Contributed by James Craig Burley (burley@gnu.org). Inspired by a first pass at translating 'g77-0.5.16/f/DOC' that was contributed to Craig by David Ronis (ronis@onsager.chem.mcgill.ca).

Using GNU Fortran

James Craig Burley

Last updated 1998-01-11

for version 0.5.21

For GNU Fortran Version 0.5.21*

Published by the Free Software Foundation
59 Temple Place - Suite 330
Boston, MA 02111-1307, USA

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) 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 this service 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 make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. 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.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute 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 and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions

for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore,

by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), 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 distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the 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 a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a

version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, 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.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE 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.

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 convey 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) 19*yy* *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 2 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, write to the Free Software
> Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) 19*yy* *name of author*
> Gnomovision 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, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program
> 'Gnomovision' (which makes passes at compilers) written by James Hacker.
>
> *signature of Ty Coon*, 1 April 1989
> Ty Coon, President of Vice

This 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 Library General Public License instead of this License.

Contributors to GNU Fortran

In addition to James Craig Burley, who wrote the front end, many people have helped create and improve GNU Fortran.

- The packaging and compiler portions of GNU Fortran are based largely on the GNU CC compiler. See section "Contributors to GNU CC" in *Using and Porting GNU CC*, for more information.
- The run-time library used by GNU Fortran is a repackaged version of the `libf2c` library (combined from the `libF77` and `libI77` libraries) provided as part of `f2c`, available for free from `netlib` sites on the Internet.
- Cygnus Support and The Free Software Foundation contributed significant money and/or equipment to Craig's efforts.
- The following individuals served as alpha testers prior to `g77`'s public release. This work consisted of testing, researching, sometimes debugging, and occasionally providing small amounts of code and fixes for `g77`, plus offering plenty of helpful advice to Craig:

  Jonathan Corbet

  Dr. Mark Fernyhough

  Takafumi Hayashi (The University of AIzu)—`takafumi@u-aizu.ac.jp`

  Kate Hedstrom

  Michel Kern (INRIA and Rice University)—`Michel.Kern@inria.fr`

  Dr. A. O. V. Le Blanc

  Dave Love

  Rick Lutowski

  Toon Moene

  Rick Niles

  Derk Reefman

  Wayne K. Schroll

  Bill Thorson

  Pedro A. M. Vazquez

  Ian Watson
- Scott Snyder (`snyder@d0sgif.fnal.gov`) provided the patch to add rudimentary support for `INTEGER*1`, `INTEGER*2`, and `LOGICAL*1`. This inspired Craig to add further support, even though the resulting support would still be incomplete, because version 0.6 is still a ways off.
- David Ronis (`ronis@onsager.chem.mcgill.ca`) inspired and encouraged Craig to rewrite the documentation in texinfo format by contributing a first pass at a translation of the old '`g77-0.5.16/f/DOC`' file.
- Toon Moene (`toon@moene.indiv.nluug.nl`) performed some analysis of generated code as part of an overall project to improve `g77` code generation to at least be as good as `f2c` used in conjunction with `gcc`. So far, this has resulted in the three, somewhat experimental, options added by `g77` to the `gcc` compiler and its back end.

- John Carr (`jfc@mit.edu`) wrote the alias analysis improvements.
- Thanks to Mary Cortani and the staff at Craftwork Solutions (`support@craftwork.com`) for all of their support.
- Many other individuals have helped debug, test, and improve `g77` over the past several years, and undoubtedly more people will be doing so in the future. If you have done so, and would like to see your name listed in the above list, please ask! The default is that people wish to remain anonymous.

1 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 C compiler 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.

2 Funding GNU Fortran

Work on GNU Fortran is still being done mostly by its author, James Craig Burley (`burley@gnu.org`), who is a volunteer for, not an employee of, the Free Software Foundation (FSF). As with other GNU software, funding is important because it can pay for needed equipment, personnel, and so on.

The FSF provides information on the best way to fund ongoing development of GNU software (such as GNU Fortran) in documents such as the "GNUS Bulletin". Email `gnu@prep.ai.mit.edu` for information on funding the FSF.

To fund specific GNU Fortran work in particular, the FSF might provide a means for that, but the FSF does not provide direct funding to the author of GNU Fortran to continue his work. The FSF has employee salary restrictions that can be incompatible with the financial needs of some volunteers, who therefore choose to remain volunteers and thus be able to be free to do contract work and otherwise make their own schedules for doing GNU work.

Still, funding the FSF at least indirectly benefits work on specific projects like GNU Fortran because it ensures the continuing operation of the FSF offices, their workstations, their network connections, and so on, which are invaluable to volunteers. (Similarly, hiring Cygnus Support can help a project like GNU Fortran—Cygnus has been a long-time donor of equipment usage to the author of GNU Fortran, and this too has been invaluable—See [Contributors], page 9.)

Currently, the only way to directly fund the author of GNU Fortran in his work on that project is to hire him for the work you want him to do, or donate money to him. Several people have done this already, with the result that he has not needed to immediately find contract work on a few occasions. If more people did this, he would be able to plan on not doing contract work for many months and could thus devote that time to work on projects (such as the planned changes for 0.6) that require longer timeframes to complete. For the latest information on the status of the author, do *`finger -l burley@gnu.org`* on a UNIX system (or any system with a command like UNIX `finger`).

Another important way to support work on GNU Fortran is to volunteer to help out. Work is needed on documentation, testing, porting to various machines, and in some cases, coding (although major changes planned for version 0.6 make it difficult to add manpower to this area). Email `fortran@gnu.org` to volunteer for this work.

See Chapter 1 [Funding Free Software], page 11, for more information.

3 Protect Your Freedom—Fight "Look And Feel"

To preserve the ability to write free software, including replacements for proprietary software, authors must be free to replicate the user interface to which users of existing software have become accustomed.

See section "Protect Your Freedom—Fight "Look And Feel"" in *Using and Porting GNU CC*, for more information.

4 Getting Started

If you don't need help getting started reading the portions of this manual that are most important to you, you should skip this portion of the manual.

If you are new to compilers, especially Fortran compilers, or new to how compilers are structured under UNIX and UNIX-like systems, you'll want to see Chapter 5 [What is GNU Fortran?], page 19.

If you are new to GNU compilers, or have used only one GNU compiler in the past and not had to delve into how it lets you manage various versions and configurations of `gcc`, you should see Chapter 6 [G77 and GCC], page 23.

Everyone except experienced `g77` users should see Chapter 7 [Invoking G77], page 25.

If you're acquainted with previous versions of `g77`, you should see Chapter 8 [News], page 47. Further, if you've actually used previous versions of `g77`, especially if you've written or modified Fortran code to be compiled by previous versions of `g77`, you should see Chapter 9 [Changes], page 65.

If you intend to write or otherwise compile code that is not already strictly conforming ANSI FORTRAN 77—and this is probably everyone—you should see Chapter 10 [Language], page 71.

If you don't already have `g77` installed on your system, you must see Chapter 15 [Installation], page 213.

If you run into trouble getting Fortran code to compile, link, run, or work properly, you might find answers if you see Chapter 16 [Debugging and Interfacing], page 239, see Chapter 17 [Collected Fortran Wisdom], page 251, and see Chapter 18 [Trouble], page 265. You might also find that the problems you are encountering are bugs in `g77`—see Chapter 20 [Bugs], page 293, for information on reporting them, after reading the other material.

If you need further help with `g77`, or with freely redistributable software in general, see Chapter 21 [Service], page 303.

If you would like to help the `g77` project, see Chapter 2 [Funding GNU Fortran], page 13, for information on helping financially, and see Chapter 23 [Projects], page 307, for information on helping in other ways.

If you're generally curious about the future of `g77`, see Chapter 23 [Projects], page 307. If you're curious about its past, see [Contributors], page 9, and see Chapter 2 [Funding GNU Fortran], page 13.

To see a few of the questions maintainers of `g77` have, and that you might be able to answer, see Chapter 19 [Open Questions], page 291.

5 What is GNU Fortran?

GNU Fortran, or `g77`, is designed initially as a free replacement for, or alternative to, the UNIX `f77` command. (Similarly, `gcc` is designed as a replacement for the UNIX `cc` command.)

`g77` also is designed to fit in well with the other fine GNU compilers and tools.

Sometimes these design goals conflict—in such cases, resolution often is made in favor of fitting in well with Project GNU. These cases are usually identified in the appropriate sections of this manual.

As compilers, `g77`, `gcc`, and `f77` share the following characteristics:

- They read a user's program, stored in a file and containing instructions written in the appropriate language (Fortran, C, and so on). This file contains *source code*.
- They 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. These instructions are called *machine code*—code designed to be efficiently translated and processed by a machine such as a computer. Humans usually aren't as good writing machine code as they are at writing Fortran or C, because it is easy to make tiny mistakes writing machine code. When writing Fortran or C, it is easy to make big mistakes.
- They 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 `gdb`).
- They locate and gather machine code already generated to perform actions requested by statements in the user's program. This machine code is organized into *libraries* and is located and gathered during the *link* phase of the compilation process. (Linking often is thought of as a separate step, because it can be directly invoked via the `ld` command. However, the `g77` and `gcc` commands, as with most compiler commands, automatically perform the linking step by calling on `ld` directly, unless asked to not do so by the user.)
- They attempt to diagnose cases where the user's program contains incorrect usages of the language. The *diagnostics* produced by the compiler indicate the problem and the location in the user's source file where the problem was first noticed. The user can use this information to locate and fix the problem. (Sometimes an incorrect usage of the language leads to a situation where the compiler can no longer make any sense of what follows—while a human might be able to—and thus ends up complaining about many "problems" it encounters that, in fact, stem from just one problem, usually the first one reported.)
- They attempt to diagnose cases where the user's program contains a correct usage of the language, but instructs the computer to do something questionable. These diagnostics often are in the form of *warnings*, instead of the *errors* that indicate incorrect usage of the language.

How these actions are performed is generally under the control of the user. Using command-line options, the user can specify how persnickety the compiler is to be regarding the program (whether to diagnose questionable usage of the language), how much time to spend making the generated machine code run faster, and so on.

`g77` consists of several components:

- A modified version of the `gcc` command, which also might be installed as the system's `cc` command. (In many cases, `cc` refers to the system's "native" C compiler, which might be a non-GNU compiler, or an older version of `gcc` considered more stable or that is used to build the operating system kernel.)
- The `g77` command itself, which also might be installed as the system's `f77` command.
- The `libf2c` run-time library. This library contains the machine code needed to support capabilities of the Fortran language that are not directly provided by the machine code generated by the `g77` compilation phase.
- The compiler itself, internally named `f771`.

  Note that `f771` does not generate machine code directly—it generates *assembly code* that is a more readable form of machine code, leaving the conversion to actual machine code to an *assembler*, usually named `as`.

`gcc` is often thought of as "the C compiler" only, but it does more than that. Based on command-line options and the names given for files on the command line, `gcc` determines which actions to perform, including preprocessing, compiling (in a variety of possible languages), assembling, and linking.

For example, the command '`gcc foo.c`' *drives* the file '`foo.c`' through the preprocessor `cpp`, then the C compiler (internally named `cc1`), then the assembler (usually `as`), then the linker (`ld`), producing an executable program named '`a.out`' (on UNIX systems).

As another example, the command '`gcc foo.cc`' would do much the same as '`gcc foo.c`', but instead of using the C compiler named `cc1`, `gcc` would use the C++ compiler (named `cc1plus`).

In a GNU Fortran installation, `gcc` recognizes Fortran source files by name just like it does C and C++ source files. It knows to use the Fortran compiler named `f771`, instead of `cc1` or `cc1plus`, to compile Fortran files.

Non-Fortran-related operation of `gcc` is generally unaffected by installing the GNU Fortran version of `gcc`. However, without the installed version of `gcc` being the GNU Fortran version, `gcc` will not be able to compile and link Fortran programs—and since `g77` uses `gcc` to do most of the actual work, neither will `g77`!

The `g77` command is essentially just a front-end for the `gcc` command. Fortran users will normally use `g77` instead of `gcc`, because `g77` knows how to specify the libraries needed to link with Fortran programs (`libf2c` and `lm`). `g77` can still compile and link programs and source files written in other languages, just like `gcc`.

The command '`g77 -v`' is a quick way to display lots of version information for the various programs used to compile a typical preprocessed Fortran source file—this produces much more output than '`gcc -v`' currently does. (If it produces an error message near the end of the output—diagnostics from the linker, usually `ld`—you might have an out-of-date `libf2c` that improperly handles complex arithmetic.) In the output of this command, the line beginning '`GNU Fortran Front End`' identifies the version number of GNU Fortran; immediately preceding that line is a line identifying the version of `gcc` with which that version of `g77` was built.

The `libf2c` library is distributed with GNU Fortran for the convenience of its users, but is not part of GNU Fortran. It contains the procedures needed by Fortran programs while they are running.

For example, while code generated by `g77` is likely to do additions, subtractions, and multiplications *in line*—in the actual compiled code—it is not likely to do trigonometric functions this way.

Instead, operations like trigonometric functions are compiled by the `f771` compiler (invoked by `g77` when compiling Fortran code) into machine code that, when run, calls on functions in `libf2c`, so `libf2c` must be linked with almost every useful program having any component compiled by GNU Fortran. (As mentioned above, the `g77` command takes care of all this for you.)

The `f771` program represents most of what is unique to GNU Fortran. While much of the `libf2c` component is really part of `f2c`, a free Fortran-to-C converter distributed by Bellcore (AT&T), plus `libU77`, provided by Dave Love, and the `g77` command is just a small front-end to `gcc`, `f771` is a combination of two rather large chunks of code.

One chunk is the so-called *GNU Back End*, or GBE, which knows how to generate fast code for a wide variety of processors. The same GBE is used by the C, C++, and Fortran compiler programs `cc1`, `cc1plus`, and `f771`, plus others. Often the GBE is referred to as the "gcc back end" or even just "gcc"—in this manual, the term GBE is used whenever the distinction is important.

The other chunk of `f771` is the majority of what is unique about GNU Fortran—the code that knows how to interpret Fortran programs to determine what they are intending to do, and then communicate that knowledge to the GBE for actual compilation of those programs. This chunk is called the *Fortran Front End* (FFE). The `cc1` and `cc1plus` programs have their own front ends, for the C and C++ languages, respectively. These fronts ends are responsible for diagnosing incorrect usage of their respective languages by the programs the process, and are responsible for most of the warnings about questionable constructs as well. (The GBE handles producing some warnings, like those concerning possible references to undefined variables.)

Because so much is shared among the compilers for various languages, much of the behavior and many of the user-selectable options for these compilers are similar. For example, diagnostics (error messages and warnings) are similar in appearance; command-line options like '`-Wall`' have generally similar effects; and the quality of generated code (in terms of speed and size) is roughly similar (since that work is done by the shared GBE).

6 Compile Fortran, C, or Other Programs

A GNU Fortran installation includes a modified version of the `gcc` command.

In a non-Fortran installation, `gcc` recognizes C, C++, and Objective-C source files.

In a GNU Fortran installation, `gcc` also recognizes Fortran source files and accepts Fortran-specific command-line options, plus some command-line options that are designed to cater to Fortran users but apply to other languages as well.

See section "Compile C; C++; or Objective-C" in *Using and Porting GNU CC*, for information on the way different languages are handled by the GNU CC compiler (`gcc`).

Also provided as part of GNU Fortran is the `g77` command. The `g77` command is designed to make compiling and linking Fortran programs somewhat easier than when using the `gcc` command for these tasks. It does this by analyzing the command line somewhat and changing it appropriately before submitting it to the `gcc` command.

Use the '`-v`' option with `g77` to see what is going on—the first line of output is the invocation of the `gcc` command. Use '`--driver=true`' to disable actual invocation of `gcc` (this works because '`true`' is the name of a UNIX command that simply returns success status).

7 GNU Fortran Command Options

The `g77` command supports all the options supported by the `gcc` command. See section "GNU CC Command Options" in *Using and Porting GNU CC*, for information on the non-Fortran-specific aspects of the `gcc` command (and, therefore, the `g77` command).

The `g77` command supports one option not supported by the `gcc` command:

`--driver=`*command*
: Specifies that *command*, rather than `gcc`, is to be invoked by `g77` to do its job. For example, within the `gcc` build directory after building GNU Fortran (but without having to install it), *`./g77 --driver=./xgcc foo.f -B./`*.

All other options are supported both by `g77` and by `gcc` as modified (and reinstalled) by the `g77` distribution. 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.

7.1 Option Summary

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

Overall Options
: See Section 7.2 [Options Controlling the Kind of Output], page 27.

```
--driver  -fversion  -fset-g77-defaults  -fno-silent
```

Shorthand Options
: See Section 7.3 [Shorthand Options], page 28.

```
-ff66  -fno-f66  -ff77  -fno-f77  -fugly  -fno-ugly
```

Fortran Language Options
: See Section 7.4 [Options Controlling Fortran Dialect], page 29.

```
-ffree-form  -fno-fixed-form  -ff90
-fvxt  -fdollar-ok  -fno-backslash
-fno-ugly-args  -fno-ugly-assign  -fno-ugly-assumed
-fugly-comma  -fugly-complex  -fugly-init  -fugly-logint
-fonetrip  -ftypeless-boz
-fintrin-case-initcap  -fintrin-case-upper
-fintrin-case-lower  -fintrin-case-any
-fmatch-case-initcap  -fmatch-case-upper
-fmatch-case-lower  -fmatch-case-any
-fsource-case-upper -fsource-case-lower  -fsource-case-preserve
-fsymbol-case-initcap  -fsymbol-case-upper
-fsymbol-case-lower  -fsymbol-case-any
-fcase-strict-upper  -fcase-strict-lower
-fcase-initcap  -fcase-upper  -fcase-lower  -fcase-preserve
-ff2c-intrinsics-delete  -ff2c-intrinsics-hide
-ff2c-intrinsics-disable  -ff2c-intrinsics-enable
-fbadu77-intrinsics-delete  -fbadu77-intrinsics-hide
```

`-fbadu77-intrinsics-disable` `-fbadu77-intrinsics-enable`
`-ff90-intrinsics-delete` `-ff90-intrinsics-hide`
`-ff90-intrinsics-disable` `-ff90-intrinsics-enable`
`-fgnu-intrinsics-delete` `-fgnu-intrinsics-hide`
`-fgnu-intrinsics-disable` `-fgnu-intrinsics-enable`
`-fmil-intrinsics-delete` `-fmil-intrinsics-hide`
`-fmil-intrinsics-disable` `-fmil-intrinsics-enable`
`-funix-intrinsics-delete` `-funix-intrinsics-hide`
`-funix-intrinsics-disable` `-funix-intrinsics-enable`
`-fvxt-intrinsics-delete` `-fvxt-intrinsics-hide`
`-fvxt-intrinsics-disable` `-fvxt-intrinsics-enable`
`-ffixed-line-length-`*n* `-ffixed-line-length-none`

Warning Options

See Section 7.5 [Options to Request or Suppress Warnings], page 35.

`-fsyntax-only` `-pedantic` `-pedantic-errors` `-fpedantic`
`-w` `-Wno-globals` `-Wimplicit` `-Wunused` `-Wuninitialized`
`-Wall` `-Wsurprising`
`-Werror` `-W`

Debugging Options

See Section 7.6 [Options for Debugging Your Program or GCC], page 38.

`-g`

Optimization Options

See Section 7.7 [Options that Control Optimization], page 38.

`-malign-double`
`-ffloat-store` `-fforce-mem` `-fforce-addr` `-fno-inline`
`-ffast-math` `-fstrength-reduce` `-frerun-cse-after-loop`
`-fexpensive-optimizations` `-fdelayed-branch`
`-fschedule-insns` `-fschedule-insn2` `-fcaller-saves`
`-funroll-loops` `-funroll-all-loops`
`-fno-move-all-movables` `-fno-reduce-all-givs`
`-fno-rerun-loop-opt`

Directory Options

See Section 7.9 [Options for Directory Search], page 40.

`-I`*dir* `-I-`

Code Generation Options

See Section 7.10 [Options for Code Generation Conventions], page 41.

`-fno-automatic` `-finit-local-zero` `-fno-f2c`
`-ff2c-library` `-fno-underscoring` `-fno-ident`
`-fpcc-struct-return` `-freg-struct-return`
`-fshort-double` `-fno-common` `-fpack-struct`
`-fzeros` `-fno-second-underscore`
`-fdebug-kludge` `-fno-emulate-complex`
`-falias-check` `-fargument-alias`
`-fargument-noalias` `-fno-argument-noalias-global`
`-fno-globals`

7.2 Options Controlling the Kind of Output

Compilation can involve as many as four stages: preprocessing, code generation (often what is really meant by the term "compilation"), assembly, and linking, always in that order. The first three stages apply to an individual source file, and end by producing an object file; linking combines all the object files (those newly compiled, and those specified as input) into an executable file.

For any given input file, the file name suffix determines what kind of program is contained in the file—that is, the language in which the program is written is generally indicated by the suffix. Suffixes specific to GNU Fortran are listed below. See Section 7.2 [gcc], page 27, for information on suffixes recognized by GNU CC.

file`.f`

file`.for` Fortran source code that should not be preprocessed.

Such source code cannot contain any preprocessor directives, such as `#include`, `#define`, `#if`, and so on.

file`.F`

file`.fpp` Fortran source code that must be preprocessed (by the C preprocessor `cpp`, which is part of GNU CC).

Note that preprocessing is not extended to the contents of files included by the `INCLUDE` directive—the `#include` preprocessor directive must be used instead.

file`.r` Ratfor source code, which must be preprocessed by the `ratfor` command, which is available separately (as it is not yet part of the GNU Fortran distribution).

UNIX users typically use the '*file*`.f`' and '*file*`.F`' nomenclature. Users of other operating systems, especially those that cannot distinguish upper-case letters from lower-case letters in their file names, typically use the '*file*`.for`' and '*file*`.fpp`' nomenclature.

Use of the preprocessor `cpp` allows use of C-like constructs such as `#define` and `#include`, but can lead to unexpected, even mistaken, results due to Fortran's source file format. It is recommended that use of the C preprocessor be limited to `#include` and, in conjunction with `#define`, only `#if` and related directives, thus avoiding in-line macro expansion entirely. This recommendation applies especially when using the traditional fixed source form. With free source form, fewer unexpected transformations are likely to happen, but use of constructs such as Hollerith and character constants can nevertheless present problems, especially when these are continued across multiple source lines. These problems result, primarily, from differences between the way such constants are interpreted by the C preprocessor and by a Fortran compiler.

Another example of a problem that results from using the C preprocessor is that a Fortran comment line that happens to contain any characters "interesting" to the C preprocessor, such as a backslash at the end of the line, is not recognized by the preprocessor as a comment line, so instead of being passed through "raw", the line is edited according to the rules for the preprocessor. For example, the backslash at the end of the line is removed, along with the subsequent newline, resulting in the next line being effectively commented out—unfortunate if that line is a non-comment line of important code!

Note: The '`-traditional`' and '`-undef`' flags are supplied to `cpp` by default, to avoid unpleasant surprises. See section "Options Controlling the Preprocessor" in *Using and Porting GNU CC*. This means that ANSI C preprocessor features (such as the '`#`' operator) aren't available, and only variables in the C reserved namespace (generally, names with a leading underscore) are liable to substitution by C predefines. Thus, if you want to do system-specific tests, use, for example, '`#ifdef __linux__`' rather than '`#ifdef linux`'. Use the '`-v`' option to see exactly how the preprocessor is invoked.

The following options that affect overall processing are recognized by the `g77` and `gcc` commands in a GNU Fortran installation:

`--driver=`*command*
: This works when invoking only the `g77` command, not when invoking the `gcc` command. See Chapter 7 [GNU Fortran Command Options], page 25, for information on this option.

`-fversion`
: Ensure that the `g77`-specific version of the compiler phase is reported, if run. (This is supplied automatically when '`-v`' or '`--verbose`' is specified as a command-line option for `g77` or `gcc` and when the resulting commands compile Fortran source files.)

`-fset-g77-defaults`
: Set up whatever `gcc` options are to apply to Fortran compilations, and avoid running internal consistency checks that might take some time.

  As of version 0.5.20, this is equivalent to '`-fmove-all-movables -freduce-all-givs -frerun-loop-opt -fargument-noalias-global`'.

  This option is supplied automatically when compiling Fortran code via the `g77` or `gcc` command. The description of this option is provided so that users seeing it in the output of, say, '`g77 -v`' understand why it is there.

  Also, developers who run `f771` directly might want to specify it by hand to get the same defaults as they would running `f771` via `g77` or `gcc`. However, such developers should, after linking a new `f771` executable, invoke it without this option once, e.g. via *`./f771 -quiet < /dev/null`*, to ensure that they have not introduced any internal inconsistencies (such as in the table of intrinsics) before proceeding—`g77` will crash with a diagnostic if it detects an inconsistency.

`-fno-silent`
: Print (to `stderr`) the names of the program units as they are compiled, in a form similar to that used by popular UNIX `f77` implementations and `f2c`.

See section "Options Controlling the Kind of Output" in *Using and Porting GNU CC*, for information on more options that control the overall operation of the `gcc` command (and, by extension, the `g77` command).

7.3 Shorthand Options

The following options serve as "shorthand" for other options accepted by the compiler:

`-fugly` Specify that certain "ugly" constructs are to be quietly accepted. Same as:

```
-fugly-args -fugly-assign -fugly-assumed
-fugly-comma -fugly-complex -fugly-init
-fugly-logint
```

These constructs are considered inappropriate to use in new or well-maintained portable Fortran code, but widely used in old code. See Section 11.9 [Distensions], page 178, for more information.

Note: The '`-fugly`' option is likely to be removed in a future version. Implicitly enabling all the '`-fugly-*`' options is unlikely to be feasible, or sensible, in the future, so users should learn to specify only those '`-fugly-*`' options they really need for a particular source file.

`-fno-ugly`

Specify that all "ugly" constructs are to be noisily rejected. Same as:

```
-fno-ugly-args -fno-ugly-assign -fno-ugly-assumed
-fno-ugly-comma -fno-ugly-complex -fno-ugly-init
-fno-ugly-logint
```

See Section 11.9 [Distensions], page 178, for more information.

`-ff66` Specify that the program is written in idiomatic FORTRAN 66. Same as '`-fonetrip -fugly-assumed`'.

The '`-fno-f66`' option is the inverse of '`-ff66`'. As such, it is the same as '`-fno-onetrip -fno-ugly-assumed`'.

The meaning of this option is likely to be refined as future versions of `g77` provide more compatibility with other existing and obsolete Fortran implementations.

`-ff77` Specify that the program is written in idiomatic UNIX FORTRAN 77 and/or the dialect accepted by the `f2c` product. Same as '`-fbackslash -fno-typeless-boz`'.

The meaning of this option is likely to be refined as future versions of `g77` provide more compatibility with other existing and obsolete Fortran implementations.

`-fno-f77` The '`-fno-f77`' option is *not* the inverse of '`-ff77`'. It specifies that the program is not written in idiomatic UNIX FORTRAN 77 or `f2c`, but in a more widely portable dialect. '`-fno-f77`' is the same as '`-fno-backslash`'.

The meaning of this option is likely to be refined as future versions of `g77` provide more compatibility with other existing and obsolete Fortran implementations.

7.4 Options Controlling Fortran Dialect

The following options control the dialect of Fortran that the compiler accepts:

`-ffree-form`

`-fno-fixed-form`

Specify that the source file is written in free form (introduced in Fortran 90) instead of the more-traditional fixed form.

`-ff90` Allow certain Fortran-90 constructs.

This option controls whether certain Fortran 90 constructs are recognized. (Other Fortran 90 constructs might or might not be recognized depending on other options such as '`-fvxt`', '`-ff90-intrinsics-enable`', and the current level of support for Fortran 90.)

See Section 11.7 [Fortran 90], page 176, for more information.

`-fvxt` Specify the treatment of certain constructs that have different meanings depending on whether the code is written in GNU Fortran (based on FORTRAN 77 and akin to Fortran 90) or VXT Fortran (more like VAX FORTRAN).

The default is '`-fno-vxt`'. '`-fvxt`' specifies that the VXT Fortran interpretations for those constructs are to be chosen.

See Section 11.6 [VXT Fortran], page 175, for more information.

`-fdollar-ok`
Allow '`$`' as a valid character in a symbol name.

`-fno-backslash`
Specify that '`\`' is not to be specially interpreted in character and Hollerith constants a la C and many UNIX Fortran compilers.

For example, with '`-fbackslash`' in effect, '`A\nB`' specifies three characters, with the second one being newline. With '`-fno-backslash`', it specifies four characters, '`A`', '`\`', '`n`', and '`B`'.

Note that `g77` implements a fairly general form of backslash processing that is incompatible with the narrower forms supported by some other compilers. For example, ''`A\003B`'' is a three-character string in `g77`, whereas other compilers that support backslash might not support the three-octal-digit form, and thus treat that string as longer than three characters.

See Section 18.5.1 [Backslash in Constants], page 284, for information on why '`-fbackslash`' is the default instead of '`-fno-backslash`'.

`-fno-ugly-args`
Disallow passing Hollerith and typeless constants as actual arguments (for example, '`CALL FOO(4HABCD)`').

See Section 11.9.1 [Ugly Implicit Argument Conversion], page 178, for more information.

`-fugly-assign`
Use the same storage for a given variable regardless of whether it is used to hold an assigned-statement label (as in '`ASSIGN 10 TO I`') or used to hold numeric data (as in '`I = 3`').

See Section 11.9.7 [Ugly Assigned Labels], page 181, for more information.

`-fugly-assumed`
Assume any dummy array with a final dimension specified as '`1`' is really an assumed-size array, as if '`*`' had been specified for the final dimension instead of '`1`'.

For example, '`DIMENSION X(1)`' is treated as if it had read '`DIMENSION X(*)`'.

See Section 11.9.2 [Ugly Assumed-Size Arrays], page 179, for more information.

`-fugly-comma`

In an external-procedure invocation, treat a trailing comma in the argument list as specification of a trailing null argument, and treat an empty argument list as specification of a single null argument.

For example, '`CALL FOO(,)`' is treated as '`CALL FOO(%VAL(0), %VAL(0))`'. That is, *two* null arguments are specified by the procedure call when '`-fugly-comma`' is in force. And '`F = FUNC()`' is treated as '`F = FUNC(%VAL(0))`'.

The default behavior, '`-fno-ugly-comma`', is to ignore a single trailing comma in an argument list. So, by default, '`CALL FOO(X,)`' is treated exactly the same as '`CALL FOO(X)`'.

See Section 11.9.4 [Ugly Null Arguments], page 180, for more information.

`-fugly-complex`

Do not complain about '`REAL(`*expr*`)`' or '`AIMAG(`*expr*`)`' when *expr* is a `COMPLEX` type other than `COMPLEX(KIND=1)`—usually this is used to permit `COMPLEX(KIND=2)` (`DOUBLE COMPLEX`) operands.

The '`-ff90`' option controls the interpretation of this construct.

See Section 11.9.3 [Ugly Complex Part Extraction], page 179, for more information.

`-fno-ugly-init`

Disallow use of Hollerith and typeless constants as initial values (in `PARAMETER` and `DATA` statements), and use of character constants to initialize numeric types and vice versa.

For example, '`DATA I/'F'/, CHRVAR/65/, J/4HABCD/`' is disallowed by '`-fno-ugly-init`'.

See Section 11.9.5 [Ugly Conversion of Initializers], page 180, for more information.

`-fugly-logint`

Treat `INTEGER` and `LOGICAL` variables and expressions as potential stand-ins for each other.

For example, automatic conversion between `INTEGER` and `LOGICAL` is enabled, for many contexts, via this option.

See Section 11.9.6 [Ugly Integer Conversions], page 181, for more information.

`-fonetrip`

Imperative executable `DO` loops are to be executed at least once each time they are reached.

ANSI FORTRAN 77 and more recent versions of the Fortran standard specify that the body of an imperative `DO` loop is not executed if the number of iterations calculated from the parameters of the loop is less than 1. (For example, '`DO 10 I = 1, 0`'.) Such a loop is called a *zero-trip loop*.

Prior to ANSI FORTRAN 77, many compilers implemented `DO` loops such that the body of a loop would be executed at least once, even if the iteration count

was zero. Fortran code written assuming this behavior is said to require *one-trip loops*. For example, some code written to the FORTRAN 66 standard expects this behavior from its `DO` loops, although that standard did not specify this behavior.

The '`-fonetrip`' option specifies that the source file(s) being compiled require one-trip loops.

This option affects only those loops specified by the (imperative) `DO` statement and by implied-`DO` lists in I/O statements. Loops specified by implied-`DO` lists in `DATA` and specification (non-executable) statements are not affected.

`-ftypeless-boz`

Specifies that prefix-radix non-decimal constants, such as '`Z'ABCD'`', are typeless instead of `INTEGER(KIND=1)`.

You can test for yourself whether a particular compiler treats the prefix form as `INTEGER(KIND=1)` or typeless by running the following program:

```
EQUIVALENCE (I, R)
R = Z'ABCD1234'
J = Z'ABCD1234'
IF (J .EQ. I) PRINT *, 'Prefix form is TYPELESS'
IF (J .NE. I) PRINT *, 'Prefix form is INTEGER'
END
```

Reports indicate that many compilers process this form as `INTEGER(KIND=1)`, though a few as typeless, and at least one based on a command-line option specifying some kind of compatibility.

`-fintrin-case-initcap`

`-fintrin-case-upper`

`-fintrin-case-lower`

`-fintrin-case-any`

Specify expected case for intrinsic names. '`-fintrin-case-lower`' is the default.

`-fmatch-case-initcap`

`-fmatch-case-upper`

`-fmatch-case-lower`

`-fmatch-case-any`

Specify expected case for keywords. '`-fmatch-case-lower`' is the default.

`-fsource-case-upper`

`-fsource-case-lower`

`-fsource-case-preserve`

Specify whether source text other than character and Hollerith constants is to be translated to uppercase, to lowercase, or preserved as is. '`-fsource-case-lower`' is the default.

`-fsymbol-case-initcap`

`-fsymbol-case-upper`

`-fsymbol-case-lower`

`-fsymbol-case-any`
: Specify valid cases for user-defined symbol names. '`-fsymbol-case-any`' is the default.

`-fcase-strict-upper`
: Same as '`-fintrin-case-upper -fmatch-case-upper -fsource-case-preserve -fsymbol-case-upper`'. (Requires all pertinent source to be in uppercase.)

`-fcase-strict-lower`
: Same as '`-fintrin-case-lower -fmatch-case-lower -fsource-case-preserve -fsymbol-case-lower`'. (Requires all pertinent source to be in lowercase.)

`-fcase-initcap`
: Same as '`-fintrin-case-initcap -fmatch-case-initcap -fsource-case-preserve -fsymbol-case-initcap`'. (Requires all pertinent source to be in initial capitals, as in '`Print *,SqRt(Value)`'.)

`-fcase-upper`
: Same as '`-fintrin-case-any -fmatch-case-any -fsource-case-upper -fsymbol-case-any`'. (Maps all pertinent source to uppercase.)

`-fcase-lower`
: Same as '`-fintrin-case-any -fmatch-case-any -fsource-case-lower -fsymbol-case-any`'. (Maps all pertinent source to lowercase.)

`-fcase-preserve`
: Same as '`-fintrin-case-any -fmatch-case-any -fsource-case-preserve -fsymbol-case-any`'. (Preserves all case in user-defined symbols, while allowing any-case matching of intrinsics and keywords. For example, '`call Foo(i,I)`' would pass two *different* variables named '`i`' and '`I`' to a procedure named '`Foo`'.)

