [IBITS], page 198, Section 9.142 [IBSET], page 199, Section 9.140 [IBCLR], page 197, Section 9.209 [NOT], page 237

9.153 IPARIETY — Bitwise XOR of array elements

Description:

Reduces with bitwise XOR (exclusive or) the elements of \texttt{ARRAY} along dimension \texttt{DIM} if the corresponding element in \texttt{MASK} is \texttt{TRUE}.

Standard: Fortran 2008 and later

Class: Transformational function

Syntax:

\[
\text{RESULT} = \text{IPARIETY} (\text{ARRAY}, \text{DIM}, \text{MASK})
\]

Arguments:

\[ \text{ARRAY} \]
Shall be an array of type \texttt{INTEGER}

\[ \text{DIM} \]
(Optional) shall be a scalar of type \texttt{INTEGER} with a value in the range from 1 to \texttt{n}, where \texttt{n} equals the rank of \texttt{ARRAY}.

\[ \text{MASK} \]
(Optional) shall be of type \texttt{LOGICAL} and either be a scalar or an array of the same shape as \texttt{ARRAY}.

Return value:

The result is of the same type as \texttt{ARRAY}.

If \texttt{DIM} is absent, a scalar with the bitwise XOR of all elements in \texttt{ARRAY} is returned. Otherwise, an array of rank \texttt{n-1}, where \texttt{n} equals the rank of \texttt{ARRAY}, and a shape similar to that of \texttt{ARRAY} with dimension \texttt{DIM} dropped is returned.

Example:
PROGRAM test_iparity
  INTEGER(1) :: a(2)
  a(1) = b'00100100'
  a(2) = b'01101010'
  ! prints 01001110
  PRINT '(b8.8)', IPARITY(a)
END PROGRAM

See also: Section 9.138 [IANY], page 196, Section 9.136 [IALL], page 195, Section 9.145 [IEOR], page 201, Section 9.214 [PARITY], page 240

9.154 IRAND — Integer pseudo-random number

Description:
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.
This intrinsic routine is provided for backwards compatibility with GNU Fortran 77. It implements a simple modulo generator as provided by g77. For new code, one should consider the use of Section 9.225 [RANDOM_NUMBER], page 246 as it implements a superior algorithm.

Standard: GNU extension
Class: Function
Syntax: RESULT = IRAND(I)

Arguments:
I Shall be a scalar INTEGER of kind 4.

Return value:
The return value is of INTEGER(kind=4) type.

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

9.155 IS_CONTIGUOUS — Test whether an array is contiguous

Description:
IS_CONTIGUOUS tests whether an array is contiguous.

Standard: Fortran 2008 and later
Class: Inquiry function
**Syntax:** RESULT = IS_CONTIGUOUS(ARRAY)

**Arguments:**
- ARRAY Shall be an array of any type.

**Return value:**
Returns a LOGICAL of the default kind, which .TRUE. if ARRAY is contiguous and false otherwise.

**Example:**
```fortran
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
```

### 9.156 IS_IOSTAT_END — Test for end-of-file value

**Description:**
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.

**Standard:** Fortran 2003 and later

**Class:** Elemental function

**Syntax:** RESULT = IS_IOSTAT_END(I)

**Arguments:**
- I Shall be of the type INTEGER.

**Return value:**
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.

**Example:**
```fortran
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
```
9.157 IS_IOSTAT_EOR — Test for end-of-record value

Description:

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.

Standard: Fortran 2003 and later

Class: Elemental function

Syntax: RESULT = IS_IOSTAT_EOR(I)

Arguments:

I Shall be of the type INTEGER.

Return value:

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.

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
```

9.158 ISATTY — Whether a unit is a terminal device.

Description:

Determine whether a unit is connected to a terminal device.

Standard: GNU extension

Class: Function

Syntax: RESULT = ISATTY(UNIT)

Arguments:

UNIT Shall be a scalar INTEGER.

Return value:

Returns .TRUE. if the UNIT is connected to a terminal device, .FALSE. otherwise.

Example:

```
PROGRAM test_isatty
  INTEGER(kind=1) :: unit
  DO unit = 1, 10
    write(*,*) isatty(unit=unit)
  END DO
END PROGRAM
```

See also: Section 9.277 [TTYNAM], page 277
9.159 ISHFT — Shift bits

Description:

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 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.

Standard: Fortran 95 and later, has overloads that are GNU extensions

Class: Elemental function

Syntax: 

RESULT = ISHFT(I, SHIFT)

Arguments:

I The type shall be INTEGER.
SHIFT The type shall be INTEGER.

Return value:

The return value is of type INTEGER and of the same kind as I.

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 95 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: Section 9.160 [ISHFTC], page 209

9.160 ISHFTC — Shift bits circularly

Description:

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 BIT_SIZE(I).

Standard: Fortran 95 and later, has overloads that are GNU extensions

Class: Elemental function

Syntax: 

RESULT = ISHFTC(I, SHIFT [, SIZE])

Arguments:

I The type shall be INTEGER.
SHIFT The type shall be INTEGER.
SIZE (Optional) The type shall be INTEGER; the value must be greater than zero and less than or equal to BIT_SIZE(I).
Return value:
The return value is of type INTEGER and of the same kind as I.

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 95 and later</td>
</tr>
<tr>
<td>BSHFTC(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: Section 9.159 [ISHFT], page 209

9.161 ISNAN — Test for a NaN

Description:
ISNAN tests whether a floating-point value is an IEEE Not-a-Number (NaN).

Standard: GNU extension

Class: Elemental function

Syntax: ISNAN(X)

Arguments:
X Variable of the type REAL.

Return value:
Returns a default-kind LOGICAL. The returned value is TRUE if X is a NaN and FALSE otherwise.

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
```

9.162 ITIME — Get current local time subroutine (hour/minutes/seconds)

Description:
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.

This intrinsic routine is provided for backwards compatibility with GNU Fortran 77. In new code, programmers should consider the use of the Section 9.82 [DATE_AND_TIME], page 161 intrinsic defined by the Fortran 95 standard.

Standard: GNU extension
**Class:** Subroutine

**Syntax:**

```
CALL ITIME(VALUES)
```

**Arguments:**

VALUES The type shall be `INTEGER, DIMENSION(3)` and the kind shall be the default integer kind.

**Return value:**

Does not return anything.

**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:** Section 9.82 [DATE_AND_TIME], page 161

### 9.163 KILL — Send a signal to a process

**Description:**

Sends the signal specified by `SIG` to the process `PID`. See `kill(2)`.

This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

**Class:** Subroutine, function

**Syntax:**

```
CALL KILL(PID, SIG [, STATUS])
STATUS = KILL(PID, SIG)
```

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PID</code></td>
<td>Shall be a scalar <code>INTEGER</code> with <code>INTENT(IN)</code>.</td>
</tr>
<tr>
<td><code>SIG</code></td>
<td>Shall be a scalar <code>INTEGER</code> with <code>INTENT(IN)</code>.</td>
</tr>
<tr>
<td><code>STATUS</code></td>
<td><a href="Optional">Subroutine</a> Shall be a scalar <code>INTEGER</code>. Returns 0 on success; otherwise a system-specific error code is returned.</td>
</tr>
<tr>
<td><code>STATUS</code></td>
<td>[Function] The kind type parameter is that of <code>PID</code>. Returns 0 on success; otherwise a system-specific error code is returned.</td>
</tr>
</tbody>
</table>

**See also:** Section 9.2 [ABORT], page 107, Section 9.101 [EXIT], page 173

### 9.164 KIND — Kind of an entity

**Description:**

`KIND(X)` returns the kind value of the entity `X`.

**Standard:** Fortran 95 and later

**Class:** Inquiry function

**Syntax:**

```
K = KIND(X)
```
Arguments:

\( X \)  
Shall be of type LOGICAL, INTEGER, REAL, COMPLEX or CHARACTER.

Return value:

The return value is a scalar of type INTEGER and of the default integer kind.

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
```

9.165 LBOUND — Lower dimension bounds of an array

Description:

Returns the lower bounds of an array, or a single lower bound along the \( DIM \) dimension.

Standard:  Fortran 95 and later, with KIND argument Fortran 2003 and later

Class:  Inquiry function

Syntax:  \( \text{RESULT} = \text{LBOUND}(\text{ARRAY} [, \text{DIM} [, \text{KIND}]]) \)

Arguments:

\( \text{ARRAY} \)  
Shall be an array, of any type.

\( \text{DIM} \)  
(Optional) Shall be a scalar INTEGER.

\( \text{KIND} \)  
(Optional) An INTEGER initialization expression indicating the kind parameter of the result.

Return value:

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.

See also:  Section 9.278 [UBOUND], page 277, Section 9.166 [LCOBOUND], page 212

9.166 LCOBOUND — Lower codimension bounds of an array

Description:

Returns the lower bounds of a coarray, or a single lower cobound along the DIM codimension.

Standard:  Fortran 2008 and later

Class:  Inquiry function
Chapter 9: Intrinsic Procedures

Syntax: \[ \text{RESULT} = \text{LCOBOUND(COARRAY [, DIM [, KIND]])} \]

Arguments:
- **ARRAY** Shall be an coarray, of any type.
- **DIM** (Optional) Shall be a scalar INTEGER.
- **KIND** (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

Return value:
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.

See also: Section 9.279 [UCOBOUND], page 278, Section 9.165 [LBOUND], page 212

9.167 **LEADZ** — Number of leading zero bits of an integer

Description:
LEADZ returns the number of leading zero bits of an integer.

Standard: Fortran 2008 and later

Class: Elemental function

Syntax: \[ \text{RESULT} = \text{LEADZ(I)} \]

Arguments:
- **I** Shall be of type INTEGER.

Return value:
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).

Example:
```fortran
PROGRAM test_leadz
  WRITE (*,*) BIT_SIZE(1) ! prints 32
  WRITE (*,*) LEADZ(1) ! prints 31
END PROGRAM
```

See also: Section 9.48 [BIT_SIZE], page 138, Section 9.273 [TRAILZ], page 275, Section 9.216 [POPCNT], page 241, Section 9.217 [POPPAR], page 242

9.168 **LEN** — Length of a character entity

Description:
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.

Standard: Fortran 77 and later, with KIND argument Fortran 2003 and later

Class: Inquiry function
**Syntax:**  
\[ L = \text{LEN} (\text{STRING} [, \text{KIND}]) \]

**Arguments:**
- **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.

**Return value:**
The return value is of type INTEGER and of kind KIND. If KIND is absent, the return value is of default integer 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:**  
Section 9.169 [LEN_TRIM], page 214, Section 9.9 [ADJUSTL], page 112, Section 9.10 [ADJUSTR], page 112

### 9.169 LEN_TRIM — Length of a character entity without trailing blank characters

**Description:**
Returns the length of a character string, ignoring any trailing blanks.

**Standard:**  
Fortran 95 and later, with KIND argument Fortran 2003 and later

**Class:**  
Elemental function

**Syntax:**  
\[ \text{RESULT} = \text{LEN} \_\text{TRIM} (\text{STRING} [, \text{KIND}]) \]

**Arguments:**
- **STRING** Shall be a scalar of type CHARACTER, with INTENT(IN)
- **KIND** (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

**Return value:**
The return value is of type INTEGER and of kind KIND. If KIND is absent, the return value is of default integer kind.

**See also:**  
Section 9.168 [LEN], page 213, Section 9.9 [ADJUSTL], page 112, Section 9.10 [ADJUSTR], page 112

### 9.170 LGE — Lexical greater than or equal

**Description:**
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.

In general, the lexical comparison intrinsics LGE, LGT, LLE, and LLT differ from the corresponding intrinsic operators .GE., .GT., .LE., and .LT., in that the latter use the processor’s character ordering (which is not ASCII on some targets), whereas the former always use the ASCII ordering.
**Standard:** Fortran 77 and later  
**Class:** Elemental function  
**Syntax:** \( \text{RESULT} = \text{LGE(STRING\_A, STRING\_B)} \)  
**Arguments:**  
- \( \text{STRING\_A} \): Shall be of default CHARACTER type.  
- \( \text{STRING\_B} \): Shall be of default CHARACTER type.  
**Return value:**  
Returns .TRUE. if \( \text{STRING\_A} \geq \text{STRING\_B} \), and .FALSE. otherwise, based on the ASCII ordering.

**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:** Section 9.171 [LGT], page 215, Section 9.173 [LLE], page 216, Section 9.174 [LLT], page 217

### 9.171 LGT — Lexical greater than

**Description:**  
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.

In general, the lexical comparison intrinsics LGE, LGT, LLE, and LLT differ from the corresponding intrinsic operators .GE., .GT., .LE., and .LT., in that the latter use the processor's character ordering (which is not ASCII on some targets), whereas the former always use the ASCII ordering.

**Standard:** Fortran 77 and later  
**Class:** Elemental function  
**Syntax:** \( \text{RESULT} = \text{LGT(STRING\_A, STRING\_B)} \)  
**Arguments:**  
- \( \text{STRING\_A} \): Shall be of default CHARACTER type.  
- \( \text{STRING\_B} \): Shall be of default CHARACTER type.  
**Return value:**  
Returns .TRUE. if \( \text{STRING\_A} > \text{STRING\_B} \), and .FALSE. otherwise, based on the ASCII ordering.

**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:** Section 9.170 [LGE], page 214, Section 9.173 [LLE], page 216, Section 9.174 [LLT], page 217
9.172 LINK — Create a hard link

Description:
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).

This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

Standard: GNU extension
Class: Subroutine, function
Syntax:
CALL LINK(PATH1, PATH2 [, STATUS])
STATUS = LINK(PATH1, PATH2)

Arguments:
PATH1 Shall be of default CHARACTER type.
PATH2 Shall be of default CHARACTER type.
STATUS (Optional) Shall be of default INTEGER type.

See also: Section 9.263 [SYMLNK], page 269, Section 9.281 [UNLINK], page 279

9.173 LLE — Lexical less than or equal

Description:
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.

In general, the lexical comparison intrinsics LGE, LGT, LLE, and LLT differ from the corresponding intrinsic operators .GE., .GT., .LE., and .LT., in that the latter use the processor’s character ordering (which is not ASCII on some targets), whereas the former always use the ASCII ordering.

Standard: Fortran 77 and later
Class: Elemental function
Syntax: RESULT = LLE(STRING_A, STRING_B)

Arguments:
STRING_A Shall be of default CHARACTER type.
STRING_B Shall be of default CHARACTER type.

Return value:
Returns .TRUE. if STRING_A <= STRING_B, and .FALSE. otherwise, based on the ASCII ordering.
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: Section 9.170 [LGE], page 214, Section 9.171 [LGT], page 215, Section 9.173 [LLE], page 216

9.174 LLT — Lexical less than

Description:

Determines whether one string is lexically less 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.

In general, the lexical comparison intrinsics LGE, LGT, LLE, and LLT differ from the corresponding intrinsic operators .GE., .GT., .LE., and .LT., in that the latter use the processor’s character ordering (which is not ASCII on some targets), whereas the former always use the ASCII ordering.

Standard: Fortran 77 and later
Class: Elemental function
Syntax: RESULT = LLT(STRING_A, STRING_B)

Arguments:

| STRING_A     | Shall be of default CHARACTER type. |
| STRING_B     | Shall be of default CHARACTER type. |

Return value:

Returns .TRUE. if STRING_A < STRING_B, and .FALSE. otherwise, based on the ASCII ordering.

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: Section 9.170 [LGE], page 214, Section 9.171 [LGT], page 215, Section 9.173 [LLE], page 216

9.175 LNBLNK — Index of the last non-blank character in a string

Description:

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.

Standard: GNU extension
Class: Elemental function
**Syntax:** RESULT = LNBLNK(STRING)

**Arguments:**

*STRING* Shall be a scalar of type CHARACTER, with INTENT(IN)

**Return value:**

The return value is of INTEGER(kind=4) type.

**See also:** Section 9.148 [INDEX intrinsic], page 202, Section 9.169 [LEN_TRIM], page 214

### 9.176 LOC — Returns the address of a variable

**Description:**

LOC(X) returns the address of X as an integer.

**Standard:** GNU extension

**Class:** Inquiry function

**Syntax:** RESULT = LOC(X)

**Arguments:**

*X* Variable of any type.

**Return value:**

The return value is of type INTEGER, with a KIND corresponding to the size (in bytes) of a memory address on the target machine.

**Example:**

```fortran
program test_loc
    integer :: i
    real :: r
    i = loc(r)
    print *, i
end program test_loc
```

### 9.177 LOG — Natural logarithm function

**Description:**

LOG(X) computes the natural logarithm of X, i.e. the logarithm to the base $e$.

**Standard:** Fortran 77 and later

**Class:** Elemental function

**Syntax:** RESULT = LOG(X)

**Arguments:**

*X* The type shall be REAL or COMPLEX.

**Return value:**

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 $-\pi < \omega \leq \pi$.

**Example:**
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)</td>
<td>REAL(4)</td>
<td>f95, gnu</td>
</tr>
<tr>
<td>DLOG(X)</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
<td>f95, gnu</td>
</tr>
<tr>
<td>CLOG(X)</td>
<td>COMPLEX(4)</td>
<td>COMPLEX(4)</td>
<td>f95, gnu</td>
</tr>
<tr>
<td>ZLOG(X)</td>
<td>COMPLEX(8)</td>
<td>COMPLEX(8)</td>
<td>f95, gnu</td>
</tr>
</tbody>
</table>

9.178 LOG10 — Base 10 logarithm function

Description:

LOG10(X) computes the base 10 logarithm of X.

Standard: Fortran 77 and later

Class: Elemental function

Syntax: RESULT = LOG10(X)

Arguments:

X The type shall be REAL.

Return value:

The return value is of type REAL or COMPLEX. The kind type parameter is the same as 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>REAL(4)</td>
<td>Fortran 95 and later</td>
</tr>
<tr>
<td>DLOG10(X)</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
<td>Fortran 95 and later</td>
</tr>
</tbody>
</table>

9.179 LOG_GAMMA — Logarithm of the Gamma function

Description:

LOG_GAMMA(X) computes the natural logarithm of the absolute value of the Gamma (Γ) function.

Standard: Fortran 2008 and later

Class: Elemental function

Syntax: X = LOG_GAMMA(X)
Arguments:

\(X\)

Shall be of type \texttt{REAL} and neither zero nor a negative integer.

Return value:

The return value is of type \texttt{REAL} of the same kind as \(X\).

Example:

```fortran
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)</td>
<td>X</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>ALGAMA(X)</td>
<td>REAL(4)</td>
<td>X</td>
<td>REAL(4)</td>
</tr>
<tr>
<td>DLGAMA(X)</td>
<td>REAL(8)</td>
<td>X</td>
<td>REAL(8)</td>
</tr>
</tbody>
</table>

See also: Gamma function: Section 9.119 [GAMMA], page 185

9.180 \texttt{LOGICAL} — Convert to logical type

Description:

Converts one kind of \texttt{LOGICAL} variable to another.