`-fbadu77-intrinsics-delete`

`-fbadu77-intrinsics-hide`

`-fbadu77-intrinsics-disable`

`-fbadu77-intrinsics-enable`
: Specify status of UNIX intrinsics having inappropriate forms. '`-fbadu77-intrinsics-enable`' is the default. See Section 12.4.1 [Intrinsic Groups], page 185.

`-ff2c-intrinsics-delete`

`-ff2c-intrinsics-hide`

`-ff2c-intrinsics-disable`

`-ff2c-intrinsics-enable`
: Specify status of f2c-specific intrinsics. '`-ff2c-intrinsics-enable`' is the default. See Section 12.4.1 [Intrinsic Groups], page 185.

`-ff90-intrinsics-delete`

`-ff90-intrinsics-hide`

`-ff90-intrinsics-disable`

`-ff90-intrinsics-enable`
: Specify status of F90-specific intrinsics. '`-ff90-intrinsics-enable`' is the default. See Section 12.4.1 [Intrinsic Groups], page 185.

`-fgnu-intrinsics-delete`

`-fgnu-intrinsics-hide`

`-fgnu-intrinsics-disable`

`-fgnu-intrinsics-enable`
: Specify status of Digital's COMPLEX-related intrinsics. '`-fgnu-intrinsics-enable`' is the default. See Section 12.4.1 [Intrinsic Groups], page 185.

`-fmil-intrinsics-delete`

`-fmil-intrinsics-hide`

`-fmil-intrinsics-disable`

`-fmil-intrinsics-enable`
: Specify status of MIL-STD-1753-specific intrinsics. '`-fmil-intrinsics-enable`' is the default. See Section 12.4.1 [Intrinsic Groups], page 185.

`-funix-intrinsics-delete`

`-funix-intrinsics-hide`

`-funix-intrinsics-disable`

`-funix-intrinsics-enable`
: Specify status of UNIX intrinsics. '`-funix-intrinsics-enable`' is the default. See Section 12.4.1 [Intrinsic Groups], page 185.

`-fvxt-intrinsics-delete`

`-fvxt-intrinsics-hide`

`-fvxt-intrinsics-disable`

`-fvxt-intrinsics-enable`
: Specify status of VXT intrinsics. '`-fvxt-intrinsics-enable`' is the default. See Section 12.4.1 [Intrinsic Groups], page 185.

`-ffixed-line-length-`*n*
: Set column after which characters are ignored in typical fixed-form lines in the source file, and 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 (corresponds to "extended-source" options in some popular compilers). *n* may 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`'.

  See Section 11.1 [Source Form], page 169, for more information.

7.5 Options to Request or Suppress Warnings

Warnings are diagnostic messages that report constructions which are not inherently erroneous but which are risky or suggest there might have been an error.

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 warnings produced by GNU Fortran:

`-fsyntax-only`
: Check the code for syntax errors, but don't do anything beyond that.

`-pedantic`
: Issue warnings for uses of extensions to ANSI FORTRAN 77. '`-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 ANSI FORTRAN 77 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 strict ANSI conformance. They soon find that it does not do quite what they want—it finds some non-ANSI practices, but not all. However, improvements to `g77` in this area are welcome.

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

`-fpedantic`
: Like '`-pedantic`', but applies only to Fortran constructs.

`-w`
: Inhibit all warning messages.

`-Wno-globals`
: Inhibit warnings about use of a name as both a global name (a subroutine, function, or block data program unit, or a common block) and implicitly as the name of an intrinsic in a source file.

  Also inhibit warnings about inconsistent invocations and/or definitions of global procedures (function and subroutines). Such inconsistencies include different numbers of arguments and different types of arguments.

`-Wimplicit`
: Warn whenever a variable, array, or function is implicitly declared. Has an effect similar to using the `IMPLICIT NONE` statement in every program unit. (Some Fortran compilers provide this feature by an option named '`-u`' or '`/WARNINGS=DECLARATIONS`'.)

`-Wunused`
: Warn whenever a variable is unused aside from its declaration.

`-Wuninitialized`
: Warn whenever an automatic variable is used without first being initialized.

  These warnings are possible only in optimizing compilation, because they require data-flow information that is computed only when optimizing. If you don't specify '`-O`', you simply won't get these warnings.

  These warnings occur only for variables that are candidates for register allocation. Therefore, they do not occur for a variable whose address is taken, or whose size is other than 1, 2, 4 or 8 bytes. Also, they do not occur for arrays, even when they are in registers.

  Note that there might be no warning about a variable that is used only to compute a value that itself is never used, because such computations may be deleted by data-flow analysis before the warnings are printed.

  These warnings are made optional because GNU Fortran is not smart enough to see all the reasons why the code might be correct despite appearing to have an error. Here is one example of how this can happen:

```
SUBROUTINE DISPAT(J)
IF (J.EQ.1) I=1
IF (J.EQ.2) I=4
IF (J.EQ.3) I=5
CALL FOO(I)
END
```

  If the value of `J` is always 1, 2 or 3, then `I` is always initialized, but GNU Fortran doesn't know this. Here is another common case:

```
SUBROUTINE MAYBE(FLAG)
LOGICAL FLAG
IF (FLAG) VALUE = 9.4
...
IF (FLAG) PRINT *, VALUE
END
```

  This has no bug because `VALUE` is used only if it is set.

`-Wall`
: The '`-Wunused`' and '`-Wuninitialized`' options combined. These are all the options which pertain to usage that we recommend avoiding and that we believe is easy to avoid. (As more warnings are added to `g77`, some might be added to the list enabled by '`-Wall`'.)

The remaining '`-W...`' options are not implied by '`-Wall`' because they warn about constructions that we consider reasonable to use, on occasion, in clean programs.

`-Wsurprising`
: Warn about "suspicious" constructs that are interpreted by the compiler in a way that might well be surprising to someone reading the code. These differences can result in subtle, compiler-dependent (even machine-dependent) behavioral differences. The constructs warned about include:

  - Expressions having two arithmetic operators in a row, such as '`X*-Y`'. Such a construct is nonstandard, and can produce unexpected results in

more complicated situations such as '`X**-Y*Z`'. `g77`, along with many other compilers, interprets this example differently than many programmers, and a few other compilers. Specifically, `g77` interprets '`X**-Y*Z`' as '`(X**(-Y))*Z`', while others might think it should be interpreted as '`X**(-(Y*Z))`'.

A revealing example is the constant expression '`2**-2*1.`', which `g77` evaluates to .25, while others might evaluate it to 0., the difference resulting from the way precedence affects type promotion.

(The '`-fpedantic`' option also warns about expressions having two arithmetic operators in a row.)

- Expressions with a unary minus followed by an operand and then a binary operator other than plus or minus. For example, '`-2**2`' produces a warning, because the precedence is '`-(2**2)`', yielding -4, not '`(-2)**2`', which yields 4, and which might represent what a programmer expects.

  An example of an expression producing different results in a surprising way is '`-I*S`', where I holds the value '`-2147483648`' and S holds '`0.5`'. On many systems, negating I results in the same value, not a positive number, because it is already the lower bound of what an `INTEGER(KIND=1)` variable can hold. So, the expression evaluates to a positive number, while the "expected" interpretation, '`(-I)*S`', would evaluate to a negative number.

  Even cases such as '`-I*J`' produce warnings, even though, in most configurations and situations, there is no computational difference between the results of the two interpretations—the purpose of this warning is to warn about differing interpretations and encourage a better style of coding, not to identify only those places where bugs might exist in the user's code.

- `DO` loops with `DO` variables that are not of integral type—that is, using `REAL` variables as loop control variables. Although such loops can be written to work in the "obvious" way, the way `g77` is required by the Fortran standard to interpret such code is likely to be quite different from the way many programmers expect. (This is true of all `DO` loops, but the differences are pronounced for non-integral loop control variables.)

  See Section 17.3 [Loops], page 254, for more information.

`-Werror` Make all warnings into errors.

`-W` Turns on "extra warnings" and, if optimization is specified via '`-O`', the '`-Wuninitialized`' option. (This might change in future versions of `g77`.)

"Extra warnings" are issued for:

- Unused parameters to a procedure (when '`-Wunused`' also is specified).
- Overflows involving floating-point constants (not available for certain configurations).

See section "Options to Request or Suppress Warnings" in *Using and Porting GNU CC*, for information on more options offered by the GBE shared by `g77`, `gcc`, and other GNU compilers.

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

`-Wcomment`

`-Wformat`

`-Wparentheses`

`-Wswitch`

`-Wtraditional`

`-Wshadow`

`-Wid-clash-`*len*

`-Wlarger-than-`*len*

`-Wconversion`

`-Waggregate-return`

`-Wredundant-decls`

These options all could have some relevant meaning for GNU Fortran programs, but are not yet supported.

7.6 Options for Debugging Your Program or GNU Fortran

GNU Fortran has various special options that are used for debugging either your program or `g77`.

`-g` Produce debugging information in the operating system's native format (stabs, COFF, XCOFF, or DWARF). GDB can work with this debugging information.

Support for this option in Fortran programs is incomplete. In particular, names of variables and arrays in common blocks or that are storage-associated via `EQUIVALENCE` are unavailable to the debugger.

However, version 0.5.19 of `g77` does provide this information in a rudimentary way, as controlled by the '`-fdebug-kludge`' option.

See Section 7.10 [Options for Code Generation Conventions], page 41, for more information.

See section "Options for Debugging Your Program or GNU CC" in *Using and Porting GNU CC*, for more information on debugging options.

7.7 Options That Control Optimization

Most Fortran users will want to use no optimization when developing and testing programs, and use '`-O`' or '`-O2`' when compiling programs for late-cycle testing and for production use.

The following flags have particular applicability when compiling Fortran programs:

`-malign-double`

(Intel 386 architecture only.)

Noticeably improves performance of `g77` programs making heavy use of `REAL(KIND=2)` (`DOUBLE PRECISION`) data on some systems. In particular, systems using Pentium, Pentium Pro, 586, and 686 implementations of the i386 architecture

execute programs faster when `REAL(KIND=2)` (`DOUBLE PRECISION`) data are aligned on 64-bit boundaries in memory.

This option can, at least, make benchmark results more consistent across various system configurations, versions of the program, and data sets.

Note: The warning in the `gcc` documentation about this option does not apply, generally speaking, to Fortran code compiled by `g77`.

Also note: `g77` fixes a `gcc` backend bug to allow '`-malign-double`' to work generally, not just with statically-allocated data.

Also also note: The negative form of '`-malign-double`' is '`-mno-align-double`', not '`-benign-double`'.

`-ffloat-store`
Might help a Fortran program that depends on exact IEEE conformance on some machines, but might slow down a program that doesn't.

`-fforce-mem`

`-fforce-addr`
Might improve optimization of loops.

`-fno-inline`
Don't compile statement functions inline. Might reduce the size of a program unit—which might be at expense of some speed (though it should compile faster). Note that if you are not optimizing, no functions can be expanded inline.

`-ffast-math`
Might allow some programs designed to not be too dependent on IEEE behavior for floating-point to run faster, or die trying.

`-fstrength-reduce`
Might make some loops run faster.

`-frerun-cse-after-loop`

`-fexpensive-optimizations`

`-fdelayed-branch`

`-fschedule-insns`

`-fschedule-insns2`

`-fcaller-saves`
Might improve performance on some code.

`-funroll-loops`
Definitely improves performance on some code.

`-funroll-all-loops`
Definitely improves performance on some code.

`-fno-move-all-movables`

`-fno-reduce-all-givs`

`-fno-rerun-loop-opt`

Each of these might improve performance on some code.

Analysis of Fortran code optimization and the resulting optimizations triggered by the above options were contributed by Toon Moene (`toon@moene.indiv.nluug.nl`).

These three options are intended to be removed someday, once they have helped determine the efficacy of various approaches to improving the performance of Fortran code.

Please let us know how use of these options affects the performance of your production code. We're particularly interested in code that runs faster when these options are *disabled*, and in non-Fortran code that benefits when they are *enabled* via the above `gcc` command-line options.

See section "Options That Control Optimization" in *Using and Porting GNU CC*, for more information on options to optimize the generated machine code.

7.8 Options Controlling the Preprocessor

These options control the C preprocessor, which is run on each C source file before actual compilation.

See section "Options Controlling the Preprocessor" in *Using and Porting GNU CC*, for information on C preprocessor options.

Some of these options also affect how `g77` processes the `INCLUDE` directive. Since this directive is processed even when preprocessing is not requested, it is not described in this section. See Section 7.9 [Options for Directory Search], page 40, for information on how `g77` processes the `INCLUDE` directive.

However, the `INCLUDE` directive does not apply preprocessing to the contents of the included file itself.

Therefore, any file that contains preprocessor directives (such as `#include`, `#define`, and `#if`) must be included via the `#include` directive, not via the `INCLUDE` directive. Therefore, any file containing preprocessor directives, if included, is necessarily included by a file that itself contains preprocessor directives.

7.9 Options for Directory Search

These options affect how the `cpp` preprocessor searches for files specified via the `#include` directive. Therefore, when compiling Fortran programs, they are meaningful when the preproecssor is used.

Some of these options also affect how `g77` searches for files specified via the `INCLUDE` directive, although files included by that directive are not, themselves, preprocessed. These options are:

`-I-`

`-I`*dir* These affect interpretation of the `INCLUDE` directive (as well as of the `#include` directive of the `cpp` preprocessor).

Note that '`-I`*dir*' must be specified *without* any spaces between '`-I`' and the directory name—that is, '`-Ifoo/bar`' is valid, but '`-I foo/bar`' is rejected by the `g77` compiler (though the preprocessor supports the latter form). 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.

See section "Options for Directory Search" in *Using and Porting GNU CC*, for information on the '`-I`' option.

7.10 Options for Code Generation Conventions

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

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

`-fno-automatic`
: Treat each program unit as if the `SAVE` statement was specified for every local variable and array referenced in it. Does not affect common blocks. (Some Fortran compilers provide this option under the name '`-static`'.)

`-finit-local-zero`
: Specify that variables and arrays that are local to a program unit (not in a common block and not passed as an argument) are to be initialized to binary zeros.

  Since there is a run-time penalty for initialization of variables that are not given the `SAVE` attribute, it might be a good idea to also use '`-fno-automatic`' with '`-finit-local-zero`'.

`-fno-f2c`
: Do not generate code designed to be compatible with code generated by `f2c`; use the GNU calling conventions instead.

  The `f2c` calling conventions require functions that return type `REAL(KIND=1)` 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 GNU calling conventions, such functions simply return their results as they would in GNU C—`REAL(KIND=1)` functions return the C type `float`, and `COMPLEX` functions return the GNU C type `complex` (or its `struct` equivalent).

  This does not affect the generation of code that interfaces with the `libf2c` library.

  However, because the `libf2c` library uses `f2c` calling conventions, `g77` rejects attempts to pass intrinsics implemented by routines in this library as actual arguments when '`-fno-f2c`' is used, to avoid bugs when they are actually called by code expecting the GNU calling conventions to work.

  For example, '`INTRINSIC ABS;CALL FOO(ABS)`' is rejected when '`-fno-f2c`' is in force. (Future versions of the `g77` run-time library might offer routines that

provide GNU-callable versions of the routines that implement the `f2c`-callable intrinsics that may be passed as actual arguments, so that valid programs need not be rejected when '`-fno-f2c`' is used.)

Caution: If '`-fno-f2c`' is used when compiling any source file used in a program, it must be used when compiling *all* Fortran source files used in that program.

`-ff2c-library`

Specify that use of `libf2c` is required. This is the default for the current version of `g77`.

Currently it is not valid to specify '`-fno-f2c-library`'. This option is provided so users can specify it in shell scripts that build programs and libraries that require the `libf2c` library, even when being compiled by future versions of `g77` that might otherwise default to generating code for an incompatible library.

`-fno-underscoring`

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

With '`-funderscoring`' in effect, `g77` appends two underscores to names with underscores and one underscore to external names with no underscores. (`g77` also appends two underscores to internal names with underscores to avoid naming collisions with external names. The '`-fno-second-underscore`' option disables appending of the second underscore in all cases.)

This is done to ensure compatibility with code produced by many UNIX Fortran compilers, including `f2c`, which perform the same transformations.

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

For example, with '`-funderscoring`', and assuming other defaults like '`-fcase-lower`' and that '`j()`' and '`max_count()`' are external functions while '`my_var`' and '`lvar`' are local variables, a statement like

```
I = J() + MAX_COUNT (MY_VAR, LVAR)
```

is implemented as something akin to:

```
i = j_() + max_count__(&my_var__, &lvar);
```

With '`-fno-underscoring`', the same statement is implemented as:

```
i = j() + max_count(&my_var, &lvar);
```

Use of '`-fno-underscoring`' allows direct specification of user-defined names while debugging and when interfacing `g77`-compiled code with other languages.

Note that just because the names match does *not* mean that the interface implemented by `g77` for an external name matches the interface implemented by some other language for that same name. That is, getting code produced by `g77` 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 `g77`, 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.

`-fno-second-underscore`

Do not append a second underscore to names of entities specified in the Fortran source file.

This option has no effect if '`-fno-underscoring`' is in effect.

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__`'.

`-fno-ident`

Ignore the '`#ident`' directive.

`-fzeros` Treat initial values of zero as if they were any other value.

As of version 0.5.18, `g77` normally treats `DATA` and other statements that are used to specify initial values of zero for variables and arrays as if no values were actually specified, in the sense that no diagnostics regarding multiple initializations are produced.

This is done to speed up compiling of programs that initialize large arrays to zeros.

Use '`-fzeros`' to revert to the simpler, slower behavior that can catch multiple initializations by keeping track of all initializations, zero or otherwise.

Caution: Future versions of `g77` might disregard this option (and its negative form, the default) or interpret it somewhat differently. The interpretation changes will affect only non-standard programs; standard-conforming programs should not be affected.

`-fdebug-kludge`

Emit information on `COMMON` and `EQUIVALENCE` members that might help users of debuggers work around lack of proper debugging information on such members.

As of version 0.5.19, `g77` offers this option to emit information on members of aggregate areas to help users while debugging. This information consists of establishing the type and contents of each such member so that, when a debugger is asked to print the contents, the printed information provides rudimentary debugging information. This information identifies the name of the aggregate area (either the `COMMON` block name, or the `g77`-assigned name for the `EQUIVALENCE` name) and the offset, in bytes, of the member from the beginning of the area.

Using `gdb`, this information is not coherently displayed in the Fortran language mode, so temporarily switching to the C language mode to display the information is suggested. Use '`set language c`' and '`set language fortran`' to accomplish this.

For example:

```
COMMON /X/A,B
EQUIVALENCE (C,D)
CHARACTER XX*50
EQUIVALENCE (I,XX(20:20))
END
```

```
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.16 (lm-gnits-dwim), Copyright 1996 Free Software Foundation, Inc...
(gdb) b MAIN__
Breakpoint 1 at 0t1200000201120112: file cd.f, line 5.
(gdb) r
Starting program: /home/user/a.out

Breakpoint 1, MAIN__ () at cd.f:5
Current language:  auto; currently fortran
(gdb) set language c
Warning: the current language does not match this frame.
(gdb) p a
$2 = "At (COMMON) `x_' plus 0 bytes"
(gdb) p b
$3 = "At (COMMON) `x_' plus 4 bytes"
(gdb) p c
$4 = "At (EQUIVALENCE) `__g77_equiv_c' plus 0 bytes"
(gdb) p d
$5 = "At (EQUIVALENCE) `__g77_equiv_c' plus 0 bytes"
(gdb) p i
$6 = "At (EQUIVALENCE) `__g77_equiv_xx' plus 20 bytes"
(gdb) p xx
$7 = "At (EQUIVALENCE) `__g77_equiv_xx' plus 1 bytes"
(gdb) set language fortran
(gdb)
```

Use '`-fdebug-kludge`' to generate this information, which might make some programs noticeably larger.

Caution: Future versions of `g77` might disregard this option (and its negative form). Current plans call for this to happen when published versions of `g77` and `gdb` exist that provide proper access to debugging information on `COMMON` and `EQUIVALENCE` members.

`-fno-emulate-complex`

Implement `COMPLEX` arithmetic using the facilities in the `gcc` back end that provide direct support of `complex` arithmetic, instead of emulating the arithmetic.

`gcc` has some known problems in its back-end support for `complex` arithmetic, due primarily to the support not being completed as of version 2.7.2.2. Other front ends for the `gcc` back end avoid this problem by emulating `complex` arithmetic at a higher level, so the back end sees arithmetic on the real and imaginary components. To make `g77` more portable to systems where `complex` support in the `gcc` back end is particularly troublesome, `g77` now defaults to performing the same kinds of emulations done by these other front ends.

Use '`-fno-emulate-complex`' to try the `complex` support in the `gcc` back end, in case it works and produces faster programs. So far, all the known bugs seem to involve compile-time crashes, rather than the generation of incorrect code.

Use of this option should not affect how Fortran code compiled by `g77` works in terms of its interfaces to other code, e.g. that compiled by `f2c`.

Caution: Future versions of `g77` are likely to change the default for this option to '`-fno-emulate-complex`', and perhaps someday ignore both forms of this option.

Also, it is possible that use of the '`-fno-emulate-complex`' option could result in incorrect code being silently produced by `g77`. But, this is generally true of compilers anyway, so, as usual, test the programs you compile before assuming they are working.

`-falias-check`

`-fargument-alias`

`-fargument-noalias`

`-fno-argument-noalias-global`

These options specify to what degree aliasing (overlap) is permitted between arguments (passed as pointers) and `COMMON` (external, or public) storage.

The default for Fortran code, as mandated by the FORTRAN 77 and Fortran 90 standards, is '`-fargument-noalias-global`'. The default for code written in the C language family is '`-fargument-alias`'.

Note that, on some systems, compiling with '`-fforce-addr`' in effect can produce more optimal code when the default aliasing options are in effect (and when optimization is enabled).

See Section 17.4.7 [Aliasing Assumed To Work], page 258, for detailed information on the implications of compiling Fortran code that depends on the ability to alias dummy arguments.

`-fno-globals`

Disable diagnostics about inter-procedural analysis problems, such as disagreements about the type of a function or a procedure's argument, that might cause a compiler crash when attempting to inline a reference to a procedure within a program unit. (The diagnostics themselves are still produced, but as warnings, unless '`-Wno-globals`' is specified, in which case no relevant diagnostics are produced.)

Further, this option disables such inlining, to avoid compiler crashes resulting from incorrect code that would otherwise be diagnosed.

As such, this option might be quite useful when compiling existing, "working" code that happens to have a few bugs that do not generally show themselves, but `g77` exposes via a diagnostic.

Use of this option therefore has the effect of instructing `g77` to behave more like it did up through version 0.5.19.1, when it paid little or no attention to disagreements between program units about a procedure's type and argument information, and when it performed no inlining of procedures (except statement functions).

Without this option, `g77` defaults to performing the potentially inlining procedures as it started doing in version 0.5.20, but as of version 0.5.21, it also diagnoses disagreements that might cause such inlining to crash the compiler.

See section "Options for Code Generation Conventions" in *Using and Porting GNU CC*, for information on more options offered by the GBE shared by `g77`, `gcc`, and other GNU compilers.

Some of these do *not* work when compiling programs written in Fortran:

`-fpcc-struct-return`

`-freg-struct-return`
: You should not use these except strictly the same way as you used them to build the version of `libf2c` with which you will be linking all code compiled by `g77` with the same option.

`-fshort-double`
: This probably either has no effect on Fortran programs, or makes them act loopy.

`-fno-common`
: Do not use this when compiling Fortran programs, or there will be Trouble.

`-fpack-struct`
: This probably will break any calls to the `libf2c` library, at the very least, even if it is built with the same option.

7.11 Environment Variables Affecting GNU Fortran

GNU Fortran 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 GNU CC" in *Using and Porting GNU CC*, for information on environment variables.

8 News About GNU Fortran

Changes made to recent versions of GNU Fortran are listed below, with the most recent version first.

The changes are generally listed in order:

1. Code-generation and run-time-library bugs
2. Compiler and run-time-library crashes involving valid code
3. New features
4. Fixes and enhancements to existing features
5. New diagnostics
6. Internal improvements
7. Miscellany

This order is not strict—for example, some items involve a combination of these elements.

In 0.5.22:

- Fix code generation for iterative `DO` loops that have one or more references to the iteration variable, or to aliases of it, in their control expressions. For example, '`DO 10 J=2,J`' now is compiled correctly.
- Fix a code-generation bug that afflicted Intel x86 targets when '`-O2`' was specified compiling, for example, an old version of the '`DNRM2`' routine.

  The x87 coprocessor stack was being mismanaged in cases involving assigned `GOTO` and `ASSIGN`.
- Fix `DTime` intrinsic so as not to truncate results to integer values (on some systems).
- Fix `SIGNAL` intrinsic so it offers portable support for 64-bit systems (such as Digital Alphas running GNU/Linux).
- Fix run-time crash involving `NAMELIST` on 64-bit machines such as Alphas.
- Fix `g77` version of `libf2c` so it no longer produces a spurious '`I/O recursion`' diagnostic at run time when an I/O operation (such as '`READ *,I`') is interrupted in a manner that causes the program to be terminated via the '`f_exit`' routine (such as via *C-c*).
- Fix `g77` crash triggered by `CASE` statement with an omitted lower or upper bound.
- Fix `g77` crash compiling references to `CPU_Time` intrinsic.
- Fix `g77` crash (or apparently infinite run-time) when compiling certain complicated expressions involving `COMPLEX` arithmetic (especially multiplication).
- Fix `g77` crash on statements such as '`PRINT *, (REAL(Z(I)),I=1,2)`', where '`Z`' is `DOUBLE COMPLEX`.
- Fix a `g++` crash.
- Support '`FORMAT(I<`*expr*`>)`' when *expr* is a compile-time constant `INTEGER` expression.
- Fix `g77` '`-g`' option so procedures that use '`ENTRY`' can be stepped through, line by line, in `gdb`.
- Fix a profiling-related bug in `gcc` back end for Intel x86 architecture.

- Allow any `REAL` argument to intrinsics `Second` and `CPU_Time`.
- Allow any numeric argument to intrinsics `Int2` and `Int8`.
- Use `tempnam`, if available, to open scratch files (as in '`OPEN(STATUS='SCRATCH')`' so that the `TMPDIR` environment variable, if present, is used.
- Rename the `gcc` keyword `restrict` to `__restrict__`, to avoid rejecting valid, existing, C programs. Support for `restrict` is now more like support for `complex`.
- Fix '`-fpedantic`' to not reject procedure invocations such as '`I=J()`' and '`CALL FOO()`'.
- Fix '`-fugly-comma`' to affect invocations of only external procedures. Restore rejection of gratuitous trailing omitted arguments to intrinsics, as in '`I=MAX(3,4,,)`'.
- Fix compiler so it accepts '`-fgnu-intrinsics-*`' and '`-fbadu77-intrinsics-*`' options.
- Improve diagnostic messages from `libf2c` so it is more likely that the printing of the active format string is limited to the string, with no trailing garbage being printed.

  (Unlike `f2c`, `g77` did not append a null byte to its compiled form of every format string specified via a `FORMAT` statement. However, `f2c` would exhibit the problem anyway for a statement like '`PRINT '(I)garbage', 1`' by printing '`(I)garbage`' as the format string.)
- Improve compilation of FORMAT expressions so that a null byte is appended to the last operand if it is a constant. This provides a cleaner run-time diagnostic as provided by `libf2c` for statements like '`PRINT '(I1', 42`'.
- Fix various crashes involving code with diagnosed errors.
- Fix cross-compilation bug when configuring `libf2c`.
- Improve diagnostics.
- Improve documentation and indexing.
- Upgrade to `libf2c` as of 1997-09-23. This fixes a formatted-I/O bug that afflicted 64-bit systems with 32-bit integers (such as Digital Alpha running GNU/Linux).

In 0.5.21:

- Fix a code-generation bug introduced by 0.5.20 caused by loop unrolling (by specifying '`-funroll-loops`' or similar). This bug afflicted all code compiled by version 2.7.2.2.f.2 of `gcc` (C, C++, Fortran, and so on).
- Fix a code-generation bug manifested when combining local `EQUIVALENCE` with a `DATA` statement that follows the first executable statement (or is treated as an executable-context statement as a result of using the '`-fpedantic`' option).
- Fix a compiler crash that occured when an integer division by a constant zero is detected. Instead, when the '`-W`' option is specified, the `gcc` back end issues a warning about such a case. This bug afflicted all code compiled by version 2.7.2.2.f.2 of `gcc` (C, C++, Fortran, and so on).
- Fix a compiler crash that occurred in some cases of procedure inlining. (Such cases became more frequent in 0.5.20.)
- Fix a compiler crash resulting from using `DATA` or similar to initialize a `COMPLEX` variable or array to zero.

- Fix compiler crashes involving use of `AND`, `OR`, or `XOR` intrinsics.
- Fix compiler bug triggered when using a `COMMON` or `EQUIVALENCE` variable as the target of an `ASSIGN` or assigned-`GOTO` statement.
- Fix compiler crashes due to using the name of a some non-standard intrinsics (such as '`FTELL`' or '`FPUTC`') as such and as the name of a procedure or common block. Such dual use of a name in a program is allowed by the standard.
- Place automatic arrays on the stack, even if `SAVE` or the '`-fno-automatic`' option is in effect. This avoids a compiler crash in some cases.
- The '`-malign-double`' option now reliably aligns `DOUBLE PRECISION` optimally on Pentium and Pentium Pro architectures (586 and 686 in `gcc`).
- New option '`-Wno-globals`' disables warnings about "suspicious" use of a name both as a global name and as the implicit name of an intrinsic, and warnings about disagreements over the number or natures of arguments passed to global procedures, or the natures of the procedures themselves.

  The default is to issue such warnings, which are new as of this version of `g77`.
- New option '`-fno-globals`' disables diagnostics about potentially fatal disagreements analysis problems, such as disagreements over the number or natures of arguments passed to global procedures, or the natures of those procedures themselves.

  The default is to issue such diagnostics and flag the compilation as unsuccessful. With this option, the diagnostics are issued as warnings, or, if '`-Wno-globals`' is specified, are not issued at all.

  This option also disables inlining of global procedures, to avoid compiler crashes resulting from coding errors that these diagnostics normally would identify.
- Diagnose cases where a reference to a procedure disagrees with the type of that procedure, or where disagreements about the number or nature of arguments exist. This avoids a compiler crash.
- Fix parsing bug whereby `g77` rejected a second initialization specification immediately following the first's closing '`/`' without an intervening comma in a `DATA` statement, and the second specification was an implied-DO list.
- Improve performance of the `gcc` back end so certain complicated expressions involving `COMPLEX` arithmetic (especially multiplication) don't appear to take forever to compile.
- Fix a couple of profiling-related bugs in `gcc` back end.
- Integrate GNU Ada's (GNAT's) changes to the back end, which consist almost entirely of bug fixes. These fixes are circa version 3.10p of GNAT.
- Include some other `gcc` fixes that seem useful in `g77`'s version of `gcc`. (See '`gcc/ChangeLog`' for details—compare it to that file in the vanilla `gcc-2.7.2.3.tar.gz` distribution.)
- Fix `libU77` routines that accept file and other names to strip trailing blanks from them, for consistency with other implementations. Blanks may be forcibly appended to such names by appending a single null character ('`CHAR(0)`') to the significant trailing blanks.
- Fix `CHMOD` intrinsic to work with file names that have embedded blanks, commas, and so on.
- Fix `SIGNAL` intrinsic so it accepts an optional third '`Status`' argument.

- Fix `IDATE()` intrinsic subroutine (VXT form) so it accepts arguments in the correct order. Documentation fixed accordingly, and for `GMTIME()` and `LTIME()` as well.
- Make many changes to `libU77` intrinsics to support existing code more directly.

  Such changes include allowing both subroutine and function forms of many routines, changing `MCLOCK()` and `TIME()` to return `INTEGER(KIND=1)` values, introducing `MCLOCK8()` and `TIME8()` to return `INTEGER(KIND=2)` values, and placing functions that are intended to perform side effects in a new intrinsic group, `badu77`.
- Improve `libU77` so it is more portable.
- Add options '`-fbadu77-intrinsics-delete`', '`-fbadu77-intrinsics-hide`', and so on.
- Fix crashes involving diagnosed or invalid code.
- `g77` and `gcc` now do a somewhat better job detecting and diagnosing arrays that are too large to handle before these cause diagnostics during the assembler or linker phase, a compiler crash, or generation of incorrect code.
- Make some fixes to alias analysis code.
- Add support for `restrict` keyword in `gcc` front end.
- Support `gcc` version 2.7.2.3 (modified by `g77` into version 2.7.2.3.f.1), and remove support for prior versions of `gcc`.
- Incorporate GNAT's patches to the `gcc` back end into `g77`'s, so GNAT users do not need to apply GNAT's patches to build both GNAT and `g77` from the same source tree.
- Modify `make` rules and related code so that generation of Info documentation doesn't require compilation using `gcc`. Now, any ANSI C compiler should be adequate to produce the `g77` documentation (in particular, the tables of intrinsics) from scratch.
- Add `INT2` and `INT8` intrinsics.
- Add `CPU_TIME` intrinsic.
- Add `ALARM` intrinsic.
- `CTIME` intrinsic now accepts any `INTEGER` argument, not just `INTEGER(KIND=2)`.
- Warn when explicit type declaration disagrees with the type of an intrinsic invocation.
- Support '`*f771`' entry in `gcc` '`specs`' file.
- Fix typo in `make` rule '`g77-cross`', used only for cross-compiling.
- Fix `libf2c` build procedure to re-archive library if previous attempt to archive was interrupted.
- Change `gcc` to unroll loops only during the last invocation (of as many as two invocations) of loop optimization.
- Improve handling of '`-fno-f2c`' so that code that attempts to pass an intrinsic as an actual argument, such as '`CALL FOO(ABS)`', is rejected due to the fact that the run-time-library routine is, effectively, compiled with '`-ff2c`' in effect.
- Fix `g77` driver to recognize '`-fsyntax-only`' as an option that inhibits linking, just like '`-c`' or '`-S`', and to recognize and properly handle the '`-nostdlib`', '`-M`', '`-MM`', '`-nodefaultlibs`', and '`-Xlinker`' options.

- Upgrade to `libf2c` as of 1997-08-16.
- Modify `libf2c` to consistently and clearly diagnose recursive I/O (at run time).
- `g77` driver now prints version information (such as produced by *g77* `-v`) to `stderr` instead of `stdout`.
- The '`.r`' suffix now designates a Ratfor source file, to be preprocessed via the `ratfor` command, available separately.
- Fix some aspects of how `gcc` determines what kind of system is being configured and what kinds are supported. For example, GNU Linux/Alpha ELF systems now are directly supported.
- Improve diagnostics.
- Improve documentation and indexing.
- Include all pertinent files for `libf2c` that come from `netlib.bell-labs.com`; give any such files that aren't quite accurate in `g77`'s version of `libf2c` the suffix '`.netlib`'.
- Reserve `INTEGER(KIND=0)` for future use.

In 0.5.20:

- The '`-fno-typeless-boz`' option is now the default.

  This option specifies that non-decimal-radix constants using the prefixed-radix form (such as '`Z'1234'`') are to be interpreted as `INTEGER` constants. Specify '`-ftypeless-boz`' to cause such constants to be interpreted as typeless.

  (Version 0.5.19 introduced '`-fno-typeless-boz`' and its inverse.)
- Options '`-ff90-intrinsics-enable`' and '`-fvxt-intrinsics-enable`' now are the defaults.

  Some programs might use names that clash with intrinsic names defined (and now enabled) by these options or by the new `libU77` intrinsics. Users of such programs might need to compile them differently (using, for example, '`-ff90-intrinsics-disable`') or, better yet, insert appropriate `EXTERNAL` statements specifying that these names are not intended to be names of intrinsics.
- The '`ALWAYS_FLUSH`' macro is no longer defined when building `libf2c`, which should result in improved I/O performance, especially over NFS.

  Note: If you have code that depends on the behavior of `libf2c` when built with '`ALWAYS_FLUSH`' defined, you will have to modify `libf2c` accordingly before building it from this and future versions of `g77`.
- Dave Love's implementation of `libU77` has been added to the version of `libf2c` distributed with and built as part of `g77`. `g77` now knows about the routines in this library as intrinsics.
- New option '`-fvxt`' specifies that the source file is written in VXT Fortran, instead of GNU Fortran.
- The '`-fvxt-not-f90`' option has been deleted, along with its inverse, '`-ff90-not-vxt`'.

  If you used one of these deleted options, you should re-read the pertinent documentation to determine which options, if any, are appropriate for compiling your code with this version of `g77`.

- The '`-fugly`' option now issues a warning, as it likely will be removed in a future version.

  (Enabling all the '`-fugly-*`' options is unlikely to be feasible, or sensible, in the future, so users should learn to specify only those '`-fugly-*`' options they really need for a particular source file.)
- The '`-fugly-assumed`' option, introduced in version 0.5.19, has been changed to better accommodate old and new code.
- Make a number of fixes to the `g77` front end and the `gcc` back end to better support Alpha (AXP) machines. This includes providing at least one bug-fix to the `gcc` back end for Alphas.
- Related to supporting Alpha (AXP) machines, the `LOC()` intrinsic and `%LOC()` construct now return values of integer type that is the same width (holds the same number of bits) as the pointer type on the machine.

  On most machines, this won't make a difference, whereas on Alphas, the type these constructs return is `INTEGER*8` instead of the more common `INTEGER*4`.
- Emulate `COMPLEX` arithmetic in the `g77` front end, to avoid bugs in `complex` support in the `gcc` back end. New option '`-fno-emulate-complex`' causes `g77` to revert the 0.5.19 behavior.
- Fix bug whereby '`REAL A(1)`', for example, caused a compiler crash if '`-fugly-assumed`' was in effect and *A* was a local (automatic) array. That case is no longer affected by the new handling of '`-fugly-assumed`'.
- Fix `g77` command driver so that '`g77 -o foo.f`' no longer deletes '`foo.f`' before issuing other diagnostics, and so the '`-x`' option is properly handled.
- Enable inlining of subroutines and functions by the `gcc` back end. This works as it does for `gcc` itself—program units may be inlined for invocations that follow them in the same program unit, as long as the appropriate compile-time options are specified.
- Dummy arguments are no longer assumed to potentially alias (overlap) other dummy arguments or `COMMON` areas when any of these are defined (assigned to) by Fortran code.

  This can result in faster and/or smaller programs when compiling with optimization enabled, though on some systems this effect is observed only when '`-fforce-addr`' also is specified.

  New options '`-falias-check`', '`-fargument-alias`', '`-fargument-noalias`', and '`-fno-argument-noalias-global`' control the way `g77` handles potential aliasing.
- The `CONJG()` and `DCONJG()` intrinsics now are compiled in-line.
- The bug-fix for 0.5.19.1 has been re-done. The `g77` compiler has been changed back to assume `libf2c` has no aliasing problems in its implementations of the `COMPLEX` (and `DOUBLE COMPLEX`) intrinsics. The `libf2c` has been changed to have no such problems.

  As a result, 0.5.20 is expected to offer improved performance over 0.5.19.1, perhaps as good as 0.5.19 in most or all cases, due to this change alone.

  Note: This change requires version 0.5.20 of `libf2c`, at least, when linking code produced by any versions of `g77` other than 0.5.19.1. Use '`g77 -v`' to determine the version numbers of the `libF77`, `libI77`, and `libU77` components of the `libf2c` library. (If these version numbers are not printed—in particular, if the linker complains about unresolved

references to names like '`g77__fvers__`'—that strongly suggests your installation has an obsolete version of `libf2c`.)

- New option '`-fugly-assign`' specifies that the same memory locations are to be used to hold the values assigned by both statements '`I = 3`' and '`ASSIGN 10 TO I`', for example. (Normally, `g77` uses a separate memory location to hold assigned statement labels.)
- `FORMAT` and `ENTRY` statements now are allowed to precede `IMPLICIT NONE` statements.
- Produce diagnostic for unsupported `SELECT CASE` on `CHARACTER` type, instead of crashing, at compile time.
- Fix crashes involving diagnosed or invalid code.
- Change approach to building `libf2c` archive ('`libf2c.a`') so that members are added to it only when truly necessary, so the user that installs an already-built `g77` doesn't need to have write access to the build tree (whereas the user doing the build might not have access to install new software on the system).
- Support `gcc` version 2.7.2.2 (modified by `g77` into version 2.7.2.2.f.2), and remove support for prior versions of `gcc`.
- Upgrade to `libf2c` as of 1997-02-08, and fix up some of the build procedures.
- Improve general build procedures for `g77`, fixing minor bugs (such as deletion of any file named '`f771`' in the parent directory of `gcc/`).
- Enable full support of `INTEGER*8` available in `libf2c` and '`f2c.h`' so that `f2c` users may make full use of its features via the `g77` version of '`f2c.h`' and the `INTEGER*8` support routines in the `g77` version of `libf2c`.
- Improve `g77` driver and `libf2c` so that '`g77 -v`' yields version information on the library.
- The `SNGL` and `FLOAT` intrinsics now are specific intrinsics, instead of synonyms for the generic intrinsic `REAL`.
- New intrinsics have been added. These are `REALPART`, `IMAGPART`, `COMPLEX`, `LONG`, and `SHORT`.
- A new group of intrinsics, '`gnu`', has been added to contain the new `REALPART`, `IMAGPART`, and `COMPLEX` intrinsics. An old group, '`dcp`', has been removed.
- Complain about industry-wide ambiguous references '`REAL(`*expr*`)`' and '`AIMAG(`*expr*`)`', where *expr* is `DOUBLE COMPLEX` (or any complex type other than `COMPLEX`), unless '`-ff90`' option specifies Fortran 90 interpretation or new '`-fugly-complex`' option, in conjunction with '`-fnot-f90`', specifies `f2c` interpretation.
- Make improvements to diagnostics.
- Speed up compiler a bit.
- Improvements to documentation and indexing, including a new chapter containing information on one, later more, diagnostics that users are directed to pull up automatically via a message in the diagnostic itself.

  (Hence the menu item '`M`' for the node '`Diagnostics`' in the top-level menu of the Info documentation.)

In 0.5.19.1:

- Code-generation bugs afflicting operations on complex data have been fixed.

  These bugs occurred when assigning the result of an operation to a complex variable (or array element) that also served as an input to that operation.

  The operations affected by this bug were: '`CONJG()`', '`DCONJG()`', '`CCOS()`', '`CDCOS()`', '`CLOG()`', '`CDLOG()`', '`CSIN()`', '`CDSIN()`', '`CSQRT()`', '`CDSQRT()`', complex division, and raising a `DOUBLE COMPLEX` operand to an `INTEGER` power. (The related generic and '`Z`'-prefixed intrinsics, such as '`ZSIN()`', also were affected.)

  For example, '`C = CSQRT(C)`', '`Z = Z/C`', and '`Z = Z**I`' (where '`C`' is `COMPLEX` and '`Z`' is `DOUBLE COMPLEX`) have been fixed.

In 0.5.19:

- Fix `FORMAT` statement parsing so negative values for specifiers such as '`P`' (e.g. '`FORMAT(-1PF8.1)`') are correctly processed as negative.
- Fix `SIGNAL` intrinsic so it once again accepts a procedure as its second argument.
- A temporary kludge option provides bare-bones information on `COMMON` and `EQUIVALENCE` members at debug time.
- New '`-fonetrip`' option specifies FORTRAN-66-style one-trip `DO` loops.
- New '`-fno-silent`' option causes names of program units to be printed as they are compiled, in a fashion similar to UNIX `f77` and `f2c`.
- New '`-fugly-assumed`' option specifies that arrays dimensioned via '`DIMENSION X(1)`', for example, are to be treated as assumed-size.
- New '`-fno-typeless-boz`' option specifies that non-decimal-radix constants using the prefixed-radix form (such as '`Z'1234'`') are to be interpreted as `INTEGER` constants.
- New '`-ff66`' option is a "shorthand" option that specifies behaviors considered appropriate for FORTRAN 66 programs.
- New '`-ff77`' option is a "shorthand" option that specifies behaviors considered appropriate for UNIX `f77` programs.
- New '`-fugly-comma`' and '`-fugly-logint`' options provided to perform some of what '`-fugly`' used to do. '`-fugly`' and '`-fno-ugly`' are now "shorthand" options, in that they do nothing more than enable (or disable) other '`-fugly-*`' options.
- Fix parsing of assignment statements involving targets that are substrings of elements of `CHARACTER` arrays having names such as '`READ`', '`WRITE`', '`GOTO`', and '`REALFUNCTIONFOO`'.
- Fix crashes involving diagnosed code.
- Fix handling of local `EQUIVALENCE` areas so certain cases of valid Fortran programs are not misdiagnosed as improperly extending the area backwards.
- Support `gcc` version 2.7.2.1.
- Upgrade to `libf2c` as of 1996-09-26, and fix up some of the build procedures.

- Change code generation for list-directed I/O so it allows for new versions of `libf2c` that might return non-zero status codes for some operations previously assumed to always return zero.

  This change not only affects how `IOSTAT=` variables are set by list-directed I/O, it also affects whether `END=` and `ERR=` labels are reached by these operations.
- Add intrinsic support for new `FTELL` and `FSEEK` procedures in `libf2c`.
- Modify `fseek_()` in `libf2c` to be more portable (though, in practice, there might be no systems where this matters) and to catch invalid '`whence`' arguments.
- Some useless warnings from the '`-Wunused`' option have been eliminated.
- Fix a problem building the '`f771`' executable on AIX systems by linking with the '`-bbigtoc`' option.
- Abort configuration if `gcc` has not been patched using the patch file provided in the '`gcc/f/gbe/`' subdirectory.
- Add options '`--help`' and '`--version`' to the `g77` command, to conform to GNU coding guidelines. Also add printing of `g77` version number when the '`--verbose`' ('`-v`') option is used.
- Change internally generated name for local `EQUIVALENCE` areas to one based on the alphabetically sorted first name in the list of names for entities placed at the beginning of the areas.
- Improvements to documentation and indexing.

In 0.5.18:

- Add some rudimentary support for `INTEGER*1`, `INTEGER*2`, `INTEGER*8`, and their `LOGICAL` equivalents. (This support works on most, maybe all, `gcc` targets.)

  Thanks to Scott Snyder (`snyder@d0sgif.fnal.gov`) for providing the patch for this!

  Among the missing elements from the support for these features are full intrinsic support and constants.
- Add some rudimentary support for the `BYTE` and `WORD` type-declaration statements. `BYTE` corresponds to `INTEGER*1`, while `WORD` corresponds to `INTEGER*2`.

  Thanks to Scott Snyder (`snyder@d0sgif.fnal.gov`) for providing the patch for this!
- The compiler code handling intrinsics has been largely rewritten to accommodate the new types. No new intrinsics or arguments for existing intrinsics have been added, so there is, at this point, no intrinsic to convert to `INTEGER*8`, for example.
- Support automatic arrays in procedures.
- Reduce space/time requirements for handling large *sparsely* initialized aggregate arrays. This improvement applies to only a subset of the general problem to be addressed in 0.6.
- Treat initial values of zero as if they weren't specified (in DATA and type-declaration statements). The initial values will be set to zero anyway, but the amount of compile time processing them will be reduced, in some cases significantly (though, again, this is only a subset of the general problem to be addressed in 0.6).

A new option, '`-fzeros`', is introduced to enable the traditional treatment of zeros as any other value.

- With '`-ff90`' in force, `g77` incorrectly interpreted '`REAL(Z)`' as returning a `REAL` result, instead of as a `DOUBLE PRECISION` result. (Here, '`Z`' is `DOUBLE COMPLEX`.)

  With '`-fno-f90`' in force, the interpretation remains unchanged, since this appears to be how at least some F77 code using the `DOUBLE COMPLEX` extension expected it to work.

  Essentially, '`REAL(Z)`' in F90 is the same as '`DBLE(Z)`', while in extended F77, it appears to be the same as '`REAL(REAL(Z))`'.

- An expression involving exponentiation, where both operands were type `INTEGER` and the right-hand operand was negative, was erroneously evaluated.

- Fix bugs involving `DATA` implied-`DO` constructs (these involved an errant diagnostic and a crash, both on good code, one involving subsequent statement-function definition).

- Close `INCLUDE` files after processing them, so compiling source files with lots of `INCLUDE` statements does not result in being unable to open `INCLUDE` files after all the available file descriptors are used up.

- Speed up compiling, especially of larger programs, and perhaps slightly reduce memory utilization while compiling (this is *not* the improvement planned for 0.6 involving large aggregate areas)—these improvements result from simply turning off some low-level code to do self-checking that hasn't been triggered in a long time.

- Introduce three new options that implement optimizations in the `gcc` back end (GBE). These options are '`-fmove-all-movables`', '`-freduce-all-givs`', and '`-frerun-loop-opt`', which are enabled, by default, for Fortran compilations. These optimizations are intended to help toon Fortran programs.

- Patch the GBE to do a better job optimizing certain kinds of references to array elements.

- Due to patches to the GBE, the version number of `gcc` also is patched to make it easier to manage installations, especially useful if it turns out a `g77` change to the GBE has a bug.

  The `g77`-modified version number is the `gcc` version number with the string '`.f.`*n*' appended, where '`f`' identifies the version as enhanced for Fortran, and *n* is '`1`' for the first Fortran patch for that version of `gcc`, '`2`' for the second, and so on.

  So, this introduces version 2.7.2.f.1 of `gcc`.

- Make several improvements and fixes to diagnostics, including the removal of two that were inappropriate or inadequate.

- Warning about two successive arithmetic operators, produced by '`-Wsurprising`', now produced *only* when both operators are, indeed, arithmetic (not relational/boolean).

- '`-Wsurprising`' now warns about the remaining cases of using non-integral variables for implied-`DO` loops, instead of these being rejected unless '`-fpedantic`' or '`-fugly`' specified.

- Allow `SAVE` of a local variable or array, even after it has been given an initial value via `DATA`, for example.

- Introduce an Info version of `g77` documentation, which supercedes '`gcc/f/CREDITS`', '`gcc/f/DOC`', and '`gcc/f/PROJECTS`'. These files will be removed in a future release. The files '`gcc/f/BUGS`', '`gcc/f/INSTALL`', and '`gcc/f/NEWS`' now are automatically built from the texinfo source when distributions are made.

  This effort was inspired by a first pass at translating '`g77-0.5.16/f/DOC`' that was contributed to Craig by David Ronis (`ronis@onsager.chem.mcgill.ca`).
- New '`-fno-second-underscore`' option to specify that, when '`-funderscoring`' is in effect, a second underscore is not to be appended to Fortran names already containing an underscore.
- Change the way iterative `DO` loops work to follow the F90 standard. In particular, calculation of the iteration count is still done by converting the start, end, and increment parameters to the type of the `DO` variable, but the result of the calculation is always converted to the default `INTEGER` type.

  (This should have no effect on existing code compiled by `g77`, but code written to assume that use of a *wider* type for the `DO` variable will result in an iteration count being fully calculated using that wider type (wider than default `INTEGER`) must be rewritten.)
- Support `gcc` version 2.7.2.
- Upgrade to `libf2c` as of 1996-03-23, and fix up some of the build procedures.

  Note that the email addresses related to `f2c` have changed—the distribution site now is named `netlib.bell-labs.com`, and the maintainer's new address is `dmg@bell-labs.com`.

In 0.5.17:

- **Fix serious bug** in '`g77 -v`' command that can cause removal of a system's '`/dev/null`' special file if run by user '`root`'.

  All users of version 0.5.16 should ensure that they have not removed '`/dev/null`' or replaced it with an ordinary file (e.g. by comparing the output of '`ls -l /dev/null`' with '`ls -l /dev/zero`'. If the output isn't basically the same, contact your system administrator about restoring '`/dev/null`' to its proper status).

  This bug is particularly insidious because removing '`/dev/null`' as a special file can go undetected for quite a while, aside from various applications and programs exhibiting sudden, strange behaviors.

  I sincerely apologize for not realizing the implications of the fact that when '`g77 -v`' runs the `ld` command with '`-o /dev/null`' that `ld` tries to *remove* the executable it is supposed to build (especially if it reports unresolved references, which it should in this case)!
- Fix crash on '`CHARACTER*(*) FOO`' in a main or block data program unit.
- Fix crash that can occur when diagnostics given outside of any program unit (such as when input file contains '`@foo`').
- Fix crashes, infinite loops (hangs), and such involving diagnosed code.
- Fix `ASSIGN`'ed variables so they can be `SAVE`'d or dummy arguments, and issue clearer error message in cases where target of `ASSIGN` or `ASSIGN`ed `GOTO`/`FORMAT` is too small (which should never happen).

- Make `libf2c` build procedures work on more systems again by eliminating unnecessary invocations of '`ld -r -x`' and '`mv`'.
- Fix omission of '`-funix-intrinsics-...`' options in list of permitted options to compiler.
- Fix failure to always diagnose missing type declaration for `IMPLICIT NONE`.
- Fix compile-time performance problem (which could sometimes crash the compiler, cause a hang, or whatever, due to a bug in the back end) involving exponentiation with a large `INTEGER` constant for the right-hand operator (e.g. '`I**32767`').
- Fix build procedures so cross-compiling `g77` (the `fini` utility in particular) is properly built using the host compiler.
- Add new '`-Wsurprising`' option to warn about constructs that are interpreted by the Fortran standard (and `g77`) in ways that are surprising to many programmers.
- Add `ERF()` and `ERFC()` as generic intrinsics mapping to existing `ERF/DERF` and `ERFC/DERFC` specific intrinsics.

  Note: You should specify '`INTRINSIC ERF,ERFC`' in any code where you might use these as generic intrinsics, to improve likelihood of diagnostics (instead of subtle run-time bugs) when using a compiler that doesn't support these as intrinsics (e.g. `f2c`).
- Remove from '`-fno-pedantic`' the diagnostic about `DO` with non-`INTEGER` index variable; issue that under '`-Wsurprising`' instead.
- Clarify some diagnostics that say things like "ignored" when that's misleading.
- Clarify diagnostic on use of `.EQ./.NE.` on `LOGICAL` operands.
- Minor improvements to code generation for various operations on `LOGICAL` operands.
- Minor improvement to code generation for some `DO` loops on some machines.
- Support `gcc` version 2.7.1.
- Upgrade to `libf2c` as of 1995-11-15.

In 0.5.16:

- Fix a code-generation bug involving complicated `EQUIVALENCE` statements not involving `COMMON`.
- Fix code-generation bugs involving invoking "gratis" library procedures in `libf2c` from code compiled with '`-fno-f2c`' by making these procedures known to `g77` as intrinsics (not affected by -fno-f2c). This is known to fix code invoking `ERF()`, `ERFC()`, `DERF()`, and `DERFC()`.
- Update `libf2c` to include netlib patches through 1995-08-16, and `#define` '`WANT_LEAD_0`' to 1 to make `g77`-compiled code more consistent with other Fortran implementations by outputting leading zeros in formatted and list-directed output.
- Fix a code-generation bug involving adjustable dummy arrays with high bounds whose primaries are changed during procedure execution, and which might well improve code-generation performance for such arrays compared to `f2c` plus `gcc` (but apparently only when using '`gcc-2.7.0`' or later).

- Fix a code-generation bug involving invocation of `COMPLEX` and `DOUBLE COMPLEX` `FUNCTION`s and doing `COMPLEX` and `DOUBLE COMPLEX` divides, when the result of the invocation or divide is assigned directly to a variable that overlaps one or more of the arguments to the invocation or divide.
- Fix crash by not generating new optimal code for '`X**I`' if '`I`' is nonconstant and the expression is used to dimension a dummy array, since the `gcc` back end does not support the necessary mechanics (and the `gcc` front end rejects the equivalent construct, as it turns out).
- Fix crash on expressions like '`COMPLEX**INTEGER`'.
- Fix crash on expressions like '`(1D0,2D0)**2`', i.e. raising a `DOUBLE COMPLEX` constant to an `INTEGER` constant power.
- Fix crashes and such involving diagnosed code.
- Diagnose, instead of crashing on, statement function definitions having duplicate dummy argument names.
- Fix bug causing rejection of good code involving statement function definitions.
- Fix bug resulting in debugger not knowing size of local equivalence area when any member of area has initial value (via `DATA`, for example).
- Fix installation bug that prevented installation of `g77` driver. Provide for easy selection of whether to install copy of `g77` as `f77` to replace the broken code.
- Fix `gcc` driver (affects `g77` thereby) to not gratuitously invoke the `f771` program (e.g. when '`-E`' is specified).
- Fix diagnostic to point to correct source line when it immediately follows an `INCLUDE` statement.
- Support more compiler options in `gcc`/`g77` when compiling Fortran files. These options include '`-p`', '`-pg`', '`-aux-info`', '`-P`', correct setting of version-number macros for preprocessing, full recognition of '`-O0`', and automatic insertion of configuration-specific linker specs.
- Add new intrinsics that interface to existing routines in `libf2c`: `ABORT`, `DERF`, `DERFC`, `ERF`, `ERFC`, `EXIT`, `FLUSH`, `GETARG`, `GETENV`, `IARGC`, `SIGNAL`, and `SYSTEM`. Note that `ABORT`, `EXIT`, `FLUSH`, `SIGNAL`, and `SYSTEM` are intrinsic subroutines, not functions (since they have side effects), so to get the return values from `SIGNAL` and `SYSTEM`, append a final argument specifying an `INTEGER` variable or array element (e.g. '`CALL SYSTEM('rm foo',ISTAT)`').
- Add new intrinsic group named '`unix`' to contain the new intrinsics, and by default enable this new group.
- Move `LOC()` intrinsic out of the '`vxt`' group to the new '`unix`' group.
- Improve `g77` so that '`g77 -v`' by itself (or with certain other options, including '`-B`', '`-b`', '`-i`', '`-nostdlib`', and '`-V`') reports lots more useful version info, and so that long-form options `gcc` accepts are understood by `g77` as well (even in truncated, unambiguous forms).
- Add new `g77` option '`--driver=name`' to specify driver when default, `gcc`, isn't appropriate.

- Add support for '`#`' directives (as output by the preprocessor) in the compiler, and enable generation of those directives by the preprocessor (when compiling '`.F`' files) so diagnostics and debugging info are more useful to users of the preprocessor.
- Produce better diagnostics, more like `gcc`, with info such as '`In function 'foo':`' and '`In file included from...:`'.
- Support `gcc`'s '`-fident`' and '`-fno-ident`' options.
- When '`-Wunused`' in effect, don't warn about local variables used as statement-function dummy arguments or `DATA` implied-`DO` iteration variables, even though, strictly speaking, these are not uses of the variables themselves.
- When '`-W -Wunused`' in effect, don't warn about unused dummy arguments at all, since there's no way to turn this off for individual cases (`g77` might someday start warning about these)—applies to `gcc` versions 2.7.0 and later, since earlier versions didn't warn about unused dummy arguments.
- New option '`-fno-underscoring`' that inhibits transformation of names (by appending one or two underscores) so users may experiment with implications of such an environment.
- Minor improvement to '`gcc/f/info`' module to make it easier to build `g77` using the native (non-`gcc`) compiler on certain machines (but definitely not all machines nor all non-`gcc` compilers). Please do not report bugs showing problems compilers have with macros defined in '`gcc/f/target.h`' and used in places like '`gcc/f/expr.c`'.
- Add warning to be printed for each invocation of the compiler if the target machine `INTEGER`, `REAL`, or `LOGICAL` size is not 32 bits, since `g77` is known to not work well for such cases (to be fixed in Version 0.6—see Section 18.2 [Actual Bugs We Haven't Fixed Yet], page 271).
- Lots of new documentation (though work is still needed to put it into canonical GNU format).
- Build `libf2c` with '`-g0`', not '`-g2`', in effect (by default), to produce smaller library without lots of debugging clutter.

In 0.5.15:

- Fix bad code generation involving '`X**I`' and temporary, internal variables generated by `g77` and the back end (such as for `DO` loops).
- Fix crash given '`CHARACTER A;DATA A/.TRUE./`'.
- Replace crash with diagnostic given '`CHARACTER A;DATA A/1.0/`'.
- Fix crash or other erratic behavior when null character constant ('`''`') is encountered.
- Fix crash or other erratic behavior involving diagnosed code.
- Fix code generation for external functions returning type `REAL` when the '`-ff2c`' option is in force (which it is by default) so that `f2c` compatibility is indeed provided.
- Disallow '`COMMON I(10)`' if '`I`' has previously been specified with an array declarator.
- New '`-ffixed-line-length-`*n*' option, where *n* is the maximum length of a typical fixed-form line, defaulting to 72 columns, such that characters beyond column *n* are ignored, or *n* is '`none`', meaning no characters are ignored. does not affect lines with

'`&`' in column 1, which are always processed as if '`-ffixed-line-length-none`' was in effect.

- No longer generate better code for some kinds of array references, as `gcc` back end is to be fixed to do this even better, and it turned out to slow down some code in some cases after all.
- In `COMMON` and `EQUIVALENCE` areas with any members given initial values (e.g. via `DATA`), uninitialized members now always initialized to binary zeros (though this is not required by the standard, and might not be done in future versions of `g77`). Previously, in some `COMMON/EQUIVALENCE` areas (essentially those with members of more than one type), the uninitialized members were initialized to spaces, to cater to `CHARACTER` types, but it seems no existing code expects that, while much existing code expects binary zeros.

In 0.5.14:

- Don't emit bad code when low bound of adjustable array is nonconstant and thus might vary as an expression at run time.
- Emit correct code for calculation of number of trips in `DO` loops for cases where the loop should not execute at all. (This bug affected cases where the difference between the begin and end values was less than the step count, though probably not for floating-point cases.)
- Fix crash when extra parentheses surround item in `DATA` implied-`DO` list.
- Fix crash over minor internal inconsistencies in handling diagnostics, just substitute dummy strings where necessary.
- Fix crash on some systems when compiling call to `MVBITS()` intrinsic.
- Fix crash on array assignment '`TYPE`*ddd*`(...)=...`', where *ddd* is a string of one or more digits.
- Fix crash on `DCMPLX()` with a single `INTEGER` argument.
- Fix various crashes involving code with diagnosed errors.
- Support '`-I`' option for `INCLUDE` statement, plus `gcc`'s '`header.gcc`' facility for handling systems like MS-DOS.
- Allow `INCLUDE` statement to be continued across multiple lines, even allow it to coexist with other statements on the same line.
- Incorporate Bellcore fixes to `libf2c` through 1995-03-15—this fixes a bug involving infinite loops reading EOF with empty list-directed I/O list.
- Remove all the `g77`-specific auto-configuration scripts, code, and so on, except for temporary substitutes for bsearch() and strtoul(), as too many configure/build problems were reported in these areas. People will have to fix their systems' problems themselves, or at least somewhere other than `g77`, which expects a working ANSI C environment (and, for now, a GNU C compiler to compile `g77` itself).
- Complain if initialized common redeclared as larger in subsequent program unit.
- Warn if blank common initialized, since its size can vary and hence related warnings that might be helpful won't be seen.

- New '`-fbackslash`' option, on by default, that causes '`\`' within `CHARACTER` and Hollerith constants to be interpreted a la GNU C. Note that this behavior is somewhat different from `f2c`'s, which supports only a limited subset of backslash (escape) sequences.
- Make '`-fugly-args`' the default.
- New '`-fugly-init`' option, on by default, that allows typeless/Hollerith to be specified as initial values for variables or named constants (`PARAMETER`), and also allows character`<->`numeric conversion in those contexts—turn off via '`-fno-ugly-init`'.
- New '`-finit-local-zero`' option to initialize local variables to binary zeros. This does not affect whether they are `SAVE`d, i.e. made automatic or static.
- New '`-Wimplicit`' option to warn about implicitly typed variables, arrays, and functions. (Basically causes all program units to default to `IMPLICIT NONE`.)
- '`-Wall`' now implies '`-Wuninitialized`' as with `gcc` (i.e. unless '`-O`' not specified, since '`-Wuninitialized`' requires '`-O`'), and implies '`-Wunused`' as well.
- '`-Wunused`' no longer gives spurious messages for unused `EXTERNAL` names (since they are assumed to refer to block data program units, to make use of libraries more reliable).
- Support `%LOC()` and `LOC()` of character arguments.
- Support null (zero-length) character constants and expressions.
- Support `f2c`'s `IMAG()` generic intrinsic.
- Support `ICHAR()`, `IACHAR()`, and `LEN()` of character expressions that are valid in assignments but not normally as actual arguments.
- Support `f2c`-style '`&`' in column 1 to mean continuation line.
- Allow `NAMELIST`, `EXTERNAL`, `INTRINSIC`, and `VOLATILE` in `BLOCK DATA`, even though these are not allowed by the standard.
- Allow `RETURN` in main program unit.
- Changes to Hollerith-constant support to obey Appendix C of the standard:
  - Now padded on the right with zeros, not spaces.
  - Hollerith "format specifications" in the form of arrays of non-character allowed.
  - Warnings issued when non-space truncation occurs when converting to another type.
  - When specified as actual argument, now passed by reference to `INTEGER` (padded on right with spaces if constant too small, otherwise fully intact if constant wider the `INTEGER` type) instead of by value.

  Warning: `f2c` differs on the interpretation of '`CALL FOO(1HX)`', which it treats exactly the same as '`CALL FOO('X')`', but which the standard and `g77` treat as '`CALL FOO(%REF('X '))`' (padded with as many spaces as necessary to widen to `INTEGER`), essentially.
- Changes and fixes to typeless-constant support:
  - Now treated as a typeless double-length `INTEGER` value.
  - Warnings issued when overflow occurs.
  - Padded on the left with zeros when converting to a larger type.

  - Should be properly aligned and ordered on the target machine for whatever type it is turned into.
  - When specified as actual argument, now passed as reference to a default `INTEGER` constant.

- `%DESCR()` of a non-`CHARACTER` expression now passes a pointer to the expression plus a length for the expression just as if it were a `CHARACTER` expression. For example, '`CALL FOO(%DESCR(D))`', where '`D`' is `REAL*8`, is the same as '`CALL FOO(D,%VAL(8))`'.
- Name of multi-entrypoint master function changed to incorporate the name of the primary entry point instead of a decimal value, so the name of the master function for '`SUBROUTINE X`' with alternate entry points is now '`__g77_masterfun_x`'.
- Remove redundant message about zero-step-count `DO` loops.
- Clean up diagnostic messages, shortening many of them.
- Fix typo in `g77` man page.
- Clarify implications of constant-handling bugs in '`f/BUGS`'.
- Generate better code for '`**`' operator with a right-hand operand of type `INTEGER`.
- Generate better code for `SQRT()` and `DSQRT()`, also when '`-ffast-math`' specified, enable better code generation for `SIN()` and `COS()`.
- Generate better code for some kinds of array references.
- Speed up lexing somewhat (this makes the compilation phase noticeably faster).

9 User-visible Changes

This section describes changes to g77 that are visible to the programmers who actually write and maintain Fortran code they compile with g77. Information on changes to installation procedures, changes to the documentation, and bug fixes is not provided here, unless it is likely to affect how users use g77. See Chapter 8 [News About GNU Fortran], page 47, for information on such changes to g77.

To find out about existing bugs and ongoing plans for GNU Fortran, retrieve ftp://alpha.gnu.org/g77.pl or, if you cannot do that, email fortran@gnu.org asking for a recent copy of the GNU Fortran '.plan' file.

In 0.5.21:

- When the '-W' option is specified, gcc, g77, and other GNU compilers that incorporate the gcc back end as modified by g77, issue a warning about integer division by constant zero.
- New option '-Wno-globals' disables warnings about "suspicious" use of a name both as a global name and as the implicit name of an intrinsic, and warnings about disagreements over the number or natures of arguments passed to global procedures, or the natures of the procedures themselves.

  The default is to issue such warnings, which are new as of this version of g77.
- New option '-fno-globals' disables diagnostics about potentially fatal disagreements analysis problems, such as disagreements over the number or natures of arguments passed to global procedures, or the natures of those procedures themselves.

  The default is to issue such diagnostics and flag the compilation as unsuccessful. With this option, the diagnostics are issued as warnings, or, if '-Wno-globals' is specified, are not issued at all.

  This option also disables inlining of global procedures, to avoid compiler crashes resulting from coding errors that these diagnostics normally would identify.
- Fix libU77 routines that accept file names to strip trailing spaces from them, for consistency with other implementations.
- Fix SIGNAL intrinsic so it accepts an optional third 'Status' argument.
- Make many changes to libU77 intrinsics to support existing code more directly.

  Such changes include allowing both subroutine and function forms of many routines, changing MCLOCK() and TIME() to return INTEGER(KIND=1) values, introducing MCLOCK8() and TIME8() to return INTEGER(KIND=2) values, and placing functions that are intended to perform side effects in a new intrinsic group, badu77.
- Add options '-fbadu77-intrinsics-delete', '-fbadu77-intrinsics-hide', and so on.
- Add INT2 and INT8 intrinsics.
- Add CPU_TIME intrinsic.
- CTIME intrinsic now accepts any INTEGER argument, not just INTEGER(KIND=2).

In 0.5.20:

- The '`-fno-typeless-boz`' option is now the default.

  This option specifies that non-decimal-radix constants using the prefixed-radix form (such as '`Z'1234'`') are to be interpreted as `INTEGER(KIND=1)` constants. Specify '`-ftypeless-boz`' to cause such constants to be interpreted as typeless.

  (Version 0.5.19 introduced '`-fno-typeless-boz`' and its inverse.)

  See Section 7.4 [Options Controlling Fortran Dialect], page 29, for information on the '`-ftypeless-boz`' option.

- Options '`-ff90-intrinsics-enable`' and '`-fvxt-intrinsics-enable`' now are the defaults.

  Some programs might use names that clash with intrinsic names defined (and now enabled) by these options or by the new `libU77` intrinsics. Users of such programs might need to compile them differently (using, for example, '`-ff90-intrinsics-disable`') or, better yet, insert appropriate `EXTERNAL` statements specifying that these names are not intended to be names of intrinsics.

- The '`ALWAYS_FLUSH`' macro is no longer defined when building `libf2c`, which should result in improved I/O performance, especially over NFS.

  Note: If you have code that depends on the behavior of `libf2c` when built with '`ALWAYS_FLUSH`' defined, you will have to modify `libf2c` accordingly before building it from this and future versions of `g77`.

  See Section 17.4.8 [Output Assumed To Flush], page 260, for more information.

- Dave Love's implementation of `libU77` has been added to the version of `libf2c` distributed with and built by `g77`. `g77` now knows about the routines in this library as intrinsics.

- New option '`-fvxt`' specifies that the source file is written in VXT Fortran, instead of GNU Fortran.

  See Section 11.6 [VXT Fortran], page 175, for more information on the constructs recognized when the '`-fvxt`' option is specified.

- The '`-fvxt-not-f90`' option has been deleted, along with its inverse, '`-ff90-not-vxt`'.

  If you used one of these deleted options, you should re-read the pertinent documentation to determine which options, if any, are appropriate for compiling your code with this version of `g77`.

  See Chapter 11 [Other Dialects], page 169, for more information.