Standard: Fortran 95 and later

Class: Elemental function

Syntax: \texttt{RESULT = LOGICAL(L [, KIND])}

Arguments:

\(L\)

The type shall be \texttt{LOGICAL}.

\(KIND\)

(Optional) An \texttt{INTEGER} initialization expression indicating the kind parameter of the result.

Return value:

The return value is a \texttt{LOGICAL} value equal to \(L\), with a kind corresponding to \(KIND\), or of the default logical kind if \(KIND\) is not given.

See also: Section 9.149 [INT], page 203, Section 9.229 [REAL], page 248, Section 9.62 [CMPLX], page 147

9.181 \texttt{LONG} — Convert to integer type

Description:

Convert to a \texttt{KIND=}4 integer type, which is the same size as a C \texttt{long} integer. This is equivalent to the standard \texttt{INT} intrinsic with an optional argument of \texttt{KIND=}4, and is only included for backwards compatibility.

Standard: GNU extension

Class: Elemental function

Syntax: \texttt{RESULT = LONG(A)}
Arguments:

A Shall be of type INTEGER, REAL, or COMPLEX.

Return value:

The return value is a INTEGER(4) variable.

See also: Section 9.149 [INT], page 203, Section 9.150 [INT2], page 204, Section 9.151 [INT8], page 204

9.182 LSHIFT — Left shift bits

Description:

LSHIFT returns a value corresponding to I with all of the bits shifted left by SHIFT places. If the absolute value of SHIFT is greater than BIT_SIZE(I), the value is undefined. Bits shifted out from the left end are lost; zeros are shifted in from the opposite end.

This function has been superseded by the ISHFT intrinsic, which is standard in Fortran 95 and later, and the SHIFTL intrinsic, which is standard in Fortran 2008 and later.

Standard: GNU extension

Class: Elemental function

Syntax: RESULT = LSHIFT(I, SHIFT)

Arguments:

I The type shall be INTEGER.
SHIFT The type shall be INTEGER.

Return value:

The return value is of type INTEGER and of the same kind as I.

See also: Section 9.159 [ISHFT], page 209, Section 9.160 [ISHFTC], page 209, Section 9.234 [RSHIFT], page 251, Section 9.245 [SHIFTA], page 258, Section 9.246 [SHIFTL], page 258, Section 9.247 [SHIFTR], page 259

9.183 LSTAT — Get file status

Description:

LSTAT is identical to Section 9.260 [STAT], page 266, except that if path is a symbolic link, then the link itself is statted, not the file that it refers to.

The elements in VALUES are the same as described by Section 9.260 [STAT], page 266.

This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

Standard: GNU extension

Class: Subroutine, function

Syntax:

CALL LSTAT(NAME, VALUES [, STATUS])
STATUS = LSTAT(NAME, VALUES)
Arguments:

**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.

Example:  See Section 9.260 [STAT], page 266 for an example.

See also:  To stat an open file:  Section 9.117 [FSTAT], page 184, to stat a file:  Section 9.260 [STAT], page 266

### 9.184 LTIME — Convert time to local time info

**Description:**

Given a system time value **TIME** (as provided by the Section 9.270 [TIME], page 273 intrinsic), fills **VALUES** with values extracted from it appropriate to the local time zone using `localtime(3)`.

This intrinsic routine is provided for backwards compatibility with GNU Fortran 77. In new code, programmers should consider the use of the Section 9.82 [DATE_AND_TIME], page 161 intrinsic defined by the Fortran 95 standard.

**Standard:**  GNU extension

**Class:**  Subroutine

**Syntax:**

```
CALL LTIME(TIME, VALUES)
```

**Arguments:**

**TIME**  An **INTEGER** scalar expression corresponding to a system time, with **INTENT(IN)**.

**VALUES**  A default **INTEGER** array with 9 elements, with **INTENT(OUT)**.

**Return value:**

The elements of **VALUES** are assigned as follows:

1. Seconds after the minute, range 0–59 or 0–61 to allow for leap seconds
2. Minutes after the hour, range 0–59
3. Hours past midnight, range 0–23
4. Day of month, range 1–31
5. Number of months since January, range 0–11
6. Years since 1900
7. Number of days since Sunday, range 0–6
8. Days since January 1, range 0–365
9. Daylight savings indicator: positive if daylight savings is in effect, zero if not, and negative if the information is not available.

See also:  Section 9.82 [DATE_AND_TIME], page 161, Section 9.81 [CTIME], page 160, Section 9.131 [GMTIME], page 192, Section 9.270 [TIME], page 273, Section 9.271 [TIMES], page 274
9.185 **MALLOC — Allocate dynamic memory**

*Description:*  
*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*.

*Standard:* GNU extension  
*Class:* Function  
*Syntax:*  
\[
\text{PTR} = \text{MALLOC(SIZE)}
\]

*Arguments:*  
\[\text{SIZE} \quad \text{The type shall be INTEGER.}\]

*Return value:*  
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 *)).

*Example:* The following example demonstrates the use of MALLOC and FREE with Cray pointers.

```fortran
program test_malloc
  implicit none
  integer i
  real*8 x(*), z
  pointer(ptr_x,x)

  ptr_x = malloc(20*8)
  do i = 1, 20
    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:* Section 9.115 [FREE], page 182

9.186 **MASKL — Left justified mask**

*Description:*  
*MASKL(I[, KIND])* has its leftmost *I* bits set to 1, and the remaining bits set to 0.

*Standard:* Fortran 2008 and later  
*Class:* Elemental function  
*Syntax:*  
\[
\text{RESULT} = \text{MASKL(I[, KIND])}
\]
Arguments:
   \( I \)  Shall be of type \texttt{INTEGER}.
   \( \text{KIND} \)  Shall be a scalar constant expression of type \texttt{INTEGER}.

Return value:
The return value is of type \texttt{INTEGER}. If \texttt{KIND} is present, it specifies the kind value of the return type; otherwise, it is of the default integer kind.

See also:  Section 9.187 [MASKR], page 224

9.187 MASKR — Right justified mask

Description:
   \( \text{MASKR}(I[, \text{KIND}]) \) has its rightmost \( I \) bits set to 1, and the remaining bits set to 0.

Standard:  Fortran 2008 and later

Class:  Elemental function

Syntax:  \texttt{RESULT = MASKR(I[, \text{KIND}])}

Arguments:
   \( I \)  Shall be of type \texttt{INTEGER}.
   \( \text{KIND} \)  Shall be a scalar constant expression of type \texttt{INTEGER}.

Return value:
The return value is of type \texttt{INTEGER}. If \texttt{KIND} is present, it specifies the kind value of the return type; otherwise, it is of the default integer kind.

See also:  Section 9.186 [MASKL], page 223

9.188 MATMUL — matrix multiplication

Description:
   Performs a matrix multiplication on numeric or logical arguments.

Standard:  Fortran 95 and later

Class:  Transformational function

Syntax:  \texttt{RESULT = MATMUL(MATRIX\_A, MATRIX\_B)}

Arguments:
   \( \text{MATRIX\_A} \)  An array of \texttt{INTEGER}, \texttt{REAL}, \texttt{COMPLEX}, or \texttt{LOGICAL} type, with a rank of one or two.
   \( \text{MATRIX\_B} \)  An array of \texttt{INTEGER}, \texttt{REAL}, or \texttt{COMPLEX} type if \texttt{MATRIX\_A} is of a numeric type; otherwise, an array of \texttt{LOGICAL} type. The rank shall be one or two, and the first (or only) dimension of \texttt{MATRIX\_B} shall be equal to the last (or only) dimension of \texttt{MATRIX\_A}. \texttt{MATRIX\_A} and \texttt{MATRIX\_B} shall not both be rank one arrays.
**Return value:**
The matrix product of \texttt{MATRIX\_A} and \texttt{MATRIX\_B}. The type and kind of the result follow the usual type and kind promotion rules, as for the \texttt{*} or \texttt{.AND.} operators.

**See also:**

\section{MAX — Maximum value of an argument list}

**Description:**
Returns the argument with the largest (most positive) value.

**Standard:** Fortran 77 and later

**Class:** Elemental function

**Syntax:** \[
\text{RESULT} = \text{MAX}(A1, A2 [, A3 [, ...]])
\]

**Arguments:**
\begin{itemize}
  \item \texttt{A1} The type shall be \texttt{INTEGER} or \texttt{REAL}.
  \item \texttt{A2, A3, ...} An expression of the same type and kind as \texttt{A1}. (As a GNU extension, arguments of different kinds are permitted.)
\end{itemize}

**Return value:**
The return value corresponds to the maximum value among the arguments, and has the same type and kind as the first argument.

**Specific names:**
\begin{tabular}{llll}
Name & Argument & Return type & Standard \\
\texttt{MAX0(A1)} & \texttt{INTEGER(4)} A1 & \texttt{INTEGER(4)} & Fortran 77 and later \\
\texttt{AMAX0(A1)} & \texttt{REAL(4)} A1 & \texttt{REAL(MAX(X))} & Fortran 77 and later \\
\texttt{MAX1(A1)} & \texttt{REAL(4)} A1 & \texttt{INT(MAX(X))} & Fortran 77 and later \\
\texttt{AMAX1(A1)} & \texttt{REAL(8)} A1 & \texttt{REAL(4)} & Fortran 77 and later \\
\texttt{DMAX1(A1)} & \texttt{REAL(8)} A1 & \texttt{REAL(8)} & Fortran 77 and later \\
\end{tabular}

**See also:** Section 9.191 \texttt{[MAXLOC]}, page 226 Section 9.192 \texttt{[MAXVAL]}, page 227, Section 9.197 \texttt{[MIN]}, page 229

\section{MAXEXPONENT — Maximum exponent of a real kind}

**Description:**
\texttt{MAXEXPONENT(X)} returns the maximum exponent in the model of the type of \texttt{X}.

**Standard:** Fortran 95 and later

**Class:** Inquiry function

**Syntax:** \[
\text{RESULT} = \text{MAXEXPONENT}(X)
\]

**Arguments:**
\[
X \quad \text{Shall be of type REAL.}
\]

**Return value:**
The return value is of type \texttt{INTEGER} and of the default integer kind.
Example:

```
program exponents
  real(kind=4) :: x
  real(kind=8) :: y

  print *, minexponent(x), maxexponent(x)
  print *, minexponent(y), maxexponent(y)
end program exponents
```

9.191 MAXLOC — Location of the maximum value within an array

**Description:**
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 if it false; otherwise, the location returned is that of the first 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.

**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:**

```
RESULT = MAXLOC(ARRAY, DIM [, MASK] [,KIND] [,BACK])
RESULT = MAXLOC(ARRAY [, MASK] [,KIND] [,BACK])
```

**Arguments:**

- **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 an array 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.

**Return value:**

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.

See also: Section 9.108 [FINDLOC], page 178, Section 9.189 [MAX], page 225, Section 9.192 [MAXVAL], page 227

9.192 MAXVAL — Maximum value of an array

Description:
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 -HUGE(ARRAY) if ARRAY is numeric, or a string of nulls if ARRAY is of character type.

Standard: Fortran 95 and later

Class: Transformational function

Syntax:
RESULT = MAXVAL(ARRAY, DIM [, MASK])
RESULT = MAXVAL(ARRAY [, MASK])

Arguments:
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 an array of type LOGICAL, and conformable with ARRAY.

Return value:
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.

See also: Section 9.189 [MAX], page 225, Section 9.191 [MAXLOC], page 226

9.193 MCLOCK — Time function

Description:
Returns the number of clock ticks since the start of the process, based on the function clock(3) in the C standard library.

This intrinsic is not fully portable, such as to systems with 32-bit INTEGER types but supporting times wider than 32 bits. Therefore, the values returned by this intrinsic might be, or become, negative, or numerically less than previous values, during a single run of the compiled program.

Standard: GNU extension
Class: Function

Syntax: `RESULT = MCLOCK()`

Return value: 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)`.

See also: Section 9.81 [CTIME], page 160, Section 9.131 [GMTIME], page 192, Section 9.184 [LTIME], page 222, Section 9.193 [MCLOCK], page 227, Section 9.270 [TIME], page 273

9.194 MCLOCK8 — Time function (64-bit)

Description: Returns the number of clock ticks since the start of the process, based on the function `clock(3)` in the C standard library.

Warning: this intrinsic does not increase the range of the timing values over that returned by `clock(3)`. On a system with a 32-bit `clock(3)`, `MCLOCK8` will return a 32-bit value, even though it is converted to a 64-bit `INTEGER(8)` value. That means overflows of the 32-bit value can still occur. Therefore, the values returned by this intrinsic might be or become negative or numerically less than previous values during a single run of the compiled program.

Standard: GNU extension

Class: Function

Syntax: `RESULT = MCLOCK8()`

Return value: 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)`.

See also: Section 9.81 [CTIME], page 160, Section 9.131 [GMTIME], page 192, Section 9.184 [LTIME], page 222, Section 9.193 [MCLOCK], page 227, Section 9.270 [TIME], page 274

9.195 MERGE — Merge variables

Description: 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.`.

Standard: Fortran 95 and later

Class: Elemental function

Syntax: `RESULT = MERGE(TSOURCE, FSOURCE, MASK)`

Arguments: `TSOURCE` May be of any type.
**FSOURCE**
Shall be of the same type and type parameters as **TSOURCE**.

**MASK**
Shall be of type **LOGICAL**.

*Return value:*
The result is of the same type and type parameters as **TSOURCE**.

### 9.196 **MERGE_BITS** — Merge of bits under mask

**Description:**

**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.

**Standard:** Fortran 2008 and later

**Class:** Elemental function

**Syntax:**

\[
\text{RESULT} = \text{MERGE_BITS}(I, J, \text{MASK})
\]

**Arguments:**

- **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.

*Return value:*
The result is of the same type and kind as I.

### 9.197 **MIN** — Minimum value of an argument list

**Description:**

Returns the argument with the smallest (most negative) value.

**Standard:** Fortran 77 and later

**Class:** Elemental function

**Syntax:**

\[
\text{RESULT} = \text{MIN}(A1, A2 [, , A3, ...])
\]

**Arguments:**

- **A1**
  The type shall be **INTEGER** or **REAL**.

- **A2, A3, ...**
  An expression of the same type and kind as A1. (As a GNU extension, arguments of different kinds are permitted.)

*Return value:*
The return value corresponds to the maximum value among the arguments, and has the same type and kind as the first argument.

**Specific names:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>MINO(A1)</td>
<td>INTEGER(4) A1</td>
<td>INTEGER(4)</td>
<td>Fortran 77 and later</td>
</tr>
<tr>
<td>AMINO(A1)</td>
<td>INTEGER(4) A1</td>
<td>REAL(4)</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>
MIN1(A1)   REAL A1   INTEGER(4)   Fortran 77 and later
AMIN1(A1)   REAL(4) A1   REAL(4)   Fortran 77 and later
DMIN1(A1)   REAL(8) A1   REAL(8)   Fortran 77 and later

See also:  Section 9.189 [MAX], page 225, Section 9.199 [MINLOC], page 230,
          Section 9.200 [MINVAL], page 231

9.198 MINEXPONENT — Minimum exponent of a real kind

Description:

MINEXPONENT(X) returns the minimum exponent in the model of the type of X.

Standard:  Fortran 95 and later

Class:  Inquiry function

Syntax:

RESULT = MINEXPONENT(X)

Arguments:

X   Shall be of type REAL.

Return value:

The return value is of type INTEGER and of the default integer kind.

Example:  See MAXEXPONENT for an example.

9.199 MINLOC — Location of the minimum value within an array

Description:

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
if it false; otherwise, the location returned is that of the first 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.

Standard:  Fortran 95 and later; ARRAY of CHARACTER and the KIND argument are avail-
able in Fortran 2003 and later. The BACK argument is available in Fortran 2008 and later.

Class:  Transformational function

Syntax:

RESULT = MINLOC(ARRAY, DIM [, MASK] [,KIND] [,BACK])
RESULT = MINLOC(ARRAY [, MASK], [,KIND] [,BACK])

Arguments:

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 an array 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**.

**Return value:**

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.

**See also:** Section 9.108 [FINDLOC], page 178, Section 9.197 [MIN], page 229, Section 9.200 [MINVAL], page 231

### 9.200 MINVAL — Minimum value of an array

**Description:**

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.

**Standard:** Fortran 95 and later

**Class:** Transformational function

**Syntax:**

```
RESULT = MINVAL(ARRAY, DIM [, MASK])
RESULT = MINVAL(ARRAY [, MASK])
```

**Arguments:**

- **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 an array of type **LOGICAL**, and conformable with **ARRAY**.

**Return value:**

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**.
See also: Section 9.197 [MIN], page 229, Section 9.199 [MINLOC], page 230

9.201 MOD — Remainder function

Description:

MOD(A,P) computes the remainder of the division of A by P.

Standard: Fortran 77 and later, has overloads that are GNU extensions

Class: Elemental function

Syntax: RESULT = MOD(A, P)

Arguments:

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.

Return value:

The return value is the result of A - (INT(A/P) * 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.

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)
  print *, mod(-17.5d0,5.5)
  print *, mod(-17.5,5.5d0)
  print *, mod(17,-3)
  print *, mod(17.5,-5.5)
  print *, mod(17.5d0,-5.5)
  print *, mod(17.5,-5.5d0)
end program test_mod
```

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 95 and later</td>
</tr>
<tr>
<td>AMOD(A,P)</td>
<td>REAL(4) A,P</td>
<td>REAL(4)</td>
<td>Fortran 95 and later</td>
</tr>
<tr>
<td>DMOD(A,P)</td>
<td>REAL(8) A,P</td>
<td>REAL(8)</td>
<td>Fortran 95 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>
9.202 MODULO — Modulo function

Description:

MODULO(A,P) computes the A modulo P.

Standard: Fortran 95 and later

Class: Elemental function

Syntax: RESULT = MODULO(A, P)

Arguments:

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.

Return value:

The type and kind of the result are those of the arguments.