- The '`-fugly`' option now issues a warning, as it likely will be removed in a future version.

  (Enabling all the '`-fugly-*`' options is unlikely to be feasible, or sensible, in the future, so users should learn to specify only those '`-fugly-*`' options they really need for a particular source file.)

- The '`-fugly-assumed`' option, introduced in version 0.5.19, has been changed to better accommodate old and new code. See Section 11.9.2 [Ugly Assumed-Size Arrays], page 179, for more information.

- Related to supporting Alpha (AXP) machines, the `LOC()` intrinsic and `%LOC()` construct now return values of `INTEGER(KIND=0)` type, as defined by the GNU Fortran language.

  This type is wide enough (holds the same number of bits) as the character-pointer type on the machine.

  On most systems, this won't make a noticable difference, whereas on Alphas and other systems with 64-bit pointers, the `INTEGER(KIND=0)` type is equivalent to `INTEGER(KIND=2)` (often referred to as `INTEGER*8`) instead of the more common `INTEGER(KIND=1)` (often referred to as `INTEGER*4`).
- Emulate `COMPLEX` arithmetic in the `g77` front end, to avoid bugs in `complex` support in the `gcc` back end. New option '`-fno-emulate-complex`' causes `g77` to revert the 0.5.19 behavior.
- Dummy arguments are no longer assumed to potentially alias (overlap) other dummy arguments or `COMMON` areas when any of these are defined (assigned to) by Fortran code.

  This can result in faster and/or smaller programs when compiling with optimization enabled, though on some systems this effect is observed only when '`-fforce-addr`' also is specified.

  New options '`-falias-check`', '`-fargument-alias`', '`-fargument-noalias`', and '`-fno-argument-noalias-global`' control the way `g77` handles potential aliasing.

  See Section 17.4.7 [Aliasing Assumed To Work], page 258, for detailed information on why the new defaults might result in some programs no longer working the way they did when compiled by previous versions of `g77`.
- New option '`-fugly-assign`' specifies that the same memory locations are to be used to hold the values assigned by both statements '`I = 3`' and '`ASSIGN 10 TO I`', for example. (Normally, `g77` uses a separate memory location to hold assigned statement labels.)

  See Section 11.9.7 [Ugly Assigned Labels], page 181, for more information.
- `FORMAT` and `ENTRY` statements now are allowed to precede `IMPLICIT NONE` statements.
- Enable full support of `INTEGER(KIND=2)` (often referred to as `INTEGER*8`) available in `libf2c` and '`f2c.h`' so that `f2c` users may make full use of its features via the `g77` version of '`f2c.h`' and the `INTEGER(KIND=2)` support routines in the `g77` version of `libf2c`.
- Improve `g77` driver and `libf2c` so that '`g77 -v`' yields version information on the library.
- The `SNGL` and `FLOAT` intrinsics now are specific intrinsics, instead of synonyms for the generic intrinsic `REAL`.
- New intrinsics have been added. These are `REALPART`, `IMAGPART`, `COMPLEX`, `LONG`, and `SHORT`.
- A new group of intrinsics, '`gnu`', has been added to contain the new `REALPART`, `IMAGPART`, and `COMPLEX` intrinsics. An old group, '`dcp`', has been removed.

In 0.5.19:

- A temporary kludge option provides bare-bones information on `COMMON` and `EQUIVALENCE` members at debug time. See Section 7.10 [Options for Code Generation Conventions], page 41, for information on the '`-fdebug-kludge`' option.
- New '`-fonetrip`' option specifies FORTRAN-66-style one-trip `DO` loops.
- New '`-fno-silent`' option causes names of program units to be printed as they are compiled, in a fashion similar to UNIX `f77` and `f2c`.
- New '`-fugly-assumed`' option specifies that arrays dimensioned via '`DIMENSION X(1)`', for example, are to be treated as assumed-size.
- New '`-fno-typeless-boz`' option specifies that non-decimal-radix constants using the prefixed-radix form (such as '`Z'1234'`') are to be interpreted as `INTEGER(KIND=1)` constants.
- New '`-ff66`' option is a "shorthand" option that specifies behaviors considered appropriate for FORTRAN 66 programs.
- New '`-ff77`' option is a "shorthand" option that specifies behaviors considered appropriate for UNIX `f77` programs.
- New '`-fugly-comma`' and '`-fugly-logint`' options provided to perform some of what '`-fugly`' used to do. '`-fugly`' and '`-fno-ugly`' are now "shorthand" options, in that they do nothing more than enable (or disable) other '`-fugly-*`' options.
- Change code generation for list-directed I/O so it allows for new versions of `libf2c` that might return non-zero status codes for some operations previously assumed to always return zero.

  This change not only affects how `IOSTAT=` variables are set by list-directed I/O, it also affects whether `END=` and `ERR=` labels are reached by these operations.
- Add intrinsic support for new `FTELL` and `FSEEK` procedures in `libf2c`.
- Add options '`--help`' and '`--version`' to the `g77` command, to conform to GNU coding guidelines. Also add printing of `g77` version number when the '`--verbose`' ('`-v`') option is used.

In 0.5.18:

- The `BYTE` and `WORD` statements now are supported, to a limited extent.
- `INTEGER*1`, `INTEGER*2`, `INTEGER*8`, and their `LOGICAL` equivalents, now are supported to a limited extent. Among the missing elements are complete intrinsic and constant support.
- Support automatic arrays in procedures. For example, '`REAL A(N)`', where '`A`' is not a dummy argument, specifies that '`A`' is an automatic array. The size of '`A`' is calculated from the value of '`N`' each time the procedure is called, that amount of space is allocated, and that space is freed when the procedure returns to its caller.
- Add '`-fno-zeros`' option, enabled by default, to reduce compile-time CPU and memory usage for code that provides initial zero values for variables and arrays.
- Introduce three new options that apply to all compilations by `g77`-aware GNU compilers—'`-fmove-all-movables`', '`-freduce-all-givs`', and '`-frerun-loop-opt`'—which can improve the run-time performance of some programs.

- Replace much of the existing documentation with a single Info document.
- New option '`-fno-second-underscore`'.

In 0.5.17:

- The `ERF()` and `ERFC()` intrinsics now are generic intrinsics, mapping to `ERF/DERF` and `ERFC/DERFC`, respectively. *Note:* Use '`INTRINSIC ERF,ERFC`' in any code that might reference these as generic intrinsics, to improve the likelihood of diagnostics (instead of subtle run-time bugs) when using compilers that don't support these as intrinsics.
- New option '`-Wsurprising`'.
- DO loops with non-`INTEGER` variables now diagnosed only when '`-Wsurprising`' specified. Previously, this was diagnosed *unless* '`-fpedantic`' or '`-fugly`' was specified.

In 0.5.16:

- `libf2c` changed to output a leading zero (0) digit for floating-point values output via list-directed and formatted output (to bring `g77` more into line with many existing Fortran implementations—the ANSI FORTRAN 77 standard leaves this choice to the implementation).
- `libf2c` no longer built with debugging information intact, making it much smaller.
- Automatic installation of the `g77` command now works.
- Diagnostic messages now more informative, a la `gcc`, including messages like '`In function 'foo':`' and '`In file included from...:`'.
- New group of intrinsics called '`unix`', including `ABORT`, `DERF`, `DERFC`, `ERF`, `ERFC`, `EXIT`, `FLUSH`, `GETARG`, `GETENV`, `SIGNAL`, and `SYSTEM`.
- '`-funix-intrinsics-{delete,hide,disable,enable}`' options added.
- '`-fno-underscoring`' option added.
- '`--driver`' option added to the `g77` command.
- Support for the `gcc` options '`-fident`' and '`-fno-ident`' added.
- '`g77 -v`' returns much more version info, making the submission of better bug reports easily.
- Many improvements to the `g77` command to better fulfill its role as a front-end to the `gcc` driver. For example, `g77` now recognizes '`--verbose`' as a verbose way of specifying '`-v`'.
- Compiling preprocessed ('`*.F`' and '`*.fpp`') files now results in better diagnostics and debugging information, as the source-location info now is passed all the way through the compilation process instead of being lost.

10 The GNU Fortran Language

GNU Fortran supports a variety of extensions to, and dialects of, the Fortran language. Its primary base is the ANSI FORTRAN 77 standard, currently available on the network at `http://kumo.swcp.com/fortran/F77_std/f77_std.html` or in `ftp://ftp.ast.cam.ac.uk/pub/michael/`. It offers some extensions that are popular among users of UNIX `f77` and `f2c` compilers, some that are popular among users of other compilers (such as Digital products), some that are popular among users of the newer Fortran 90 standard, and some that are introduced by GNU Fortran.

(If you need a text on Fortran, a few freely available electronic references have pointers from `http://www.fortran.com/fortran/Books/`.)

Part of what defines a particular implementation of a Fortran system, such as `g77`, is the particular characteristics of how it supports types, constants, and so on. Much of this is left up to the implementation by the various Fortran standards and accepted practice in the industry.

The GNU Fortran *language* is described below. Much of the material is organized along the same lines as the ANSI FORTRAN 77 standard itself.

See Chapter 11 [Other Dialects], page 169, for information on features `g77` supports that are not part of the GNU Fortran language.

Note: This portion of the documentation definitely needs a lot of work!

10.1 Direction of Language Development

The purpose of the following description of the GNU Fortran language is to promote wide portability of GNU Fortran programs.

GNU Fortran is an evolving language, due to the fact that `g77` itself is in beta test. Some current features of the language might later be redefined as dialects of Fortran supported by `g77` when better ways to express these features are added to `g77`, for example. Such features would still be supported by `g77`, but would be available only when one or more command-line options were used.

The GNU Fortran *language* is distinct from the GNU Fortran *compilation system* (`g77`).

For example, `g77` supports various dialects of Fortran—in a sense, these are languages other than GNU Fortran—though its primary purpose is to support the GNU Fortran language, which also is described in its documentation and by its implementation.

On the other hand, non-GNU compilers might offer support for the GNU Fortran language, and are encouraged to do so.

Currently, the GNU Fortran language is a fairly fuzzy object. It represents something of a cross between what `g77` accepts when compiling using the prevailing defaults and what this document describes as being part of the language.

Future versions of `g77` are expected to clarify the definition of the language in the documentation. Often, this will mean adding new features to the language, in the form of both new documentation and new support in `g77`. However, it might occasionally mean removing a feature from the language itself to "dialect" status. In such a case, the documentation

would be adjusted to reflect the change, and `g77` itself would likely be changed to require one or more command-line options to continue supporting the feature.

The development of the GNU Fortran language is intended to strike a balance between:

- Serving as a mostly-upwards-compatible language from the de facto UNIX Fortran dialect as supported by `f77`.
- Offering new, well-designed language features. Attributes of such features include not making existing code any harder to read (for those who might be unaware that the new features are not in use) and not making state-of-the-art compilers take longer to issue diagnostics, among others.
- Supporting existing, well-written code without gratuitously rejecting non-standard constructs, regardless of the origin of the code (its dialect).
- Offering default behavior and command-line options to reduce and, where reasonable, eliminate the need for programmers to make any modifications to code that already works in existing production environments.
- Diagnosing constructs that have different meanings in different systems, languages, and dialects, while offering clear, less ambiguous ways to express each of the different meanings so programmers can change their code appropriately.

One of the biggest practical challenges for the developers of the GNU Fortran language is meeting the sometimes contradictory demands of the above items.

For example, a feature might be widely used in one popular environment, but the exact same code that utilizes that feature might not work as expected—perhaps it might mean something entirely different—in another popular environment.

Traditionally, Fortran compilers—even portable ones—have solved this problem by simply offering the appropriate feature to users of the respective systems. This approach treats users of various Fortran systems and dialects as remote "islands", or camps, of programmers, and assume that these camps rarely come into contact with each other (or, especially, with each other's code).

Project GNU takes a radically different approach to software and language design, in that it assumes that users of GNU software do not necessarily care what kind of underlying system they are using, regardless of whether they are using software (at the user-interface level) or writing it (for example, writing Fortran or C code).

As such, GNU users rarely need consider just what kind of underlying hardware (or, in many cases, operating system) they are using at any particular time. They can use and write software designed for a general-purpose, widely portable, heteregenous environment—the GNU environment.

In line with this philosophy, GNU Fortran must evolve into a product that is widely ported and portable not only in the sense that it can be successfully built, installed, and run by users, but in the larger sense that its users can use it in the same way, and expect largely the same behaviors from it, regardless of the kind of system they are using at any particular time.

This approach constrains the solutions `g77` can use to resolve conflicts between various camps of Fortran users. If these two camps disagree about what a particular construct should mean, `g77` cannot simply be changed to treat that particular construct as having

one meaning without comment (such as a warning), lest the users expecting it to have the other meaning are unpleasantly surprised that their code misbehaves when executed.

The use of the ASCII backslash character in character constants is an excellent (and still somewhat unresolved) example of this kind of controversy. See Section 18.5.1 [Backslash in Constants], page 284. Other examples are likely to arise in the future, as `g77` developers strive to improve its ability to accept an ever-wider variety of existing Fortran code without requiring significant modifications to said code.

Development of GNU Fortran is further constrained by the desire to avoid requiring programmers to change their code. This is important because it allows programmers, administrators, and others to more faithfully evaluate and validate `g77` (as an overall product and as new versions are distributed) without having to support multiple versions of their programs so that they continue to work the same way on their existing systems (non-GNU perhaps, but possibly also earlier versions of `g77`).

10.2 ANSI FORTRAN 77 Standard Support

GNU Fortran supports ANSI FORTRAN 77 with the following caveats. In summary, the only ANSI FORTRAN 77 features `g77` doesn't support are those that are probably rarely used in actual code, some of which are explicitly disallowed by the Fortran 90 standard.

10.2.1 No Passing External Assumed-length

`g77` disallows passing of an external procedure as an actual argument if the procedure's type is declared `CHARACTER*(*)`. For example:

```
CHARACTER*(*) CFUNC
EXTERNAL CFUNC
CALL FOO(CFUNC)
END
```

It isn't clear whether the standard considers this conforming.

10.2.2 No Passing Dummy Assumed-length

`g77` disallows passing of a dummy procedure as an actual argument if the procedure's type is declared `CHARACTER*(*)`.

```
SUBROUTINE BAR(CFUNC)
CHARACTER*(*) CFUNC
EXTERNAL CFUNC
CALL FOO(CFUNC)
END
```

It isn't clear whether the standard considers this conforming.

10.2.3 No Pathological Implied-DO

The `DO` variable for an implied-`DO` construct in a `DATA` statement may not be used as the `DO` variable for an outer implied-`DO` construct. For example, this fragment is disallowed by `g77`:

```
DATA ((A(I, I), I= 1, 10), I= 1, 10) /.../
```

This also is disallowed by Fortran 90, as it offers no additional capabilities and would have a variety of possible meanings.

Note that it is *very* unlikely that any production Fortran code tries to use this unsupported construct.

10.2.4 No Useless Implied-DO

An array element initializer in an implied-`DO` construct in a `DATA` statement must contain at least one reference to the `DO` variables of each outer implied-`DO` construct. For example, this fragment is disallowed by `g77`:

```
DATA (A, I= 1, 1) /1./
```

This also is disallowed by Fortran 90, as FORTRAN 77's more permissive requirements offer no additional capabilities. However, `g77` doesn't necessarily diagnose all cases where this requirement is not met.

Note that it is *very* unlikely that any production Fortran code tries to use this unsupported construct.

10.3 Conformance

(The following information augments or overrides the information in Section 1.4 of ANSI X3.9-1978 FORTRAN 77 in specifying the GNU Fortran language. Chapter 1 of that document otherwise serves as the basis for the relevant aspects of GNU Fortran.)

The definition of the GNU Fortran language is akin to that of the ANSI FORTRAN 77 language in that it does not generally require conforming implementations to diagnose cases where programs do not conform to the language.

However, `g77` as a compiler is being developed in a way that is intended to enable it to diagnose such cases in an easy-to-understand manner.

A program that conforms to the GNU Fortran language should, when compiled, linked, and executed using a properly installed `g77` system, perform as described by the GNU Fortran language definition. Reasons for different behavior include, among others:

- Use of resources (memory—heap, stack, and so on; disk space; CPU time; etc.) exceeds those of the system.
- Range and/or precision of calculations required by the program exceeds that of the system.
- Excessive reliance on behaviors that are system-dependent (non-portable Fortran code).
- Bugs in the program.
- Bug in `g77`.
- Bugs in the system.

Despite these "loopholes", the availability of a clear specification of the language of programs submitted to `g77`, as this document is intended to provide, is considered an important aspect of providing a robust, clean, predictable Fortran implementation.

The definition of the GNU Fortran language, while having no special legal status, can therefore be viewed as a sort of contract, or agreement. This agreement says, in essence, "if you write a program in this language, and run it in an environment (such as a `g77` system) that supports this language, the program should behave in a largely predictable way".

10.4 Notation Used in This Chapter

(The following information augments or overrides the information in Section 1.5 of ANSI X3.9-1978 FORTRAN 77 in specifying the GNU Fortran language. Chapter 1 of that document otherwise serves as the basis for the relevant aspects of GNU Fortran.)

In this chapter, "must" denotes a requirement, "may" denotes permission, and "must not" and "may not" denote prohibition. Terms such as "might", "should", and "can" generally add little or nothing in the way of weight to the GNU Fortran language itself, but are used to explain or illustrate the language.

For example:

> "The `FROBNITZ` statement must precede all executable statements in a program unit, and may not specify any dummy arguments. It may specify local or common variables and arrays. Its use should be limited to portions of the program designed to be non-portable and system-specific, because it might cause the containing program unit to behave quite differently on different systems."

Insofar as the GNU Fortran language is specified, the requirements and permissions denoted by the above sample statement are limited to the placement of the statement and the kinds of things it may specify. The rest of the statement—the content regarding non-portable portions of the program and the differing behavior of program units containing the `FROBNITZ` statement—does not pertain the GNU Fortran language itself. That content offers advice and warnings about the `FROBNITZ` statement.

Remember: The GNU Fortran language definition specifies both what constitutes a valid GNU Fortran program and how, given such a program, a valid GNU Fortran implementation is to interpret that program.

It is *not* incumbent upon a valid GNU Fortran implementation to behave in any particular way, any consistent way, or any predictable way when it is asked to interpret input that is *not* a valid GNU Fortran program.

Such input is said to have *undefined* behavior when interpreted by a valid GNU Fortran implementation, though an implementation may choose to specify behaviors for some cases of inputs that are not valid GNU Fortran programs.

Other notation used herein is that of the GNU texinfo format, which is used to generate printed hardcopy, on-line hypertext (Info), and on-line HTML versions, all from a single source document. This notation is used as follows:

- Keywords defined by the GNU Fortran language are shown in uppercase, as in: `COMMON`, `INTEGER`, and `BLOCK DATA`.

  Note that, in practice, many Fortran programs are written in lowercase—uppercase is used in this manual as a means to readily distinguish keywords and sample Fortran-related text from the prose in this document.

- Portions of actual sample program, input, or output text look like this: '`Actual program text`'.

  Generally, uppercase is used for all Fortran-specific and Fortran-related text, though this does not always include literal text within Fortran code.

  For example: '`PRINT *, 'My name is Bob'`'.

- A metasyntactic variable—that is, a name used in this document to serve as a placeholder for whatever text is used by the user or programmer–appears as shown in the following example:

  "The `INTEGER` *ivar* statement specifies that *ivar* is a variable or array of type `INTEGER`."

  In the above example, any valid text may be substituted for the metasyntactic variable *ivar* to make the statement apply to a specific instance, as long as the same text is substituted for *both* occurrences of *ivar*.

- Ellipses ("...") are used to indicate further text that is either unimportant or expanded upon further, elsewhere.

- Names of data types are in the style of Fortran 90, in most cases.

  See Section 10.7.1.3 [Kind Notation], page 84, for information on the relationship between Fortran 90 nomenclature (such as `INTEGER(KIND=1)`) and the more traditional, less portably concise nomenclature (such as `INTEGER*4`).

10.5 Fortran Terms and Concepts

(The following information augments or overrides the information in Chapter 2 of ANSI X3.9-1978 FORTRAN 77 in specifying the GNU Fortran language. Chapter 2 of that document otherwise serves as the basis for the relevant aspects of GNU Fortran.)

10.5.1 Syntactic Items

(Corresponds to Section 2.2 of ANSI X3.9-1978 FORTRAN 77.)

In GNU Fortran, a symbolic name is at least one character long, and has no arbitrary upper limit on length. However, names of entities requiring external linkage (such as external functions, external subroutines, and `COMMON` areas) might be restricted to some arbitrary length by the system. Such a restriction is no more constrained than that of one through six characters.

Underscores ('`_`') are accepted in symbol names after the first character (which must be a letter).

10.5.2 Statements, Comments, and Lines

(Corresponds to Section 2.3 of ANSI X3.9-1978 FORTRAN 77.)

Use of an exclamation point ('`!`') to begin a trailing comment (a comment that extends to the end of the same source line) is permitted under the following conditions:

- The exclamation point does not appear in column 6. Otherwise, it is treated as an indicator of a continuation line.

- The exclamation point appears outside a character or hollerith constant. Otherwise, the exclamation point is considered part of the constant.
- The exclamation point appears to the left of any other possible trailing comment. That is, a trailing comment may contain exclamation points in their commentary text.

Use of a semicolon (';') as a statement separator is permitted under the following conditions:

- The semicolon appears outside a character or hollerith constant. Otherwise, the semicolon is considered part of the constant.
- The semicolon appears to the left of a trailing comment. Otherwise, the semicolon is considered part of that comment.
- Neither a logical `IF` statement nor a non-construct `WHERE` statement (a Fortran 90 feature) may be followed (in the same, possibly continued, line) by a semicolon used as a statement separator.

  This restriction avoids the confusion that can result when reading a line such as:

  ```
  IF (VALIDP) CALL FOO; CALL BAR
  ```

  Some readers might think the '`CALL BAR`' is executed only if '`VALIDP`' is `.TRUE.`, while others might assume its execution is unconditional.

  (At present, `g77` does not diagnose code that violates this restriction.)

10.5.3 Scope of Symbolic Names and Statement Labels

(Corresponds to Section 2.9 of ANSI X3.9-1978 FORTRAN 77.)

Included in the list of entities that have a scope of a program unit are construct names (a Fortran 90 feature). See Section 10.10.3 [Construct Names], page 89, for more information.

10.6 Characters, Lines, and Execution Sequence

(The following information augments or overrides the information in Chapter 3 of ANSI X3.9-1978 FORTRAN 77 in specifying the GNU Fortran language. Chapter 3 of that document otherwise serves as the basis for the relevant aspects of GNU Fortran.)

10.6.1 GNU Fortran Character Set

(Corresponds to Section 3.1 of ANSI X3.9-1978 FORTRAN 77.)

Letters include uppercase letters (the twenty-six characters of the English alphabet) and lowercase letters (their lowercase equivalent). Generally, lowercase letters may be used in place of uppercase letters, though in character and hollerith constants, they are distinct.

Special characters include:

- Semicolon (';')
- Exclamation point ('!')
- Double quote ('"')
- Backslash ('\')
- Question mark ('?')

- Hash mark (‘#’)
- Ampersand (‘&’)
- Percent sign (‘%’)
- Underscore (‘_’)
- Open angle (‘<’)
- Close angle (‘>’)
- The FORTRAN 77 special characters (⟨SPC⟩, ‘=’, ‘+’, ‘-’, ‘*’, ‘/’, ‘(’, ‘)’, ‘,’, ‘.’, ‘$’, ‘’’, and ‘:’)

Note that this document refers to ⟨SPC⟩ as *space*, while X3.9-1978 FORTRAN 77 refers to it as *blank*.

10.6.2 Lines

(Corresponds to Section 3.2 of ANSI X3.9-1978 FORTRAN 77.)

The way a Fortran compiler views source files depends entirely on the implementation choices made for the compiler, since those choices are explicitly left to the implementation by the published Fortran standards.

The GNU Fortran language mandates a view applicable to UNIX-like text files—files that are made up of an arbitrary number of lines, each with an arbitrary number of characters (sometimes called stream-based files).

This view does not apply to types of files that are specified as having a particular number of characters on every single line (sometimes referred to as record-based files).

Because a “line in a program unit is a sequence of 72 characters”, to quote X3.9-1978, the GNU Fortran language specifies that a stream-based text file is translated to GNU Fortran lines as follows:

- A newline in the file is the character that represents the end of a line of text to the underlying system. For example, on ASCII-based systems, a newline is the ⟨NL⟩ character, which has ASCII value 12 (decimal).
- Each newline in the file serves to end the line of text that precedes it (and that does not contain a newline).
- The end-of-file marker (`EOF`) also serves to end the line of text that precedes it (and that does not contain a newline).
- Any line of text that is shorter than 72 characters is padded to that length with spaces (called “blanks” in the standard).
- Any line of text that is longer than 72 characters is truncated to that length, but the truncated remainder must consist entirely of spaces.
- Characters other than newline and the GNU Fortran character set are invalid.

For the purposes of the remainder of this description of the GNU Fortran language, the translation described above has already taken place, unless otherwise specified.

The result of the above translation is that the source file appears, in terms of the remainder of this description of the GNU Fortran language, as if it had an arbitrary number of 72-character lines, each character being among the GNU Fortran character set.

For example, if the source file itself has two newlines in a row, the second newline becomes, after the above translation, a single line containing 72 spaces.

10.6.3 Continuation Line

(Corresponds to Section 3.2.3 of ANSI X3.9-1978 FORTRAN 77.)

A continuation line is any line that both

- Contains a continuation character, and
- Contains only spaces in columns 1 through 5

A continuation character is any character of the GNU Fortran character set other than space (⟨SPC⟩) or zero ('0') in column 6, or a digit ('0' through '9') in column 7 through 72 of a line that has only spaces to the left of that digit.

The continuation character is ignored as far as the content of the statement is concerned.

The GNU Fortran language places no limit on the number of continuation lines in a statement. In practice, the limit depends on a variety of factors, such as available memory, statement content, and so on, but no GNU Fortran system may impose an arbitrary limit.

10.6.4 Statements

(Corresponds to Section 3.3 of ANSI X3.9-1978 FORTRAN 77.)

Statements may be written using an arbitrary number of continuation lines.

Statements may be separated using the semicolon (';'), except that the logical `IF` and non-construct `WHERE` statements may not be separated from subsequent statements using only a semicolon as statement separator.

The `END PROGRAM`, `END SUBROUTINE`, `END FUNCTION`, and `END BLOCK DATA` statements are alternatives to the `END` statement. These alternatives may be written as normal statements—they are not subject to the restrictions of the `END` statement.

However, no statement other than `END` may have an initial line that appears to be an `END` statement—even `END PROGRAM`, for example, must not be written as:

```
 END
&PROGRAM
```

10.6.5 Statement Labels

(Corresponds to Section 3.4 of ANSI X3.9-1978 FORTRAN 77.)

A statement separated from its predecessor via a semicolon may be labeled as follows:

- The semicolon is followed by the label for the statement, which in turn follows the label.
- The label must be no more than five digits in length.
- The first digit of the label for the statement is not the first non-space character on a line. Otherwise, that character is treated as a continuation character.

A statement may have only one label defined for it.

10.6.6 Order of Statements and Lines

(Corresponds to Section 3.5 of ANSI X3.9-1978 FORTRAN 77.)

Generally, `DATA` statements may precede executable statements. However, specification statements pertaining to any entities initialized by a `DATA` statement must precede that `DATA` statement. For example, after '`DATA I/1/`', '`INTEGER I`' is not permitted, but '`INTEGER J`' is permitted.

The last line of a program unit may be an `END` statement, or may be:

- An `END PROGRAM` statement, if the program unit is a main program.
- An `END SUBROUTINE` statement, if the program unit is a subroutine.
- An `END FUNCTION` statement, if the program unit is a function.
- An `END BLOCK DATA` statement, if the program unit is a block data.

10.6.7 Including Source Text

Additional source text may be included in the processing of the source file via the `INCLUDE` directive:

```
INCLUDE filename
```

The source text to be included is identified by *filename*, which is a literal GNU Fortran character constant. The meaning and interpretation of *filename* depends on the implementation, but typically is a filename.

(`g77` treats it as a filename that it searches for in the current directory and/or directories specified via the '`-I`' command-line option.)

The effect of the `INCLUDE` directive is as if the included text directly replaced the directive in the source file prior to interpretation of the program. Included text may itself use `INCLUDE`. The depth of nested `INCLUDE` references depends on the implementation, but typically is a positive integer.

This virtual replacement treats the statements and `INCLUDE` directives in the included text as syntactically distinct from those in the including text.

Therefore, the first non-comment line of the included text must not be a continuation line. The included text must therefore have, after the non-comment lines, either an initial line (statement), an `INCLUDE` directive, or nothing (the end of the included text).

Similarly, the including text may end the `INCLUDE` directive with a semicolon or the end of the line, but it cannot follow an `INCLUDE` directive at the end of its line with a continuation line. Thus, the last statement in an included text may not be continued.

Any statements between two `INCLUDE` directives on the same line are treated as if they appeared in between the respective included texts. For example:

```
INCLUDE 'A'; PRINT *, 'B'; INCLUDE 'C'; END PROGRAM
```

If the text included by '`INCLUDE 'A'`' constitutes a '`PRINT *, 'A'`' statement and the text included by '`INCLUDE 'C'`' constitutes a '`PRINT *, 'C'`' statement, then the output of the above sample program would be

```
A
B
```

```
C
```

(with suitable allowances for how an implementation defines its handling of output).

Included text must not include itself directly or indirectly, regardless of whether the *filename* used to reference the text is the same.

Note that `INCLUDE` is *not* a statement. As such, it is neither a non-executable or executable statement. However, if the text it includes constitutes one or more executable statements, then the placement of `INCLUDE` is subject to effectively the same restrictions as those on executable statements.

An `INCLUDE` directive may be continued across multiple lines as if it were a statement. This permits long names to be used for *filename*.

10.7 Data Types and Constants

(The following information augments or overrides the information in Chapter 4 of ANSI X3.9-1978 FORTRAN 77 in specifying the GNU Fortran language. Chapter 4 of that document otherwise serves as the basis for the relevant aspects of GNU Fortran.)

To more concisely express the appropriate types for entities, this document uses the more concise Fortran 90 nomenclature such as `INTEGER(KIND=1)` instead of the more traditional, but less portably concise, byte-size-based nomenclature such as `INTEGER*4`, wherever reasonable.

When referring to generic types—in contexts where the specific precision and range of a type are not important—this document uses the generic type names `INTEGER`, `LOGICAL`, `REAL`, `COMPLEX`, and `CHARACTER`.

In some cases, the context requires specification of a particular type. This document uses the '`KIND=`' notation to accomplish this throughout, sometimes supplying the more traditional notation for clarification, though the traditional notation might not work the same way on all GNU Fortran implementations.

Use of '`KIND=`' makes this document more concise because `g77` is able to define values for '`KIND=`' that have the same meanings on all systems, due to the way the Fortran 90 standard specifies these values are to be used.

(In particular, that standard permits an implementation to arbitrarily assign nonnegative values. There are four distinct sets of assignments: one to the `CHARACTER` type; one to the `INTEGER` type; one to the `LOGICAL` type; and the fourth to both the `REAL` and `COMPLEX` types. Implementations are free to assign these values in any order, leave gaps in the ordering of assignments, and assign more than one value to a representation.)

This makes '`KIND=`' values superior to the values used in non-standard statements such as '`INTEGER*4`', because the meanings of the values in those statements vary from machine to machine, compiler to compiler, even operating system to operating system.

However, use of '`KIND=`' is *not* generally recommended when writing portable code (unless, for example, the code is going to be compiled only via `g77`, which is a widely ported compiler). GNU Fortran does not yet have adequate language constructs to permit use of '`KIND=`' in a fashion that would make the code portable to Fortran 90 implementations; and, this construct is known to *not* be accepted by many popular FORTRAN 77 implementations, so it cannot be used in code that is to be ported to those.

The distinction here is that this document is able to use specific values for 'KIND=' to concisely document the types of various operations and operands.

A Fortran program should use the FORTRAN 77 designations for the appropriate GNU Fortran types—such as INTEGER for INTEGER(KIND=1), REAL for REAL(KIND=1), and DOUBLE COMPLEX for COMPLEX(KIND=2)—and, where no such designations exist, make use of appropriate techniques (preprocessor macros, parameters, and so on) to specify the types in a fashion that may be easily adjusted to suit each particular implementation to which the program is ported. (These types generally won't need to be adjusted for ports of g77.)

Further details regarding GNU Fortran data types and constants are provided below.

10.7.1 Data Types

(Corresponds to Section 4.1 of ANSI X3.9-1978 FORTRAN 77.)

GNU Fortran supports these types:

1. Integer (generic type INTEGER)
2. Real (generic type REAL)
3. Double precision
4. Complex (generic type COMPLEX)
5. Logical (generic type LOGICAL)
6. Character (generic type CHARACTER)
7. Double Complex

(The types numbered 1 through 6 above are standard FORTRAN 77 types.)

The generic types shown above are referred to in this document using only their generic type names. Such references usually indicate that any specific type (kind) of that generic type is valid.

For example, a context described in this document as accepting the COMPLEX type also is likely to accept the DOUBLE COMPLEX type.

The GNU Fortran language supports three ways to specify a specific kind of a generic type.

10.7.1.1 Double Notation

The GNU Fortran language supports two uses of the keyword DOUBLE to specify a specific kind of type:

- DOUBLE PRECISION, equivalent to REAL(KIND=2)
- DOUBLE COMPLEX, equivalent to COMPLEX(KIND=2)

Use one of the above forms where a type name is valid.

While use of this notation is popular, it doesn't scale well in a language or dialect rich in intrinsic types, as is the case for the GNU Fortran language (especially planned future versions of it).

After all, one rarely sees type names such as 'DOUBLE INTEGER', 'QUADRUPLE REAL', or 'QUARTER INTEGER'. Instead, INTEGER*8, REAL*16, and INTEGER*1 often are substituted for

these, respectively, even though they do not always have the same meanings on all systems. (And, the fact that '`DOUBLE REAL`' does not exist as such is an inconsistency.)

Therefore, this document uses "double notation" only on occasion for the benefit of those readers who are accustomed to it.

10.7.1.2 Star Notation

The following notation specifies the storage size for a type:

generic-type`*`*n*

generic-type must be a generic type—one of `INTEGER`, `REAL`, `COMPLEX`, `LOGICAL`, or `CHARACTER`. *n* must be one or more digits comprising a decimal integer number greater than zero.

Use the above form where a type name is valid.

The '`*`*n*' notation specifies that the amount of storage occupied by variables and array elements of that type is *n* times the storage occupied by a `CHARACTER*1` variable.

This notation might indicate a different degree of precision and/or range for such variables and array elements, and the functions that return values of types using this notation. It does not limit the precision or range of values of that type in any particular way—use explicit code to do that.

Further, the GNU Fortran language requires no particular values for *n* to be supported by an implementation via the '`*`*n*' notation. `g77` supports `INTEGER*1` (as `INTEGER(KIND=3)`) on all systems, for example, but not all implementations are required to do so, and `g77` is known to not support `REAL*1` on most (or all) systems.

As a result, except for *generic-type* of `CHARACTER`, uses of this notation should be limited to isolated portions of a program that are intended to handle system-specific tasks and are expected to be non-portable.

(Standard FORTRAN 77 supports the '`*`*n*' notation for only `CHARACTER`, where it signifies not only the amount of storage occupied, but the number of characters in entities of that type. However, almost all Fortran compilers have supported this notation for generic types, though with a variety of meanings for *n*.)

Specifications of types using the '`*`*n*' notation always are interpreted as specifications of the appropriate types described in this document using the '`KIND=`*n*' notation, described below.

While use of this notation is popular, it doesn't serve well in the context of a widely portable dialect of Fortran, such as the GNU Fortran language.

For example, even on one particular machine, two or more popular Fortran compilers might well disagree on the size of a type declared `INTEGER*2` or `REAL*16`. Certainly there is known to be disagreement over such things among Fortran compilers on *different* systems.

Further, this notation offers no elegant way to specify sizes that are not even multiples of the "byte size" typically designated by `INTEGER*1`. Use of "absurd" values (such as `INTEGER*1000`) would certainly be possible, but would perhaps be stretching the original intent of this notation beyond the breaking point in terms of widespread readability of documentation and code making use of it.

Therefore, this document uses "star notation" only on occasion for the benefit of those readers who are accustomed to it.

10.7.1.3 Kind Notation

The following notation specifies the kind-type selector of a type:

generic-type`(KIND=`*n*`)`

Use the above form where a type name is valid.

generic-type must be a generic type—one of `INTEGER`, `REAL`, `COMPLEX`, `LOGICAL`, or `CHARACTER`. *n* must be an integer initialization expression that is a positive, nonzero value.

Programmers are discouraged from writing these values directly into their code. Future versions of the GNU Fortran language will offer facilities that will make the writing of code portable to `g77` *and* Fortran 90 implementations simpler.

However, writing code that ports to existing FORTRAN 77 implementations depends on avoiding the '`KIND=`' construct.

The '`KIND=`' construct is thus useful in the context of GNU Fortran for two reasons:

- It provides a means to specify a type in a fashion that is portable across all GNU Fortran implementations (though not other FORTRAN 77 and Fortran 90 implementations).
- It provides a sort of Rosetta stone for this document to use to concisely describe the types of various operations and operands.

The values of *n* in the GNU Fortran language are assigned using a scheme that:

- Attempts to maximize the ability of readers of this document to quickly familiarize themselves with assignments for popular types
- Provides a unique value for each specific desired meaning
- Provides a means to automatically assign new values so they have a "natural" relationship to existing values, if appropriate, or, if no such relationship exists, will not interfere with future values assigned on the basis of such relationships
- Avoids using values that are similar to values used in the existing, popular '`*`*n*' notation, to prevent readers from expecting that these implied correspondences work on all GNU Fortran implementations

The assignment system accomplishes this by assigning to each "fundamental meaning" of a specific type a unique prime number. Combinations of fundamental meanings—for example, a type that is two times the size of some other type—are assigned values of *n* that are the products of the values for those fundamental meanings.

A prime value of *n* is never given more than one fundamental meaning, to avoid situations where some code or system cannot reasonably provide those meanings in the form of a single type.

The values of *n* assigned so far are:

`KIND=0` This value is reserved for future use.

The planned future use is for this value to designate, explicitly, context-sensitive kind-type selection. For example, the expression '`1D0 * 0.1_0`' would be equivalent to '`1D0 * 0.1D0`'.

`KIND=1` This corresponds to the default types for `REAL`, `INTEGER`, `LOGICAL`, `COMPLEX`, and `CHARACTER`, as appropriate.

These are the "default" types described in the Fortran 90 standard, though that standard does not assign any particular '`KIND=`' value to these types.

(Typically, these are `REAL*4`, `INTEGER*4`, `LOGICAL*4`, and `COMPLEX*8`.)

`KIND=2` This corresponds to types that occupy twice as much storage as the default types. `REAL(KIND=2)` is `DOUBLE PRECISION` (typically `REAL*8`), `COMPLEX(KIND=2)` is `DOUBLE COMPLEX` (typically `COMPLEX*16`),

These are the "double precision" types described in the Fortran 90 standard, though that standard does not assign any particular '`KIND=`' value to these types.

n of 4 thus corresponds to types that occupy four times as much storage as the default types, *n* of 8 to types that occupy eight times as much storage, and so on.

The `INTEGER(KIND=2)` and `LOGICAL(KIND=2)` types are not necessarily supported by every GNU Fortran implementation.

`KIND=3` This corresponds to types that occupy as much storage as the default `CHARACTER` type, which is the same effective type as `CHARACTER(KIND=1)` (making that type effectively the same as `CHARACTER(KIND=3)`).

(Typically, these are `INTEGER*1` and `LOGICAL*1`.)

n of 6 thus corresponds to types that occupy twice as much storage as the *n*=3 types, *n* of 12 to types that occupy four times as much storage, and so on.

These are not necessarily supported by every GNU Fortran implementation.

`KIND=5` This corresponds to types that occupy half the storage as the default (*n*=1) types.

(Typically, these are `INTEGER*2` and `LOGICAL*2`.)

n of 25 thus corresponds to types that occupy one-quarter as much storage as the default types.

These are not necessarily supported by every GNU Fortran implementation.

`KIND=7` This is valid only as `INTEGER(KIND=7)` and denotes the `INTEGER` type that has the smallest storage size that holds a pointer on the system.

A pointer representable by this type is capable of uniquely addressing a `CHARACTER*1` variable, array, array element, or substring.

(Typically this is equivalent to `INTEGER*4` or, on 64-bit systems, `INTEGER*8`. In a compatible C implementation, it typically would be the same size and semantics of the C type `void *`.)

Note that these are *proposed* correspondences and might change in future versions of `g77`—avoid writing code depending on them while `g77`, and therefore the GNU Fortran language it defines, is in beta testing.

Values not specified in the above list are reserved to future versions of the GNU Fortran language.

Implementation-dependent meanings will be assigned new, unique prime numbers so as to not interfere with other implementation-dependent meanings, and offer the possibility of increasing the portability of code depending on such types by offering support for them in other GNU Fortran implementations.

Other meanings that might be given unique values are:

- Types that make use of only half their storage size for representing precision and range.

  For example, some compilers offer options that cause `INTEGER` types to occupy the amount of storage that would be needed for `INTEGER(KIND=2)` types, but the range remains that of `INTEGER(KIND=1)`.
- The IEEE single floating-point type.
- Types with a specific bit pattern (endianness), such as the little-endian form of `INTEGER(KIND=1)`. These could permit, conceptually, use of portable code and implementations on data files written by existing systems.

Future *prime* numbers should be given meanings in as incremental a fashion as possible, to allow for flexibility and expressiveness in combining types.

For example, instead of defining a prime number for little-endian IEEE doubles, one prime number might be assigned the meaning "little-endian", another the meaning "IEEE double", and the value of *n* for a little-endian IEEE double would thus naturally be the product of those two respective assigned values. (It could even be reasonable to have IEEE values result from the products of prime values denoting exponent and fraction sizes and meanings, hidden bit usage, availability and representations of special values such as subnormals, infinities, and Not-A-Numbers (NaNs), and so on.)

This assignment mechanism, while not inherently required for future versions of the GNU Fortran language, is worth using because it could ease management of the "space" of supported types much easier in the long run.

The above approach suggests a mechanism for specifying inheritance of intrinsic (built-in) types for an entire, widely portable product line. It is certainly reasonable that, unlike programmers of other languages offering inheritance mechanisms that employ verbose names for classes and subclasses, along with graphical browsers to elucidate the relationships, Fortran programmers would employ a mechanism that works by multiplying prime numbers together and finding the prime factors of such products.

Most of the advantages for the above scheme have been explained above. One disadvantage is that it could lead to the defining, by the GNU Fortran language, of some fairly large prime numbers. This could lead to the GNU Fortran language being declared "munitions" by the United States Department of Defense.

10.7.2 Constants

(Corresponds to Section 4.2 of ANSI X3.9-1978 FORTRAN 77.)

A *typeless constant* has one of the following forms:

'*binary-digits*'`B`
'*octal-digits*'`O`
'*hexadecimal-digits*'`Z`
'*hexadecimal-digits*'`X`

binary-digits, *octal-digits*, and *hexadecimal-digits* are nonempty strings of characters in the set '`01`', '`01234567`', and '`0123456789ABCDEFabcdef`', respectively. (The value for '`A`' (and '`a`') is 10, for '`B`' and '`b`' is 11, and so on.)

Typeless constants have values that depend on the context in which they are used.

All other constants, called *typed constants*, are interpreted—converted to internal form—according to their inherent type. Thus, context is *never* a determining factor for the type, and hence the interpretation, of a typed constant. (All constants in the ANSI FORTRAN 77 language are typed constants.)

For example, '`1`' is always type `INTEGER(KIND=1)` in GNU Fortran (called default INTEGER in Fortran 90), '`9.435784839284958`' is always type `REAL(KIND=1)` (even if the additional precision specified is lost, and even when used in a `REAL(KIND=2)` context), '`1E0`' is always type `REAL(KIND=2)`, and '`1D0`' is always type `REAL(KIND=2)`.

10.7.3 Integer Type

(Corresponds to Section 4.3 of ANSI X3.9-1978 FORTRAN 77.)

An integer constant also may have one of the following forms:

`B'`*binary-digits*`'`
`O'`*octal-digits*`'`
`Z'`*hexadecimal-digits*`'`
`X'`*hexadecimal-digits*`'`

binary-digits, *octal-digits*, and *hexadecimal-digits* are nonempty strings of characters in the set '`01`', '`01234567`', and '`0123456789ABCDEFabcdef`', respectively. (The value for '`A`' (and '`a`') is 10, for '`B`' and '`b`' is 11, and so on.)

10.7.4 Character Type

(Corresponds to Section 4.8 of ANSI X3.9-1978 FORTRAN 77.)

A character constant may be delimited by a pair of double quotes ('`"`') instead of apostrophes. In this case, an apostrophe within the constant represents a single apostrophe, while a double quote is represented in the source text of the constant by two consecutive double quotes with no intervening spaces.

A character constant may be empty (have a length of zero).

A character constant may include a substring specification, The value of such a constant is the value of the substring—for example, the value of '`'hello'(3:5)`' is the same as the value of '`''llo''`'.

10.8 Expressions

(The following information augments or overrides the information in Chapter 6 of ANSI X3.9-1978 FORTRAN 77 in specifying the GNU Fortran language. Chapter 6 of that document otherwise serves as the basis for the relevant aspects of GNU Fortran.)

10.8.1 The `%LOC()` Construct

`%LOC(`*arg*`)`

The `%LOC()` construct is an expression that yields the value of the location of its argument, *arg*, in memory. The size of the type of the expression depends on the system—typically, it is equivalent to either `INTEGER(KIND=1)` or `INTEGER(KIND=2)`, though it is actually type `INTEGER(KIND=7)`.

The argument to `%LOC()` must be suitable as the left-hand side of an assignment statement. That is, it may not be a general expression involving operators such as addition, subtraction, and so on, nor may it be a constant.

Use of `%LOC()` is recommended 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.

Do not depend on `%LOC()` returning a pointer that can be safely used to *define* (change) the argument. While this might work in some circumstances, it is hard to predict whether it will continue to work when a program (that works using this unsafe behavior) is recompiled using different command-line options or a different version of `g77`.

Generally, `%LOC()` is safe when used as an argument to a procedure that makes use of the value of the corresponding dummy argument only during its activation, and only when such use is restricted to referencing (reading) the value of the argument to `%LOC()`.

Implementation Note: Currently, `g77` passes arguments (those not passed using a construct such as `%VAL()`) by reference or descriptor, depending on the type of the actual argument. Thus, given '`INTEGER I`', '`CALL FOO(I)`' would seem to mean the same thing as '`CALL FOO(%LOC(I))`', and in fact might compile to identical code.

However, '`CALL FOO(%LOC(I))`' emphatically means "pass the address of '`I`' in memory". While '`CALL FOO(I)`' might use that same approach in a particular version of `g77`, another version or compiler might choose a different implementation, such as copy-in/copy-out, to effect the desired behavior—and which will therefore not necessarily compile to the same code as would '`CALL FOO(%LOC(I))`' using the same version or compiler.

See Chapter 16 [Debugging and Interfacing], page 239, for detailed information on how this particular version of `g77` implements various constructs.

10.9 Specification Statements

(The following information augments or overrides the information in Chapter 8 of ANSI X3.9-1978 FORTRAN 77 in specifying the GNU Fortran language. Chapter 8 of that document otherwise serves as the basis for the relevant aspects of GNU Fortran.)

10.9.1 `NAMELIST` Statement

The `NAMELIST` statement, and related I/O constructs, are supported by the GNU Fortran language in essentially the same way as they are by `f2c`.

10.9.2 `DOUBLE COMPLEX` Statement

`DOUBLE COMPLEX` is a type-statement (and type) that specifies the type `COMPLEX(KIND=2)` in GNU Fortran.

10.10 Control Statements

(The following information augments or overrides the information in Chapter 11 of ANSI X3.9-1978 FORTRAN 77 in specifying the GNU Fortran language. Chapter 11 of that document otherwise serves as the basis for the relevant aspects of GNU Fortran.)

10.10.1 DO WHILE

The `DO WHILE` statement, a feature of both the MIL-STD 1753 and Fortran 90 standards, is provided by the GNU Fortran language.

10.10.2 END DO

The `END DO` statement is provided by the GNU Fortran language.

This statement is used in one of two ways:

- The Fortran 90 meaning, in which it specifies the termination point of a single `DO` loop started with a `DO` statement that specifies no termination label.
- The MIL-STD 1753 meaning, in which it specifies the termination point of one or more `DO` loops, all of which start with a `DO` statement that specify the label defined for the `END DO` statement.

  This kind of `END DO` statement is merely a synonym for `CONTINUE`, except it is permitted only when the statement is labeled and a target of one or more labeled `DO` loops.

  It is expected that this use of `END DO` will be removed from the GNU Fortran language in the future, though it is likely that it will long be supported by `g77` as a dialect form.

10.10.3 Construct Names

The GNU Fortran language supports construct names as defined by the Fortran 90 standard. These names are local to the program unit and are defined as follows:

construct-name: *block-statement*

Here, *construct-name* is the construct name itself; its definition is connoted by the single colon (':'); and *block-statement* is an `IF`, `DO`, or `SELECT CASE` statement that begins a block.

A block that is given a construct name must also specify the same construct name in its termination statement:

`END` *block construct-name*

Here, *block* must be `IF`, `DO`, or `SELECT`, as appropriate.

10.10.4 The CYCLE and EXIT Statements

The CYCLE and EXIT statements specify that the remaining statements in the current iteration of a particular active (enclosing) DO loop are to be skipped.

CYCLE specifies that these statements are skipped, but the END DO statement that marks the end of the DO loop be executed—that is, the next iteration, if any, is to be started. If the statement marking the end of the DO loop is not END DO—in other words, if the loop is not a block DO—the CYCLE statement does not execute that statement, but does start the next iteration (if any).

EXIT specifies that the loop specified by the DO construct is terminated.

The DO loop affected by CYCLE and EXIT is the innermost enclosing DO loop when the following forms are used:

```
CYCLE
EXIT
```

Otherwise, the following forms specify the construct name of the pertinent DO loop:

```
CYCLE construct-name
EXIT construct-name
```

CYCLE and EXIT can be viewed as glorified GO TO statements. However, they cannot be easily thought of as GO TO statements in obscure cases involving FORTRAN 77 loops. For example:

```
      DO 10 I = 1, 5
      DO 10 J = 1, 5
         IF (J .EQ. 5) EXIT
      DO 10 K = 1, 5
         IF (K .EQ. 3) CYCLE
10    PRINT *, 'I=', I, ' J=', J, ' K=', K
20    CONTINUE
```

In particular, neither the EXIT nor CYCLE statements above are equivalent to a GO TO statement to either label '10' or '20'.

To understand the effect of CYCLE and EXIT in the above fragment, it is helpful to first translate it to its equivalent using only block DO loops:

```
      DO I = 1, 5
         DO J = 1, 5
            IF (J .EQ. 5) EXIT
            DO K = 1, 5
               IF (K .EQ. 3) CYCLE
10             PRINT *, 'I=', I, ' J=', J, ' K=', K
            END DO
         END DO
      END DO
20    CONTINUE
```

Adding new labels allows translation of CYCLE and EXIT to GO TO so they may be more easily understood by programmers accustomed to FORTRAN coding:

```
      DO I = 1, 5
         DO J = 1, 5
```

```
            IF (J .EQ. 5) GOTO 18
            DO K = 1, 5
               IF (K .EQ. 3) GO TO 12
10             PRINT *, 'I=', I, ' J=', J, ' K=', K
12          END DO
         END DO
18    END DO
20    CONTINUE
```

Thus, the `CYCLE` statement in the innermost loop skips over the `PRINT` statement as it begins the next iteration of the loop, while the `EXIT` statement in the middle loop ends that loop but *not* the outermost loop.

10.11 Functions and Subroutines

(The following information augments or overrides the information in Chapter 15 of ANSI X3.9-1978 FORTRAN 77 in specifying the GNU Fortran language. Chapter 15 of that document otherwise serves as the basis for the relevant aspects of GNU Fortran.)

10.11.1 The `%VAL()` Construct

`%VAL(`*arg*`)`

The `%VAL()` construct specifies that an argument, *arg*, is to be passed by value, instead of by reference or descriptor.

`%VAL()` is restricted to actual arguments in invocations of external procedures.

Use of `%VAL()` is recommended 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 the 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.

Implementation Note: Currently, `g77` passes all arguments either by reference or by descriptor.

Thus, use of `%VAL()` tends to be restricted to cases where the called procedure is written in a language other than Fortran that supports call-by-value semantics. (C is an example of such a language.)

See Section 16.2 [Procedures (SUBROUTINE and FUNCTION)], page 240, for detailed information on how this particular version of `g77` passes arguments to procedures.

10.11.2 The `%REF()` Construct

`%REF(`*arg*`)`

The `%REF()` construct specifies that an argument, *arg*, is to be passed by reference, instead of by value or descriptor.

`%REF()` is restricted to actual arguments in invocations of external procedures.

Use of `%REF()` is recommended 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 the 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.

Do not depend on `%REF()` supplying a pointer to the procedure being invoked. While that is a likely implementation choice, other implementation choices are available that preserve Fortran pass-by-reference semantics without passing a pointer to the argument, *arg*. (For example, a copy-in/copy-out implementation.)

Implementation Note: Currently, `g77` passes all arguments (other than variables and arrays of type `CHARACTER`) by reference. Future versions of, or dialects supported by, `g77` might not pass `CHARACTER` functions by reference.

Thus, use of `%REF()` tends to be restricted to cases where *arg* is type `CHARACTER` but the called procedure accesses it via a means other than the method used for Fortran `CHARACTER` arguments.

See Section 16.2 [Procedures (SUBROUTINE and FUNCTION)], page 240, for detailed information on how this particular version of `g77` passes arguments to procedures.

10.11.3 The `%DESCR()` Construct

`%DESCR(`*arg*`)`

The `%DESCR()` construct specifies that an argument, *arg*, is to be passed by descriptor, instead of by value or reference.

`%DESCR()` is restricted to actual arguments in invocations of external procedures.

Use of `%DESCR()` is recommended 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 the 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.

Do not depend on `%DESCR()` supplying a pointer and/or a length passed by value to the procedure being invoked. While that is a likely implementation choice, other implementation choices are available that preserve the pass-by-reference semantics without passing a pointer to the argument, *arg*. (For example, a copy-in/copy-out implementation.) And, future versions of `g77` might change the way descriptors are implemented, such as passing a single argument pointing to a record containing the pointer/length information instead of passing that same information via two arguments as it currently does.

Implementation Note: Currently, `g77` passes all variables and arrays of type `CHARACTER` by descriptor. Future versions of, or dialects supported by, `g77` might pass `CHARACTER` functions by descriptor as well.

Thus, use of `%DESCR()` tends to be restricted to cases where *arg* is not type `CHARACTER` but the called procedure accesses it via a means similar to the method used for Fortran `CHARACTER` arguments.

See Section 16.2 [Procedures (SUBROUTINE and FUNCTION)], page 240, for detailed information on how this particular version of g77 passes arguments to procedures.

10.11.4 Generics and Specifics

The ANSI FORTRAN 77 language defines generic and specific intrinsics. In short, the distinctions are:

- *Specific* intrinsics have specific types for their arguments and a specific return type.
- *Generic* intrinsics are treated, on a case-by-case basis in the program's source code, as one of several possible specific intrinsics.

  Typically, a generic intrinsic has a return type that is determined by the type of one or more of its arguments.

The GNU Fortran language generalizes these concepts somewhat, especially by providing intrinsic subroutines and generic intrinsics that are treated as either a specific intrinsic subroutine or a specific intrinsic function (e.g. `SECOND`).

However, GNU Fortran avoids generalizing this concept to the point where existing code would be accepted as meaning something possibly different than what was intended.

For example, `ABS` is a generic intrinsic, so all working code written using `ABS` of an `INTEGER` argument expects an `INTEGER` return value. Similarly, all such code expects that `ABS` of an `INTEGER*2` argument returns an `INTEGER*2` return value.

Yet, `IABS` is a *specific* intrinsic that accepts only an `INTEGER(KIND=1)` argument. Code that passes something other than an `INTEGER(KIND=1)` argument to `IABS` is not valid GNU Fortran code, because it is not clear what the author intended.

For example, if '`J`' is `INTEGER(KIND=6)`, '`IABS(J)`' is not defined by the GNU Fortran language, because the programmer might have used that construct to mean any of the following, subtly different, things:

- Convert '`J`' to `INTEGER(KIND=1)` first (as if '`IABS(INT(J))`' had been written).
- Convert the result of the intrinsic to `INTEGER(KIND=1)` (as if '`INT(ABS(J))`' had been written).
- No conversion (as if '`ABS(J)`' had been written).

The distinctions matter especially when types and values wider than `INTEGER(KIND=1)` (such as `INTEGER(KIND=2)`), or when operations performing more "arithmetic" than absolute-value, are involved.

The following sample program is not a valid GNU Fortran program, but might be accepted by other compilers. If so, the output is likely to be revealing in terms of how a given compiler treats intrinsics (that normally are specific) when they are given arguments that do not conform to their stated requirements:

```
      PROGRAM JCB002
C Version 1:
C Modified 1997-05-21 (Burley) to accommodate compilers that implement
C INT(I1-I2) as INT(I1)-INT(I2) given INTEGER*2 I1,I2.
C
C Version 0:
```

```
C Written by James Craig Burley 1997-02-20.
C Contact via Internet email: burley@gnu.org
C
C Purpose:
C Determine how compilers handle non-standard IDIM
C on INTEGER*2 operands, which presumably can be
C extrapolated into understanding how the compiler
C generally treats specific intrinsics that are passed
C arguments not of the correct types.
C
C If your compiler implements INTEGER*2 and INTEGER
C as the same type, change all INTEGER*2 below to
C INTEGER*1.
C
      INTEGER*2 I0, I4
      INTEGER I1, I2, I3
      INTEGER*2 ISMALL, ILARGE
      INTEGER*2 ITOOLG, ITWO
      INTEGER*2 ITMP
      LOGICAL L2, L3, L4
C
C Find smallest INTEGER*2 number.
C
      ISMALL=0
 10   I0 = ISMALL-1
      IF ((I0 .GE. ISMALL) .OR. (I0+1 .NE. ISMALL)) GOTO 20
      ISMALL = I0
      GOTO 10
 20   CONTINUE
C
C Find largest INTEGER*2 number.
C
      ILARGE=0
 30   I0 = ILARGE+1
      IF ((I0 .LE. ILARGE) .OR. (I0-1 .NE. ILARGE)) GOTO 40
      ILARGE = I0
      GOTO 30
 40   CONTINUE
C
C Multiplying by two adds stress to the situation.
C
      ITWO = 2
C
C Need a number that, added to -2, is too wide to fit in I*2.
C
      ITOOLG = ISMALL
C
C Use IDIM the straightforward way.
```

```
C
      I1 = IDIM (ILARGE, ISMALL) * ITWO + ITOOLG
C
C Calculate result for first interpretation.
C
      I2 = (INT (ILARGE) - INT (ISMALL)) * ITWO + ITOOLG
C
C Calculate result for second interpretation.
C
      ITMP = ILARGE - ISMALL
      I3 = (INT (ITMP)) * ITWO + ITOOLG
C
C Calculate result for third interpretation.
C
      I4 = (ILARGE - ISMALL) * ITWO + ITOOLG
C
C Print results.
C
      PRINT *, 'ILARGE=', ILARGE
      PRINT *, 'ITWO=', ITWO
      PRINT *, 'ITOOLG=', ITOOLG
      PRINT *, 'ISMALL=', ISMALL
      PRINT *, 'I1=', I1
      PRINT *, 'I2=', I2
      PRINT *, 'I3=', I3
      PRINT *, 'I4=', I4
      PRINT *
      L2 = (I1 .EQ. I2)
      L3 = (I1 .EQ. I3)
      L4 = (I1 .EQ. I4)
      IF (L2 .AND. .NOT.L3 .AND. .NOT.L4) THEN
         PRINT *, 'Interp 1: IDIM(I*2,I*2) => IDIM(INT(I*2),INT(I*2))'
         STOP
      END IF
      IF (L3 .AND. .NOT.L2 .AND. .NOT.L4) THEN
         PRINT *, 'Interp 2: IDIM(I*2,I*2) => INT(DIM(I*2,I*2))'
         STOP
      END IF
      IF (L4 .AND. .NOT.L2 .AND. .NOT.L3) THEN
         PRINT *, 'Interp 3: IDIM(I*2,I*2) => DIM(I*2,I*2)'
         STOP
      END IF
      PRINT *, 'Results need careful analysis.'
      END
```

No future version of the GNU Fortran language will likely permit specific intrinsic invocations with wrong-typed arguments (such as `IDIM` in the above example), since it has been determined that disagreements exist among many production compilers on the interpreta-

tion of such invocations. These disagreements strongly suggest that Fortran programmers, and certainly existing Fortran programs, disagree about the meaning of such invocations.

The first version of '`JCB002`' didn't accommodate some compilers' treatment of '`INT(I1-I2)`' where '`I1`' and '`I2`' are `INTEGER*2`. In such a case, these compilers apparently convert both operands to `INTEGER*4` and then do an `INTEGER*4` subtraction, instead of doing an `INTEGER*2` subtraction on the original values in '`I1`' and '`I2`'.

However, the results of the careful analyses done on the outputs of programs compiled by these various compilers show that they all implement either '`Interp 1`' or '`Interp 2`' above.

Specifically, it is believed that the new version of '`JCB002`' above will confirm that:

- Digital Semiconductor ("DEC") Alpha OSF/1, HP-UX 10.0.1, AIX 3.2.5 `f77` compilers all implement '`Interp 1`'.
- IRIX 5.3 `f77` compiler implements '`Interp 2`'.
- Solaris 2.5, SunOS 4.1.3, DECstation ULTRIX 4.3, and IRIX 6.1 `f77` compilers all implement '`Interp 3`'.

If you get different results than the above for the stated compilers, or have results for other compilers that might be worth adding to the above list, please let us know the details (compiler product, version, machine, results, and so on).

10.11.5 `REAL()` and `AIMAG()` of Complex

The GNU Fortran language disallows `REAL(`*expr*`)` and `AIMAG(`*expr*`)`, where *expr* is any `COMPLEX` type other than `COMPLEX(KIND=1)`, except when they are used in the following way:

```
REAL(REAL(expr))
REAL(AIMAG(expr))
```

The above forms explicitly specify that the desired effect is to convert the real or imaginary part of *expr*, which might be some `REAL` type other than `REAL(KIND=1)`, to type `REAL(KIND=1)`, and have that serve as the value of the expression.

The GNU Fortran language offers clearly named intrinsics to extract the real and imaginary parts of a complex entity without any conversion:

```
REALPART(expr)
IMAGPART(expr)
```

To express the above using typical extended FORTRAN 77, use the following constructs (when *expr* is `COMPLEX(KIND=2)`):

```
DBLE(expr)
DIMAG(expr)
```

The FORTRAN 77 language offers no way to explicitly specify the real and imaginary parts of a complex expression of arbitrary type, apparently as a result of requiring support for only one `COMPLEX` type (`COMPLEX(KIND=1)`). The concepts of converting an expression to type `REAL(KIND=1)` and of extracting the real part of a complex expression were thus "smooshed" by FORTRAN 77 into a single intrinsic, since they happened to have the exact same effect in that language (due to having only one `COMPLEX` type).

Note: When '`-ff90`' is in effect, `g77` treats '`REAL(`*expr*`)`', where *expr* is of type `COMPLEX`, as '`REALPART(`*expr*`)`', whereas with '`-fugly-complex -fno-f90`' in effect, it is treated as '`REAL(REALPART(`*expr*`))`'.

See Section 11.9.3 [Ugly Complex Part Extraction], page 179, for more information.

10.11.6 `CMPLX()` of `DOUBLE PRECISION`

In accordance with Fortran 90 and at least some (perhaps all) other compilers, the GNU Fortran language defines `CMPLX()` as always returning a result that is type `COMPLEX(KIND=1)`.

This means '`CMPLX(D1,D2)`', where '`D1`' and '`D2`' are `REAL(KIND=2)` (`DOUBLE PRECISION`), is treated as:

```
CMPLX(SNGL(D1), SNGL(D2))
```

(It was necessary for Fortran 90 to specify this behavior for `DOUBLE PRECISION` arguments, since that is the behavior mandated by FORTRAN 77.)

The GNU Fortran language also provides the `DCMPLX()` intrinsic, which is provided by some FORTRAN 77 compilers to construct a `DOUBLE COMPLEX` entity from of `DOUBLE PRECISION` operands. However, this solution does not scale well when more `COMPLEX` types (having various precisions and ranges) are offered by Fortran implementations.

Fortran 90 extends the `CMPLX()` intrinsic by adding an extra argument used to specify the desired kind of complex result. However, this solution is somewhat awkward to use, and `g77` currently does not support it.

The GNU Fortran language provides a simple way to build a complex value out of two numbers, with the precise type of the value determined by the types of the two numbers (via the usual type-promotion mechanism):

```
COMPLEX(real, imag)
```

When *real* and *imag* are the same `REAL` types, `COMPLEX()` performs no conversion other than to put them together to form a complex result of the same (complex version of real) type.

See Section 10.11.9.44 [Complex Intrinsic], page 110, for more information.

10.11.7 MIL-STD 1753 Support

The GNU Fortran language includes the MIL-STD 1753 intrinsics `BTEST`, `IAND`, `IBCLR`, `IBITS`, `IBSET`, `IEOR`, `IOR`, `ISHFT`, `ISHFTC`, `MVBITS`, and `NOT`.

10.11.8 `f77`/`f2c` Intrinsics

The bit-manipulation intrinsics supported by traditional `f77` and by `f2c` are available in the GNU Fortran language. These include `AND`, `LSHIFT`, `OR`, `RSHIFT`, and `XOR`.

Also supported are the intrinsics `CDABS`, `CDCOS`, `CDEXP`, `CDLOG`, `CDSIN`, `CDSQRT`, `DCMPLX`, `DCONJG`, `DFLOAT`, `DIMAG`, `DREAL`, and `IMAG`, `ZABS`, `ZCOS`, `ZEXP`, `ZLOG`, `ZSIN`, and `ZSQRT`.

10.11.9 Table of Intrinsic Functions

(Corresponds to Section 15.10 of ANSI X3.9-1978 FORTRAN 77.)

The GNU Fortran language adds various functions, subroutines, types, and arguments to the set of intrinsic functions in ANSI FORTRAN 77. The complete set of intrinsics supported by the GNU Fortran language is described below.

Note that a name is not treated as that of an intrinsic if it is specified in an `EXTERNAL` statement in the same program unit; if a command-line option is used to disable the groups to which the intrinsic belongs; or if the intrinsic is not named in an `INTRINSIC` statement and a command-line option is used to hide the groups to which the intrinsic belongs.

So, it is recommended that any reference in a program unit to an intrinsic procedure that is not a standard FORTRAN 77 intrinsic be accompanied by an appropriate `INTRINSIC` statement in that program unit. This sort of defensive programming makes it more likely that an implementation will issue a diagnostic rather than generate incorrect code for such a reference.

The terminology used below is based on that of the Fortran 90 standard, so that the text may be more concise and accurate:

- `OPTIONAL` means the argument may be omitted.
- '`A-1, A-2, ..., A-n`' means more than one argument (generally named '`A`') may be specified.
- '`scalar`' means the argument must not be an array (must be a variable or array element, or perhaps a constant if expressions are permitted).
- '`DIMENSION(4)`' means the argument must be an array having 4 elements.
- `INTENT(IN)` means the argument must be an expression (such as a constant or a variable that is defined upon invocation of the intrinsic).
- `INTENT(OUT)` means the argument must be definable by the invocation of the intrinsic (that is, must not be a constant nor an expression involving operators other than array reference and substring reference).
- `INTENT(INOUT)` means the argument must be defined prior to, and definable by, invocation of the intrinsic (a combination of the requirements of `INTENT(IN)` and `INTENT(OUT)`.
- See Section 10.7.1.3 [Kind Notation], page 84 for explanation of `KIND`.

10.11.9.1 Abort Intrinsic

```
CALL Abort()
```

Intrinsic groups: `unix`.

Description:

Prints a message and potentially causes a core dump via `abort(3)`.

10.11.9.2 Abs Intrinsic

```
Abs(A)
```

Abs: `INTEGER` or `REAL` function. The exact type depends on that of argument *A*—if *A* is `COMPLEX`, this function's type is `REAL` with the same '`KIND=`' value as the type of *A*. Otherwise, this function's type is the same as that of *A*.

A: `INTEGER`, `REAL`, or `COMPLEX`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns the absolute value of *A*.

If *A* is type `COMPLEX`, the absolute value is computed as:

```
SQRT(REALPART(A)**2, IMAGPART(A)**2)
```

Otherwise, it is computed by negating the *A* if it is negative, or returning *A*.

See Section 10.11.9.227 [Sign Intrinsic], page 158, for how to explicitly compute the positive or negative form of the absolute value of an expression.

10.11.9.3 Access Intrinsic

```
Access(Name, Mode)
```

Access: `INTEGER(KIND=1)` function.

Name: `CHARACTER`; scalar; INTENT(IN).

Mode: `CHARACTER`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Checks file *Name* for accessibility in the mode specified by *Mode* and returns 0 if the file is accessible in that mode, otherwise an error code if the file is inaccessible or *Mode* is invalid. See `access(2)`. A null character ('`CHAR(0)`') marks the end of the name in *Name*—otherwise, trailing blanks in *Name* are ignored. *Mode* may be a concatenation of any of the following characters:

'`r`' Read permission

'`w`' Write permission

'`x`' Execute permission

'*SPC*' Existence

10.11.9.4 AChar Intrinsic

```
AChar(I)
```

AChar: `CHARACTER*1` function.

I: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: `f2c`, `f90`.

Description:

Returns the ASCII character corresponding to the code specified by *I*.

See Section 10.11.9.131 [IAChar Intrinsic], page 133, for the inverse of this function.

See Section 10.11.9.39 [Char Intrinsic], page 108, for the function corresponding to the system's native character set.

10.11.9.5 ACos Intrinsic

`ACos(`*X*`)`

ACos: `REAL` function, the '`KIND=`' value of the type being that of argument *X*.

X: `REAL`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns the arc-cosine (inverse cosine) of *X* in radians.

See Section 10.11.9.46 [Cos Intrinsic], page 110, for the inverse of this function.

10.11.9.6 AdjustL Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL AdjustL`' to use this name for an external procedure.

10.11.9.7 AdjustR Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL AdjustR`' to use this name for an external procedure.

10.11.9.8 AImag Intrinsic

`AImag(`*Z*`)`

AImag: `REAL` function. This intrinsic is valid when argument *Z* is `COMPLEX(KIND=1)`. When *Z* is any other `COMPLEX` type, this intrinsic is valid only when used as the argument to `REAL()`, as explained below.