If A and P are of type INTEGER:

MODULO(A,P) has the value R such that A=Q*P+R, where Q is an integer and R is between 0 (inclusive) and P (exclusive).

If A and P are of type REAL:

MODULO(A,P) has the value of A - FLOOR (A / P) * P.

The returned value has the same sign as P and a magnitude less than the magnitude of 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)

print *, modulo(17,-3)
print *, modulo(17.5,-5.5)
end program

See also: Section 9.201 [MOD], page 232

9.203 MOVE_ALLOC — Move allocation from one object to another

Description:

MOVE_ALLOC(FROM, TO) moves the allocation from FROM to TO. FROM will become deallocated in the process.

Standard: Fortran 2003 and later

Class: Pure subroutine

Syntax: CALL MOVE_ALLOC(FROM, TO)
Arguments:

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.

Return value:
None

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
```

9.204 MVBITS — Move bits from one integer to another

Description:
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).

Standard: Fortran 95 and later, has overloads that are GNU extensions

Class: Elemental subroutine

Syntax: CALL MVBITS(FROM, FROMPOS, LEN, TO, TOPOS)

Arguments:

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.

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 95 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: Section 9.140 [IBCLR], page 197, Section 9.142 [IBSET], page 199,
Section 9.141 [IBITS], page 198, Section 9.137 [IAND], page 195, Section 9.152 [IOR], page 204, Section 9.145 [IEOR], page 201
9.205 **NEAREST — Nearest representable number**

**Description:**

NEAREST(X, S) returns the processor-representable number nearest to X in the direction indicated by the sign of S.

**Standard:** Fortran 95 and later

**Class:** Elemental function

**Syntax:**

RESULT = NEAREST(X, S)

**Arguments:**

- X
  - Shall be of type REAL.
- S
  - Shall be of type REAL and not equal to zero.

**Return value:**

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.

**Example:**

```fortran
program test_nearest
  real :: x, y
  x = nearest(42.0, 1.0)
  y = nearest(42.0, -1.0)
  write (*,"(3(G20.15))") x, y, x - y
end program test_nearest
```

9.206 **NEW_LINE — New line character**

**Description:**

NEW_LINE(C) returns the new-line character.

**Standard:** Fortran 2003 and later

**Class:** Inquiry function

**Syntax:**

RESULT = NEW_LINE(C)

**Arguments:**

- C
  - The argument shall be a scalar or array of the type CHARACTER.

**Return value:**

Returns a CHARACTER scalar of length one with the new-line character of the same kind as parameter C.

**Example:**

```fortran
program newline
  implicit none
  write(*,'(A)') 'This is record 1.'//NEW_LINE('A')//''This is record 2.'
end program newline
```
9.207 NINT — Nearest whole number

Description:
NINT(A) rounds its argument to the nearest whole number.

Standard: Fortran 77 and later, with KIND argument Fortran 90 and later

Class: Elemental function

Syntax: RESULT = NINT(A [, KIND])

Arguments:
A The type of the argument shall be REAL.
KIND (Optional) An INTEGER initialization expression indicating
the kind parameter of the result.

Return value:
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.

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 95 and later</td>
</tr>
<tr>
<td>IDNINT(A)</td>
<td>REAL(8) A</td>
<td>INTEGER</td>
<td>Fortran 95 and later</td>
</tr>
</tbody>
</table>

See also: Section 9.58 [CEILING], page 144, Section 9.109 [FLOOR], page 178

9.208 NORM2 — Euclidean vector norms

Description:
Calculates the Euclidean vector norm ($L_2$ norm) of of ARRAY along dimension DIM.

Standard: Fortran 2008 and later

Class: Transformational function

Syntax:
RESULT = NORM2(ARRAY[, DIM])

Arguments:
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.
Return value:
The result is of the same type as ARRAY.
If DIM is absent, a scalar with the square root of the sum of all elements in ARRAY squared is returned. Otherwise, an array of rank $n - 1$, where $n$ equals the rank of ARRAY, and a shape similar to that of ARRAY with dimension DIM dropped is returned.

Example:
```fortran
PROGRAM test_sum
  REAL :: x(5) = [ real :: 1, 2, 3, 4, 5 ]
  print *, NORM2(x) ! = sqrt(55.) ~ 7.416
END PROGRAM
```

9.209 NOT — Logical negation

Description:
NOT returns the bitwise Boolean inverse of I.

Standard: Fortran 95 and later, has overloads that are GNU extensions

Class: Elemental function

Syntax:
RESULT = NOT(I)

Arguments:
I The type shall be INTEGER.

Return value:
The return type is INTEGER, of the same kind as the argument.

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</td>
<td>INTEGER</td>
<td>Fortran 95 and later</td>
</tr>
<tr>
<td>BNOT(A)</td>
<td>INTEGER(1)</td>
<td>INTEGER(1)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>INOT(A)</td>
<td>INTEGER(2)</td>
<td>INTEGER(2)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>JNOT(A)</td>
<td>INTEGER(4)</td>
<td>INTEGER(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>KNOT(A)</td>
<td>INTEGER(8)</td>
<td>INTEGER(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>

See also: Section 9.137 [IAND], page 195, Section 9.145 [IEOR], page 201, Section 9.152 [IOR], page 204, Section 9.141 [IBITS], page 198, Section 9.142 [IBSET], page 199, Section 9.140 [IBCLR], page 197

9.210 NULL — Function that returns an disassociated pointer

Description:
Returns a disassociated pointer.

If MOLD is present, a disassociated pointer of the same type is returned, otherwise the type is determined by context.

In Fortran 95, MOLD is optional. Please note that Fortran 2003 includes cases where it is required.

Standard: Fortran 95 and later
Transformational function

Syntax:

PTR => NULL([MOLD])

Arguments:

MOLD (Optional) shall be a pointer of any association status and of any type.

Return value:

A disassociated pointer.

Example:

REAL, POINTER, DIMENSION(:) :: VEC => NULL ()

See also: Section 9.22 [ASSOCIATED], page 120

9.211 NUM_IMAGES — Function that returns the number of images

Description:

Returns the number of images.

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)

Arguments:

DISTANCE (optional, intent(in)) Nonnegative scalar integer

FAILED (optional, intent(in)) Scalar logical expression

Return value:

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.

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: Section 9.269 [THIS_IMAGE], page 272, Section 9.147 [IMAGE_INDEX], page 202
9.212 OR — Bitwise logical OR

Description:
Bitwise logical OR.
This intrinsic routine is provided for backwards compatibility with GNU Fortran 77. For integer arguments, programmers should consider the use of the Section 9.152 [IOR], page 204 intrinsic defined by the Fortran standard.

Standard: GNU extension
Class: Function
Syntax:
RESULT = OR(I, J)

Arguments:
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.

Return value:
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 Section 9.149 [INT], page 203 occurred.

Example:

```fortran
PROGRAM test_or
 LOGICAL :: T = .TRUE., F = .FALSE.
 INTEGER :: a, b
 DATA a / Z'F' /, b / Z'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: Section 9.152 [IOR], page 204

9.213 PACK — Pack an array into an array of rank one

Description:
Stores the elements of ARRAY in an array of rank one.
The beginning of the resulting array is made up of elements whose MASK equals TRUE. Afterwards, positions are filled with elements taken from VECTOR.

Standard: Fortran 95 and later
Class: Transformational function
Syntax:
RESULT = PACK(ARRAY, MASK[,VECTOR])
Arguments:

- **ARRAY** Shall be an array of any type.
- **MASK** Shall be an array of type `LOGICAL` and of the same size as **ARRAY**. Alternatively, it may be a `LOGICAL` scalar.
- **VECTOR** (Optional) shall be an array of the same type as **ARRAY** and of rank one. If present, the number of elements in **VECTOR** shall be equal to or greater than the number of true elements in **MASK**. If **MASK** is scalar, the number of elements in **VECTOR** shall be equal to or greater than the number of elements in **ARRAY**.

Return value:

The result is an array of rank one and the same type as that of **ARRAY**. If **VECTOR** is present, the result size is that of **VECTOR**, the number of `TRUE` values in **MASK** otherwise.

Example: Gathering nonzero elements from an array:

```fortran
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**:

```fortran
PROGRAM test_pack_2
  INTEGER :: m(4)
  m = (/ 1, 0, 0, 2 /)
  WRITE(*, FMT="(4(I0, ' '))") pack(m, m /= 0, (/ 0, 0, 3, 4 /)) ! "1 2 3 4"
END PROGRAM
```

See also: Section 9.282 [UNPACK], page 279

### 9.214 PARITY — Reduction with exclusive OR

**Description:**

Calculates the parity, i.e. the reduction using `.XOR.` of **MASK** along dimension **DIM**.

**Standard:** Fortran 2008 and later

**Class:** Transformational function

**Syntax:**

```fortran
RESULT = PARITY(MASK[, DIM])
```

**Arguments:**

- **LOGICAL** 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**.

**Return value:**

The result is of the same type as **MASK**.
If $DIM$ is absent, a scalar with the parity of all elements in $MASK$ is returned, i.e. true if an odd number of elements is `.true.` and false otherwise. If $DIM$ is present, an array of rank $n - 1$, where $n$ equals the rank of $ARRAY$, and a shape similar to that of $MASK$ with dimension $DIM$ dropped is returned.

**Example:**

```fortran
PROGRAM test_sum
  LOGICAL :: x(2) = [ .true., .false. ]
  print *, PARITY(x) ! prints "T" (true).
END PROGRAM
```

### 9.215 PERROR — Print system error message

**Description:**

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)`.

**Standard:** GNU extension

**Class:** Subroutine

**Syntax:**

```fortran
CALL PERROR(STRING)
```

**Arguments:**

- `STRING` A scalar of type `CHARACTER` and of the default kind.

**See also:** Section 9.146 [IERRNO], page 201

### 9.216 POPCNT — Number of bits set

**Description:**

$POPCNT(I)$ returns the number of bits set (`1` bits) in the binary representation of $I$.

**Standard:** Fortran 2008 and later

**Class:** Elemental function

**Syntax:**

```fortran
RESULT = POPCNT(I)
```

**Arguments:**

- $I$ Shall be of type `INTEGER`.

**Return value:**

The return value is of type `INTEGER` and of the default integer kind.

**See also:** Section 9.217 [POPPAR], page 242, Section 9.167 [LEADZ], page 213, Section 9.273 [TRAILZ], page 275

**Example:**

```fortran
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
```
**9.217 POPPAR — Parity of the number of bits set**

*Description:*

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.

*Standard:* Fortran 2008 and later

*Class:* Elemental function

*Syntax:* 

\[
\text{RESULT} = \text{POPPAR}(I)
\]

*Arguments:*

\( I \)  

Shall be of type INTEGER.

*Return value:*

The return value is of type INTEGER and of the default integer kind.

*See also:* Section 9.216 [POPCNT], page 241, Section 9.167 [LEADZ], page 213, Section 9.273 [TRAILZ], page 275

*Example:*

```fortran
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
```

---

**9.218 PRECISION — Decimal precision of a real kind**

*Description:*

PRECISION(X) returns the decimal precision in the model of the type of X.

*Standard:* Fortran 95 and later

*Class:* Inquiry function

*Syntax:*

\[
\text{RESULT} = \text{PRECISION}(X)
\]

*Arguments:*

\( X \)  

Shall be of type REAL or COMPLEX.

*Return value:*

The return value is of type INTEGER and of the default integer kind.

*See also:* Section 9.242 [SELECTED_REAL_KIND], page 256, Section 9.227 [RANGE], page 248

*Example:*

```fortran
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
```
Chapter 9: Intrinsic Procedures

9.219 PRESENT — Determine whether an optional dummy argument is specified

**Description:**
Determine whether an optional dummy argument is present.

**Standard:** Fortran 95 and later

**Class:** Inquiry function

**Syntax:**
RESULT = PRESENT(A)

**Arguments:**
- 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.

**Return value:**
Returns either TRUE if the optional argument A is present, or FALSE otherwise.

**Example:**
```fortran
PROGRAM test_present
    WRITE(*,*) f(), f(42) ! "F T"
    CONTAINS
    LOGICAL FUNCTION f(x)
        INTEGER, INTENT(IN), OPTIONAL :: x
        f = PRESENT(x)
    END FUNCTION
END PROGRAM
```

9.220 PRODUCT — Product of array elements

**Description:**
Multiplies the elements of ARRAY along dimension DIM if the corresponding element in MASK is TRUE.

**Standard:** Fortran 95 and later

**Class:** Transformational function

**Syntax:**
RESULT = PRODUCT(ARRAY[, MASK])
RESULT = PRODUCT(ARRAY, DIM[, MASK])

**Arguments:**
- 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.

**Return value:**
The result is of the same type as ARRAY.
If DIM is absent, a scalar with the product of all elements in ARRAY is returned. Otherwise, an array of rank n-1, where n equals the rank of ARRAY,
and a shape similar to that of ARRAY with dimension DIM dropped is returned.

Example:

```fortran
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: Section 9.262 [SUM], page 268

### 9.221 RADIX — Base of a model number

**Description:**

RADIX(X) returns the base of the model representing the entity X.

**Standard:** Fortran 95 and later

**Class:** Inquiry function

**Syntax:**

```
RESULT = RADIX(X)
```

**Arguments:**

- **X**
  - Shall be of type INTEGER or REAL

**Return value:**

The return value is a scalar of type INTEGER and of the default integer kind.

**See also:** Section 9.242 [SELECTED_REAL_KIND], page 256

**Example:**

```fortran
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
```

### 9.222 RAN — Real pseudo-random number

**Description:**

For compatibility with HP FORTRAN 77/iX, the RAN intrinsic is provided as an alias for RAND. See Section 9.223 [RAND], page 244 for complete documentation.

**Standard:** GNU extension

**Class:** Function

**See also:** Section 9.223 [RAND], page 244, Section 9.225 [RANDOM_NUMBER], page 246

### 9.223 RAND — Real pseudo-random number

**Description:**

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 \textit{FLAG} is 1, the generator is restarted by \texttt{CALL SRAND(0)}; if \textit{FLAG} has any other value, it is used as a new seed with \texttt{SRAND}.

This intrinsic routine is provided for backwards compatibility with GNU Fortran 77. It implements a simple modulo generator as provided by \texttt{g77}. For new code, one should consider the use of \textsection 9.225 [\texttt{RANDOM\_NUMBER}], page 246 as it implements a superior algorithm.

\begin{description}
\item[Standard:] GNU extension
\item[Class:] Function
\item[Syntax:] \texttt{RESULT = RAND(I)}
\item[Arguments:] \texttt{I} \hspace{1em} Shall be a scalar \texttt{INTEGER} of kind 4.
\item[Return value:] The return value is of \texttt{REAL} type and the default kind.
\item[Example:] 
\begin{verbatim}
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
\end{verbatim}
\item[See also:] \textsection 9.259 [\texttt{SRAND}], page 266, \textsection 9.225 [\texttt{RANDOM\_NUMBER}], page 246
\end{description}

\subsection*{9.224 \texttt{RANDOM\_INIT} — Initialize a pseudo-random number generator}

\textbf{Description:}
Initializes the state of the pseudorandom number generator used by \texttt{RANDOM\_NUMBER}.

\begin{description}
\item[Standard:] Fortran 2018
\item[Class:] Subroutine
\item[Syntax:] \texttt{CALL RANDOM\_INIT(REPEATABLE, IMAGE\_DISTINCT)}
\item[Arguments:] \texttt{REPEATABLE} \hspace{1em} Shall be a scalar with a \texttt{LOGICAL} type, and it is \texttt{INTENT(IN)}. If it is \texttt{.true.}, the seed is set to a processor-dependent value that is the same each time \texttt{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 \texttt{.false.}, the seed is set to a processor-dependent value.
\end{description}
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 value that does depend which image called `RANDOM_INIT`.

Example:

```fortran
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: Section 9.225 [RANDOM_NUMBER], page 246, Section 9.226 [RANDOM_SEED], page 247

### 9.225 RANDOM_NUMBER — Pseudo-random number

**Description:**

Returns a single pseudorandom number or an array of pseudorandom numbers from the uniform distribution over the range $0 \leq x < 1$.

The runtime-library implements the xorshift1024* random number generator (RNG). This generator has a period of $2^{1024} - 1$, and when using multiple threads up to $2^{512}$ threads can each generate $2^{512}$ random numbers before any aliasing occurs.

Note that in a multi-threaded program (e.g. using OpenMP directives), each thread will have its own random number state. For details of the seeding procedure, see the documentation for the `RANDOM_SEED` intrinsic.

**Standard:** Fortran 95 and later

**Class:** Subroutine

**Syntax:** `RANDOM_NUMBER(HARVEST)`

**Arguments:**

- `HARVEST` Shall be a scalar or an array of type `REAL`.

Example:

```fortran
program test_random_number
  REAL :: r(5,5)
  CALL RANDOM_NUMBER(r)
end program
```

See also: Section 9.226 [RANDOM_SEED], page 247, Section 9.224 [RANDOM_INIT], page 245
9.226 **RANDOM_SEED** — Initialize a pseudo-random number sequence

*Description:*  
Restarts or queries the state of the pseudorandom number generator used by WRITE_NUMBER.

If **RANDOM_SEED** is called without arguments, it is seeded with random data retrieved from the operating system.

As an extension to the Fortran standard, the GFortran **RANDOM_NUMBER** supports multiple threads. Each thread in a multi-threaded program has its own seed. When **RANDOM_SEED** is called either without arguments or with the **PUT** argument, the given seed is copied into a master seed as well as the seed of the current thread. When a new thread uses **RANDOM_NUMBER** for the first time, the seed is copied from the master seed, and forwarded $N \times 2^{12}$ steps to guarantee that the random stream does not alias any other stream in the system, where $N$ is the number of threads that have used **RANDOM_NUMBER** so far during the program execution.

*Standard:* Fortran 95 and later

*Class:* Subroutine

*Syntax:*  
CALL **RANDOM_SEED**([SIZE, PUT, GET])

*Arguments:*

- **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 **INTENT(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 **INTENT(OUT)** and the size of the array must be larger than or equal to the number returned by the **SIZE** argument.

*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:*  
Section 9.225 [RANDOM_NUMBER], page 246, Section 9.224 [RANDOM_INIT], page 245
9.227 RANGE — Decimal exponent range

Description:
RANGE(X) returns the decimal exponent range in the model of the type of X.

Standard: Fortran 95 and later
Class: Inquiry function
Syntax: RESULT = RANGE(X)

Arguments:
X Shall be of type INTEGER, REAL or COMPLEX.

Return value:
The return value is of type INTEGER and of the default integer kind.

See also: Section 9.242 [SELECTED_REAL_KIND], page 256, Section 9.218 [PRECISION], page 242

Example: See PRECISION for an example.

9.228 RANK — Rank of a data object

Description:
RANK(A) returns the rank of a scalar or array data object.

Standard: Technical Specification (TS) 29113
Class: Inquiry function
Syntax: RESULT = RANK(A)

Arguments:
A can be of any type

Return value:
The return value is of type INTEGER and of the default integer kind. For arrays, their rank is returned; for scalars zero is returned.

Example:

```
program test_rank
  integer :: a
  real, allocatable :: b(:,;)

  print *, rank(a), rank(b) ! Prints: 0 2
end program test_rank
```

9.229 REAL — Convert to real type

Description:
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.

Standard: Fortran 77 and later
Class: Elemental function
Syntax:

\[
\begin{align*}
\text{RESULT} &= \text{REAL}(A [, \text{KIND}]) \\
\text{RESULT} &= \text{REALPART}(Z)
\end{align*}
\]

Arguments:

- \(A\)  
  Shall be \text{INTEGER}, \text{REAL}, or \text{COMPLEX}.

- \(\text{KIND}\)  
  (Optional) An \text{INTEGER} initialization expression indicating the kind parameter of the result.

Return value:

These functions return a \text{REAL} variable or array under the following rules:

(A) \(\text{REAL}(A)\) is converted to a default real type if \(A\) is an integer or real variable.

(B) \(\text{REAL}(A)\) is converted to a real type with the kind type parameter of \(A\) if \(A\) is a complex variable.

(C) \(\text{REAL}(A, \text{KIND})\) is converted to a real type with kind type parameter \(\text{KIND}\) if \(A\) is a complex, integer, or real variable.

Example:

```
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>Fortran 77 and later</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(8)</td>
<td>REAL(4)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>SNGL(A)</td>
<td>INTEGER(8)</td>
<td>REAL(4)</td>
<td>Fortran 77 and later</td>
</tr>
</tbody>
</table>

See also:  
Section 9.83 [DBLE], page 162

9.230 RENAME — Rename a file

Description:

Renames a file from file \(\text{PATH1}\) to \(\text{PATH2}\). A null character (\text{CHAR(0)}\) can be used to mark the end of the names in \(\text{PATH1}\) and \(\text{PATH2}\); otherwise, trailing blanks in the file names are ignored. If the \text{STATUS} argument is supplied, it contains 0 on success or a nonzero error code upon return; see \text{rename(2)}.

This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

Standard: GNU extension

Class: Subroutine, function

Syntax:
CALL RENAME(PATH1, PATH2 [, STATUS])
STATUS = RENAME(PATH1, PATH2)

Arguments:

PATH1 Shall be of default CHARACTER type.
PATH2 Shall be of default CHARACTER type.
STATUS (Optional) Shall be of default INTEGER type.

See also: Section 9.172 [LINK], page 216

9.231 REPEAT — Repeated string concatenation

Description:
Concatenates NCOPIES copies of a string.

Standard: Fortran 95 and later

Class: Transformational function

Syntax: RESULT = REPEAT(STRING, NCOPIES)

Arguments:

STRING Shall be scalar and of type CHARACTER.
NCOPIES Shall be scalar and of type INTEGER.

Return value:
A new scalar of type CHARACTER built up from NCOPIES copies of STRING.

Example:

program test_repeat
    write(*,*) repeat("x", 5)  ! "xxxxx"
end program

9.232 RESHAPE — Function to reshape an array

Description:
Reshapes SOURCE to correspond to SHAPE. If necessary, the new array may be padded with elements from PAD or permuted as defined by ORDER.

Standard: Fortran 95 and later

Class: Transformational function

Syntax: RESULT = RESHAPE(SOURCE, SHAPE[, PAD, ORDER])

Arguments:

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.
Return value:
The result is an array of shape \( SHAPE \) with the same type as \( SOURCE \).

Example:

```fortran
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: Section 9.244 [SHAPE], page 257

9.233 RRSPACING — Reciprocal of the relative spacing

Description:

\( \text{RRSPACING}(X) \) returns the reciprocal of the relative spacing of model numbers near \( X \).

Standard: Fortran 95 and later

Class: Elemental function

Syntax:

\[ \text{RESULT} = \text{RRSPACING}(X) \]

Arguments:

\( X \) Shall be of type \texttt{REAL}.

Return value:

The return value is of the same type and kind as \( X \). The value returned is equal to
\[
\text{ABS} (\text{FRACTION}(X)) \times \text{FLOAT} (\text{RADIX}(X))^{\text{DIGITS}(X)}.
\]

See also: Section 9.256 [SPACING], page 264

9.234 RSHIFT — Right shift bits

Description:

\( \text{RSHIFT} \) returns a value corresponding to \( I \) with all of the bits shifted right by \( SHIFT \) places. If the absolute value of \( SHIFT \) is greater than \( \text{BIT}_\text{SIZE}(I) \), the 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.

This function has been superseded by the \texttt{SHIFTA} intrinsic, which is standard in Fortran 2008 and later.

Standard: GNU extension

Class: Elemental function

Syntax:

\[ \text{RESULT} = \text{RSHIFT}(I, \text{SHIFT}) \]

Arguments:

\( I \) The type shall be \texttt{INTEGER}.

\( SHIFT \) The type shall be \texttt{INTEGER}.

Return value:

The return value is of type \texttt{INTEGER} and of the same kind as \( I \).
9.235 SAME_TYPE_AS — Query dynamic types for equality

Description:
Query dynamic types for equality.

Standard: Fortran 2003 and later

Class: Inquiry function

Syntax: RESULT = SAME_TYPE_AS(A, B)

Arguments:
A Shall be an object of extensible declared type or unlimited polymorphic.
B Shall be an object of extensible declared type or unlimited polymorphic.

Return value:
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.

See also: Section 9.104 [EXTENDS_TYPE_OF], page 175

9.236 SCALE — Scale a real value

Description:
SCALE(X, I) returns X * RADIX(X)**I.

Standard: Fortran 95 and later

Class: Elemental function

Syntax: RESULT = SCALE(X, I)

Arguments:
X The type of the argument shall be a REAL.
I The type of the argument shall be a INTEGER.

Return value:
The return value is of the same type and kind as X. Its value is X * RADIX(X)**I.

Example:

```fortran
program test_scale
    real :: x = 178.1387e-4
    integer :: i = 5
    print *, scale(x,i), x*radix(x)**i
end program test_scale
```
9.237 SCAN — Scan a string for the presence of a set of characters

Description:
Scans a STRING for any of the characters in a SET of characters.
If BACK is either absent or equals FALSE, this function returns the position of
the leftmost character of STRING that is in SET. If BACK equals TRUE, the
rightmost position is returned. If no character of SET is found in STRING, the
result is zero.

Standard: Fortran 95 and later, with KIND argument Fortran 2003 and later
Class: Elemental function
Syntax: RESULT = SCAN(STRING, SET[, BACK [, KIND]])
Arguments:
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.

Return value:
The return value is of type INTEGER and of kind KIND. If KIND is absent, the
return value is of default integer 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: Section 9.148 [INDEX intrinsic], page 202, Section 9.283 [VERIFY], page 280

9.238 SECNDS — Time function

Description:
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.

Standard: GNU extension
Class: Function
Syntax: RESULT = SECNDS (X)
Arguments:
T Shall be of type REAL(4).
X Shall be of type REAL(4).

Return value:
None
Example:

```
program test_seconds
  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_seconds
```

9.239 SECOND — CPU time function

Description:

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.

This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

Standard: GNU extension

Class: Subroutine, function

Syntax:

```
CALL SECOND(TIME)
TIME = SECOND()
```

Arguments:

- **TIME** Shall be of type REAL(4).

Return value:

In either syntax, *TIME* is set to the process’s current runtime in seconds.

See also: Section 9.79 [CPU_TIME], page 158

9.240 SELECTED_CHAR_KIND — Choose character kind

Description:

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.

Standard: Fortran 2003 and later

Class: Transformational function

Syntax: RESULT = SELECTED_CHAR_KIND(NAME)

Arguments:

- **NAME** Shall be a scalar and of the default character type.
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
```

9.241 SELECTED_INT_KIND — Choose integer kind

Description:

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 \(-1\).

Standard: Fortran 95 and later

Class: Transformational function

Syntax: RESULT = SELECTED_INT_KIND(R)

Arguments:

\( R \) Shall be a scalar and of type INTEGER.

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(i15) >= 10_k5**5-1
    print *, huge(i15) >= 10_k15**15-1
end program large_integers
```
9.242 SELECTED_REAL_KIND — Choose real kind

Description:
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 RADIX.

Standard: Fortran 95 and later, with RADIX Fortran 2008 or later
Class: Transformational function
Syntax: RESULT = SELECTED_REAL_KIND([P, R, RADIX])
Arguments:
P (Optional) shall be a scalar and of type INTEGER.
R (Optional) shall be a scalar and of type INTEGER.
RADIX (Optional) shall be a scalar and of type INTEGER.
Before Fortran 2008, at least one of the arguments R or P shall be present;
since Fortran 2008, they are assumed to be zero if absent.

Return value:
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 RADIX. If the 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

-1 if the processor does not support a real data type with a
   precision greater than or equal to P, but the R and RADIX require-
   ments can be fulfilled

-2 if the processor does not support a real type with an exponent
   range greater than or equal to R, but P and RADIX are fulfillable

-3 if RADIX but not P and R requirements
   are fulfillable

-4 if RADIX and either P or R requirements
   are fulfillable

-5 if there is no real type with the given RADIX

See also: Section 9.218 [PRECISION], page 242, Section 9.227 [RANGE], page 248,
Section 9.221 [RADIX], page 244

Example:
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
9.243 SET_EXPONENT — Set the exponent of the model

Description:

SET_EXPONENT(X, I) returns the real number whose fractional part is that of X and whose exponent part is I.

Standard: Fortran 95 and later

Class: Elemental function

Syntax: RESULT = SET_EXPONENT(X, I)

Arguments:

X Shall be of type REAL.
I Shall be of type INTEGER.

Return value:

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 FRACTION(X) * RADIX(X)**I.

Example:

PROGRAM test_setexp
  REAL :: x = 178.1387e-4
  INTEGER :: i = 17
  PRINT *, SET_EXPONENT(x, i), FRACTION(x) * RADIX(x)**i
END PROGRAM

9.244 SHAPE — Determine the shape of an array

Description:

Determines the shape of an array.

Standard: Fortran 95 and later, with KIND argument Fortran 2003 and later

Class: Inquiry function

Syntax: RESULT = SHAPE(SOURCE [, KIND])

Arguments:

SOURCE Shall be an array or scalar of any type. If SOURCE is a pointer it must be associated and allocatable arrays must be allocated.
KIND (Optional) An INTEGER initialization expression indicating the kind parameter of the result.

Return value:

An INTEGER array of rank one with as many elements as SOURCE has dimensions. The elements of the resulting array correspond to the extent of SOURCE along the respective dimensions. If SOURCE is a scalar, the result is the rank
one array of size zero. If KIND is absent, the return value has the default integer kind otherwise the specified 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: Section 9.232 [RESHAPE], page 250, Section 9.253 [SIZE], page 262

### 9.245 SHIFTA — Right shift with fill

**Description:**

SHIFTA returns a value corresponding to I with all of the bits shifted right by SHIFT places. If the absolute value of SHIFT is greater than BIT_SIZE(I), the 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.

**Standard:** Fortran 2008 and later

**Class:** Elemental function

**Syntax:**

```fortran
RESULT = SHIFTA(I, SHIFT)
```

**Arguments:**

- I
  - The type shall be INTEGER.
- SHIFT
  - The type shall be INTEGER.

**Return value:**

The return value is of type INTEGER and of the same kind as I.

See also: Section 9.246 [SHIFTL], page 258, Section 9.247 [SHIFTR], page 259

### 9.246 SHIFTL — Left shift

**Description:**

SHIFTL returns a value corresponding to I with all of the bits shifted left by SHIFT places. If the absolute value of SHIFT is greater than BIT_SIZE(I), the value is undefined. Bits shifted out from the left end are lost, and bits shifted in from the right end are set to 0.

**Standard:** Fortran 2008 and later

**Class:** Elemental function

**Syntax:**

```fortran
RESULT = SHIFTL(I, SHIFT)
```

**Arguments:**

- I
  - The type shall be INTEGER.
- SHIFT
  - The type shall be INTEGER.

**Return value:**

The return value is of type INTEGER and of the same kind as I.

See also: Section 9.245 [SHIFTA], page 258, Section 9.247 [SHIFTR], page 259
9.247 SHIFTR — Right shift

Description:

SHIFTR returns a value corresponding to $I$ with all of the bits shifted right by $SHIFT$ places. If the absolute value of $SHIFT$ is greater than BIT_SIZE($I$), the value is undefined. Bits shifted out from the right end are lost, and bits shifted in from the left end are set to 0.

Standard: Fortran 2008 and later

Class: Elemental function

Syntax: 

\[
\text{RESULT} = \text{SHIFTR}(I, \text{SHIFT})
\]

Arguments:

- \(I\) The type shall be INTEGER.
- \(SHIFT\) The type shall be INTEGER.

Return value:

The return value is of type INTEGER and of the same kind as $I$.

See also: Section 9.245 [SHIFTA], page 258, Section 9.246 [SHIFTL], page 258

9.248 SIGN — Sign copying function

Description:

SIGN($A$, $B$) returns the value of $A$ with the sign of $B$.

Standard: Fortran 77 and later

Class: Elemental function

Syntax: 

\[
\text{RESULT} = \text{SIGN}(A, B)
\]

Arguments:

- $A$ Shall be of type INTEGER or REAL
- $B$ Shall be of the same type and kind as $A$

Return value:

The kind of the return value is that of $A$ and $B$. If $B \geq 0$ then the result is ABS($A$), else it is -ABS($A$).

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.)
    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>f77, gnu</td>
</tr>
</tbody>
</table>
9.249 SIGNAL — Signal handling subroutine (or function)

Description:
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).

If SIGNAL is called as a subroutine and the STATUS argument is supplied, it is set to the value returned by signal(2).

Standard: GNU extension
Class: Subroutine, function
Syntax:
CALL SIGNAL(NUMBER, HANDLER [, STATUS])
STATUS = SIGNAL(NUMBER, HANDLER)

Arguments:
NUMBER Shall be a scalar integer, with INTENT(IN)
HANDLER Signal handler (INTEGER FUNCTION or SUBROUTINE) or dummy/global INTEGER scalar. INTEGER. It is INTENT(IN).
STATUS (Optional) STATUS shall be a scalar integer. It has INTENT(OUT).

Return value:
The SIGNAL function returns the value returned by signal(2).

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

9.250 SIN — Sine function

Description:
SIN(X) computes the sine of X.

Standard: Fortran 77 and later
Class: Elemental function
Syntax: RESULT = SIN(X)
Arguments:

\( X \) The type shall be \texttt{REAL} or \texttt{COMPLEX}.

Return value:

The return value has same type and kind as \( X \).

Example:

```
program test_sin
  real :: x = 0.0
  x = sin(x)
end program test_sin
```

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>REAL(4) X</td>
<td>REAL(4)</td>
<td>f77, gnu</td>
</tr>
<tr>
<td>DSIN(X)</td>
<td>REAL(8) X</td>
<td>REAL(8)</td>
<td>f95, gnu</td>
</tr>
<tr>
<td>CSIN(X)</td>
<td>COMPLEX(4) X</td>
<td>COMPLEX(4)</td>
<td>f95, gnu</td>
</tr>
<tr>
<td>ZSIN(X)</td>
<td>COMPLEX(8) X</td>
<td>COMPLEX(8)</td>
<td>f95, gnu</td>
</tr>
<tr>
<td>CDSIN(X)</td>
<td>COMPLEX(8) X</td>
<td>COMPLEX(8)</td>
<td>f95, gnu</td>
</tr>
</tbody>
</table>

See also: Inverse function: Section 9.19 \[ASIN\], page 119 Degrees function: Section 9.251 \[SIND\], page 261

9.251 SIND — Sine function, degrees

Description:

\texttt{SIND(X)} computes the sine of \( X \) in degrees.

This function is for compatibility only and should be avoided in favor of standard constructs wherever possible.

Standard: GNU Extension, enabled with \texttt{’-fdec-math’}.

Class: Elemental function

Syntax: \( \text{RESULT} = \text{SIND}(X) \)

Arguments:

\( X \) The type shall be \texttt{REAL} or \texttt{COMPLEX}.

Return value:

The return value has same type and kind as \( X \), and its value is in degrees.

Example:

```
program test_sind
  real :: x = 0.0
  x = sind(x)
end program test_sind
```

Specific names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Return type</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIND(X)</td>
<td>REAL(4) X</td>
<td>REAL(4)</td>
<td>GNU Extension</td>
</tr>
<tr>
<td>DSIND(X)</td>
<td>REAL(8) X</td>
<td>REAL(8)</td>
<td>GNU Extension</td>
</tr>
<tr>
<td>CSIND(X)</td>
<td>COMPLEX(4) X</td>
<td>COMPLEX(4)</td>
<td>GNU Extension</td>
</tr>
<tr>
<td>ZSIND(X)</td>
<td>COMPLEX(8) X</td>
<td>COMPLEX(8)</td>
<td>GNU Extension</td>
</tr>
<tr>
<td>CDSIND(X)</td>
<td>COMPLEX(8) X</td>
<td>COMPLEX(8)</td>
<td>GNU Extension</td>
</tr>
</tbody>
</table>
9.252 SINH — Hyperbolic sine function

Description:
SINH(X) computes the hyperbolic sine of X.

Standard: Fortran 95 and later, for a complex argument Fortran 2008 or later

Class: Elemental function

Syntax: RESULT = SINH(X)

Arguments:
X The type shall be REAL or COMPLEX.

Return value:
The return value has same type and kind as X.

Example:
```
program test_sinh
    real(8) :: 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>SINH(X)</td>
<td>REAL(4)</td>
<td>REAL(4)</td>
<td>Fortran 95 and later</td>
</tr>
<tr>
<td>DSINH(X)</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
<td>Fortran 95 and later</td>
</tr>
</tbody>
</table>

See also: Section 9.21 [ASINH], page 120

9.253 SIZE — Determine the size of an array

Description:
Determine the extent of ARRAY along a specified dimension DIM, or the total number of elements in ARRAY if DIM is absent.