Z: `COMPLEX`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns the (possibly converted) imaginary part of *Z*.

Use of `AIMAG()` with an argument of a type other than `COMPLEX(KIND=1)` is restricted to the following case:

```
REAL(AIMAG(Z))
```

This expression converts the imaginary part of Z to `REAL(KIND=1)`.

See Section 10.11.5 [REAL() and AIMAG() of Complex], page 96, for more information.

10.11.9.9 AInt Intrinsic

`AInt(`*A*`)`

AInt: `REAL` function, the '`KIND=`' value of the type being that of argument *A*.

A: `REAL`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns *A* with the fractional portion of its magnitude truncated and its sign preserved. (Also called "truncation towards zero".)

See Section 10.11.9.21 [ANInt Intrinsic], page 103, for how to round to nearest whole number.

See Section 10.11.9.148 [Int Intrinsic], page 138, for how to truncate and then convert number to `INTEGER`.

10.11.9.10 Alarm Intrinsic

`CALL Alarm(`*Seconds*`, `*Handler*`, `*Status*`)`

Seconds: `INTEGER`; scalar; INTENT(IN).

Handler: Signal handler (`INTEGER FUNCTION` or `SUBROUTINE`) or dummy/global `INTEGER(KIND=1)` scalar.

Status: `INTEGER(KIND=1)`; OPTIONAL; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Causes external subroutine *Handler* to be executed after a delay of *Seconds* seconds by using `alarm(1)` to set up a signal and `signal(2)` to catch it. If *Status* is supplied, it will be returned with the the number of seconds remaining until any previously scheduled alarm was due to be delivered, or zero if there was no previously scheduled alarm. See Section 10.11.9.228 [Signal Intrinsic (subroutine)], page 158.

10.11.9.11 All Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL All`' to use this name for an external procedure.

10.11.9.12 Allocated Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Allocated`' to use this name for an external procedure.

10.11.9.13 ALog Intrinsic

`ALog(`*X*`)`

ALog: `REAL(KIND=1)` function.

X: `REAL(KIND=1)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `LOG()` that is specific to one type for *X*. See Section 10.11.9.170 [Log Intrinsic], page 145.

10.11.9.14 ALog10 Intrinsic

`ALog10(`*X*`)`

ALog10: `REAL(KIND=1)` function.

X: `REAL(KIND=1)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `LOG10()` that is specific to one type for *X*. See Section 10.11.9.171 [Log10 Intrinsic], page 145.

10.11.9.15 AMax0 Intrinsic

`AMax0(`*A*`-1, `*A*`-2, ..., `*A*`-n)`

AMax0: `REAL(KIND=1)` function.

A: `INTEGER(KIND=1)`; at least two such arguments must be provided; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `MAX()` that is specific to one type for *A* and a different return type. See Section 10.11.9.179 [Max Intrinsic], page 149.

10.11.9.16 AMax1 Intrinsic

`AMax1(`*A*`-1, `*A*`-2, ..., `*A*`-n)`

AMax1: `REAL(KIND=1)` function.

A: `REAL(KIND=1)`; at least two such arguments must be provided; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `MAX()` that is specific to one type for *A*. See Section 10.11.9.179 [Max Intrinsic], page 149.

10.11.9.17 AMin0 Intrinsic

`AMin0(`*A*`-1, `*A*`-2, ..., `*A*`-n)`

AMin0: `REAL(KIND=1)` function.

A: `INTEGER(KIND=1)`; at least two such arguments must be provided; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `MIN()` that is specific to one type for *A* and a different return type. See Section 10.11.9.188 [Min Intrinsic], page 150.

10.11.9.18 AMin1 Intrinsic

`AMin1(`*A*`-1, `*A*`-2, ..., `*A*`-n)`

AMin1: `REAL(KIND=1)` function.

A: `REAL(KIND=1)`; at least two such arguments must be provided; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `MIN()` that is specific to one type for *A*. See Section 10.11.9.188 [Min Intrinsic], page 150.

10.11.9.19 AMod Intrinsic

`AMod(`*A*`, `*P*`)`

AMod: `REAL(KIND=1)` function.

A: `REAL(KIND=1)`; scalar; INTENT(IN).

P: `REAL(KIND=1)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `MOD()` that is specific to one type for *A*. See Section 10.11.9.194 [Mod Intrinsic], page 151.

10.11.9.20 And Intrinsic

`And(`*I*`, `*J*`)`

And: `INTEGER` or `LOGICAL` function, the exact type being the result of cross-promoting the types of all the arguments.

I: `INTEGER` or `LOGICAL`; scalar; INTENT(IN).

J: `INTEGER` or `LOGICAL`; scalar; INTENT(IN).

Intrinsic groups: `f2c`.

Description:

Returns value resulting from boolean AND of pair of bits in each of *I* and *J*.

10.11.9.21 ANInt Intrinsic

`ANInt(`*A*`)`

ANInt: `REAL` function, the '`KIND=`' value of the type being that of argument *A*.

A: `REAL`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns *A* with the fractional portion of its magnitude eliminated by rounding to the nearest whole number and with its sign preserved.

A fractional portion exactly equal to '`.5`' is rounded to the whole number that is larger in magnitude. (Also called "Fortran round".)

See Section 10.11.9.9 [AInt Intrinsic], page 100, for how to truncate to whole number.

See Section 10.11.9.198 [NInt Intrinsic], page 152, for how to round and then convert number to `INTEGER`.

10.11.9.22 Any Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Any`' to use this name for an external procedure.

10.11.9.23 ASin Intrinsic

`ASin(`*X*`)`

ASin: `REAL` function, the '`KIND=`' value of the type being that of argument *X*.

X: `REAL`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns the arc-sine (inverse sine) of *X* in radians.

See Section 10.11.9.229 [Sin Intrinsic], page 159, for the inverse of this function.

10.11.9.24 Associated Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Associated`' to use this name for an external procedure.

10.11.9.25 ATan Intrinsic

`ATan(`*X*`)`

ATan: `REAL` function, the '`KIND=`' value of the type being that of argument *X*.

X: `REAL`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns the arc-tangent (inverse tangent) of *X* in radians.

See Section 10.11.9.243 [Tan Intrinsic], page 164, for the inverse of this function.

10.11.9.26 ATan2 Intrinsic

`ATan2(`*Y*`, `*X*`)`

ATan2: `REAL` function, the exact type being the result of cross-promoting the types of all the arguments.

Y: `REAL`; scalar; INTENT(IN).

X: `REAL`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns the arc-tangent (inverse tangent) of the complex number (*Y*, *X*) in radians.

See Section 10.11.9.243 [Tan Intrinsic], page 164, for the inverse of this function.

10.11.9.27 BesJ0 Intrinsic

`BesJ0(`*X*`)`

BesJ0: `REAL` function, the '`KIND=`' value of the type being that of argument *X*.

X: `REAL`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Calculates the Bessel function of the first kind of order 0 of *X*. See `bessel(3m)`, on whose implementation the function depends.

10.11.9.28 BesJ1 Intrinsic

`BesJ1(`*X*`)`

BesJ1: `REAL` function, the '`KIND=`' value of the type being that of argument *X*.

X: `REAL`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Calculates the Bessel function of the first kind of order 1 of *X*. See `bessel(3m)`, on whose implementation the function depends.

10.11.9.29 BesJN Intrinsic

`BesJN(`*N*`, `*X*`)`

BesJN: `REAL` function, the '`KIND=`' value of the type being that of argument *X*.

N: `INTEGER`; scalar; INTENT(IN).

X: `REAL`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Calculates the Bessel function of the first kind of order *N* of *X*. See `bessel(3m)`, on whose implementation the function depends.

10.11.9.30 BesY0 Intrinsic

`BesY0(`*X*`)`

BesY0: `REAL` function, the '`KIND=`' value of the type being that of argument *X*.

X: `REAL`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Calculates the Bessel function of the second kind of order 0 of *X*. See `bessel(3m)`, on whose implementation the function depends.

10.11.9.31 BesY1 Intrinsic

`BesY1(`*X*`)`

BesY1: `REAL` function, the '`KIND=`' value of the type being that of argument *X*.

X: `REAL`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Calculates the Bessel function of the second kind of order 1 of *X*. See `bessel(3m)`, on whose implementation the function depends.

10.11.9.32 BesYN Intrinsic

`BesYN(`*N*`, `*X*`)`

BesYN: `REAL` function, the '`KIND=`' value of the type being that of argument *X*.

N: `INTEGER`; scalar; INTENT(IN).

X: `REAL`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Calculates the Bessel function of the second kind of order *N* of *X*. See `bessel(3m)`, on whose implementation the function depends.

10.11.9.33 Bit_Size Intrinsic

`Bit_Size(`*I*`)`

Bit_Size: `INTEGER` function, the '`KIND=`' value of the type being that of argument *I*.

I: `INTEGER`; scalar.

Intrinsic groups: `f90`.

Description:

Returns the number of bits (integer precision plus sign bit) represented by the type for *I*.

See Section 10.11.9.34 [BTest Intrinsic], page 106, for how to test the value of a bit in a variable or array.

See Section 10.11.9.136 [IBSet Intrinsic], page 134, for how to set a bit in a variable to 1.

See Section 10.11.9.134 [IBClr Intrinsic], page 134, for how to set a bit in a variable to 0.

10.11.9.34 BTest Intrinsic

`BTest(`*I*`, `*Pos*`)`

BTest: `LOGICAL(KIND=1)` function.

I: `INTEGER`; scalar; INTENT(IN).

Pos: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: `mil`, `f90`, `vxt`.

Description:

Returns `.TRUE.` if bit *Pos* in *I* is 1, `.FALSE.` otherwise.

(Bit 0 is the low-order (rightmost) bit, adding the value 2^0, or 1, to the number if set to 1; bit 1 is the next-higher-order bit, adding 2^1, or 2; bit 2 adds 2^2, or 4; and so on.)

See Section 10.11.9.33 [Bit_Size Intrinsic], page 106, for how to obtain the number of bits in a type. The leftmost bit of *I* is '`BIT_SIZE(`*I*`-1)`'.

10.11.9.35 CAbs Intrinsic

`CAbs(`*A*`)`

CAbs: `REAL(KIND=1)` function.

A: `COMPLEX(KIND=1)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `ABS()` that is specific to one type for *A*. See Section 10.11.9.2 [Abs Intrinsic], page 98.

10.11.9.36 CCos Intrinsic

`CCos(`*X*`)`

CCos: `COMPLEX(KIND=1)` function.

X: `COMPLEX(KIND=1)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `COS()` that is specific to one type for *X*. See Section 10.11.9.46 [Cos Intrinsic], page 110.

10.11.9.37 Ceiling Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Ceiling`' to use this name for an external procedure.

10.11.9.38 CExp Intrinsic

`CExp(`*X*`)`

CExp: `COMPLEX(KIND=1)` function.

X: `COMPLEX(KIND=1)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `EXP()` that is specific to one type for *X*. See Section 10.11.9.99 [Exp Intrinsic], page 123.

10.11.9.39 Char Intrinsic

`Char(`*I*`)`

Char: `CHARACTER*1` function.

I: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns the character corresponding to the code specified by *I*, using the system's native character set.

Because the system's native character set is used, the correspondence between character and their codes is not necessarily the same between GNU Fortran implementations.

Note that no intrinsic exists to convert a numerical value to a printable character string. For example, there is no intrinsic that, given an `INTEGER` or `REAL` argument with the value '`154`', returns the `CHARACTER` result ''`154`''.

Instead, you can use internal-file I/O to do this kind of conversion. For example:

```
INTEGER VALUE
CHARACTER*10 STRING
VALUE = 154
WRITE (STRING, '(I10)'), VALUE
PRINT *, STRING
END
```

The above program, when run, prints:

```
       154
```

See Section 10.11.9.137 [IChar Intrinsic], page 134, for the inverse of the `CHAR` function.

See Section 10.11.9.4 [AChar Intrinsic], page 99, for the function corresponding to the ASCII character set.

10.11.9.40 ChDir Intrinsic (subroutine)

`CALL ChDir(`*Dir*`, `*Status*`)`

Dir: `CHARACTER`; scalar; INTENT(IN).

Status: `INTEGER(KIND=1)`; OPTIONAL; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Sets the current working directory to be *Dir*. If the *Status* argument is supplied, it contains 0 on success or a non-zero error code otherwise upon return. See `chdir(3)`.

Caution: Using this routine during I/O to a unit connected with a non-absolute file name can cause subsequent I/O on such a unit to fail because the I/O library may reopen files by name.

Some non-GNU implementations of Fortran provide this intrinsic as only a function, not as a subroutine, or do not support the (optional) *Status* argument.

For information on other intrinsics with the same name: See Section 12.4.2.17 [ChDir Intrinsic (function)], page 189.

10.11.9.41 ChMod Intrinsic (subroutine)

`CALL ChMod(`*Name*`, `*Mode*`, `*Status*`)`

Name: `CHARACTER`; scalar; INTENT(IN).

Mode: `CHARACTER`; scalar; INTENT(IN).

Status: `INTEGER(KIND=1)`; OPTIONAL; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Changes the access mode of file *Name* according to the specification *Mode*, which is given in the format of `chmod(1)`. A null character ('`CHAR(0)`') marks the end of the name in *Name*—otherwise, trailing blanks in *Name* are ignored. Currently, *Name* must not contain the single quote character.

If the *Status* argument is supplied, it contains 0 on success or a non-zero error code upon return.

Note that this currently works by actually invoking `/bin/chmod` (or the `chmod` found when the library was configured) and so may fail in some circumstances and will, anyway, be slow.

Some non-GNU implementations of Fortran provide this intrinsic as only a function, not as a subroutine, or do not support the (optional) *Status* argument.

For information on other intrinsics with the same name: See Section 12.4.2.18 [ChMod Intrinsic (function)], page 189.

10.11.9.42 CLog Intrinsic

`CLog(`*X*`)`

CLog: `COMPLEX(KIND=1)` function.

X: `COMPLEX(KIND=1)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `LOG()` that is specific to one type for *X*. See Section 10.11.9.170 [Log Intrinsic], page 145.

10.11.9.43 Cmplx Intrinsic

`Cmplx(`*X*`, `*Y*`)`

Cmplx: `COMPLEX(KIND=1)` function.

X: `INTEGER`, `REAL`, or `COMPLEX`; scalar; INTENT(IN).

Y: `INTEGER` or `REAL`; OPTIONAL (must be omitted if *X* is `COMPLEX`); scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

If *X* is not type `COMPLEX`, constructs a value of type `COMPLEX(KIND=1)` from the real and imaginary values specified by *X* and *Y*, respectively. If *Y* is omitted, '`0.`' is assumed.

If *X* is type `COMPLEX`, converts it to type `COMPLEX(KIND=1)`.

See Section 10.11.9.44 [Complex Intrinsic], page 110, for information on easily constructing a `COMPLEX` value of arbitrary precision from `REAL` arguments.

10.11.9.44 Complex Intrinsic

`Complex(`*Real*`, `*Imag*`)`

Complex: `COMPLEX` function, the exact type being the result of cross-promoting the types of all the arguments.

Real: `INTEGER` or `REAL`; scalar; INTENT(IN).

Imag: `INTEGER` or `REAL`; scalar; INTENT(IN).

Intrinsic groups: `gnu`.

Description:

Returns a `COMPLEX` value that has '`Real`' and '`Imag`' as its real and imaginary parts, respectively.

If *Real* and *Imag* are the same type, and that type is not `INTEGER`, no data conversion is performed, and the type of the resulting value has the same kind value as the types of *Real* and *Imag*.

If *Real* and *Imag* are not the same type, the usual type-promotion rules are applied to both, converting either or both to the appropriate `REAL` type. The type of the resulting value has the same kind value as the type to which both *Real* and *Imag* were converted, in this case.

If *Real* and *Imag* are both `INTEGER`, they are both converted to `REAL(KIND=1)`, and the result of the `COMPLEX()` invocation is type `COMPLEX(KIND=1)`.

Note: The way to do this in standard Fortran 90 is too hairy to describe here, but it is important to note that '`CMPLX(D1,D2)`' returns a `COMPLEX(KIND=1)` result even if '`D1`' and '`D2`' are type `REAL(KIND=2)`. Hence the availability of `COMPLEX()` in GNU Fortran.

10.11.9.45 Conjg Intrinsic

`Conjg(`*Z*`)`

Conjg: `COMPLEX` function, the '`KIND=`' value of the type being that of argument *Z*.

Z: `COMPLEX`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns the complex conjugate:

`COMPLEX(REALPART(`*Z*`), -IMAGPART(`*Z*`))`

10.11.9.46 Cos Intrinsic

`Cos(`*X*`)`

Cos: `REAL` or `COMPLEX` function, the exact type being that of argument *X*.

X: `REAL` or `COMPLEX`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns the cosine of *X*, an angle measured in radians.

See Section 10.11.9.5 [ACos Intrinsic], page 100, for the inverse of this function.

10.11.9.47 CosH Intrinsic

`CosH(`*X*`)`

CosH: `REAL` function, the '`KIND=`' value of the type being that of argument *X*.

X: `REAL`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns the hyperbolic cosine of *X*.

10.11.9.48 Count Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Count`' to use this name for an external procedure.

10.11.9.49 CPU_Time Intrinsic

`CALL CPU_Time(`*Seconds*`)`

Seconds: `REAL`; scalar; INTENT(OUT).

Intrinsic groups: `f90`.

Description:

Returns in *Seconds* the current value of the system time. This implementation of the Fortran 95 intrinsic is just an alias for `second` See Section 10.11.9.221 [Second Intrinsic (subroutine)], page 157.

10.11.9.50 CShift Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL CShift`' to use this name for an external procedure.

10.11.9.51 CSin Intrinsic

`CSin(`*X*`)`

CSin: `COMPLEX(KIND=1)` function.

X: `COMPLEX(KIND=1)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `SIN()` that is specific to one type for *X*. See Section 10.11.9.229 [Sin Intrinsic], page 159.

10.11.9.52 CSqRt Intrinsic

`CSqRt(`*X*`)`

CSqRt: `COMPLEX(KIND=1)` function.

X: `COMPLEX(KIND=1)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `SQRT()` that is specific to one type for *X*. See Section 10.11.9.235 [SqRt Intrinsic], page 160.

10.11.9.53 CTime Intrinsic (subroutine)

`CALL CTime(`*Result*`, `*STime*`)`

Result: `CHARACTER`; scalar; INTENT(OUT).

STime: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Converts *STime*, a system time value, such as returned by `TIME8()`, to a string of the form '`Sat Aug 19 18:13:14 1995`', and returns that string in *Result*.

See Section 10.11.9.246 [Time8 Intrinsic], page 164.

Some non-GNU implementations of Fortran provide this intrinsic as only a function, not as a subroutine.

For information on other intrinsics with the same name: See Section 10.11.9.54 [CTime Intrinsic (function)], page 112.

10.11.9.54 CTime Intrinsic (function)

`CTime(`*STime*`)`

CTime: `CHARACTER*(*)` function.

STime: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Converts *STime*, a system time value, such as returned by `TIME8()`, to a string of the form '`Sat Aug 19 18:13:14 1995`', and returns that string as the function value.

See Section 10.11.9.246 [Time8 Intrinsic], page 164.

For information on other intrinsics with the same name: See Section 10.11.9.53 [CTime Intrinsic (subroutine)], page 112.

10.11.9.55 DAbs Intrinsic

`DAbs(`*A*`)`

DAbs: `REAL(KIND=2)` function.

A: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `ABS()` that is specific to one type for *A*. See Section 10.11.9.2 [Abs Intrinsic], page 98.

10.11.9.56 DACos Intrinsic

`DACos(`*X*`)`

DACos: `REAL(KIND=2)` function.

X: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `ACOS()` that is specific to one type for *X*. See Section 10.11.9.5 [ACos Intrinsic], page 100.

10.11.9.57 DASin Intrinsic

`DASin(`*X*`)`

DASin: `REAL(KIND=2)` function.

X: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `ASIN()` that is specific to one type for *X*. See Section 10.11.9.23 [ASin Intrinsic], page 104.

10.11.9.58 DATan Intrinsic

`DATan(`*X*`)`

DATan: `REAL(KIND=2)` function.

X: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `ATAN()` that is specific to one type for *X*. See Section 10.11.9.25 [ATan Intrinsic], page 104.

10.11.9.59 DATan2 Intrinsic

`DATan2(`*Y*`, `*X*`)`

DATan2: `REAL(KIND=2)` function.

Y: `REAL(KIND=2)`; scalar; INTENT(IN).

X: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `ATAN2()` that is specific to one type for *Y* and *X*. See Section 10.11.9.26 [ATan2 Intrinsic], page 104.

10.11.9.60 Date_and_Time Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Date_and_Time`' to use this name for an external procedure.

10.11.9.61 DbesJ0 Intrinsic

`DbesJ0(`*X*`)`

DbesJ0: `REAL(KIND=2)` function.

X: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Archaic form of `BESJ0()` that is specific to one type for *X*. See Section 10.11.9.27 [BesJ0 Intrinsic], page 105.

10.11.9.62 DbesJ1 Intrinsic

`DbesJ1(`*X*`)`

DbesJ1: `REAL(KIND=2)` function.

X: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Archaic form of `BESJ1()` that is specific to one type for *X*. See Section 10.11.9.28 [BesJ1 Intrinsic], page 105.

10.11.9.63 DbesJN Intrinsic

`DbesJN(`*N*`, `*X*`)`

DbesJN: `REAL(KIND=2)` function.

N: `INTEGER`; scalar; INTENT(IN).

X: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Archaic form of `BESJN()` that is specific to one type for *X*. See Section 10.11.9.29 [BesJN Intrinsic], page 105.

10.11.9.64 DbesY0 Intrinsic

`DbesY0(`*X*`)`

DbesY0: `REAL(KIND=2)` function.

X: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Archaic form of `BESY0()` that is specific to one type for *X*. See Section 10.11.9.30 [BesY0 Intrinsic], page 105.

10.11.9.65 DbesY1 Intrinsic

`DbesY1(`*X*`)`

DbesY1: `REAL(KIND=2)` function.

X: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Archaic form of `BESY1()` that is specific to one type for *X*. See Section 10.11.9.31 [BesY1 Intrinsic], page 106.

10.11.9.66 DbesYN Intrinsic

`DbesYN(`*N*`, `*X*`)`

DbesYN: `REAL(KIND=2)` function.

N: `INTEGER`; scalar; INTENT(IN).

X: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Archaic form of `BESYN()` that is specific to one type for *X*. See Section 10.11.9.32 [BesYN Intrinsic], page 106.

10.11.9.67 Dble Intrinsic

`Dble(`*A*`)`

Dble: `REAL(KIND=2)` function.

A: `INTEGER`, `REAL`, or `COMPLEX`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns A converted to double precision (`REAL(KIND=2)`). If A is `COMPLEX`, the real part of A is used for the conversion and the imaginary part disregarded.

See Section 10.11.9.232 [Sngl Intrinsic], page 160, for the function that converts to single precision.

See Section 10.11.9.148 [Int Intrinsic], page 138, for the function that converts to `INTEGER`.

See Section 10.11.9.44 [Complex Intrinsic], page 110, for the function that converts to `COMPLEX`.

10.11.9.68 DCos Intrinsic

`DCos(`X`)`

DCos: `REAL(KIND=2)` function.

X: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `COS()` that is specific to one type for X. See Section 10.11.9.46 [Cos Intrinsic], page 110.

10.11.9.69 DCosH Intrinsic

`DCosH(`X`)`

DCosH: `REAL(KIND=2)` function.

X: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `COSH()` that is specific to one type for X. See Section 10.11.9.47 [CosH Intrinsic], page 111.

10.11.9.70 DDiM Intrinsic

`DDiM(`X`, `Y`)`

DDiM: `REAL(KIND=2)` function.

X: `REAL(KIND=2)`; scalar; INTENT(IN).

Y: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `DIM()` that is specific to one type for X and Y. See Section 10.11.9.75 [DiM Intrinsic], page 117.

10.11.9.71 DErF Intrinsic

`DErF(`*X*`)`

DErF: `REAL(KIND=2)` function.

X: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Archaic form of `ERF()` that is specific to one type for *X*. See Section 10.11.9.94 [ErF Intrinsic], page 122.

10.11.9.72 DErFC Intrinsic

`DErFC(`*X*`)`

DErFC: `REAL(KIND=2)` function.

X: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Archaic form of `ERFC()` that is specific to one type for *X*. See Section 10.11.9.95 [ErFC Intrinsic], page 122.

10.11.9.73 DExp Intrinsic

`DExp(`*X*`)`

DExp: `REAL(KIND=2)` function.

X: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `EXP()` that is specific to one type for *X*. See Section 10.11.9.99 [Exp Intrinsic], page 123.

10.11.9.74 Digits Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Digits`' to use this name for an external procedure.

10.11.9.75 DiM Intrinsic

`DiM(`*X*`, `*Y*`)`

DiM: `INTEGER` or `REAL` function, the exact type being the result of cross-promoting the types of all the arguments.

X: `INTEGER` or `REAL`; scalar; INTENT(IN).

Y: `INTEGER` or `REAL`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns '*X*-*Y*' if *X* is greater than *Y*; otherwise returns zero.

10.11.9.76 DInt Intrinsic

`DInt(`*A*`)`

DInt: `REAL(KIND=2)` function.

A: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `AINT()` that is specific to one type for *A*. See Section 10.11.9.9 [AInt Intrinsic], page 100.

10.11.9.77 DLog Intrinsic

`DLog(`*X*`)`

DLog: `REAL(KIND=2)` function.

X: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `LOG()` that is specific to one type for *X*. See Section 10.11.9.170 [Log Intrinsic], page 145.

10.11.9.78 DLog10 Intrinsic

`DLog10(`*X*`)`

DLog10: `REAL(KIND=2)` function.

X: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `LOG10()` that is specific to one type for *X*. See Section 10.11.9.171 [Log10 Intrinsic], page 145.

10.11.9.79 DMax1 Intrinsic

`DMax1(`*A*`-1, `*A*`-2, ..., `*A*`-n)`

DMax1: `REAL(KIND=2)` function.

A: `REAL(KIND=2)`; at least two such arguments must be provided; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `MAX()` that is specific to one type for *A*. See Section 10.11.9.179 [Max Intrinsic], page 149.

10.11.9.80 DMin1 Intrinsic

`DMin1(`*A*`-1, `*A*`-2, ..., `*A*`-n)`

DMin1: `REAL(KIND=2)` function.

A: `REAL(KIND=2)`; at least two such arguments must be provided; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `MIN()` that is specific to one type for *A*. See Section 10.11.9.188 [Min Intrinsic], page 150.

10.11.9.81 DMod Intrinsic

`DMod(`*A*`, `*P*`)`

DMod: `REAL(KIND=2)` function.

A: `REAL(KIND=2)`; scalar; INTENT(IN).

P: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `MOD()` that is specific to one type for *A*. See Section 10.11.9.194 [Mod Intrinsic], page 151.

10.11.9.82 DNInt Intrinsic

`DNInt(`*A*`)`

DNInt: `REAL(KIND=2)` function.

A: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `ANINT()` that is specific to one type for *A*. See Section 10.11.9.21 [ANInt Intrinsic], page 103.

10.11.9.83 Dot_Product Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Dot_Product`' to use this name for an external procedure.

10.11.9.84 DProd Intrinsic

`DProd(`*X*`, `*Y*`)`

DProd: `REAL(KIND=2)` function.

X: `REAL(KIND=1)`; scalar; INTENT(IN).

Y: `REAL(KIND=1)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns '`DBLE(`*X*`)*DBLE(`*Y*`)`'.

10.11.9.85 DSign Intrinsic

`DSign(`*A*`, `*B*`)`

DSign: `REAL(KIND=2)` function.

A: `REAL(KIND=2)`; scalar; INTENT(IN).

B: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `SIGN()` that is specific to one type for *A* and *B*. See Section 10.11.9.227 [Sign Intrinsic], page 158.

10.11.9.86 DSin Intrinsic

`DSin(`*X*`)`

DSin: `REAL(KIND=2)` function.

X: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `SIN()` that is specific to one type for *X*. See Section 10.11.9.229 [Sin Intrinsic], page 159.

10.11.9.87 DSinH Intrinsic

`DSinH(`*X*`)`

DSinH: `REAL(KIND=2)` function.

X: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `SINH()` that is specific to one type for *X*. See Section 10.11.9.230 [SinH Intrinsic], page 159.

10.11.9.88 DSqRt Intrinsic

`DSqRt(`*X*`)`

DSqRt: `REAL(KIND=2)` function.

X: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `SQRT()` that is specific to one type for *X*. See Section 10.11.9.235 [SqRt Intrinsic], page 160.

10.11.9.89 DTan Intrinsic

`DTan(`*X*`)`

DTan: `REAL(KIND=2)` function.

X: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `TAN()` that is specific to one type for *X*. See Section 10.11.9.243 [Tan Intrinsic], page 164.

10.11.9.90 DTanH Intrinsic

`DTanH(`*X*`)`

DTanH: `REAL(KIND=2)` function.

X: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `TANH()` that is specific to one type for *X*. See Section 10.11.9.244 [TanH Intrinsic], page 164.

10.11.9.91 Dtime Intrinsic (subroutine)

`CALL Dtime(`*Result*`, `*TArray*`)`

Result: `REAL(KIND=1)`; scalar; INTENT(OUT).

TArray: `REAL(KIND=1)`; DIMENSION(2); INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Initially, return the number of seconds of runtime since the start of the process's execution in *Result*, and the user and system components of this in '*TArray*`(1)`' and '*TArray*`(2)`' respectively. The value of *Result* is equal to '*TArray*`(1) + `*TArray*`(2)`'.

Subsequent invocations of '`DTIME()`' set values based on accumulations since the previous invocation.

Some non-GNU implementations of Fortran provide this intrinsic as only a function, not as a subroutine.

For information on other intrinsics with the same name: See Section 12.4.2.36 [Dtime Intrinsic (function)], page 193.

10.11.9.92 EOShift Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL EOShift`' to use this name for an external procedure.

10.11.9.93 Epsilon Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Epsilon`' to use this name for an external procedure.

10.11.9.94 ErF Intrinsic

`ErF(`*X*`)`

ErF: `REAL` function, the '`KIND=`' value of the type being that of argument *X*.

X: `REAL`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Returns the error function of *X*. See `erf(3m)`, which provides the implementation.

10.11.9.95 ErFC Intrinsic

`ErFC(`*X*`)`

ErFC: `REAL` function, the '`KIND=`' value of the type being that of argument *X*.

X: `REAL`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Returns the complementary error function of *X*: '`ERFC(R) = 1 - ERF(R)`' (except that the result may be more accurate than explicitly evaluating that formulae would give). See `erfc(3m)`, which provides the implementation.

10.11.9.96 ETime Intrinsic (subroutine)

`CALL ETime(`*Result*`, `*TArray*`)`

Result: `REAL(KIND=1)`; scalar; INTENT(OUT).

TArray: `REAL(KIND=1)`; DIMENSION(2); INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Return the number of seconds of runtime since the start of the process's execution in *Result*, and the user and system components of this in '*TArray*`(1)`' and '*TArray*`(2)`' respectively. The value of *Result* is equal to '*TArray*`(1) + `*TArray*`(2)`'.

Some non-GNU implementations of Fortran provide this intrinsic as only a function, not as a subroutine.

For information on other intrinsics with the same name: See Section 10.11.9.97 [ETime Intrinsic (function)], page 123.

10.11.9.97 ETime Intrinsic (function)

`ETime(`*TArray*`)`

ETime: `REAL(KIND=1)` function.

TArray: `REAL(KIND=1)`; DIMENSION(2); INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Return the number of seconds of runtime since the start of the process's execution as the function value, and the user and system components of this in '*TArray*`(1)`' and '*TArray*`(2)`' respectively. The functions' value is equal to '*TArray*`(1) + `*TArray*`(2)`'.

For information on other intrinsics with the same name: See Section 10.11.9.96 [ETime Intrinsic (subroutine)], page 122.

10.11.9.98 Exit Intrinsic

`CALL Exit(`*Status*`)`

Status: `INTEGER`; OPTIONAL; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Exit the program with status *Status* after closing open Fortran I/O units and otherwise behaving as `exit(2)`. If *Status* is omitted the canonical 'success' value will be returned to the system.

10.11.9.99 Exp Intrinsic

`Exp(`*X*`)`

Exp: `REAL` or `COMPLEX` function, the exact type being that of argument *X*.

X: `REAL` or `COMPLEX`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns '*e*`**`*X*', where *e* is approximately 2.7182818.

See Section 10.11.9.170 [Log Intrinsic], page 145, for the inverse of this function.

10.11.9.100 Exponent Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Exponent`' to use this name for an external procedure.

10.11.9.101 Fdate Intrinsic (subroutine)

`CALL Fdate(`*Date*`)`

Date: `CHARACTER`; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Returns the current date (using the same format as `CTIME()`) in *Date*.

Equivalent to:

```
CALL CTIME(Date, TIME8())
```

See Section 10.11.9.53 [CTime Intrinsic (subroutine)], page 112.

Some non-GNU implementations of Fortran provide this intrinsic as only a function, not as a subroutine.

For information on other intrinsics with the same name: See Section 10.11.9.102 [Fdate Intrinsic (function)], page 124.

10.11.9.102 Fdate Intrinsic (function)

```
Fdate()
```

Fdate: `CHARACTER*(*)` function.

Intrinsic groups: `unix`.

Description:

Returns the current date (using the same format as `CTIME()`).

Equivalent to:

```
CTIME(TIME8())
```

See Section 10.11.9.54 [CTime Intrinsic (function)], page 112.

For information on other intrinsics with the same name: See Section 10.11.9.101 [Fdate Intrinsic (subroutine)], page 123.

10.11.9.103 FGet Intrinsic (subroutine)

```
CALL FGet(C, Status)
```

C: `CHARACTER`; scalar; INTENT(OUT).

Status: `INTEGER(KIND=1)`; OPTIONAL; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Reads a single character into *C* in stream mode from unit 5 (by-passing normal formatted output) using `getc(3)`. Returns in *Status* 0 on success, -1 on end-of-file, and the error code from `ferror(3)` otherwise.

Stream I/O should not be mixed with normal record-oriented (formatted or unformatted) I/O on the same unit; the results are unpredictable.

For information on other intrinsics with the same name: See Section 12.4.2.37 [FGet Intrinsic (function)], page 194.

10.11.9.104 FGetC Intrinsic (subroutine)

```
CALL FGetC(Unit, C, Status)
```

Unit: `INTEGER`; scalar; INTENT(IN).

C: `CHARACTER`; scalar; INTENT(OUT).

Status: `INTEGER(KIND=1)`; OPTIONAL; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Reads a single character into *C* in stream mode from unit *Unit* (by-passing normal formatted output) using `getc(3)`. Returns in *Status* 0 on success, -1 on end-of-file, and the error code from `ferror(3)` otherwise.

Stream I/O should not be mixed with normal record-oriented (formatted or unformatted) I/O on the same unit; the results are unpredictable.

For information on other intrinsics with the same name: See Section 12.4.2.38 [FGetC Intrinsic (function)], page 194.

10.11.9.105 Float Intrinsic

```
Float(A)
```

Float: `REAL(KIND=1)` function.

A: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `REAL()` that is specific to one type for *A*. See Section 10.11.9.211 [Real Intrinsic], page 154.

10.11.9.106 Floor Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Floor`' to use this name for an external procedure.