Standard: Fortran 95 and later, with KIND argument Fortran 2003 and later

Class: Inquiry function

Syntax: RESULT = SIZE(ARRAY[, DIM [, KIND]])

Arguments:

| 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. |
Return value:
The return value is of type INTEGER and of kind KIND. If KIND is absent, the return value is of default integer kind.

Example:

```
PROGRAM test_size
  WRITE(*,*) SIZE((/ 1, 2 /)) ! 2
END PROGRAM
```

See also: Section 9.244 [SHAPE], page 257, Section 9.232 [RESHAPE], page 250

### 9.254 SIZEOF — Size in bytes of an expression

**Description:**
SIZEOF(X) calculates the number of bytes of storage the expression X occupies.

**Standard:** GNU extension

**Class:** Inquiry function

**Syntax:**

```
N = SIZEOF(X)
```

**Arguments:**
X The argument shall be of any type, rank or shape.

**Return value:**
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.

**Example:**

```
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: Section 9.57 [C_SIZEOF], page 143, Section 9.261 [STORAGE_SIZE], page 267

### 9.255 SLEEP — Sleep for the specified number of seconds

**Description:**
Calling this subroutine causes the process to pause for SECONDS seconds.

**Standard:** GNU extension
Class: Subroutine

Syntax: CALL SLEEP(SECONDS)

Arguments:

SECONDS The type shall be of default INTEGER.

Example:

program test_sleep
    call sleep(5)
end

9.256 SPACING — Smallest distance between two numbers of a given type

Description:
Determines the distance between the argument X and the nearest adjacent number of the same type.

Standard: Fortran 95 and later

Class: Elemental function

Syntax: RESULT = SPACING(X)

Arguments:
X Shall be of type REAL.

Return value:
The result is of the same type as the input argument 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: Section 9.233 [RRSPACING], page 251

9.257 SPREAD — Add a dimension to an array

Description:
Replicates a SOURCE array NCOPIES times along a specified dimension DIM.

Standard: Fortran 95 and later

Class: Transformational function

Syntax: RESULT = SPREAD(SOURCE, DIM, NCOPIES)

Arguments:

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.

**Return value:**
The result is an array of the same type as SOURCE and has rank n+1 where n equals the rank of SOURCE.

**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:** Section 9.282 [UNPACK], page 279

### 9.258 SQRT — Square-root function

**Description:**
\( \text{SQRT}(X) \) computes the square root of \( X \).

**Standard:** Fortran 77 and later

**Class:** Elemental function

**Syntax:**
\[
\text{RESULT} = \text{SQRT}(X)
\]

**Arguments:**
- \( X \) The type shall be REAL or COMPLEX.

**Return value:**
The return value is of type REAL or COMPLEX. The kind type parameter is the same as \( X \).

**Example:**
```
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 95 and later</td>
</tr>
<tr>
<td>D SQRT(X)</td>
<td>REAL(8) X</td>
<td>REAL(8)</td>
<td>Fortran 95 and later</td>
</tr>
<tr>
<td>CSQRT(X)</td>
<td>COMPLEX(4) X</td>
<td>COMPLEX(4)</td>
<td>Fortran 95 and later</td>
</tr>
<tr>
<td>Z SQRT(X)</td>
<td>COMPLEX(8) X</td>
<td>COMPLEX(8)</td>
<td>GNU extension</td>
</tr>
<tr>
<td>C DS QRT(X)</td>
<td>COMPLEX(8) X</td>
<td>COMPLEX(8)</td>
<td>GNU extension</td>
</tr>
</tbody>
</table>
9.259 SRAND — Reinitialize the random number generator

Description:
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.

Standard: GNU extension
Class: Subroutine
Syntax: CALL SRAND(SEED)
Arguments:
SEED Shall be a scalar INTEGER(kind=4).

Return value:
Does not return anything.

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: Section 9.223 [RAND], page 244, Section 9.226 [RANDOM_SEED], page 247, Section 9.225 [RANDOM_NUMBER], page 246

9.260 STAT — Get file status

Description:
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.

The elements that are obtained and stored in the array VALUES:
VALUES(1) Device ID
VALUES(2) Inode number
VALUES(3) File mode
VALUES(4) Number of links
VALUES(5) Owner’s uid
VALUES(6) Owner’s gid
VALUES(7) ID of device containing directory entry for file (0 if not available)
VALUES(8) File size (bytes)
VALUES(9) Last access time
VALUES(10) Last modification time
VALUES(11) Last file status change time
VALUES(12) Preferred I/O block size (-1 if not available)
VALUES(13)  Number of blocks allocated (-1 if not available)
Not all these elements are relevant on all systems. If an element is not relevant, it is returned as 0.
This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

Standard: GNU extension
Class: Subroutine, function
Syntax:

CALL STAT(NAME, VALUES [, STATUS])
STATUS = STAT(NAME, VALUES)

Arguments:

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.

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))
    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: Section 9.117 [FSTAT], page 184, to stat a link: Section 9.183 [LSTAT], page 221

9.261 STORAGE_SIZE — Storage size in bits

Description:

Returns the storage size of argument A in bits.

Standard: Fortran 2008 and later
Class: Inquiry function

Syntax:  
RESULT = STORAGE_SIZE(A [, KIND])

Arguments:  
A  Shall be a scalar or array of any type.
KIND (Optional) shall be a scalar integer constant expression.

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:  
Section 9.57 [C SIZEOF], page 143, Section 9.254 [SIZEOF], page 263

9.262 SUM — Sum of array elements

Description:  
Adds the elements of ARRAY along dimension DIM if the corresponding element in MASK is TRUE.

Standard:  Fortran 95 and later

Class: Transformational function

Syntax:  
RESULT = SUM(ARRAY[, MASK])
RESULT = SUM(ARRAY, DIM[, MASK])

Arguments:  
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.

Return value:  
The result is of the same type as ARRAY.
If DIM is absent, a scalar with the sum of all elements in ARRAY is returned. Otherwise, an array of rank n-1, where n equals the rank of ARRAY, and a shape similar to that of ARRAY with dimension DIM dropped is returned.

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:  
Section 9.220 [PRODUCT], page 243
9.263 SYMLNK — Create a symbolic link

Description:
Makes a symbolic link from file \textit{PATH1} to \textit{PATH2}. A null character (\texttt{CHAR(0)}) can be used to mark the end of the names in \textit{PATH1} and \textit{PATH2}; otherwise, trailing blanks in the file names are ignored. If the \textit{STATUS} argument is supplied, it contains 0 on success or a nonzero error code upon return; see \texttt{symlink(2)}. If the system does not supply \texttt{symlink(2)}, \texttt{ENOSYS} is returned.

This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

\textit{Standard}: GNU extension

\textit{Class}: Subroutine, function

\textit{Syntax}:
\begin{verbatim}
CALL SYMLNK(PATH1, PATH2 [, STATUS])
STATUS = SYMLNK(PATH1, PATH2)
\end{verbatim}

\textit{Arguments}:
\begin{itemize}
  \item \textit{PATH1} Shall be of default \texttt{CHARACTER} type.
  \item \textit{PATH2} Shall be of default \texttt{CHARACTER} type.
  \item \textit{STATUS} (Optional) Shall be of default \texttt{INTEGER} type.
\end{itemize}

\textit{See also}: Section 9.172 [LINK], page 216, Section 9.281 [UNLINK], page 279

9.264 SYSTEM — Execute a shell command

Description:
Passes the command \textit{COMMAND} to a shell (see \texttt{system(3)}). If argument \textit{STATUS} is present, it contains the value returned by \texttt{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.

This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

Note that the \texttt{system} function need not be thread-safe. It is the responsibility of the user to ensure that \texttt{system} is not called concurrently.

\textit{Standard}: GNU extension

\textit{Class}: Subroutine, function

\textit{Syntax}:
\begin{verbatim}
CALL SYSTEM(COMMAND [, STATUS])
STATUS = SYSTEM(COMMAND)
\end{verbatim}

\textit{Arguments}:
\begin{itemize}
  \item \textit{COMMAND} Shall be of default \texttt{CHARACTER} type.
  \item \textit{STATUS} (Optional) Shall be of default \texttt{INTEGER} type.
\end{itemize}

\textit{See also}: Section 9.100 [EXECUTE_COMMAND_LINE], page 172, which is part of the Fortran 2008 standard and should considered in new code for future portability.
9.265  SYSTEM_CLOCK — Time function

Description:
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.

COUNT_RATE is system dependent and can vary depending on the kind of the arguments. For kind=4 arguments (and smaller integer kinds), COUNT represents milliseconds, while for kind=8 arguments (and larger integer kinds), COUNT typically represents micro- or nanoseconds depending on resolution of the underlying platform clock. COUNT_MAX usually equals HUGE(COUNT_MAX). Note that the millisecond resolution of the kind=4 version implies that the COUNT will wrap around in roughly 25 days. In order to avoid issues with the wrap around and for more precise timing, please use the kind=8 version.

If there is no clock, or querying the clock fails, COUNT is set to -HUGE(COUNT), and COUNT_RATE and COUNT_MAX are set to zero.

When running on a platform using the GNU C library (glibc) version 2.16 or older, or a derivative thereof, the high resolution monotonic clock is available only when linking with the rt library. This can be done explicitly by adding the -lrt flag when linking the application, but is also done implicitly when using OpenMP.

On the Windows platform, the version with kind=4 arguments uses the GetTickCount function, whereas the kind=8 version uses QueryPerformanceCounter and QueryPerformanceCounterFrequency. For more information, and potential caveats, please see the platform documentation.

Standard:  Fortran 95 and later
Class:  Subroutine
Syntax:  CALL SYSTEM_CLOCK([COUNT, COUNT_RATE, COUNT_MAX])
Arguments:
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).

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
9.266 TAN — Tangent function

Description:
  TAN(X) computes the tangent of X.

Standard: Fortran 77 and later, for a complex argument Fortran 2008 or later

Class: Elemental function

Syntax:
  RESULT = TAN(X)

Arguments:
  X  The type shall be REAL or COMPLEX.

Return value:
  The return value has same type and kind as X, and its value is in radians.

Example:

```
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 95 and later</td>
</tr>
<tr>
<td>DTAN(X)</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
<td>Fortran 95 and later</td>
</tr>
</tbody>
</table>

See also: Inverse function: Section 9.23 [ATAN], page 122 Degrees function: Section 9.267 [TAND], page 271

9.267 TAND — Tangent function, degrees

Description:
  TAND(X) computes the tangent of X in degrees.

  This function is for compatibility only and should be avoided in favor of standard constructs wherever possible.

Standard: GNU Extension, enabled with ‘-fdec-math’.

Class: Elemental function

Syntax:
  RESULT = TAND(X)

Arguments:
  X  The type shall be REAL or COMPLEX.

Return value:
  The return value has same type and kind as X, and its value is in degrees.

Example:
```
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)</td>
<td>REAL(4)</td>
<td>GNU Extension</td>
</tr>
<tr>
<td>DTAND(X)</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
<td>GNU Extension</td>
</tr>
</tbody>
</table>

**See also:**
Inverse function: Section 9.24 [ATAND], page 122
Radians function: Section 9.266 [TAN], page 271

### 9.268 TANH — Hyperbolic tangent function

**Description:**

TANH(X) computes the hyperbolic tangent of X.

**Standard:** Fortran 77 and later, for a complex argument Fortran 2008 or later

**Class:** Elemental function

**Syntax:**

```
X = TANH(X)
```

**Arguments:**

- **X**
  - The type shall be REAL or COMPLEX.

**Return value:**

- 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 lies in the range $-1 \leq \tanh(x) \leq 1$.

**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)</td>
<td>REAL(4)</td>
<td>Fortran 95 and later</td>
</tr>
<tr>
<td>DTANH(X)</td>
<td>REAL(8)</td>
<td>REAL(8)</td>
<td>Fortran 95 and later</td>
</tr>
</tbody>
</table>

**See also:** Section 9.27 [ATANH], page 125

### 9.269 THIS_IMAGE — Function that returns the cosubscript index of this image

**Description:**

Returns the cosubscript for this image.

**Standard:** Fortran 2008 and later. With DISTANCE argument, Technical Specification (TS) 18508 or later

**Class:** Transformational function
Chapter 9: Intrinsic Procedures

Syntax:

\[
\text{RESULT} = \text{THIS\_IMAGE}() \\
\text{RESULT} = \text{THIS\_IMAGE}(	ext{DISTANCE}) \\
\text{RESULT} = \text{THIS\_IMAGE}(	ext{COARRAY [, DIM]})
\]

Arguments:

- **DISTANCE** (optional, intent(in)) Nonnegative scalar integer (not permitted together with 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.

Return value:

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).

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: Section 9.211 [NUM\_IMAGES], page 238, Section 9.147 [IMAGE\_INDEX], page 202

9.270 TIME — Time function

Description:

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 Section 9.81 [CTIME], page 160, Section 9.131 [GMTIME], page 192, and Section 9.184 [LTIME], page 222.

This intrinsic is not fully portable, such as to systems with 32-bit INTEGER types but supporting times wider than 32 bits. Therefore, the values returned
by this intrinsic might be, or become, negative, or numerically less than previous values, during a single run of the compiled program.

See Section 9.271 [TIME8], page 274, for information on a similar intrinsic that might be portable to more GNU Fortran implementations, though to fewer Fortran compilers.

**Standard:** GNU extension

**Class:** Function

**Syntax:** 
RESULT = TIME()

**Return value:**
The return value is a scalar of type INTEGER(4).

**See also:** Section 9.82 [DATE_AND_TIME], page 161, Section 9.81 [CTIME], page 160, Section 9.131 [GMTIME], page 192, Section 9.184 [LTIME], page 222, Section 9.193 [MCLOCK], page 227, Section 9.271 [TIME8], page 274

### 9.271 TIME8 — Time function (64-bit)

**Description:**
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 Section 9.81 [CTIME], page 160, Section 9.131 [GMTIME], page 192, and Section 9.184 [LTIME], page 222.

**Warning:** this intrinsic does not increase the range of the timing values over that returned by time(3). On a system with a 32-bit time(3), TIME8 will return a 32-bit value, even though it is converted to a 64-bit INTEGER(8) value. That means overflows of the 32-bit value can still occur. Therefore, the values returned by this intrinsic might be or become negative or numerically less than previous values during a single run of the compiled program.

**Standard:** GNU extension

**Class:** Function

**Syntax:** 
RESULT = TIME8()

**Return value:**
The return value is a scalar of type INTEGER(8).

**See also:** Section 9.82 [DATE_AND_TIME], page 161, Section 9.81 [CTIME], page 160, Section 9.131 [GMTIME], page 192, Section 9.184 [LTIME], page 222, Section 9.194 [MCLOCK8], page 228, Section 9.270 [TIME], page 273

### 9.272 TINY — Smallest positive number of a real kind

**Description:**
TINY(X) returns the smallest positive (non zero) number in the model of the type of X.

**Standard:** Fortran 95 and later
Class: Inquiry function
Syntax: RESULT = TINY(X)

Arguments:
X Shall be of type REAL.

Return value:
The return value is of the same type and kind as X

Example: See HUGE for an example.

9.273 TRAILZ — Number of trailing zero bits of an integer

Description:
TRAILZ returns the number of trailing zero bits of an integer.

Standard: Fortran 2008 and later
Class: Elemental function
Syntax: RESULT = TRAILZ(I)

Arguments:
I Shall be of type INTEGER.

Return value:
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).

Example:

```fortran
PROGRAM test_trailz
WRITE (*,*) TRAILZ(8) ! prints 3
END PROGRAM
```

See also: Section 9.48 [BIT_SIZE], page 138, Section 9.167 [LEADZ], page 213, Section 9.217 [POPPAR], page 242, Section 9.216 [POPCNT], page 241

9.274 TRANSFER — Transfer bit patterns

Description:
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.
This is approximately equivalent to the C concept of casting one type to another.

Standard: Fortran 95 and later
Class: Transformational function
Syntax: RESULT = TRANSFER(SOURCE, MOLD[, SIZE])

Arguments:
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.
Return value:
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.

If the bitwise representation of the result is longer than that of SOURCE, then the leading bits of the result correspond to those of SOURCE and any trailing bits are filled arbitrarily.
When the resulting bit representation does not correspond to a valid representation of a variable of the same type as MOLD, the results are undefined, and subsequent operations on the result cannot be guaranteed to produce sensible behavior. For example, it is possible to create LOGICAL variables for which VAR and .NOT. VAR both appear to be true.

Example:

```
PROGRAM test_transfer
  integer :: x = 2143289344
  print *, transfer(x, 1.0) ! prints "NaN" on i686
END PROGRAM
```

9.275 TRANSPOSE — Transpose an array of rank two

Description:
Transpose an array of rank two. Element (i, j) of the result has the value MATRIX(j, i), for all i, j.

Standard: Fortran 95 and later

Class: Transformational function

Syntax:
RESULT = TRANSPOSE(MATRIX)

Arguments:
MATRIX Shall be an array of any type and have a rank of two.

Return value:
The result has the same type as MATRIX, and has shape (/ m, n /) if MATRIX has shape (/ n, m /).

9.276 TRIM — Remove trailing blank characters of a string

Description:
Removes trailing blank characters of a string.

Standard: Fortran 95 and later

Class: Transformational function

Syntax:
RESULT = TRIM(STRING)

Arguments:
STRING Shall be a scalar of type CHARACTER.
Return value:
A scalar of type CHARACTER which length is that of STRING less the number of trailing blanks.

Example:

```
PROGRAM test_trim
  CHARACTER(len=10), PARAMETER :: s = "GFORTAN  
  WRITE(*,*) LEN(s), LEN(TRIM(s)) ! "10 8", with/without trailing blanks
END PROGRAM
```

See also: Section 9.9 [ADJUSTL], page 112, Section 9.10 [ADJUSTR], page 112

9.277 TTYNAM — Get the name of a terminal device.

Description:
Get the name of a terminal device. For more information, see ttymame(3).
This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

Standard: GNU extension
Class: Subroutine, function
Syntax:
```
CALL TTYNAM(UNIT, NAME)
NAME = TTYNAM(UNIT)
```
Arguments:
- UNIT shall be a scalar INTEGER.
- NAME shall be of type CHARACTER.

Example:

```
PROGRAM test_ttynam
  INTEGER :: unit
  DO unit = 1, 10
    IF (isatty(unit=unit)) write(*,*) ttynam(unit)
  END DO
END PROGRAM
```

See also: Section 9.158 [ISATTY], page 208

9.278 UBOUND — Upper dimension bounds of an array

Description:
Returns the upper bounds of an array, or a single upper bound along the DIM dimension.

Standard: Fortran 95 and later, with KIND argument Fortran 2003 and later
Class: Inquiry function
Syntax:
```
RESULT = UBOUND(ARRAY [, DIM [, KIND]])
```
Arguments:
- ARRAY shall be an array, of any type.
9.279 UCOBOUND — Upper codimension bounds of an array

Description:
Returns the upper cobounds of a coarray, or a single upper cobound along the DIM codimension.

Standard: Fortran 2008 and later
Class: Inquiry function
Syntax: RESULT = UCOBOUND(COARRAY [, DIM [, KIND]])
Arguments:
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.

Return value:
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.

See also: Section 9.165 [LBOUND], page 212, Section 9.166 [LCOBOUND], page 212

9.280 UMASK — Set the file creation mask

Description:
Sets the file creation mask to MASK. If called as a function, it returns the old value. If called as a subroutine and argument OLD if it is supplied, it is set to the old value. See umask(2).

Standard: GNU extension
Class: Subroutine, function
Syntax:
CALL UMASK(MASK [, OLD])
OLD = UMASK(MASK)
Arguments:

- MASK Shall be a scalar of type INTEGER.
- OLD (Optional) Shall be a scalar of type INTEGER.

9.281 UNLINK — Remove a file from the file system

Description:
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).

This intrinsic is provided in both subroutine and function forms; however, only one form can be used in any given program unit.

Standard: GNU extension
Class: Subroutine, function
Syntax:

CALL UNLINK(PATH [, STATUS])
STATUS = UNLINK(PATH)

Arguments:

- PATH Shall be of default CHARACTER type.
- STATUS (Optional) Shall be of default INTEGER type.

See also: Section 9.172 [LINK], page 216, Section 9.263 [SYMLNK], page 269

9.282 UNPACK — Unpack an array of rank one into an array

Description:
Store the elements of VECTOR in an array of higher rank.

Standard: Fortran 95 and later
Class: Transformational function
Syntax: RESULT = UNPACK(VECTOR, MASK, FIELD)

Arguments:

- 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.

Return value:
The resulting array corresponds to FIELD with TRUE elements of MASK replaced by values from VECTOR in array element order.

Example:
PROGRAM test_unpack
  integer :: vector(2) = (/1,1/)
  logical :: mask(4) = (/ .TRUE., .FALSE., .FALSE., .TRUE. /)
  integer :: field(2,2) = 0, unity(2,2)

  ! result: unity matrix
  unity = unpack(vector, reshape(mask, (/2,2/)), field)
END PROGRAM

See also:  Section 9.213 [PACK], page 239, Section 9.257 [SPREAD], page 264

9.283 VERIFY — Scan a string for characters not a given set

Description:
Verifies that all the characters in STRING belong to the set of characters in SET.
If BACK is either absent or equals FALSE, this function returns the position of the leftmost character of STRING that is not in SET. If BACK equals TRUE, the rightmost position is returned. If all characters of STRING are found in SET, the result is zero.

Standard: Fortran 95 and later, with KIND argument Fortran 2003 and later
Class: Elemental function
Syntax: RESULT = VERIFY(STRING, SET[, BACK [, KIND]])
Arguments:
  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.

Return value:
The return value is of type INTEGER and of kind KIND. If KIND is absent, the return value is of default integer kind.

Example:
PROGRAM test_verify
  WRITE(*,*) VERIFY("FORTRAN", "AO") ! 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:  Section 9.237 [SCAN], page 253, Section 9.148 [INDEX intrinsic], page 202

9.284 XOR — Bitwise logical exclusive OR

Description:
Bitwise logical exclusive or.
This intrinsic routine is provided for backwards compatibility with GNU Fortran 77. For integer arguments, programmers should consider the use of the
Section 9.145 [IEOR], page 201 intrinsic and for logical arguments the .NEQV. operator, which are both defined by the Fortran standard.

**Standard:** GNU extension  
**Class:** Function  
**Syntax:** \( \text{RESULT} = \text{XOR}(I, J) \)

**Arguments:**
- \( I \) 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.

**Return value:**
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 Section 9.149 [INT], page 203 occurred.

**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: Section 9.145 [IEOR], page 201
10 Intrinsic Modules

10.1 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 that a 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 has 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: Section 9.69 [COMPILER_OPTIONS], page 152 and Section 9.70 [COMPILER_VERSION], page 153.
10.2 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_ASSOCIATED`
- `C_F_Pointer`
- `C_F_PROCPOINTER`
- `C_FUNLOC`
- `C_LOC`
- `C_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_INTMAX_T</td>
<td>intmax_t</td>
<td></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>
</tbody>
</table>
REAL C_LONG_DOUBLE long double
REAL C_FLOAT128 __float128 Ext.
COMPLEX C_FLOAT_COMPLEX float _Complex
COMPLEX C_DOUBLE_COMPLEX double _Complex
COMPLEX C_LONG_DOUBLE_COMPLEX long double _Complex
REAL C_FLOAT128_COMPLEX __float128 _Complex Ext.
LOGICAL C_BOOL _Bool
CHARACTER C_CHAR char

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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_NULL_PTR</td>
<td>C_PTR</td>
</tr>
<tr>
<td>C_NULL_FUNPTR</td>
<td>C_FUNPTR</td>
</tr>
</tbody>
</table>

Both are equivalent to the value NULL in C.

10.3 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.
10.4 OpenMP Modules OMP_LIB and OMP_LIB_KINDS

*Standard:* OpenMP Application Program Interface v4.5

The OpenMP Fortran runtime library routines are provided both in a form of two Fortran 90 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 Section “Introduction” in GNU 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.

OMP_LIB_KINDS provides the following scalar default-integer named constants:

omp_lock_kind
omp_nest_lock_kind
omp_proc_bind_kind
omp_sched_kind

OMP_LIB provides the scalar default-integer named constant openmp_version with a value of the form yyyyymm, where yyyy is the year and mm the month of the OpenMP version; for OpenMP v4.5 the value is 201511.

The following scalar integer named constants of the kind omp_sched_kind:

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:

omp_proc_bind_false
omp_proc_bind_true
omp_proc_bind_master
omp_proc_bind_close
omp_proc_bind_spread

10.5 OpenACC Module OPENACC

*Standard:* OpenACC Application Programming Interface v2.0

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 Section “Introduction” in GNU 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 v2.0.

OPENACC provides the scalar default-integer named constant openacc_version with a value of the form yyyyymm, where yyyy is the year and mm the month of the OpenACC version; for OpenACC v2.0 the value is 201306.
Contributing

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.

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
- Thomas Koenig
- Asher Langton
- H. J. Lu
- Toon Moene
- Brooks Moses
- Andrew Pinski
- Tim Prince
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.

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.

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.

**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.

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


Everyone is permitted to copy and distribute verbatim copies of this
license document, but changing it is not allowed.

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 copy-
right 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 incom-
patible 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 pre-
cisely 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.

**TERMS AND CONDITIONS**

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.


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.

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 otherwise 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.


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.

   
   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.

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.

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.

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.

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.

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.


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.

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.

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. 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.

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.

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. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

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.

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

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 http://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 http://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 http://www.gnu.org/philosophy/why-not-lgpl.html.
GNU Free Documentation License

Version 1.3, 3 November 2008

http://fsf.org/

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. 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 non-commercially. 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. Any member 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 vari-
ous 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 individu-
ally 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.
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.

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 pub-
lished 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:

Copyright (C) year your name.
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 no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with...Texts.” line with this:

  with the Invariant Sections being list their titles, with the Front-Cover Texts being list, and with the Back-Cover Texts being list.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.
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.

Copyright © 1994 Free Software Foundation, Inc.
Verbatim copying and redistribution of this section is permitted without royalty; alteration is not permitted.
Option Index

gfortran’s command line options are indexed here without any initial ‘-’ or ‘--’. Where an option has both positive and negative forms (such as -foption and -fno-option), relevant entries in the manual are indexed under the most appropriate form; it may sometimes be useful to look up both forms.

A
A-predicate=answer ........................................ 14
A-predicate=answer ........................................ 14

B
backslash .................................................. 9

C
c-prototypes ............................................. 29
cpp ......................................................... 13
c .......................................................... 14
CC ........................................................... 15

d
D
dD .............................................................. 13
dI .............................................................. 13
dM .............................................................. 13
Dname ........................................................ 15
Dname=definition .......................................... 15
dN .............................................................. 13
dU .............................................................. 13

F
faggressive-function-elimination ...................... 28
fallign-commons ......................................... 28
fall-intrinsics ........................................... 8
fblas-matmul-limit ....................................... 27
fbounds-check ........................................... 25
fcheck ....................................................... 25
fcheck-array-temporaries ............................... 26
fcoarray ................................................... 24
fconvert=conversion .................................... 22
fcray-pointer ............................................ 10
fd-lines-as-code ......................................... 9
fd-lines-as-comments .................................... 9
fdec ........................................................ 9
fdec-include ............................................. 9
fdec-intrinsic-ints .................................... 9
fdec-math ................................................ 9
fdec-static ............................................... 9
fdec-structure ......................................... 9
fdefault-double-8 ...................................... 12
fdefault-integer-8 ..................................... 11
fdefault-real-8 .......................................... 11
fdefault-real-10 ....................................... 11
fdefault-real-16 ....................................... 11
fdefault-real-8 .......................................... 11
fdollar-ok ................................................ 9
dumpfortran-optimized ................................ 20
dumpfortran-original ................................... 20
dump-parse-tree .......................................... 20
fexternal-blas .......................................... 27
ff2c ........................................................ 23
ffixed-form .............................................. 8
ffixed-line-length-n .................................... 10
ffpe-summary=list ...................................... 21
ffpe-trap=list .......................................... 20
ffree-form ............................................... 8
ffree-line-length-n .................................... 10
fimplicit-none ......................................... 10
finit-character ......................................... 27
finit-derived ........................................... 27
finit-integer ............................................ 27
finit-local-zero ........................................ 27
finit-logical ............................................ 27
finit-real ............................................... 27
finit-matmul-limit .................................... 27
finteger-4-integer-8 .................................. 12
f intrinsic-modules-path dir ......................... 22
fxarray-constructor .................................... 26
fmax-errors=n .......................................... 16
fmax-identifier-length=n ............................. 10
fmax-stack-var-size ................................... 26
fmax-subrecord-length=length ...................... 22
fmodule-private ........................................ 10
fno-automatic .......................................... 23
fno-backtrace .......................................... 21
fno-protect-paren ...................................... 28
fno-underscoring ...................................... 23
fopenacc ............................................... 10
fopenmp ............................................... 11
fpack-derived .......................................... 26
fpad-source ............................................. 10
fpp ......................................................... 13
frange-check ............................................ 11
freal-4-real-10 ....................................... 12
freal-4-real-16 ....................................... 12
freal-4-real-8 ........................................ 12
freal-8-real-10 ....................................... 12
freal-8-real-16 ....................................... 12
freal-8-real-4 ........................................ 12
frealloc-lhs ............................................ 28
frecompile-marker=length ......................... 22
frecurse ............................................... 27
fremerge ............................................... 27
frepack-arrays ........................................ 26
# Keyword Index

<table>
<thead>
<tr>
<th>$</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>17</td>
</tr>
<tr>
<td>%</td>
<td>38</td>
</tr>
<tr>
<td>%LOC</td>
<td>57</td>
</tr>
<tr>
<td>%REF</td>
<td>57</td>
</tr>
<tr>
<td>%VAL</td>
<td>57</td>
</tr>
<tr>
<td>&amp;</td>
<td>77</td>
</tr>
<tr>
<td>&amp;</td>
<td>78</td>
</tr>
<tr>
<td>&amp;</td>
<td>79</td>
</tr>
<tr>
<td>&amp;</td>
<td>79</td>
</tr>
<tr>
<td>&amp;</td>
<td>79</td>
</tr>
<tr>
<td>&amp;</td>
<td>79</td>
</tr>
<tr>
<td>&amp;</td>
<td>79</td>
</tr>
<tr>
<td>&amp;</td>
<td>79</td>
</tr>
<tr>
<td>A</td>
<td>108</td>
</tr>
<tr>
<td>A</td>
<td>109</td>
</tr>
<tr>
<td>A</td>
<td>110</td>
</tr>
<tr>
<td>A</td>
<td>111</td>
</tr>
<tr>
<td>A</td>
<td>111</td>
</tr>
<tr>
<td>A</td>
<td>111</td>
</tr>
<tr>
<td>A</td>
<td>112</td>
</tr>
<tr>
<td>A</td>
<td>112</td>
</tr>
<tr>
<td>A</td>
<td>113</td>
</tr>
<tr>
<td>A</td>
<td>113</td>
</tr>
<tr>
<td>A</td>
<td>114</td>
</tr>
<tr>
<td>A</td>
<td>219</td>
</tr>
<tr>
<td>A</td>
<td>16</td>
</tr>
<tr>
<td>A</td>
<td>19, 28</td>
</tr>
<tr>
<td>A</td>
<td>16</td>
</tr>
<tr>
<td>A</td>
<td>38</td>
</tr>
<tr>
<td>A</td>
<td>38</td>
</tr>
<tr>
<td>A</td>
<td>38</td>
</tr>
<tr>
<td>A</td>
<td>116</td>
</tr>
<tr>
<td>A</td>
<td>233</td>
</tr>
<tr>
<td>A</td>
<td>116</td>
</tr>
<tr>
<td>A</td>
<td>115</td>
</tr>
<tr>
<td>A</td>
<td>218</td>
</tr>
<tr>
<td>A</td>
<td>219</td>
</tr>
<tr>
<td>A</td>
<td>225</td>
</tr>
<tr>
<td>A</td>
<td>225</td>
</tr>
<tr>
<td>A</td>
<td>229</td>
</tr>
<tr>
<td>A</td>
<td>229</td>
</tr>
<tr>
<td>A</td>
<td>232</td>
</tr>
<tr>
<td>A</td>
<td>116</td>
</tr>
<tr>
<td>A</td>
<td>117</td>
</tr>
<tr>
<td>A</td>
<td>118</td>
</tr>
<tr>
<td>A</td>
<td>111</td>
</tr>
<tr>
<td>A</td>
<td>120</td>
</tr>
<tr>
<td>A</td>
<td>125</td>
</tr>
<tr>
<td>A</td>
<td>57</td>
</tr>
<tr>
<td>A</td>
<td>152, 186, 187, 197</td>
</tr>
<tr>
<td>A</td>
<td>268</td>
</tr>
<tr>
<td>A</td>
<td>195</td>
</tr>
<tr>
<td>A</td>
<td>115, 118</td>
</tr>
<tr>
<td>A</td>
<td>25</td>
</tr>
<tr>
<td>A</td>
<td>250</td>
</tr>
<tr>
<td>A</td>
<td>228</td>
</tr>
<tr>
<td>A</td>
<td>115, 118</td>
</tr>
<tr>
<td>A</td>
<td>268</td>
</tr>
<tr>
<td>A</td>
<td>158</td>
</tr>
<tr>
<td>A</td>
<td>243</td>
</tr>
<tr>
<td>A</td>
<td>38</td>
</tr>
<tr>
<td>A</td>
<td>206</td>
</tr>
<tr>
<td>A</td>
<td>262</td>
</tr>
<tr>
<td>A</td>
<td>264</td>
</tr>
<tr>
<td>A</td>
<td>264</td>
</tr>
<tr>
<td>A</td>
<td>158</td>
</tr>
<tr>
<td>A</td>
<td>239</td>
</tr>
<tr>
<td>A</td>
<td>264, 279</td>
</tr>
<tr>
<td>A</td>
<td>52</td>
</tr>
<tr>
<td>A</td>
<td>226</td>
</tr>
<tr>
<td>A</td>
<td>230</td>
</tr>
<tr>
<td>A</td>
<td>212</td>
</tr>
<tr>
<td>A</td>
<td>227</td>
</tr>
<tr>
<td>A</td>
<td>228</td>
</tr>
<tr>
<td>A</td>
<td>231</td>
</tr>
<tr>
<td>A</td>
<td>243</td>
</tr>
<tr>
<td>A</td>
<td>158, 262</td>
</tr>
<tr>
<td>A</td>
<td>196</td>
</tr>
<tr>
<td>A</td>
<td>239</td>
</tr>
<tr>
<td>A</td>
<td>205</td>
</tr>
<tr>
<td>A</td>
<td>159</td>
</tr>
<tr>
<td>A</td>
<td>243</td>
</tr>
<tr>
<td>A</td>
<td>239</td>
</tr>
<tr>
<td>A</td>
<td>159</td>
</tr>
<tr>
<td>A</td>
<td>279</td>
</tr>
<tr>
<td>A</td>
<td>257</td>
</tr>
<tr>
<td>A</td>
<td>168</td>
</tr>
<tr>
<td>A</td>
<td>159</td>
</tr>
<tr>
<td>Function / Operation</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>array, size</td>
<td>262</td>
</tr>
<tr>
<td>array, sum</td>
<td>268</td>
</tr>
<tr>
<td>array, transmogrify</td>
<td>250</td>
</tr>
<tr>
<td>array, transpose</td>
<td>276</td>
</tr>
<tr>
<td>array, unpacking</td>
<td>279</td>
</tr>
<tr>
<td>array, upper bound</td>
<td>277</td>
</tr>
<tr>
<td>array, XOR</td>
<td>205</td>
</tr>
<tr>
<td>ASCII collating sequence</td>
<td>109, 194</td>
</tr>
<tr>
<td>ASIN</td>
<td>119</td>
</tr>
<tr>
<td>ASIND</td>
<td>119</td>
</tr>
<tr>
<td>ASINH</td>
<td>120</td>
</tr>
<tr>
<td>ASSOCIATED</td>
<td>120</td>
</tr>
<tr>
<td>association status</td>
<td>120</td>
</tr>
<tr>
<td>association status, C pointer</td>
<td>140</td>
</tr>
<tr>
<td>asynchronous I/O</td>
<td>47</td>
</tr>
<tr>
<td>ATAN</td>
<td>122</td>
</tr>
<tr>
<td>ATAN2</td>
<td>123</td>
</tr>
<tr>
<td>ATAN2D</td>
<td>124</td>
</tr>
<tr>
<td>ATAND</td>
<td>122</td>
</tr>
<tr>
<td>ATANH</td>
<td>125</td>
</tr>
<tr>
<td>Atomic subroutine, add</td>
<td>125</td>
</tr>
<tr>
<td>Atomic subroutine, ADD with fetch</td>
<td>128</td>
</tr>
<tr>
<td>Atomic subroutine, AND</td>
<td>126</td>
</tr>
<tr>
<td>Atomic subroutine, AND with fetch</td>
<td>129</td>
</tr>
<tr>
<td>Atomic subroutine, compare and swap</td>
<td>127</td>
</tr>
<tr>
<td>Atomic subroutine, define</td>
<td>127</td>
</tr>
<tr>
<td>Atomic subroutine, OR</td>
<td>131</td>
</tr>
<tr>
<td>Atomic subroutine, OR with fetch</td>
<td>130</td>
</tr>
<tr>
<td>Atomic subroutine, reference</td>
<td>132</td>
</tr>
<tr>
<td>Atomic subroutine, XOR</td>
<td>133</td>
</tr>
<tr>
<td>Atomic subroutine, XOR with fetch</td>
<td>130</td>
</tr>
<tr>
<td>ATOMIC_ADD</td>
<td>125</td>
</tr>
<tr>
<td>ATOMIC_AND</td>
<td>126</td>
</tr>
<tr>
<td>ATOMICDEFINE</td>
<td>127</td>
</tr>
<tr>
<td>ATOMIC_FETCH_ADD</td>
<td>128</td>
</tr>
<tr>
<td>ATOMIC_FETCH_AND</td>
<td>129</td>
</tr>
<tr>
<td>ATOMIC_FETCH_OR</td>
<td>130</td>
</tr>
<tr>
<td>ATOMIC_FETCH_XOR</td>
<td>130</td>
</tr>
<tr>
<td>ATOMIC_OR</td>
<td>131</td>
</tr>
<tr>
<td>ATOMIC_REF</td>
<td>132</td>
</tr>
<tr>
<td>ATOMIC_XOR</td>
<td>133</td>
</tr>
<tr>
<td>Authors</td>
<td>289</td>
</tr>
<tr>
<td>AUTOMATIC</td>
<td>63</td>
</tr>
<tr>
<td>BESJN</td>
<td>135</td>
</tr>
<tr>
<td>Bessel function, first kind</td>
<td>134, 135</td>
</tr>
<tr>
<td>Bessel function, second kind</td>
<td>135, 136</td>
</tr>
<tr>
<td>BESSEL_J0</td>
<td>134</td>
</tr>
<tr>
<td>BESSEL_J1</td>
<td>134</td>
</tr>
<tr>
<td>BESSEL_JN</td>
<td>135</td>
</tr>
<tr>
<td>BESSEL_Y0</td>
<td>135</td>
</tr>
<tr>
<td>BESSEL_Y1</td>
<td>136</td>
</tr>
<tr>
<td>BESSEL_YN</td>
<td>136</td>
</tr>
<tr>
<td>BESY0</td>
<td>135</td>
</tr>
<tr>
<td>BESY1</td>
<td>136</td>
</tr>
<tr>
<td>BESYN</td>
<td>136</td>
</tr>
<tr>
<td>BGE</td>
<td>137</td>
</tr>
<tr>
<td>BGT</td>
<td>138</td>
</tr>
<tr>
<td>BIAND</td>
<td>195</td>
</tr>
<tr>
<td>BIEOR</td>
<td>201</td>
</tr>
<tr>
<td>binary representation</td>
<td>241, 242</td>
</tr>
<tr>
<td>BIT_SIZE</td>
<td>138</td>
</tr>
<tr>
<td>BITTEST</td>
<td>139</td>
</tr>
<tr>
<td>bits set</td>
<td>241</td>
</tr>
<tr>
<td>bits, AND of array elements</td>
<td>195</td>
</tr>
<tr>
<td>bits, clear</td>
<td>197</td>
</tr>
<tr>
<td>bits, extract</td>
<td>198</td>
</tr>
<tr>
<td>bits, get</td>
<td>198</td>
</tr>
<tr>
<td>bits, merge</td>
<td>229</td>
</tr>
<tr>
<td>bits, move</td>
<td>234, 257</td>
</tr>
<tr>
<td>bits, negate</td>
<td>237</td>
</tr>
<tr>
<td>bits, number of</td>
<td>138</td>
</tr>
<tr>
<td>bits, OR of array elements</td>
<td>196</td>
</tr>
<tr>
<td>bits, set</td>
<td>199</td>
</tr>
<tr>
<td>bits, shift</td>
<td>209</td>
</tr>
<tr>
<td>bits, shift circular</td>
<td>209</td>
</tr>
<tr>
<td>bits, shift left</td>
<td>221, 258</td>
</tr>
<tr>
<td>bits, shift right</td>
<td>251, 258, 259</td>
</tr>
<tr>
<td>bits, testing</td>
<td>139</td>
</tr>
<tr>
<td>bits, unset</td>
<td>197</td>
</tr>
<tr>
<td>bits, XOR of array elements</td>
<td>205</td>
</tr>
<tr>
<td>bitwise comparison</td>
<td>137, 138, 139</td>
</tr>
<tr>
<td>bitwise logical and</td>
<td>116, 195</td>
</tr>
<tr>
<td>bitwise logical exclusive or</td>
<td>201, 280</td>
</tr>
<tr>
<td>bitwise logical not</td>
<td>237</td>
</tr>
<tr>
<td>bitwise logical or</td>
<td>204, 239</td>
</tr>
<tr>
<td>BJTEST</td>
<td>139</td>
</tr>
<tr>
<td>BKTEST</td>
<td>139</td>
</tr>
<tr>
<td>BLE</td>
<td>138</td>
</tr>
<tr>
<td>BLT</td>
<td>139</td>
</tr>
<tr>
<td>BMOD</td>
<td>232</td>
</tr>
<tr>
<td>BMVBITS</td>
<td>234</td>
</tr>
<tr>
<td>BNOT</td>
<td>237</td>
</tr>
<tr>
<td>bounds checking</td>
<td>25</td>
</tr>
<tr>
<td>BOZ literal constants</td>
<td>51</td>
</tr>
<tr>
<td>BSHFT</td>
<td>209</td>
</tr>
<tr>
<td>BSHFTC</td>
<td>209</td>
</tr>
<tr>
<td>BTES</td>
<td>139</td>
</tr>
</tbody>
</table>