10.11.9.107 Flush Intrinsic

```
CALL Flush(Unit)
```

Unit: `INTEGER`; OPTIONAL; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

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

Some non-GNU implementations of Fortran provide this intrinsic as a library procedure that might or might not support the (optional) *Unit* argument.

10.11.9.108 FNum Intrinsic

`FNum(`*Unit*`)`

FNum: `INTEGER(KIND=1)` function.

Unit: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Returns the Unix file descriptor number corresponding to the open Fortran I/O unit *Unit*. This could be passed to an interface to C I/O routines.

10.11.9.109 FPut Intrinsic (subroutine)

`CALL FPut(`*C*`, `*Status*`)`

C: `CHARACTER`; scalar; INTENT(IN).

Status: `INTEGER(KIND=1)`; OPTIONAL; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Writes the single character *C* in stream mode to unit 6 (by-passing normal formatted output) using `putc(3)`. Returns in *Status* 0 on success, the error code from `ferror(3)` otherwise.

Stream I/O should not be mixed with normal record-oriented (formatted or unformatted) I/O on the same unit; the results are unpredictable.

For information on other intrinsics with the same name: See Section 12.4.2.41 [FPut Intrinsic (function)], page 195.

10.11.9.110 FPutC Intrinsic (subroutine)

`CALL FPutC(`*Unit*`, `*C*`, `*Status*`)`

Unit: `INTEGER`; scalar; INTENT(IN).

C: `CHARACTER`; scalar; INTENT(IN).

Status: `INTEGER(KIND=1)`; OPTIONAL; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Writes the single character *Unit* in stream mode to unit 6 (by-passing normal formatted output) using `putc(3)`. Returns in *C* 0 on success, the error code from `ferror(3)` otherwise.

Stream I/O should not be mixed with normal record-oriented (formatted or unformatted) I/O on the same unit; the results are unpredictable.

For information on other intrinsics with the same name: See Section 12.4.2.42 [FPutC Intrinsic (function)], page 195.

10.11.9.111 Fraction Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Fraction`' to use this name for an external procedure.

10.11.9.112 FSeek Intrinsic

`CALL FSeek(`*Unit*`, `*Offset*`, `*Whence*`, `*ErrLab*`)`

Unit: `INTEGER`; scalar; INTENT(IN).

Offset: `INTEGER`; scalar; INTENT(IN).

Whence: `INTEGER`; scalar; INTENT(IN).

ErrLab: '`*`*label*', where *label* is the label of an executable statement; OPTIONAL.

Intrinsic groups: `unix`.

Description:

Attempts to move Fortran unit *Unit* to the specified *Offset*: absolute offset if *Offset*=0; relative to the current offset if *Offset*=1; relative to the end of the file if *Offset*=2. It branches to label *Whence* if *Unit* is not open or if the call otherwise fails.

10.11.9.113 FStat Intrinsic (subroutine)

`CALL FStat(`*Unit*`, `*SArray*`, `*Status*`)`

Unit: `INTEGER`; scalar; INTENT(IN).

SArray: `INTEGER(KIND=1)`; DIMENSION(13); INTENT(OUT).

Status: `INTEGER(KIND=1)`; OPTIONAL; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Obtains data about the file open on Fortran I/O unit *Unit* and places them in the array *SArray*. The values in this array are extracted from the `stat` structure as returned by `fstat(2)` q.v., as follows:

1. File mode
2. Inode number
3. ID of device containing directory entry for file
4. Device id (if relevant)
5. Number of links
6. Owner's uid
7. Owner's gid
8. File size (bytes)
9. Last access time
10. Last modification time
11. Last file status change time
12. Preferred I/O block size
13. Number of blocks allocated

Not all these elements are relevant on all systems. If an element is not relevant, it is returned as 0.

If the *Status* argument is supplied, it contains 0 on success or a non-zero error code upon return.

Some non-GNU implementations of Fortran provide this intrinsic as only a function, not as a subroutine, or do not support the (optional) *Status* argument.

For information on other intrinsics with the same name: See Section 10.11.9.114 [FStat Intrinsic (function)], page 128.

10.11.9.114 FStat Intrinsic (function)

`FStat(`*Unit*`, `*SArray*`)`

FStat: `INTEGER(KIND=1)` function.

Unit: `INTEGER`; scalar; INTENT(IN).

SArray: `INTEGER(KIND=1)`; DIMENSION(13); INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Obtains data about the file open on Fortran I/O unit *Unit* and places them in the array *SArray*. The values in this array are extracted from the `stat` structure as returned by `fstat(2)` q.v., as follows:

1. File mode
2. Inode number
3. ID of device containing directory entry for file
4. Device id (if relevant)
5. Number of links
6. Owner's uid
7. Owner's gid
8. File size (bytes)
9. Last access time
10. Last modification time
11. Last file status change time
12. Preferred I/O block size
13. Number of blocks allocated

Not all these elements are relevant on all systems. If an element is not relevant, it is returned as 0.

Returns 0 on success or a non-zero error code.

For information on other intrinsics with the same name: See Section 10.11.9.113 [FStat Intrinsic (subroutine)], page 127.

10.11.9.115 FTell Intrinsic (subroutine)

`CALL FTell(`*Unit*`, `*Offset*`)`

Unit: `INTEGER`; scalar; INTENT(IN).

Offset: `INTEGER(KIND=1)`; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Sets *Offset* to the current offset of Fortran unit *Unit* (or to −1 if *Unit* is not open).

Some non-GNU implementations of Fortran provide this intrinsic as only a function, not as a subroutine.

For information on other intrinsics with the same name: See Section 10.11.9.116 [FTell Intrinsic (function)], page 129.

10.11.9.116 FTell Intrinsic (function)

`FTell(`*Unit*`)`

FTell: `INTEGER(KIND=1)` function.

Unit: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Returns the current offset of Fortran unit *Unit* (or −1 if *Unit* is not open).

For information on other intrinsics with the same name: See Section 10.11.9.115 [FTell Intrinsic (subroutine)], page 128.

10.11.9.117 GError Intrinsic

`CALL GError(`*Message*`)`

Message: `CHARACTER`; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Returns the system error message corresponding to the last system error (C `errno`).

10.11.9.118 GetArg Intrinsic

`CALL GetArg(`*Pos*`, `*Value*`)`

Pos: `INTEGER`; scalar; INTENT(IN).

Value: `CHARACTER`; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Sets *Value* to the *Pos*-th command-line argument (or to all blanks if there are fewer than *Value* command-line arguments); `CALL GETARG(0,` *value*`)` sets *value* to the name of the program (on systems that support this feature).

See Section 10.11.9.133 [IArgC Intrinsic], page 133, for information on how to get the number of arguments.

10.11.9.119 GetCWD Intrinsic (subroutine)

`CALL GetCWD(`*Name*`, `*Status*`)`

Name: `CHARACTER`; scalar; INTENT(OUT).

Status: `INTEGER(KIND=1)`; OPTIONAL; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Places the current working directory in *Name*. If the *Status* argument is supplied, it contains 0 success or a non-zero error code upon return (`ENOSYS` if the system does not provide `getcwd(3)` or `getwd(3)`).

Some non-GNU implementations of Fortran provide this intrinsic as only a function, not as a subroutine, or do not support the (optional) *Status* argument.

For information on other intrinsics with the same name: See Section 10.11.9.120 [GetCWD Intrinsic (function)], page 130.

10.11.9.120 GetCWD Intrinsic (function)

`GetCWD(`*Name*`)`

GetCWD: `INTEGER(KIND=1)` function.

Name: `CHARACTER`; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Places the current working directory in *Name*. Returns 0 on success, otherwise a non-zero error code (`ENOSYS` if the system does not provide `getcwd(3)` or `getwd(3)`).

For information on other intrinsics with the same name: See Section 10.11.9.119 [GetCWD Intrinsic (subroutine)], page 130.

10.11.9.121 GetEnv Intrinsic

`CALL GetEnv(`*Name*`, `*Value*`)`

Name: `CHARACTER`; scalar; INTENT(IN).

Value: `CHARACTER`; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Sets *Value* to the value of environment variable given by the value of *Name* (`$name` in shell terms) or to blanks if `$name` has not been set. A null character ('`CHAR(0)`') marks the end of the name in *Name*—otherwise, trailing blanks in *Name* are ignored.

10.11.9.122 GetGId Intrinsic

`GetGId()`

GetGId: `INTEGER(KIND=1)` function.

Intrinsic groups: `unix`.

Description:

Returns the group id for the current process.

10.11.9.123 GetLog Intrinsic

```
CALL GetLog(Login)
```

Login: `CHARACTER`; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Returns the login name for the process in *Login*.

Caution: On some systems, the `getlogin(3)` function, which this intrinsic calls at run time, is either not implemented or returns a null pointer. In the latter case, this intrinsic returns blanks in *Login*.

10.11.9.124 GetPId Intrinsic

```
GetPId()
```

GetPId: `INTEGER(KIND=1)` function.

Intrinsic groups: `unix`.

Description:

Returns the process id for the current process.

10.11.9.125 GetUId Intrinsic

```
GetUId()
```

GetUId: `INTEGER(KIND=1)` function.

Intrinsic groups: `unix`.

Description:

Returns the user id for the current process.

10.11.9.126 GMTime Intrinsic

```
CALL GMTime(STime, TArray)
```

STime: `INTEGER(KIND=1)`; scalar; INTENT(IN).

TArray: `INTEGER(KIND=1)`; DIMENSION(9); INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Given a system time value *STime*, fills *TArray* with values extracted from it appropriate to the GMT time zone using `gmtime(3)`.

The array elements are 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 0–31
5. Number of months since January, range 0–12

6. Years since 1900
7. Number of days since Sunday, range 0–6
8. Days since January 1
9. Daylight savings indicator: positive if daylight savings is in effect, zero if not, and negative if the information isn't available.

10.11.9.127 HostNm Intrinsic (subroutine)

`CALL HostNm(`*Name*`, `*Status*`)`

Name: `CHARACTER`; scalar; INTENT(OUT).

Status: `INTEGER(KIND=1)`; OPTIONAL; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Fills *Name* with the system's host name returned by `gethostname(2)`. If the *Status* argument is supplied, it contains 0 on success or a non-zero error code upon return (`ENOSYS` if the system does not provide `gethostname(2)`).

Some non-GNU implementations of Fortran provide this intrinsic as only a function, not as a subroutine, or do not support the (optional) *Status* argument.

For information on other intrinsics with the same name: See Section 10.11.9.128 [HostNm Intrinsic (function)], page 132.

10.11.9.128 HostNm Intrinsic (function)

`HostNm(`*Name*`)`

HostNm: `INTEGER(KIND=1)` function.

Name: `CHARACTER`; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Fills *Name* with the system's host name returned by `gethostname(2)`, returning 0 on success or a non-zero error code (`ENOSYS` if the system does not provide `gethostname(2)`).

For information on other intrinsics with the same name: See Section 10.11.9.127 [HostNm Intrinsic (subroutine)], page 132.

10.11.9.129 Huge Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Huge`' to use this name for an external procedure.

10.11.9.130 IAbs Intrinsic

`IAbs(`*A*`)`

IAbs: `INTEGER(KIND=1)` function.

A: `INTEGER(KIND=1)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `ABS()` that is specific to one type for *A*. See Section 10.11.9.2 [Abs Intrinsic], page 98.

10.11.9.131 IAChar Intrinsic

`IAChar(`*C*`)`

IAChar: `INTEGER(KIND=1)` function.

C: `CHARACTER`; scalar; INTENT(IN).

Intrinsic groups: `f2c`, `f90`.

Description:

Returns the code for the ASCII character in the first character position of *C*.

See Section 10.11.9.4 [AChar Intrinsic], page 99, for the inverse of this function.

See Section 10.11.9.137 [IChar Intrinsic], page 134, for the function corresponding to the system's native character set.

10.11.9.132 IAnd Intrinsic

`IAnd(`*I*`, `*J*`)`

IAnd: `INTEGER` function, the exact type being the result of cross-promoting the types of all the arguments.

I: `INTEGER`; scalar; INTENT(IN).

J: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: `mil`, `f90`, `vxt`.

Description:

Returns value resulting from boolean AND of pair of bits in each of *I* and *J*.

10.11.9.133 IArgC Intrinsic

`IArgC()`

IArgC: `INTEGER(KIND=1)` function.

Intrinsic groups: `unix`.

Description:

Returns the number of command-line arguments.

This count does not include the specification of the program name itself.

10.11.9.134 IBClr Intrinsic

`IBClr(`*I*`, `*Pos*`)`

IBClr: `INTEGER` function, the '`KIND=`' value of the type being that of argument *I*.

I: `INTEGER`; scalar; INTENT(IN).

Pos: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: `mil`, `f90`, `vxt`.

Description:

Returns the value of *I* with bit *Pos* cleared (set to zero). See Section 10.11.9.34 [BTest Intrinsic], page 106 for information on bit positions.

10.11.9.135 IBits Intrinsic

`IBits(`*I*`, `*Pos*`, `*Len*`)`

IBits: `INTEGER` function, the '`KIND=`' value of the type being that of argument *I*.

I: `INTEGER`; scalar; INTENT(IN).

Pos: `INTEGER`; scalar; INTENT(IN).

Len: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: `mil`, `f90`, `vxt`.

Description:

Extracts a subfield of length *Len* from *I*, starting from bit position *Pos* and extending left for *Len* bits. The result is right-justified and the remaining bits are zeroed. The value of '*Pos*+*Len*' must be less than or equal to the value '`BIT_SIZE(`*I*`)`'. See Section 10.11.9.33 [Bit_Size Intrinsic], page 106.

10.11.9.136 IBSet Intrinsic

`IBSet(`*I*`, `*Pos*`)`

IBSet: `INTEGER` function, the '`KIND=`' value of the type being that of argument *I*.

I: `INTEGER`; scalar; INTENT(IN).

Pos: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: `mil`, `f90`, `vxt`.

Description:

Returns the value of *I* with bit *Pos* set (to one). See Section 10.11.9.34 [BTest Intrinsic], page 106 for information on bit positions.

10.11.9.137 IChar Intrinsic

`IChar(`*C*`)`

IChar: `INTEGER(KIND=1)` function.

C: `CHARACTER`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns the code for the character in the first character position of *C*.

Because the system's native character set is used, the correspondence between character and their codes is not necessarily the same between GNU Fortran implementations.

Note that no intrinsic exists to convert a printable character string to a numerical value. For example, there is no intrinsic that, given the `CHARACTER` value ''`154`'', returns an `INTEGER` or `REAL` value with the value '`154`'.

Instead, you can use internal-file I/O to do this kind of conversion. For example:

```
INTEGER VALUE
CHARACTER*10 STRING
STRING = '154'
READ (STRING, '(I10)'), VALUE
PRINT *, VALUE
END
```

The above program, when run, prints:

```
 154
```

See Section 10.11.9.39 [Char Intrinsic], page 108, for the inverse of the `ICHAR` function.

See Section 10.11.9.131 [IAChar Intrinsic], page 133, for the function corresponding to the ASCII character set.

10.11.9.138 IDate Intrinsic (UNIX)

`CALL IDate(`*TArray*`)`

TArray: `INTEGER(KIND=1)`; DIMENSION(3); INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Fills *TArray* with the numerical values at the current local time of day, month (in the range 1–12), and year in elements 1, 2, and 3, respectively. The year has four significant digits.

For information on other intrinsics with the same name: See Section 12.4.2.43 [IDate Intrinsic (VXT)], page 195.

10.11.9.139 IDiM Intrinsic

`IDiM(`*X*`, `*Y*`)`

IDiM: `INTEGER(KIND=1)` function.

X: `INTEGER(KIND=1)`; scalar; INTENT(IN).

Y: `INTEGER(KIND=1)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `DIM()` that is specific to one type for *X* and *Y*. See Section 10.11.9.75 [DiM Intrinsic], page 117.

10.11.9.140 IDInt Intrinsic

`IDInt(`*A*`)`

IDInt: `INTEGER(KIND=1)` function.

A: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `INT()` that is specific to one type for *A*. See Section 10.11.9.148 [Int Intrinsic], page 138.

10.11.9.141 IDNInt Intrinsic

`IDNInt(`*A*`)`

IDNInt: `INTEGER(KIND=1)` function.

A: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `NINT()` that is specific to one type for *A*. See Section 10.11.9.198 [NInt Intrinsic], page 152.

10.11.9.142 IEOr Intrinsic

`IEOr(`*I*`, `*J*`)`

IEOr: `INTEGER` function, the exact type being the result of cross-promoting the types of all the arguments.

I: `INTEGER`; scalar; INTENT(IN).

J: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: `mil`, `f90`, `vxt`.

Description:

Returns value resulting from boolean exclusive-OR of pair of bits in each of *I* and *J*.

10.11.9.143 IErrNo Intrinsic

`IErrNo()`

IErrNo: `INTEGER(KIND=1)` function.

Intrinsic groups: `unix`.

Description:

Returns the last system error number (corresponding to the C `errno`).

10.11.9.144 IFix Intrinsic

`IFix(`*A*`)`

IFix: `INTEGER(KIND=1)` function.

A: `REAL(KIND=1)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `INT()` that is specific to one type for *A*. See Section 10.11.9.148 [Int Intrinsic], page 138.

10.11.9.145 Imag Intrinsic

`Imag(`*Z*`)`

Imag: `REAL` function, the '`KIND=`' value of the type being that of argument *Z*.

Z: `COMPLEX`; scalar; INTENT(IN).

Intrinsic groups: `f2c`.

Description:

The imaginary part of *Z* is returned, without conversion.

Note: The way to do this in standard Fortran 90 is '`AIMAG(`*Z*`)`'. However, when, for example, *Z* is `DOUBLE COMPLEX`, '`AIMAG(`*Z*`)`' means something different for some compilers that are not true Fortran 90 compilers but offer some extensions standardized by Fortran 90 (such as the `DOUBLE COMPLEX` type, also known as `COMPLEX(KIND=2)`).

The advantage of `IMAG()` is that, while not necessarily more or less portable than `AIMAG()`, it is more likely to cause a compiler that doesn't support it to produce a diagnostic than generate incorrect code.

See Section 10.11.5 [REAL() and AIMAG() of Complex], page 96, for more information.

10.11.9.146 ImagPart Intrinsic

`ImagPart(`*Z*`)`

ImagPart: `REAL` function, the '`KIND=`' value of the type being that of argument *Z*.

Z: `COMPLEX`; scalar; INTENT(IN).

Intrinsic groups: `gnu`.

Description:

The imaginary part of *Z* is returned, without conversion.

Note: The way to do this in standard Fortran 90 is '`AIMAG(`*Z*`)`'. However, when, for example, *Z* is `DOUBLE COMPLEX`, '`AIMAG(`*Z*`)`' means something different for some compilers that are not true Fortran 90 compilers but offer some extensions standardized by Fortran 90 (such as the `DOUBLE COMPLEX` type, also known as `COMPLEX(KIND=2)`).

The advantage of `IMAGPART()` is that, while not necessarily more or less portable than `AIMAG()`, it is more likely to cause a compiler that doesn't support it to produce a diagnostic than generate incorrect code.

See Section 10.11.5 [REAL() and AIMAG() of Complex], page 96, for more information.

10.11.9.147 Index Intrinsic

`Index(`*String*`, `*Substring*`)`

Index: `INTEGER(KIND=1)` function.

String: `CHARACTER`; scalar; INTENT(IN).

Substring: `CHARACTER`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns the position of the start of the first occurrence of string *Substring* as a substring in *String*, counting from one. If *Substring* doesn't occur in *String*, zero is returned.

10.11.9.148 Int Intrinsic

`Int(`*A*`)`

Int: `INTEGER(KIND=1)` function.

A: `INTEGER`, `REAL`, or `COMPLEX`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns *A* with the fractional portion of its magnitude truncated and its sign preserved, converted to type `INTEGER(KIND=1)`.

If *A* is type `COMPLEX`, its real part is truncated and converted, and its imaginary part is disregarded.

See Section 10.11.9.198 [NInt Intrinsic], page 152, for how to convert, rounded to nearest whole number.

See Section 10.11.9.9 [AInt Intrinsic], page 100, for how to truncate to whole number without converting.

10.11.9.149 Int2 Intrinsic

`Int2(`*A*`)`

Int2: `INTEGER(KIND=6)` function.

A: `INTEGER`, `REAL`, or `COMPLEX`; scalar; INTENT(IN).

Intrinsic groups: `gnu`.

Description:

Returns *A* with the fractional portion of its magnitude truncated and its sign preserved, converted to type `INTEGER(KIND=6)`.

If *A* is type `COMPLEX`, its real part is truncated and converted, and its imaginary part is disgregarded.

See Section 10.11.9.148 [Int Intrinsic], page 138.

The precise meaning of this intrinsic might change in a future version of the GNU Fortran language, as more is learned about how it is used.

10.11.9.150 Int8 Intrinsic

`Int8(`*A*`)`

Int8: `INTEGER(KIND=2)` function.

A: `INTEGER`, `REAL`, or `COMPLEX`; scalar; INTENT(IN).

Intrinsic groups: `gnu`.

Description:

Returns *A* with the fractional portion of its magnitude truncated and its sign preserved, converted to type `INTEGER(KIND=2)`.

If *A* is type `COMPLEX`, its real part is truncated and converted, and its imaginary part is disgregarded.

See Section 10.11.9.148 [Int Intrinsic], page 138.

The precise meaning of this intrinsic might change in a future version of the GNU Fortran language, as more is learned about how it is used.

10.11.9.151 IOr Intrinsic

`IOr(`*I*`, `*J*`)`

IOr: `INTEGER` function, the exact type being the result of cross-promoting the types of all the arguments.

I: `INTEGER`; scalar; INTENT(IN).

J: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: `mil`, `f90`, `vxt`.

Description:

Returns value resulting from boolean OR of pair of bits in each of *I* and *J*.

10.11.9.152 IRand Intrinsic

`IRand(`*Flag*`)`

IRand: `INTEGER(KIND=1)` function.

Flag: `INTEGER`; OPTIONAL; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Returns a uniform quasi-random number up to a system-dependent limit. If *Flag* is 0, the next number in sequence is returned; if *Flag* is 1, the generator is restarted by calling the UNIX function '`srand(0)`'; if *Flag* has any other value, it is used as a new seed with `srand()`.

See Section 10.11.9.236 [SRand Intrinsic], page 161.

Note: As typically implemented (by the routine of the same name in the C library), this random number generator is a very poor one, though the BSD and GNU libraries provide a much better implementation than the 'traditional' one. On a different system you almost certainly want to use something better.

10.11.9.153 IsaTty Intrinsic

`IsaTty(`*Unit*`)`

IsaTty: `LOGICAL(KIND=1)` function.

Unit: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Returns `.TRUE.` if and only if the Fortran I/O unit specified by *Unit* is connected to a terminal device. See `isatty(3)`.

10.11.9.154 IShft Intrinsic

`IShft(`*I, Shift*`)`

IShft: `INTEGER` function, the '`KIND=`' value of the type being that of argument *I*.

I: `INTEGER`; scalar; INTENT(IN).

Shift: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: `mil`, `f90`, `vxt`.

Description:

All bits representing *I* are shifted *Shift* places. '*Shift*`.GT.0`' indicates a left shift, '*Shift*`.EQ.0`' indicates no shift and '*Shift*`.LT.0`' indicates a right shift. If the absolute value of the shift count is greater than '`BIT_SIZE(`*I*`)`', the result is undefined. Bits shifted out from the left end or the right end, as the case may be, are lost. Zeros are shifted in from the opposite end.

See Section 10.11.9.155 [IShftC Intrinsic], page 140 for the circular-shift equivalent.

10.11.9.155 IShftC Intrinsic

`IShftC(`*I, Shift, Size*`)`

IShftC: `INTEGER` function, the '`KIND=`' value of the type being that of argument *I*.

I: `INTEGER`; scalar; INTENT(IN).

Shift: `INTEGER`; scalar; INTENT(IN).

Size: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: `mil`, `f90`, `vxt`.

Description:

The rightmost *Size* bits of the argument *I* are shifted circularly *Shift* places, i.e. the bits shifted out of one end are shifted into the opposite end. No bits are lost. The unshifted bits of the result are the same as the unshifted bits of *I*. The absolute value of the argument *Shift* must be less than or equal to *Size*. The value of *Size* must be greater than or equal to one and less than or equal to '`BIT_SIZE(`*I*`)`'.

See Section 10.11.9.154 [IShft Intrinsic], page 140 for the logical shift equivalent.

10.11.9.156 ISign Intrinsic

`ISign(`*A*`, `*B*`)`

ISign: `INTEGER(KIND=1)` function.

A: `INTEGER(KIND=1)`; scalar; INTENT(IN).

B: `INTEGER(KIND=1)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `SIGN()` that is specific to one type for *A* and *B*. See Section 10.11.9.227 [Sign Intrinsic], page 158.

10.11.9.157 ITime Intrinsic

`CALL ITime(`*TArray*`)`

TArray: `INTEGER(KIND=1)`; DIMENSION(3); INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Returns the current local time hour, minutes, and seconds in elements 1, 2, and 3 of *TArray*, respectively.

10.11.9.158 Kill Intrinsic (subroutine)

`CALL Kill(`*Pid*`, `*Signal*`, `*Status*`)`

Pid: `INTEGER`; scalar; INTENT(IN).

Signal: `INTEGER`; scalar; INTENT(IN).

Status: `INTEGER(KIND=1)`; OPTIONAL; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Sends the signal specified by *Signal* to the process *Pid*. If the *Status* argument is supplied, it contains 0 on success or a non-zero error code upon return. See `kill(2)`.

Some non-GNU implementations of Fortran provide this intrinsic as only a function, not as a subroutine, or do not support the (optional) *Status* argument.

For information on other intrinsics with the same name: See Section 12.4.2.93 [Kill Intrinsic (function)], page 201.

10.11.9.159 Kind Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Kind`' to use this name for an external procedure.

10.11.9.160 LBound Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL LBound`' to use this name for an external procedure.

10.11.9.161 Len Intrinsic

`Len(`*String*`)`

Len: `INTEGER(KIND=1)` function.

String: `CHARACTER`; scalar.

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns the length of *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.

See Section 10.11.9.33 [Bit_Size Intrinsic], page 106, for the function that determines the size of its argument in bits.

10.11.9.162 Len_Trim Intrinsic

`Len_Trim(`*String*`)`

Len_Trim: `INTEGER(KIND=1)` function.

String: `CHARACTER`; scalar; INTENT(IN).

Intrinsic groups: `f90`.

Description:

Returns the index of the last non-blank character in *String*. `LNBLNK` and `LEN_TRIM` are equivalent.

10.11.9.163 LGe Intrinsic

`LGe(`*String_A*`, `*String_B*`)`

LGe: `LOGICAL(KIND=1)` function.

String_A: `CHARACTER`; scalar; INTENT(IN).

String_B: `CHARACTER`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns '`.TRUE.`' if '*String_A*`.GE.`*String_B*', '`.FALSE.`' otherwise. *String_A* and *String_B* are interpreted as containing ASCII character codes. If either value contains a character not in the ASCII character set, the result is processor dependent.

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.

The lexical comparison intrinsics `LGe`, `LGt`, `LLe`, and `LLt` differ from the corresponding intrinsic operators `.GE.`, `.GT.`, `.LE.`, `.LT.`. Because the ASCII collating sequence is assumed, the following expressions always return '`.TRUE.`':

```
LGE ('0', ' ')
LGE ('A', '0')
LGE ('a', 'A')
```

The following related expressions do *not* always return '`.TRUE.`', as they are not necessarily evaluated assuming the arguments use ASCII encoding:

```
'0' .GE. ' '
'A' .GE. '0'
'a' .GE. 'A'
```

The same difference exists between `LGt` and `.GT.`; between `LLe` and `.LE.`; and between `LLt` and `.LT.`.

10.11.9.164 LGt Intrinsic

`LGt(`*String_A*`, `*String_B*`)`

LGt: `LOGICAL(KIND=1)` function.

String_A: `CHARACTER`; scalar; INTENT(IN).

String_B: `CHARACTER`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns '`.TRUE.`' if '*String_A* `.GT.` *String_B*', '`.FALSE.`' otherwise. *String_A* and *String_B* are interpreted as containing ASCII character codes. If either value contains a character not in the ASCII character set, the result is processor dependent.

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.

See Section 10.11.9.163 [LGe Intrinsic], page 142, for information on the distinction between the `LGT` intrinsic and the `.GT.` operator.

10.11.9.165 Link Intrinsic (subroutine)

`CALL Link(`*Path1*`, `*Path2*`, `*Status*`)`

Path1: `CHARACTER`; scalar; INTENT(IN).

Path2: `CHARACTER`; scalar; INTENT(IN).

Status: `INTEGER(KIND=1)`; OPTIONAL; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Makes a (hard) link from file *Path1* to *Path2*. A null character ('`CHAR(0)`') marks the end of the names in *Path1* and *Path2*—otherwise, trailing blanks in *Path1* and *Path2* are ignored. If the *Status* argument is supplied, it contains 0 on success or a non-zero error code upon return. See `link(2)`.

Some non-GNU implementations of Fortran provide this intrinsic as only a function, not as a subroutine, or do not support the (optional) *Status* argument.

For information on other intrinsics with the same name: See Section 12.4.2.94 [Link Intrinsic (function)], page 201.

10.11.9.166 LLe Intrinsic

`LLe(`*String_A*`, `*String_B*`)`

LLe: `LOGICAL(KIND=1)` function.

String_A: `CHARACTER`; scalar; INTENT(IN).

String_B: `CHARACTER`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns '`.TRUE.`' if '*String_A*`.LE.`*String_B*', '`.FALSE.`' otherwise. *String_A* and *String_B* are interpreted as containing ASCII character codes. If either value contains a character not in the ASCII character set, the result is processor dependent.

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.

See Section 10.11.9.163 [LGe Intrinsic], page 142, for information on the distinction between the `LLE` intrinsic and the `.LE.` operator.

10.11.9.167 LLt Intrinsic

`LLt(`*String_A*`, `*String_B*`)`

LLt: `LOGICAL(KIND=1)` function.

String_A: `CHARACTER`; scalar; INTENT(IN).

String_B: `CHARACTER`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns '`.TRUE.`' if '*String_A*`.LT.`*String_B*', '`.FALSE.`' otherwise. *String_A* and *String_B* are interpreted as containing ASCII character codes. If either value contains a character not in the ASCII character set, the result is processor dependent.

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.

See Section 10.11.9.163 [LGe Intrinsic], page 142, for information on the distinction between the `LLT` intrinsic and the `.LT.` operator.

10.11.9.168 LnBlnk Intrinsic

`LnBlnk(`*String*`)`

LnBlnk: `INTEGER(KIND=1)` function.

String: `CHARACTER`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Returns the index of the last non-blank character in *String*. `LNBLNK` and `LEN_TRIM` are equivalent.

10.11.9.169 Loc Intrinsic

`Loc(`*Entity*`)`

Loc: `INTEGER(KIND=7)` function.

Entity: Any type; cannot be a constant or expression.

Intrinsic groups: `unix`.

Description:

The `LOC()` intrinsic works the same way as the `%LOC()` construct. See Section 10.8.1 [The `%LOC()` Construct], page 88, for more information.

10.11.9.170 Log Intrinsic

`Log(`*X*`)`

Log: `REAL` or `COMPLEX` function, the exact type being that of argument *X*.

X: `REAL` or `COMPLEX`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns the natural logarithm of *X*, which must be greater than zero or, if type `COMPLEX`, must not be zero.

See Section 10.11.9.99 [Exp Intrinsic], page 123, for the inverse of this function.

See Section 10.11.9.171 [Log10 Intrinsic], page 145, for the base-10 logarithm function.

10.11.9.171 Log10 Intrinsic

`Log10(`*X*`)`

Log10: `REAL` function, the '`KIND=`' value of the type being that of argument *X*.

X: `REAL`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns the natural logarithm of *X*, which must be greater than zero or, if type `COMPLEX`, must not be zero.

The inverse of this function is '`10. ** LOG10(`*X*`)`'.

See Section 10.11.9.170 [Log Intrinsic], page 145, for the natural logarithm function.

10.11.9.172 Logical Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Logical`' to use this name for an external procedure.

10.11.9.173 Long Intrinsic

`Long(`*A*`)`

Long: `INTEGER(KIND=1)` function.

A: `INTEGER(KIND=6)`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Archaic form of `INT()` that is specific to one type for *A*. See Section 10.11.9.148 [Int Intrinsic], page 138.

The precise meaning of this intrinsic might change in a future version of the GNU Fortran language, as more is learned about how it is used.

10.11.9.174 LShift Intrinsic

`LShift(`*I*`, `*Shift*`)`

LShift: `INTEGER` function, the '`KIND=`' value of the type being that of argument *I*.

I: `INTEGER`; scalar; INTENT(IN).

Shift: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: `f2c`.

Description:

Returns *I* shifted to the left *Shift* bits.

Although similar to the expression '*I*`*(2**`*Shift*`)`', there are important differences. For example, the sign of the result is not necessarily the same as the sign of *I*.

Currently this intrinsic is defined assuming the underlying representation of *I* is as a two's-complement integer. It is unclear at this point whether that definition will apply when a different representation is involved.

See Section 10.11.9.174 [LShift Intrinsic], page 146, for the inverse of this function.

See Section 10.11.9.154 [IShft Intrinsic], page 140, for information on a more widely available left-shifting intrinsic that is also more precisely defined.

10.11.9.175 LStat Intrinsic (subroutine)

`CALL LStat(`*File*`, `*SArray*`, `*Status*`)`

File: `CHARACTER`; scalar; INTENT(IN).

SArray: `INTEGER(KIND=1)`; DIMENSION(13); INTENT(OUT).

Status: `INTEGER(KIND=1)`; OPTIONAL; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Obtains data about the given file *File* and places them in the array *SArray*. A null character ('`CHAR(0)`') marks the end of the name in *File*—otherwise, trailing blanks in *File* are ignored. If *File* is a symbolic link it returns data on the link itself, so the routine is available only on systems that support symbolic links. The values in this array are extracted from the `stat` structure as returned by `fstat(2)` q.v., as follows:

1. File mode
2. Inode number
3. ID of device containing directory entry for file
4. Device id (if relevant)
5. Number of links
6. Owner's uid
7. Owner's gid
8. File size (bytes)
9. Last access time
10. Last modification time
11. Last file status change time
12. Preferred I/O block size
13. Number of blocks allocated

Not all these elements are relevant on all systems. If an element is not relevant, it is returned as 0.

If the *Status* argument is supplied, it contains 0 on success or a non-zero error code upon return (`ENOSYS` if the system does not provide `lstat(2)`).

Some non-GNU implementations of Fortran provide this intrinsic as only a function, not as a subroutine, or do not support the (optional) *Status* argument.

For information on other intrinsics with the same name: See Section 10.11.9.176 [LStat Intrinsic (function)], page 147.

10.11.9.176 LStat Intrinsic (function)

`LStat(`*File*`, `*SArray*`)`

LStat: `INTEGER(KIND=1)` function.

File: `CHARACTER`; scalar; INTENT(IN).

SArray: `INTEGER(KIND=1)`; DIMENSION(13); INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Obtains data about the given file *File* and places them in the array *SArray*. A null character ('`CHAR(0)`') marks the end of the name in *File*—otherwise, trailing blanks in *File* are ignored. If *File* is a symbolic link it returns data on the link itself, so the routine is available only on systems that support symbolic links. The values in this array are extracted from the `stat` structure as returned by `fstat(2)` q.v., as follows:

1. File mode
2. Inode number
3. ID of device containing directory entry for file
4. Device id (if relevant)
5. Number of links

6. Owner's uid
7. Owner's gid
8. File size (bytes)
9. Last access time
10. Last modification time
11. Last file status change time
12. Preferred I/O block size
13. Number of blocks allocated

Not all these elements are relevant on all systems. If an element is not relevant, it is returned as 0.