B

BABS ........................................ 108
backslash .................................. 9
BACKSPACE ................................ 57
backtrace .................................. 21, 133
BACKTRACE .................................. 133
base 10 logarithm function ............. 219
BBCLR .................................... 197
BBBITS ................................... 198
BBSET .................................... 199
BBTEST ................................... 139
BESJ0 ................................... 134
BESJ1 ................................... 134

The GNU Fortran Compiler
C

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_ASSOCIATED</td>
<td>140</td>
</tr>
<tr>
<td>C_F_POINTER</td>
<td>141</td>
</tr>
<tr>
<td>C_F_PROCPOINTER</td>
<td>141</td>
</tr>
<tr>
<td>C_FUNLOC</td>
<td>142</td>
</tr>
<tr>
<td>C_LOC</td>
<td>143</td>
</tr>
<tr>
<td>C_SIZEOF</td>
<td>143</td>
</tr>
<tr>
<td>CABS</td>
<td>108</td>
</tr>
<tr>
<td>calling convention</td>
<td>23</td>
</tr>
<tr>
<td>CARRIAGECONTROL</td>
<td>65</td>
</tr>
<tr>
<td>CCOS</td>
<td>154</td>
</tr>
<tr>
<td>CCOSD</td>
<td>155</td>
</tr>
<tr>
<td>CDABS</td>
<td>108</td>
</tr>
<tr>
<td>CDCOS</td>
<td>154</td>
</tr>
<tr>
<td>CDCOSD</td>
<td>155</td>
</tr>
<tr>
<td>CDEXP</td>
<td>174</td>
</tr>
<tr>
<td>CDLG</td>
<td>218</td>
</tr>
<tr>
<td>CDSIN</td>
<td>260</td>
</tr>
<tr>
<td>CDSIND</td>
<td>261</td>
</tr>
<tr>
<td>CDSQRT</td>
<td>265</td>
</tr>
<tr>
<td>ceiling</td>
<td>117, 144</td>
</tr>
<tr>
<td>CEILING</td>
<td>144</td>
</tr>
<tr>
<td>CEXP</td>
<td>174</td>
</tr>
<tr>
<td>character kind</td>
<td>254</td>
</tr>
<tr>
<td>character set</td>
<td>9</td>
</tr>
<tr>
<td>CHAR</td>
<td>145</td>
</tr>
<tr>
<td>CHDIR</td>
<td>145</td>
</tr>
<tr>
<td>checking array temporaries</td>
<td>25</td>
</tr>
<tr>
<td>checking subscripts</td>
<td>25</td>
</tr>
<tr>
<td>CHMOD</td>
<td>146</td>
</tr>
<tr>
<td>clock ticks</td>
<td>227, 228, 270</td>
</tr>
<tr>
<td>CLOG</td>
<td>218</td>
</tr>
<tr>
<td>CMPLX</td>
<td>147</td>
</tr>
<tr>
<td>CD_BROADCAST</td>
<td>147</td>
</tr>
<tr>
<td>CD_MAX</td>
<td>148</td>
</tr>
<tr>
<td>CD_MIN</td>
<td>149</td>
</tr>
<tr>
<td>CD_REDUCE</td>
<td>150</td>
</tr>
<tr>
<td>CD_SUM</td>
<td>151</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_atomic_cas</td>
<td>102</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_atomic_define</td>
<td>101</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_atomic_op</td>
<td>103</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_atomic_ref</td>
<td>102</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_co_broadcast</td>
<td>103</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_co_max</td>
<td>104</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_co_min</td>
<td>104</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_co_reduce</td>
<td>105</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_co_sum</td>
<td>105</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_deregister</td>
<td>89</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_error_stop</td>
<td>101</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_error_stop_str</td>
<td>101</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_event_post</td>
<td>98</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_event_query</td>
<td>99</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_event_wait</td>
<td>98</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_fail_image</td>
<td>101</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_failed_images</td>
<td>87</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_finish</td>
<td>86</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_get</td>
<td>91</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_get_by_ref</td>
<td>94</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_image_status</td>
<td>87</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_init</td>
<td>86</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_is_present</td>
<td>90</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_lock</td>
<td>97</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_num_images</td>
<td>87</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_register</td>
<td>88</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_send</td>
<td>90</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_send_by_ref</td>
<td>93</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_sendget</td>
<td>92</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_sendget_by_ref</td>
<td>95</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_stopped_images</td>
<td>88</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_sync_all</td>
<td>99</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_sync_images</td>
<td>100</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_sync_memory</td>
<td>100</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_this_image</td>
<td>86</td>
</tr>
<tr>
<td>Coarray._gfortran_caf_unlock</td>
<td>97</td>
</tr>
<tr>
<td>coarray.IMAGE_INDEX</td>
<td>202</td>
</tr>
<tr>
<td>coarray.lower bound</td>
<td>212</td>
</tr>
<tr>
<td>coarray.NUM_IMAGES</td>
<td>238</td>
</tr>
<tr>
<td>coarray.THIS_IMAGE</td>
<td>272</td>
</tr>
<tr>
<td>coarray.upper bound</td>
<td>278</td>
</tr>
<tr>
<td>Coarrays</td>
<td>83</td>
</tr>
<tr>
<td>coarrays</td>
<td>24</td>
</tr>
<tr>
<td>code generation, conventions</td>
<td>23</td>
</tr>
<tr>
<td>collating sequence, ASCII</td>
<td>109, 194</td>
</tr>
<tr>
<td>Collectives, generic reduction</td>
<td>150</td>
</tr>
<tr>
<td>Collectives, maximal value</td>
<td>148</td>
</tr>
<tr>
<td>Collectives, minimal value</td>
<td>149</td>
</tr>
<tr>
<td>Collectives, sum of values</td>
<td>151</td>
</tr>
<tr>
<td>Collectives, value broadcasting</td>
<td>147</td>
</tr>
<tr>
<td>command line</td>
<td>172</td>
</tr>
<tr>
<td>command options</td>
<td>7</td>
</tr>
<tr>
<td>command-line arguments</td>
<td>152, 186, 187, 197</td>
</tr>
<tr>
<td>command-line arguments, number of</td>
<td>152, 197</td>
</tr>
<tr>
<td>COMMAND_ARGUMENT_COUNT</td>
<td>152</td>
</tr>
<tr>
<td>COMMON</td>
<td>68</td>
</tr>
<tr>
<td>compiler flags inquiry function</td>
<td>152</td>
</tr>
<tr>
<td>compiler, name and version</td>
<td>153</td>
</tr>
<tr>
<td>COMPILER_OPTIONS</td>
<td>152</td>
</tr>
<tr>
<td>COMPILER_VERSION</td>
<td>153</td>
</tr>
<tr>
<td>complex conjugate</td>
<td>154</td>
</tr>
<tr>
<td>Complex function</td>
<td>68</td>
</tr>
<tr>
<td>complex numbers, conversion to</td>
<td>147, 153, 162</td>
</tr>
<tr>
<td>complex numbers, imaginary part</td>
<td>113</td>
</tr>
<tr>
<td>complex numbers, real part</td>
<td>165, 248</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>153</td>
</tr>
<tr>
<td>Conditional compilation</td>
<td>2</td>
</tr>
<tr>
<td>CONJG</td>
<td>154</td>
</tr>
<tr>
<td>consistency, durability</td>
<td>45</td>
</tr>
<tr>
<td>Contributing</td>
<td>289</td>
</tr>
<tr>
<td>Contributors</td>
<td>289</td>
</tr>
<tr>
<td>conversion</td>
<td>17</td>
</tr>
<tr>
<td>conversion, to character</td>
<td>145</td>
</tr>
<tr>
<td>conversion, to complex</td>
<td>147, 153, 162</td>
</tr>
<tr>
<td>conversion, to integer</td>
<td>52, 194, 199, 203, 204, 220</td>
</tr>
<tr>
<td>conversion, to logical</td>
<td>52, 220</td>
</tr>
<tr>
<td>conversion, to real</td>
<td>162, 248</td>
</tr>
<tr>
<td>conversion, to string</td>
<td>160</td>
</tr>
<tr>
<td>CONVERT specifier</td>
<td>55</td>
</tr>
<tr>
<td>core, dump</td>
<td>107</td>
</tr>
<tr>
<td>COSD</td>
<td>55</td>
</tr>
<tr>
<td>COSH</td>
<td>56</td>
</tr>
<tr>
<td>cosine</td>
<td>54</td>
</tr>
<tr>
<td>cosine, degrees</td>
<td>55</td>
</tr>
<tr>
<td>cosine, hyperbolic</td>
<td>56</td>
</tr>
<tr>
<td>cosine, hyperbolic, inverse</td>
<td>111</td>
</tr>
<tr>
<td>cosine, inverse</td>
<td>110</td>
</tr>
<tr>
<td>cosine, inverse, degrees</td>
<td>111</td>
</tr>
<tr>
<td>CSOS</td>
<td>54</td>
</tr>
<tr>
<td>COTAN</td>
<td>57</td>
</tr>
<tr>
<td>cotangent</td>
<td>56</td>
</tr>
<tr>
<td>cotangent, degrees</td>
<td>57</td>
</tr>
<tr>
<td>DATAN</td>
<td>56</td>
</tr>
<tr>
<td>CDAT</td>
<td>58</td>
</tr>
<tr>
<td>CPU_TIME</td>
<td>158</td>
</tr>
<tr>
<td>Credits</td>
<td>289</td>
</tr>
<tr>
<td>CSHIFT</td>
<td>159</td>
</tr>
<tr>
<td>CSIN</td>
<td>260</td>
</tr>
<tr>
<td>CSIND</td>
<td>261</td>
</tr>
<tr>
<td>CSQRT</td>
<td>265</td>
</tr>
<tr>
<td>CTIME</td>
<td>160</td>
</tr>
<tr>
<td>current date</td>
<td>161, 175, 200</td>
</tr>
<tr>
<td>current time</td>
<td>161, 175, 210, 273, 274</td>
</tr>
</tbody>
</table>

**D**

| DABS                     |  108 |
| DACOS                   |  110 |
| DACOSD                  |  111 |
| DACOSH                  |  111 |
| DASIN                   |  119 |
| DASIND                  |  119 |
| DASINH                  |  120 |
| DATAN                   |  122 |
| DATAN2                   |  123 |
| DATAN2D                 |  124 |
| DATAND                  |  122 |
| DATANH                  |  125 |
| date, current           |  161, 175, 200 |
| DATE_AND_TIME           |  161 |
| DBESJ0                  |  134 |
| DBESJ1                  |  134 |
| DBESJN                  |  135 |
| DBESYO                  |  135 |
| DBESY1                  |  136 |
| DBESYN                  |  136 |
| DBLE                    |  162 |
| DCMPLX                  |  162 |
| DCNJG                   |  154 |
| DCOS                    |  154 |
| DCOSD                   |  155 |
| DCOSH                   |  156 |
| DCOTAN                  |  156 |
| DCOTAND                 |  157 |
| DDIM                    |  163 |

**debugging information options** : 20
**debugging, preprocessor** : 13
**DECODE** : 67
**delayed execution** : 114, 263
**DEXP** : 174
**DFLOAT** : 248
**DGAMMA** : 185
**dialect options** : 8
**DIGITS** : 163
**DIM** : 163
**DIMAG** : 113
**DINT** : 113
**directive, INCLUDE** : 21
**directory, options** : 21
**directory, search paths for inclusion** : 21
**division, modulo** : 233
**division, remainder** : 232
**DLOG** : 219
**DLOG10** : 219
**DMAX1** : 225
**DMIN1** : 229
**DMOD** : 232
**DNINT** : 117
**dot product** : 164
**DOT_PRODUCT** : 164
**DPROD** : 165
**DREAL** : 165
**DSHIFT** : 166
**DSHIFTR** : 166
**DSIGN** : 259
**DSIN** : 260
**DSIND** : 261
**DSINH** : 262
**DSQRT** : 265
**DTAN** : 271
**DTAND** : 271
**DTANH** : 272
**DTIME** : 167
**dummy argument, unused** : 19

**E**

| elapsed time            | 167, 253, 254 |
| Elimination of functions with identical argument lists | 28 |
| ENCODE                  | 67 |
| ENUM statement          | 38 |
| ENUMERATOR statement    | 38 |
| environment variable    | 29, 31, 189, 190 |
| EOF                     | 57 |
| EOSHIFT                 | 168 |
| EPSILON                 | 169 |
| ERF                     | 169 |
| ERFC                    | 170 |
| ERFC_SCALED             | 170 |
| error function          | 169 |
| error function, complementary | 170 |
| Error function, complementary, exponentially-scaled | 170 |
| Errors, limiting | 16 |
| Escape characters | 9 |
| ETIME | 171 |
| Euclidean distance | 194 |
| Euclidean vector norm | 236 |
| EVENT_QUERY | 172 |
| Events, EVENT_QUERY | 172 |
| EXECUTE_COMMAND_LINE | 172 |
| EXIT | 173 |
| Exponent | 67 |
| Exponential function | 174 |
| Exponential function, inverse | 218, 219 |
| EXPONENT | 174 |
| Expression size | 143, 263 |
| EXP | 174 |
| EXTENDS_TYPE_OF | 175 |
| Extensions | 49 |
| Extensions, implemented | 49 |
| Extensions, not implemented | 67 |
| Extra warnings | 17 |
| Floating point, set exponent | 257 |
| FDATE | 248 |
| FDOT | 248 |
| FDOTI | 248 |
| FDOTJ | 248 |
| FDOTK | 248 |
| Floor | 113, 178 |
| FLOOR | 178 |
| Flush | 179 |
| Flush statement | 38 |
| FNUM | 180 |
| Form feed whitespace | 64 |
| FORMAT | 68 |
| Fortran 77 | 3 |
| FPP | 2 |
| FPUT | 180 |
| FPUTC | 181 |
| FRACTION | 182 |
| FREE | 182 |
| Front-end optimization | 29 |
| FSEEK | 183 |
| FSTAT | 184 |
| FTELL | 184 |
| Function elimination | 19 |
| F2C calling convention | 23, 24 |
| Factorial function | 185 |
| FDATE | 175 |
| FDL, GNU Free Documentation License | 305 |
| FGET | 176 |
| FGETC | 177 |
| File format, fixed | 8, 10 |
| File format, free | 8, 10 |
| File operation, file number | 180 |
| File operation, flush | 179 |
| File operation, position | 183, 184 |
| File operation, read character | 176, 177 |
| File operation, seek | 183 |
| File operation, write character | 180, 181 |
| File system, access mode | 109 |
| File system, change access mode | 146 |
| File system, create link | 216, 269 |
| File system, file creation mask | 278 |
| File system, file status | 184, 221, 266 |
| File system, hard link | 216 |
| File system, remove file | 279 |
| File system, rename file | 249 |
| File system, soft link | 269 |
| File, symbolic link | 46 |
| File, unformatted sequential | 46 |
| FINDLOC | 178 |
| Flags inquiry function | 152 |
| Floating point, exponent | 174 |
| Floating point, fraction | 182 |
| Floating point, nearest different | 235 |
| Floating point, relative spacing | 251, 264 |
| Floating point, scale | 252 |
| Floating point, exponent | 257 |
| FDATE | 248 |
| FDOT | 248 |
| FDOTI | 248 |
| FDOTJ | 248 |
| FDOTK | 248 |
| Floor | 113, 178 |
| FLOOR | 178 |
| Flush | 179 |
| Flush statement | 38 |
| FNUM | 180 |
| Form feed whitespace | 64 |
| FORMAT | 68 |
| Fortran 77 | 3 |
| FPP | 2 |
| FPUT | 180 |
| FPUTC | 181 |
| FRACTION | 182 |
| FREE | 182 |
| Front-end optimization | 29 |
| FSEEK | 183 |
| FSTAT | 184 |
| FTELL | 184 |
| Function elimination | 19 |
| F2C calling convention | 23, 24 |
| Gamma function | 185 |
| Gamma function, logarithm of | 219 |
| GAMMA | 185 |
| GCC | 2 |
| Generating C prototypes from Fortran source code | 29 |
| GERROR | 186 |
| GET_COMMAND | 187 |
| GET_COMMAND_ARGUMENT | 187 |
| GET_ENVIRONMENT_VARIABLE | 190 |
| GETARG | 186 |
| GETCWD | 188 |
| GETENV | 189 |
| GETGID | 190 |
| GETLOG | 191 |
| GETPID | 191 |
| GETUID | 192 |
| GMTIME | 192 |
| GNU Compiler Collection | 2 |
| GNU Fortran command options | 7 |
| Hollerith constants | 53 |
| HOSTNM | 193 |
| HUGE | 193 |
| Hyperbolic cosine | 156 |
| Hyperbolic function, cosine | 156 |
| Hyperbolic function, cosine, inverse | 111 |
| Hyperbolic function, sine, inverse | 262 |
hyperbolic function, sine, inverse 120
hyperbolic function, tangent 272
hyperbolic function, tangent, inverse 125
hyperbolic sine 262
hyperbolic tangent 272
HYPOT 194

I
I/O item lists 51
I/O specifiers 65
IABS 108
IACHAR 194
IALL 195
IAND 195
IANY 196
IARGC 197
IBCLR 197
IBITS 198
IBSET 199
ICHAR 199
IDATE 200
IDIM 163
IDINT 203
IDNINT 236
IEEE, ISNAN 210
IEOR 201
IERROO 201
IFIX 203
IIABS 108
IIAND 195
IIBCLR 197
IIBITS 198
IIBSET 199
IIER 201
IIOR 204
IIISHFT 209
IIISHFTC 209
IMAGE_INDEX 202
images, cosubscript to image index conversion 202
images, index of this image 272
images, number of 238
IMAG 113
IMAPART 113
IMOD 232
IMPORT statement 38
IMVBITS 234
INCLUDE directive 21
inclusion, directory search paths for 21
INDEX 202
INOT 237
input/output, asynchronous 47
INT 203
INT2 204
INT8 204
integer kind 255
Interoperability 69
intrinsic 19
intrinsic Modules 283
intrinsics procedures 107
intrinsics, integer 62
intrinsics, math 64
intrinsics, trigonometric functions 64
inverse hyperbolic cosine 111
inverse hyperbolic sine 120
inverse hyperbolic tangent 25
IONMSG= specifier 38
IOR 204
IOSTAT, end of file 207
IOSTAT, end of record 208
IPARITY 205
IRAND 206
IS_IOSTAT_END 207
IS_IOSTAT_EOR 206, 208
ISATTY 208
ISHFT 209
ISHFTC 209
ISIGN 259
ISNAN 210
ISO_FORTRAN_ENV statement 38
ITIME 210

J
JIABS 108
JIAND 195
JIBCLR 197
JIBITS 198
JIBSET 199
JIOR 201
JISHFT 204
JISHFTC 209
JMOD 232
JMVBITS 234
JNOT 237

K
KIABS 108
KIAND 195
KIBCLR 197
KIBITS 198
KIBSET 199
KIEOR 201
KILL 211
kind 43, 211
kind, character 254
kind, integer 255
kind, old-style 49
kind, real 256
KIND 211
KIOR 204
KISHFT 209
KISHFTC 209
NEAREST ................................................. 235
NEW_LINE ................................................. 235
newline ................................................. 235
NINT .................................................. 236
norm, Euclidean ....................................... 236
NORM ................................................... 236
NOSHARED ............................................... 65
NOT ..................................................... 237
NULL ..................................................... 237
NUM.Images ............................................. 238

O
open, action ............................................. 46
OpenACC ................................................ 10, 56
OpenMP .................................................. 11, 56
operators, unary ....................................... 52
operators, xor .......................................... 65
options inquiry function ............................... 152
options, code generation ................................ 23
options, debugging ..................................... 20
options, dialect ......................................... 8
options, directory search ............................... 21
options, errors ......................................... 15
options, Fortran dialect ................................ 8
options, gfortran command ............................. 7
options, linking ......................................... 22
options, negative forms ................................ 7
options, preprocessor .................................. 13
options, real kind type promotion ..................... 12
options, run-time ....................................... 23
options, runtime ........................................ 22
options, warnings ...................................... 15
OR ......................................................... 239
output, newline .......................................... 235

P
PACK ..................................................... 239
PARAMETER ............................................. 67
Parity ..................................................... 240
parity ..................................................... 242
PARITY ................................................... 240
paths, search ............................................ 21, 22
PERM ...................................................... 241
pointer checking ....................................... 25
pointer, C address of pointers .......................... 141
pointer, C address of procedures ...................... 142
pointer, C association status ........................... 140
pointer, convert C to Fortran ........................... 141
pointer, Cray ............................................ 53
pointer, cray ............................................ 182, 223
pointer, disassociated ................................... 237
pointer, status ........................................... 120, 237
POPCNT ................................................. 241
POPPAR ............................................... 242
positive difference ..................................... 163
PRECISION ............................................. 242

Preprocessing .......................................... 2
preprocessing, assertion ................................ 14
preprocessing, define macros ......................... 15
preprocessing, include path ........................... 13, 14
preprocessing, keep comments ....................... 14, 15
preprocessing, no linemarkers ....................... 15
preprocessing, undefine macros ...................... 15
preprocessor ............................................. 13
preprocessor, debugging ................................ 13
preprocessor, disable .................................. 13
preprocessor, enable ................................... 13
preprocessor, include file handling ................. 2
preprocessor, working directory ...................... 13
PRESENT ............................................... 243
private .................................................. 10
procedure pointer, convert C to Fortran ............. 143
process ID .............................................. 191
product, double-precision .............................. 165
product, matrix ......................................... 224
product, vector ......................................... 164
PRODUCT ............................................... 243
program termination .................................... 173
program termination, with core dump ............... 107
PROTECTED statement .................................. 38

Q
Q exponent-letter ...................................... 51

R
radix, real .............................................. 256
RADIX ................................................... 244
random number generation ............................. 206, 244, 246
random number generation, initialization .......... 245
random number generation, seeding .................. 247, 266
RAN ...................................................... 244
RAI ....................................................... 244
RANDOM_INIT .......................................... 245
RANDOM_NUMBER ....................................... 246
RANDOM_SEED .......................................... 247
range checking .......................................... 25
RANGE ................................................... 248
rank ....................................................... 248
RANK ..................................................... 248
re-association of parenthesized expressions ........ 28
read character, stream mode ......................... 176, 177
READONLY ............................................... 65
real kind ............................................... 256
real number, exponent ................................ 174
real number, fraction ................................ 182
real number, nearest different ....................... 235
real number, relative spacing ......................... 251, 264
real number, scale .................................... 252
real number, set exponent ............................ 257
Reallocate the LHS in assignments ................... 28
Reallocate the LHS in assignments, notification ..... 19
S

SAME_TYPE_A S 252
SAVE statement 23
SCALE 252
SCAN 253
search path 21
search paths, for included files 21
SECONDS 253
SECOND 254
seeded a random number generator 247, 266
SELECTED_CHAR_KIND 254
SELECTED_INT_KIND 255
SELECTED_REAL_KIND 256
sequential, unformatted 46
SET_EXPONENT 257
SHAPE 257
SHARE 65
SHARED 65
shift, left 166, 258
shift, right 166, 259
shift, right with fill 258
SHIFTA 258
SHIFTL 258
SHIFTR 259
SHORT 204
sign copying 259
SIGN 259
SIGNAL 260
SIND 261
sine 260
sine, degrees 261
sine, hyperbolic 262
sine, hyperbolic, inverse 120
sine, inverse 119
sine, inverse, degrees 119
SIN 260
SINH 262
size of a variable, in bits 138
size of an expression 143, 263
SIZE 262
SIZEOF 263
SLEEP 263
SNGL 248
SPACING 264
SPREAD 264
SORT 265
SPRINT 265
square-root 265
SRAND 266
Standard 4
statement, ENUM 38
statement, ENUMERATOR 38
statement, FLUSH 38
statement, IMPORT 38
statement, ISO_FORTRAN_ENV 38
statement, PROTECTED 38
statement, SAVE 23
statement, USE, INTRINSIC 38
statement, VALUE 38
statement, VOLATILE 38
STAT 266
STATIC 63
storage size 267
STORAGE_SIZE 267
stream mode, read character 176, 177
stream mode, write character 180, 181
STREAM I/O 38
string, adjust left 112
string, adjust right 112
string, comparison 214, 215, 216, 217
string, concatenate 250
string, find missing set 280
string, find non-blank character 217
string, find substring 202
string, length 213
string, length, without trailing whitespace 214
string, remove trailing whitespace 276
string, repeat 250
strings, varying length 4
structure packing 26
STRUCTURE 57
subrecord 46
substring checking 25
substring position 202
sum array elements 268
SUN 268
suppressing warnings 15
symbol names 9
symbol names, transforming 23, 24
symbol names, underscores 23, 24
SYMLNK 269
syntax checking 16
system, error handling 186, 201, 241
system, group ID 190
system, host name 193
system, login name ........................................ 191
system, process ID ........................................ 191
system, signal handling .................................. 260
system, system call ...................................... 172, 269
system, terminal ........................................ 208, 277
system, user ID ........................................... 192
system, working directory .............................. 145, 188
SYSTEM ................................................... 269
SYSTEM_CLOCK .......................................... 270

T

tabulators .................................................... 18
tangent ..................................................... 271
tangent, degrees .......................................... 271
tangent, hyperbolic ....................................... 272
tangent, hyperbolic, inverse ......................... 125
tangent, inverse .......................................... 122, 123
tangent, inverse, degrees ......................... 122, 124
TAN .......................................................... 271
TAND .......................................................... 271
TANH .......................................................... 272
terminate program ........................................ 173
terminate program, with core dump ............... 107
THIS_IMAGE ................................................. 272
thread-safety, threads .................................. 44
time, clock ticks ......................................... 227, 228, 270
time, conversion to GMT info ....................... 192
time, conversion to local time info ................. 222
time, conversion to string ............................ 160
time, current ............................................ 161, 175, 210, 273, 274
time, elapsed ............................................. 158, 167, 171, 253, 254
TIME ........................................................ 273
TIMES ...................................................... 274
TINY ........................................................ 274
TR 15581 ................................................... 38
trace ......................................................... 21
TRAILZ ....................................................... 275
TRANSFER .................................................. 275
transforming symbol names ......................... 23, 24
transpose .................................................. 276
TRANSPOSE ................................................ 276
trigonometric function, cosine .................... 154
trigonometric function, cosine, degrees .......... 155
trigonometric function, cosine, inverse .......... 110
trigonometric function, cosine, inverse, degrees 
   ......................................................... 111
trigonometric function, cotangent ............... 156
trigonometric function, cotangent, degrees .... 157
trigonometric function, sine ......................... 260
trigonometric function, sine, degrees ........... 261
trigonometric function, sine, inverse .......... 119
trigonometric function, sine, inverse, degrees . 119
trigonometric function, tangent .................. 271
trigonometric function, tangent, degrees ....... 271
trigonometric function, tangent, inverse ........ 122, 123
trigonometric function, tangent, inverse, degrees 
   ......................................................... 122, 124

TRIM ....................................................... 276
TTYNAM .................................................... 277
type alias print ....................................... 64
type cast .................................................. 275

U

UBOUND .................................................... 277
UCBOUND .................................................. 278
UMASK ...................................................... 278
underflow ............................................... 18
underscore ............................................... 23, 24
unformatted sequential ............................ 46
UNION ..................................................... 60
UNLINK ..................................................... 279
UNPACK .................................................... 279
unused dummy argument ............................. 19
unused parameter ....................................... 19
USE, INTRINSIC statement ......................... 38
user id ..................................................... 192

V

VALUE statement ....................................... 38
variable attributes .................................... 63
Varying length character strings .................. 4
Varying length strings ................................ 4
vector product ......................................... 164
VERIFY .................................................... 280
version of the compiler .............................. 153
VOLATILE .................................................. 68
VOLATILE statement ................................. 38

W

warning, C binding type .............................. 17
warnings, aliasing .................................. 16
warnings, alignment of COMMON blocks ......... 19
warnings, all ........................................... 16
warnings, ampersand ................................. 17
warnings, argument mismatch ...................... 17
warnings, array temporaries ...................... 17
warnings, character truncation ................. 17
warnings, conversion ............................... 17
warnings, division of integers .................. 18
warnings, extra ....................................... 17
warnings, function elimination .................. 19
warnings, implicit interface ...................... 18
warnings, implicit procedure ..................... 18
warnings, integer division ......................... 18
warnings, interface mismatch .................... 17
warnings, intrinsic ................................... 19
warnings, intrinsics of other standards ........ 18
warnings, line truncation ......................... 17
warnings, loop interchange ......................... 17
warnings, non-standard intrinsics ............... 18
warnings, parameter mismatch ................... 17
warnings, q exponent-letter ..................... 18
<table>
<thead>
<tr>
<th>Keyword</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>warnings, suppressing</td>
<td>15</td>
</tr>
<tr>
<td>warnings, suspicious code</td>
<td>18</td>
</tr>
<tr>
<td>warnings, tabs</td>
<td>18</td>
</tr>
<tr>
<td>warnings, to errors</td>
<td>20</td>
</tr>
<tr>
<td>warnings, undefined do loop</td>
<td>18</td>
</tr>
<tr>
<td>warnings, underflow</td>
<td>18</td>
</tr>
<tr>
<td>warnings, unused dummy argument</td>
<td>19</td>
</tr>
<tr>
<td>warnings, unused parameter</td>
<td>19</td>
</tr>
<tr>
<td>warnings, use statements</td>
<td>180, 181</td>
</tr>
<tr>
<td>write character, stream mode</td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td>280</td>
</tr>
<tr>
<td>XOR reduction</td>
<td>240</td>
</tr>
<tr>
<td>Z</td>
<td></td>
</tr>
<tr>
<td>ZABS</td>
<td>108</td>
</tr>
<tr>
<td>ZCOS</td>
<td>154</td>
</tr>
<tr>
<td>ZCOSD</td>
<td>155</td>
</tr>
<tr>
<td>zero bits</td>
<td>213, 275</td>
</tr>
<tr>
<td>ZEXP</td>
<td>174</td>
</tr>
<tr>
<td>ZLOG</td>
<td>218</td>
</tr>
<tr>
<td>ZSIN</td>
<td>260</td>
</tr>
<tr>
<td>ZSIND</td>
<td>261</td>
</tr>
<tr>
<td>ZSQRT</td>
<td>265</td>
</tr>
</tbody>
</table>