Returns 0 on success or a non-zero error code (`ENOSYS` if the system does not provide `lstat(2)`).

For information on other intrinsics with the same name: See Section 10.11.9.175 [LStat Intrinsic (subroutine)], page 146.

10.11.9.177 LTime Intrinsic

`CALL LTime(`*STime*`, `*TArray*`)`

STime: `INTEGER(KIND=1)`; scalar; INTENT(IN).

TArray: `INTEGER(KIND=1)`; DIMENSION(9); INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Given a system time value *STime*, fills *TArray* with values extracted from it appropriate to the GMT time zone using `localtime(3)`.

The array elements are 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 0–31
5. Number of months since January, range 0–12
6. Years since 1900
7. Number of days since Sunday, range 0–6
8. Days since January 1
9. Daylight savings indicator: positive if daylight savings is in effect, zero if not, and negative if the information isn't available.

10.11.9.178 MatMul Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL MatMul`' to use this name for an external procedure.

10.11.9.179 Max Intrinsic

`Max(`*A*`-1, `*A*`-2, ..., `*A*`-n)`

Max: `INTEGER` or `REAL` function, the exact type being the result of cross-promoting the types of all the arguments.

A: `INTEGER` or `REAL`; at least two such arguments must be provided; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns the argument with the largest value.

See Section 10.11.9.188 [Min Intrinsic], page 150, for the opposite function.

10.11.9.180 Max0 Intrinsic

`Max0(`*A*`-1, `*A*`-2, ..., `*A*`-n)`

Max0: `INTEGER(KIND=1)` function.

A: `INTEGER(KIND=1)`; at least two such arguments must be provided; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `MAX()` that is specific to one type for *A*. See Section 10.11.9.179 [Max Intrinsic], page 149.

10.11.9.181 Max1 Intrinsic

`Max1(`*A*`-1, `*A*`-2, ..., `*A*`-n)`

Max1: `INTEGER(KIND=1)` function.

A: `REAL(KIND=1)`; at least two such arguments must be provided; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `MAX()` that is specific to one type for *A* and a different return type. See Section 10.11.9.179 [Max Intrinsic], page 149.

10.11.9.182 MaxExponent Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL MaxExponent`' to use this name for an external procedure.

10.11.9.183 MaxLoc Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL MaxLoc`' to use this name for an external procedure.

10.11.9.184 MaxVal Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL MaxVal`' to use this name for an external procedure.

10.11.9.185 MClock Intrinsic

```
MClock()
```

MClock: `INTEGER(KIND=1)` function.

Intrinsic groups: `unix`.

Description:

Returns the number of clock ticks since the start of the process. Supported on systems with `clock(3)` (q.v.).

This intrinsic is not fully portable, such as to systems with 32-bit `INTEGER` types but supporting times wider than 32 bits. See Section 10.11.9.186 [MClock8 Intrinsic], page 150, for information on a similar intrinsic that might be portable to more GNU Fortran implementations, though to fewer Fortran compilers.

If the system does not support `clock(3)`, -1 is returned.

10.11.9.186 MClock8 Intrinsic

```
MClock8()
```

MClock8: `INTEGER(KIND=2)` function.

Intrinsic groups: `unix`.

Description:

Returns the number of clock ticks since the start of the process. Supported on systems with `clock(3)` (q.v.).

No Fortran implementations other than GNU Fortran are known to support this intrinsic at the time of this writing. See Section 10.11.9.185 [MClock Intrinsic], page 150, for information on a similar intrinsic that might be portable to more Fortran compilers, though to fewer GNU Fortran implementations.

If the system does not support `clock(3)`, -1 is returned.

10.11.9.187 Merge Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Merge`' to use this name for an external procedure.

10.11.9.188 Min Intrinsic

```
Min(A-1, A-2, ..., A-n)
```

Min: `INTEGER` or `REAL` function, the exact type being the result of cross-promoting the types of all the arguments.

A: `INTEGER` or `REAL`; at least two such arguments must be provided; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns the argument with the smallest value.

See Section 10.11.9.179 [Max Intrinsic], page 149, for the opposite function.

10.11.9.189 Min0 Intrinsic

`Min0(`*A*`-1, `*A*`-2, ..., `*A*`-n)`

Min0: `INTEGER(KIND=1)` function.

A: `INTEGER(KIND=1)`; at least two such arguments must be provided; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `MIN()` that is specific to one type for *A*. See Section 10.11.9.188 [Min Intrinsic], page 150.

10.11.9.190 Min1 Intrinsic

`Min1(`*A*`-1, `*A*`-2, ..., `*A*`-n)`

Min1: `INTEGER(KIND=1)` function.

A: `REAL(KIND=1)`; at least two such arguments must be provided; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `MIN()` that is specific to one type for *A* and a different return type. See Section 10.11.9.188 [Min Intrinsic], page 150.

10.11.9.191 MinExponent Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL MinExponent`' to use this name for an external procedure.

10.11.9.192 MinLoc Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL MinLoc`' to use this name for an external procedure.

10.11.9.193 MinVal Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL MinVal`' to use this name for an external procedure.

10.11.9.194 Mod Intrinsic

`Mod(`*A*`, `*P*`)`

Mod: `INTEGER` or `REAL` function, the exact type being the result of cross-promoting the types of all the arguments.

A: `INTEGER` or `REAL`; scalar; INTENT(IN).

P: `INTEGER` or `REAL`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns remainder calculated as:

```
A - (INT(A / P) * P)
```

P must not be zero.

10.11.9.195 Modulo Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Modulo`' to use this name for an external procedure.

10.11.9.196 MvBits Intrinsic

```
CALL MvBits(From, FromPos, Len, TO, ToPos)
```

From: `INTEGER`; scalar; INTENT(IN).

FromPos: `INTEGER`; scalar; INTENT(IN).

Len: `INTEGER`; scalar; INTENT(IN).

TO: `INTEGER` with same '`KIND=`' value as for *From*; scalar; INTENT(INOUT).

ToPos: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: `mil`, `f90`, `vxt`.

Description:

Moves *Len* bits from positions *FromPos* through '*FromPos*+*Len*`-1`' of *From* to positions *ToPos* through '*FromPos*+*Len*`-1`' of *TO*. The portion of argument *TO* not affected by the movement of bits is unchanged. Arguments *From* and *TO* are permitted to be the same numeric storage unit. The values of '*FromPos*+*Len*' and '*ToPos*+*Len*' must be less than or equal to '`BIT_SIZE(`*From*`)`'.

10.11.9.197 Nearest Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Nearest`' to use this name for an external procedure.

10.11.9.198 NInt Intrinsic

```
NInt(A)
```

NInt: `INTEGER(KIND=1)` function.

A: `REAL`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns *A* with the fractional portion of its magnitude eliminated by rounding to the nearest whole number and with its sign preserved, converted to type `INTEGER(KIND=1)`.

If *A* is type `COMPLEX`, its real part is rounded and converted.

A fractional portion exactly equal to '`.5`' is rounded to the whole number that is larger in magnitude. (Also called "Fortran round".)

See Section 10.11.9.148 [Int Intrinsic], page 138, for how to convert, truncate to whole number.

See Section 10.11.9.21 [ANInt Intrinsic], page 103, for how to round to nearest whole number without converting.

10.11.9.199 Not Intrinsic

`Not(`*I*`)`

Not: `INTEGER` function, the '`KIND=`' value of the type being that of argument *I*.

I: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: `mil`, `f90`, `vxt`.

Description:

Returns value resulting from boolean NOT of each bit in *I*.

10.11.9.200 Or Intrinsic

`Or(`*I*`, `*J*`)`

Or: `INTEGER` or `LOGICAL` function, the exact type being the result of cross-promoting the types of all the arguments.

I: `INTEGER` or `LOGICAL`; scalar; INTENT(IN).

J: `INTEGER` or `LOGICAL`; scalar; INTENT(IN).

Intrinsic groups: `f2c`.

Description:

Returns value resulting from boolean OR of pair of bits in each of *I* and *J*.

10.11.9.201 Pack Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Pack`' to use this name for an external procedure.

10.11.9.202 PError Intrinsic

`CALL PError(`*String*`)`

String: `CHARACTER`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

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)`.

10.11.9.203 Precision Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Precision`' to use this name for an external procedure.

10.11.9.204 Present Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Present`' to use this name for an external procedure.

10.11.9.205 Product Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Product`' to use this name for an external procedure.

10.11.9.206 Radix Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Radix`' to use this name for an external procedure.

10.11.9.207 Rand Intrinsic

`Rand(`*Flag*`)`

Rand: `REAL(KIND=1)` function.

Flag: `INTEGER`; OPTIONAL; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Returns a uniform quasi-random number between 0 and 1. If *Flag* is 0, the next number in sequence is returned; if *Flag* is 1, the generator is restarted by calling '`srand(0)`'; if *Flag* has any other value, it is used as a new seed with `srand`.

See Section 10.11.9.236 [SRand Intrinsic], page 161.

Note: As typically implemented (by the routine of the same name in the C library), this random number generator is a very poor one, though the BSD and GNU libraries provide a much better implementation than the 'traditional' one. On a different system you almost certainly want to use something better.

10.11.9.208 Random_Number Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Random_Number`' to use this name for an external procedure.

10.11.9.209 Random_Seed Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Random_Seed`' to use this name for an external procedure.

10.11.9.210 Range Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Range`' to use this name for an external procedure.

10.11.9.211 Real Intrinsic

`Real(`*A*`)`

Real: `REAL` function. The exact type is '`REAL(KIND=1)`' when argument *A* is any type other than `COMPLEX`, or when it is `COMPLEX(KIND=1)`. When *A* is any `COMPLEX` type other than

`COMPLEX(KIND=1)`, this intrinsic is valid only when used as the argument to `REAL()`, as explained below.

A: `INTEGER`, `REAL`, or `COMPLEX`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Converts *A* to `REAL(KIND=1)`.

Use of `REAL()` with a `COMPLEX` argument (other than `COMPLEX(KIND=1)`) is restricted to the following case:

```
REAL(REAL(A))
```

This expression converts the real part of A to `REAL(KIND=1)`.

See Section 10.11.9.212 [RealPart Intrinsic], page 155, for information on a GNU Fortran intrinsic that extracts the real part of an arbitrary `COMPLEX` value.

See Section 10.11.5 [REAL() and AIMAG() of Complex], page 96, for more information.

10.11.9.212 RealPart Intrinsic

```
RealPart(Z)
```

RealPart: `REAL` function, the '`KIND=`' value of the type being that of argument *Z*.

Z: `COMPLEX`; scalar; INTENT(IN).

Intrinsic groups: `gnu`.

Description:

The real part of *Z* is returned, without conversion.

Note: The way to do this in standard Fortran 90 is '`REAL(Z)`'. However, when, for example, *Z* is `COMPLEX(KIND=2)`, '`REAL(Z)`' means something different for some compilers that are not true Fortran 90 compilers but offer some extensions standardized by Fortran 90 (such as the `DOUBLE COMPLEX` type, also known as `COMPLEX(KIND=2)`).

The advantage of `REALPART()` is that, while not necessarily more or less portable than `REAL()`, it is more likely to cause a compiler that doesn't support it to produce a diagnostic than generate incorrect code.

See Section 10.11.5 [REAL() and AIMAG() of Complex], page 96, for more information.

10.11.9.213 Rename Intrinsic (subroutine)

```
CALL Rename(Path1, Path2, Status)
```

Path1: `CHARACTER`; scalar; INTENT(IN).

Path2: `CHARACTER`; scalar; INTENT(IN).

Status: `INTEGER(KIND=1)`; OPTIONAL; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Renames the file *Path1* to *Path2*. A null character ('`CHAR(0)`') marks the end of the names in *Path1* and *Path2*—otherwise, trailing blanks in *Path1* and *Path2* are ignored. See

`rename(2)`. If the *Status* argument is supplied, it contains 0 on success or a non-zero error code upon return.

Some non-GNU implementations of Fortran provide this intrinsic as only a function, not as a subroutine, or do not support the (optional) *Status* argument.

For information on other intrinsics with the same name: See Section 12.4.2.126 [Rename Intrinsic (function)], page 204.

10.11.9.214 Repeat Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Repeat`' to use this name for an external procedure.

10.11.9.215 Reshape Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Reshape`' to use this name for an external procedure.

10.11.9.216 RRSpacing Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL RRSpacing`' to use this name for an external procedure.

10.11.9.217 RShift Intrinsic

`RShift(`*I*`, `*Shift*`)`

RShift: `INTEGER` function, the '`KIND=`' value of the type being that of argument *I*.

I: `INTEGER`; scalar; INTENT(IN).

Shift: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: `f2c`.

Description:

Returns *I* shifted to the right *Shift* bits.

Although similar to the expression '*I*`/(2**`*Shift*`)`', there are important differences. For example, the sign of the result is undefined.

Currently this intrinsic is defined assuming the underlying representation of *I* is as a two's-complement integer. It is unclear at this point whether that definition will apply when a different representation is involved.

See Section 10.11.9.217 [RShift Intrinsic], page 156, for the inverse of this function.

See Section 10.11.9.154 [IShft Intrinsic], page 140, for information on a more widely available right-shifting intrinsic that is also more precisely defined.

10.11.9.218 Scale Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Scale`' to use this name for an external procedure.

10.11.9.219 Scan Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Scan`' to use this name for an external procedure.

10.11.9.220 Second Intrinsic (function)

```
Second()
```

Second: `REAL(KIND=1)` function.

Intrinsic groups: `unix`.

Description:

Returns the process's runtime in seconds—the same value as the UNIX function `etime` returns.

For information on other intrinsics with the same name: See Section 10.11.9.221 [Second Intrinsic (subroutine)], page 157.

10.11.9.221 Second Intrinsic (subroutine)

```
CALL Second(Seconds)
```

Seconds: `REAL`; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Returns the process's runtime in seconds in *Seconds*—the same value as the UNIX function `etime` returns.

This routine is known from Cray Fortran. See Section 10.11.9.49 [CPU_Time Intrinsic], page 111 for a standard equivalent.

For information on other intrinsics with the same name: See Section 10.11.9.220 [Second Intrinsic (function)], page 157.

10.11.9.222 Selected_Int_Kind Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Selected_Int_Kind`' to use this name for an external procedure.

10.11.9.223 Selected_Real_Kind Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Selected_Real_Kind`' to use this name for an external procedure.

10.11.9.224 Set_Exponent Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Set_Exponent`' to use this name for an external procedure.

10.11.9.225 Shape Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Shape`' to use this name for an external procedure.

10.11.9.226 Short Intrinsic

`Short(`*A*`)`

Short: `INTEGER(KIND=6)` function.

A: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Returns *A* with the fractional portion of its magnitude truncated and its sign preserved, converted to type `INTEGER(KIND=6)`.

If *A* is type `COMPLEX`, its real part is truncated and converted, and its imaginary part is disgregarded.

See Section 10.11.9.148 [Int Intrinsic], page 138.

The precise meaning of this intrinsic might change in a future version of the GNU Fortran language, as more is learned about how it is used.

10.11.9.227 Sign Intrinsic

`Sign(`*A*`, `*B*`)`

Sign: `INTEGER` or `REAL` function, the exact type being the result of cross-promoting the types of all the arguments.

A: `INTEGER` or `REAL`; scalar; INTENT(IN).

B: `INTEGER` or `REAL`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns '`ABS(`*A*`)*`*s*', where *s* is `+1` if '*B*`.GE.0`', `-1` otherwise.

See Section 10.11.9.2 [Abs Intrinsic], page 98, for the function that returns the magnitude of a value.

10.11.9.228 Signal Intrinsic (subroutine)

`CALL Signal(`*Number*`, `*Handler*`, `*Status*`)`

Number: `INTEGER`; scalar; INTENT(IN).

Handler: Signal handler (`INTEGER FUNCTION` or `SUBROUTINE`) or dummy/global `INTEGER(KIND=1)` scalar.

Status: `INTEGER(KIND=7)`; OPTIONAL; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

If *Handler* is a an `EXTERNAL` routine, arranges for it to be invoked with a single integer argument (of system-dependent length) 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)`.

Note that *Handler* will be called using C conventions, so the value of its argument in Fortran terms Fortran terms is obtained by applying `%LOC()` (or *LOC()*) to it.

The value returned by `signal(2)` is written to *Status*, if that argument is supplied. Otherwise the return value is ignored.

Some non-GNU implementations of Fortran provide this intrinsic as only a function, not as a subroutine, or do not support the (optional) *Status* argument.

Warning: Use of the `libf2c` run-time library function '`signal_`' directly (such as via '`EXTERNAL SIGNAL`') requires use of the `%VAL()` construct to pass an `INTEGER` value (such as '`SIG_IGN`' or '`SIG_DFL`') for the *Handler* argument.

However, while '`CALL SIGNAL(`*signum*`, %VAL(SIG_IGN))`' works when '`SIGNAL`' is treated as an external procedure (and resolves, at link time, to `libf2c`'s '`signal_`' routine), this construct is not valid when '`SIGNAL`' is recognized as the intrinsic of that name.

Therefore, for maximum portability and reliability, code such references to the '`SIGNAL`' facility as follows:

```
INTRINSIC SIGNAL
...
CALL SIGNAL(signum, SIG_IGN)
```

`g77` will compile such a call correctly, while other compilers will generally either do so as well or reject the '`INTRINSIC SIGNAL`' statement via a diagnostic, allowing you to take appropriate action.

For information on other intrinsics with the same name: See Section 12.4.2.128 [Signal Intrinsic (function)], page 205.

10.11.9.229 Sin Intrinsic

`Sin(`*X*`)`

Sin: `REAL` or `COMPLEX` function, the exact type being that of argument *X*.

X: `REAL` or `COMPLEX`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns the sine of *X*, an angle measured in radians.

See Section 10.11.9.23 [ASin Intrinsic], page 104, for the inverse of this function.

10.11.9.230 SinH Intrinsic

`SinH(`*X*`)`

SinH: `REAL` function, the '`KIND=`' value of the type being that of argument *X*.

X: `REAL`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns the hyperbolic sine of *X*.

10.11.9.231 Sleep Intrinsic

```
CALL Sleep(Seconds)
```

Seconds: `INTEGER(KIND=1)`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Causes the process to pause for *Seconds* seconds. See `sleep(2)`.

10.11.9.232 Sngl Intrinsic

```
Sngl(A)
```

Sngl: `REAL(KIND=1)` function.

A: `REAL(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Archaic form of `REAL()` that is specific to one type for *A*. See Section 10.11.9.211 [Real Intrinsic], page 154.

10.11.9.233 Spacing Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Spacing`' to use this name for an external procedure.

10.11.9.234 Spread Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Spread`' to use this name for an external procedure.

10.11.9.235 SqRt Intrinsic

```
SqRt(X)
```

SqRt: `REAL` or `COMPLEX` function, the exact type being that of argument *X*.

X: `REAL` or `COMPLEX`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns the square root of *X*, which must not be negative.

To calculate and represent the square root of a negative number, complex arithmetic must be used. For example, '`SQRT(COMPLEX(X))`'.

The inverse of this function is '`SQRT(X) * SQRT(X)`'.

10.11.9.236 SRand Intrinsic

```
CALL SRand(Seed)
```

Seed: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Reinitialises the generator with the seed in *Seed*. See Section 10.11.9.152 [IRand Intrinsic], page 139. See Section 10.11.9.207 [Rand Intrinsic], page 154.

10.11.9.237 Stat Intrinsic (subroutine)

```
CALL Stat(File, SArray, Status)
```

File: `CHARACTER`; scalar; INTENT(IN).

SArray: `INTEGER(KIND=1)`; DIMENSION(13); INTENT(OUT).

Status: `INTEGER(KIND=1)`; OPTIONAL; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Obtains data about the given file *File* and places them in the array *SArray*. A null character ('`CHAR(0)`') marks the end of the name in *File*—otherwise, trailing blanks in *File* are ignored. The values in this array are extracted from the `stat` structure as returned by `fstat(2)` q.v., as follows:

1. File mode
2. Inode number
3. ID of device containing directory entry for file
4. Device id (if relevant)
5. Number of links
6. Owner's uid
7. Owner's gid
8. File size (bytes)
9. Last access time
10. Last modification time
11. Last file status change time
12. Preferred I/O block size
13. Number of blocks allocated

Not all these elements are relevant on all systems. If an element is not relevant, it is returned as 0.

If the *Status* argument is supplied, it contains 0 on success or a non-zero error code upon return.

Some non-GNU implementations of Fortran provide this intrinsic as only a function, not as a subroutine, or do not support the (optional) *Status* argument.

For information on other intrinsics with the same name: See Section 10.11.9.238 [Stat Intrinsic (function)], page 162.

10.11.9.238 Stat Intrinsic (function)

`Stat(`*File*`, `*SArray*`)`

Stat: `INTEGER(KIND=1)` function.

File: `CHARACTER`; scalar; INTENT(IN).

SArray: `INTEGER(KIND=1)`; DIMENSION(13); INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Obtains data about the given file *File* and places them in the array *SArray*. A null character ('`CHAR(0)`') marks the end of the name in *File*—otherwise, trailing blanks in *File* are ignored. The values in this array are extracted from the `stat` structure as returned by `fstat(2)` q.v., as follows:

1. File mode
2. Inode number
3. ID of device containing directory entry for file
4. Device id (if relevant)
5. Number of links
6. Owner's uid
7. Owner's gid
8. File size (bytes)
9. Last access time
10. Last modification time
11. Last file status change time
12. Preferred I/O block size
13. Number of blocks allocated

Not all these elements are relevant on all systems. If an element is not relevant, it is returned as 0.

Returns 0 on success or a non-zero error code.

For information on other intrinsics with the same name: See Section 10.11.9.237 [Stat Intrinsic (subroutine)], page 161.

10.11.9.239 Sum Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Sum`' to use this name for an external procedure.

10.11.9.240 SymLnk Intrinsic (subroutine)

`CALL SymLnk(`*Path1*`, `*Path2*`, `*Status*`)`

Path1: `CHARACTER`; scalar; INTENT(IN).

Path2: `CHARACTER`; scalar; INTENT(IN).

Status: `INTEGER(KIND=1)`; OPTIONAL; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Makes a symbolic link from file *Path1* to *Path2*. A null character ('`CHAR(0)`') marks the end of the names in *Path1* and *Path2*—otherwise, trailing blanks in *Path1* and *Path2* are ignored. If the *Status* argument is supplied, it contains 0 on success or a non-zero error code upon return (`ENOSYS` if the system does not provide `symlink(2)`).

Some non-GNU implementations of Fortran provide this intrinsic as only a function, not as a subroutine, or do not support the (optional) *Status* argument.

For information on other intrinsics with the same name: See Section 12.4.2.131 [SymLnk Intrinsic (function)], page 206.

10.11.9.241 System Intrinsic (subroutine)

`CALL System(`*Command*`, `*Status*`)`

Command: `CHARACTER`; scalar; INTENT(IN).

Status: `INTEGER(KIND=1)`; OPTIONAL; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Passes the command *Command* to a shell (see `system(3)`). If argument *Status* is present, it contains the value returned by `system(3)`, presumably 0 if the shell command succeeded. Note that which shell is used to invoke the command is system-dependent and environment-dependent.

Some non-GNU implementations of Fortran provide this intrinsic as only a function, not as a subroutine, or do not support the (optional) *Status* argument.

For information on other intrinsics with the same name: See Section 12.4.2.132 [System Intrinsic (function)], page 207.

10.11.9.242 System_Clock Intrinsic

`CALL System_Clock(`*Count*`, `*Rate*`, `*Max*`)`

Count: `INTEGER(KIND=1)`; scalar; INTENT(OUT).

Rate: `INTEGER(KIND=1)`; scalar; INTENT(OUT).

Max: `INTEGER(KIND=1)`; scalar; INTENT(OUT).

Intrinsic groups: `f90`.

Description:

Returns in *Count* the current value of the system clock; this is the value returned by the UNIX function `times(2)` in this implementation, but isn't in general. *Rate* is the number of clock ticks per second and *Max* is the maximum value this can take, which isn't very useful in this implementation since it's just the maximum C `unsigned int` value.

10.11.9.243 Tan Intrinsic

`Tan(`*X*`)`

Tan: `REAL` function, the '`KIND=`' value of the type being that of argument *X*.

X: `REAL`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns the tangent of *X*, an angle measured in radians.

See Section 10.11.9.25 [ATan Intrinsic], page 104, for the inverse of this function.

10.11.9.244 TanH Intrinsic

`TanH(`*X*`)`

TanH: `REAL` function, the '`KIND=`' value of the type being that of argument *X*.

X: `REAL`; scalar; INTENT(IN).

Intrinsic groups: (standard FORTRAN 77).

Description:

Returns the hyperbolic tangent of *X*.

10.11.9.245 Time Intrinsic (UNIX)

`Time()`

Time: `INTEGER(KIND=1)` function.

Intrinsic groups: `unix`.

Description:

Returns the current time encoded as an integer (in the manner of the UNIX function `time(3)`). This value is suitable for passing to `CTIME`, `GMTIME`, and `LTIME`.

This intrinsic is not fully portable, such as to systems with 32-bit `INTEGER` types but supporting times wider than 32 bits. See Section 10.11.9.246 [Time8 Intrinsic], page 164, for information on a similar intrinsic that might be portable to more GNU Fortran implementations, though to fewer Fortran compilers.

For information on other intrinsics with the same name: See Section 12.4.2.134 [Time Intrinsic (VXT)], page 207.

10.11.9.246 Time8 Intrinsic

`Time8()`

Time8: `INTEGER(KIND=2)` function.

Intrinsic groups: `unix`.

Description:

Returns the current time encoded as a long integer (in the manner of the UNIX function `time(3)`). This value is suitable for passing to `CTIME`, `GMTIME`, and `LTIME`.

No Fortran implementations other than GNU Fortran are known to support this intrinsic at the time of this writing. See Section 10.11.9.245 [Time Intrinsic (UNIX)], page 164, for information on a similar intrinsic that might be portable to more Fortran compilers, though to fewer GNU Fortran implementations.

10.11.9.247 Tiny Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Tiny`' to use this name for an external procedure.

10.11.9.248 Transfer Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Transfer`' to use this name for an external procedure.

10.11.9.249 Transpose Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Transpose`' to use this name for an external procedure.

10.11.9.250 Trim Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Trim`' to use this name for an external procedure.

10.11.9.251 TtyNam Intrinsic (subroutine)

`CALL TtyNam(`*Name*`, `*Unit*`)`

Name: `CHARACTER`; scalar; INTENT(OUT).

Unit: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Sets *Name* to the name of the terminal device open on logical unit *Unit* or a blank string if *Unit* is not connected to a terminal.

Some non-GNU implementations of Fortran provide this intrinsic as only a function, not as a subroutine.

For information on other intrinsics with the same name: See Section 10.11.9.252 [TtyNam Intrinsic (function)], page 165.

10.11.9.252 TtyNam Intrinsic (function)

`TtyNam(`*Unit*`)`

TtyNam: `CHARACTER*(*)` function.

Unit: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: `unix`.

Description:

Returns the name of the terminal device open on logical unit *Unit* or a blank string if *Unit* is not connected to a terminal.

For information on other intrinsics with the same name: See Section 10.11.9.251 [TtyNam Intrinsic (subroutine)], page 165.

10.11.9.253 UBound Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL UBound`' to use this name for an external procedure.

10.11.9.254 UMask Intrinsic (subroutine)

```
CALL UMask(Mask, Old)
```

Mask: `INTEGER`; scalar; INTENT(IN).

Old: `INTEGER(KIND=1)`; OPTIONAL; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Sets the file creation mask to *Mask* and returns the old value in argument *Old* if it is supplied. See `umask(2)`.

Some non-GNU implementations of Fortran provide this intrinsic as only a function, not as a subroutine.

For information on other intrinsics with the same name: See Section 12.4.2.135 [UMask Intrinsic (function)], page 208.

10.11.9.255 Unlink Intrinsic (subroutine)

```
CALL Unlink(File, Status)
```

File: `CHARACTER`; scalar; INTENT(IN).

Status: `INTEGER(KIND=1)`; OPTIONAL; scalar; INTENT(OUT).

Intrinsic groups: `unix`.

Description:

Unlink the file *File*. A null character ('`CHAR(0)`') marks the end of the name in *File*—otherwise, trailing blanks in *File* are ignored. If the *Status* argument is supplied, it contains 0 on success or a non-zero error code upon return. See `unlink(2)`.

Some non-GNU implementations of Fortran provide this intrinsic as only a function, not as a subroutine, or do not support the (optional) *Status* argument.

For information on other intrinsics with the same name: See Section 12.4.2.136 [Unlink Intrinsic (function)], page 208.

10.11.9.256 Unpack Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Unpack`' to use this name for an external procedure.

10.11.9.257 Verify Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL Verify`' to use this name for an external procedure.

10.11.9.258 XOr Intrinsic

`XOr(`*I*`, `*J*`)`

XOr: `INTEGER` or `LOGICAL` function, the exact type being the result of cross-promoting the types of all the arguments.

I: `INTEGER` or `LOGICAL`; scalar; INTENT(IN).

J: `INTEGER` or `LOGICAL`; scalar; INTENT(IN).

Intrinsic groups: `f2c`.

Description:

Returns value resulting from boolean exclusive-OR of pair of bits in each of *I* and *J*.

10.11.9.259 ZAbs Intrinsic

`ZAbs(`*A*`)`

ZAbs: `REAL(KIND=2)` function.

A: `COMPLEX(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: `f2c`.

Description:

Archaic form of `ABS()` that is specific to one type for *A*. See Section 10.11.9.2 [Abs Intrinsic], page 98.

10.11.9.260 ZCos Intrinsic

`ZCos(`*X*`)`

ZCos: `COMPLEX(KIND=2)` function.

X: `COMPLEX(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: `f2c`.

Description:

Archaic form of `COS()` that is specific to one type for *X*. See Section 10.11.9.46 [Cos Intrinsic], page 110.

10.11.9.261 ZExp Intrinsic

`ZExp(`*X*`)`

ZExp: `COMPLEX(KIND=2)` function.

X: `COMPLEX(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: `f2c`.

Description:

Archaic form of `EXP()` that is specific to one type for *X*. See Section 10.11.9.99 [Exp Intrinsic], page 123.

10.11.9.262 ZLog Intrinsic

`ZLog(`*X*`)`

ZLog: `COMPLEX(KIND=2)` function.

X: `COMPLEX(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: `f2c`.

Description:

Archaic form of `LOG()` that is specific to one type for *X*. See Section 10.11.9.170 [Log Intrinsic], page 145.

10.11.9.263 ZSin Intrinsic

`ZSin(`*X*`)`

ZSin: `COMPLEX(KIND=2)` function.

X: `COMPLEX(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: `f2c`.

Description:

Archaic form of `SIN()` that is specific to one type for *X*. See Section 10.11.9.229 [Sin Intrinsic], page 159.

10.11.9.264 ZSqRt Intrinsic

`ZSqRt(`*X*`)`

ZSqRt: `COMPLEX(KIND=2)` function.

X: `COMPLEX(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: `f2c`.

Description:

Archaic form of `SQRT()` that is specific to one type for *X*. See Section 10.11.9.235 [SqRt Intrinsic], page 160.

10.12 Scope and Classes of Symbolic Names

(The following information augments or overrides the information in Chapter 18 of ANSI X3.9-1978 FORTRAN 77 in specifying the GNU Fortran language. Chapter 18 of that document otherwise serves as the basis for the relevant aspects of GNU Fortran.)

10.12.1 Underscores in Symbol Names

Underscores ('_') are accepted in symbol names after the first character (which must be a letter).

11 Other Dialects

GNU Fortran supports a variety of features that are not considered part of the GNU Fortran language itself, but are representative of various dialects of Fortran that `g77` supports in whole or in part.

Any of the features listed below might be disallowed by `g77` unless some command-line option is specified. Currently, some of the features are accepted using the default invocation of `g77`, but that might change in the future.

Note: This portion of the documentation definitely needs a lot of work!

11.1 Source Form

GNU Fortran accepts programs written in either fixed form or free form.

Fixed form corresponds to ANSI FORTRAN 77 (plus popular extensions, such as allowing tabs) and Fortran 90's fixed form.

Free form corresponds to Fortran 90's free form (though possibly not entirely up-to-date, and without complaining about some things that for which Fortran 90 requires diagnostics, such as the spaces in the constant in '`R = 3 . 1`').

The way a Fortran compiler views source files depends entirely on the implementation choices made for the compiler, since those choices are explicitly left to the implementation by the published Fortran standards. GNU Fortran currently tries to be somewhat like a few popular compilers (`f2c`, Digital ("DEC") Fortran, and so on), though a cleaner default definition along with more flexibility offered by command-line options is likely to be offered in version 0.6.

This section describes how `g77` interprets source lines.

11.1.1 Carriage Returns

Carriage returns ('`\r`') in source lines are ignored. This is somewhat different from `f2c`, which seems to treat them as spaces outside character/Hollerith constants, and encodes them as '`\r`' inside such constants.

11.1.2 Tabs

A source line with a ⟨TAB⟩ character anywhere in it is treated as entirely significant—however long it is—instead of ending in column 72 (for fixed-form source) or 132 (for free-form source). This also is different from `f2c`, which encodes tabs as '`\t`' (the ASCII ⟨TAB⟩ character) inside character and Hollerith constants, but nevertheless seems to treat the column position as if it had been affected by the canonical tab positioning.

`g77` effectively translates tabs to the appropriate number of spaces (a la the default for the UNIX `expand` command) before doing any other processing, other than (currently) noting whether a tab was found on a line and using this information to decide how to interpret the length of the line and continued constants.

Note that this default behavior probably will change for version 0.6, when it will presumably be available via a command-line option. The default as of version 0.6 is planned

to be a "pure visual" model, where tabs are immediately converted to spaces and otherwise have no effect, so the way a typical user sees source lines produces a consistent result no matter how the spacing in those source lines is actually implemented via tabs, spaces, and trailing tabs/spaces before newline. Command-line options are likely to be added to specify whether all or just-tabbed lines are to be extended to 132 or full input-line length, and perhaps even an option will be added to specify the truncated-line behavior to which some Digital compilers default (and which affects the way continued character/Hollerith constants are interpreted).

11.1.3 Short Lines

Source lines shorter than the applicable fixed-form length are treated as if they were padded with spaces to that length. (None of this is relevant to source files written in free form.)

This affects only continued character and Hollerith constants, and is a different interpretation than provided by some other popular compilers (although a bit more consistent with the traditional punched-card basis of Fortran and the way the Fortran standard expressed fixed source form).

`g77` might someday offer an option to warn about cases where differences might be seen as a result of this treatment, and perhaps an option to specify the alternate behavior as well.

Note that this padding cannot apply to lines that are effectively of infinite length—such lines are specified using command-line options like '`-ffixed-line-length-none`', for example.

11.1.4 Long Lines

Source lines longer than the applicable length are truncated to that length. Currently, `g77` does not warn if the truncated characters are not spaces, to accommodate existing code written for systems that treated truncated text as commentary (especially in columns 73 through 80).

See Section 7.4 [Options Controlling Fortran Dialect], page 29, for information on the '`-ffixed-line-length-`*n*' option, which can be used to set the line length applicable to fixed-form source files.

11.1.5 Ampersand Continuation Line

A '`&`' in column 1 of fixed-form source denotes an arbitrary-length continuation line, imitating the behavior of `f2c`.

11.2 Trailing Comment

`g77` supports use of '`/*`' to start a trailing comment. In the GNU Fortran language, '`!`' is used for this purpose.

'`/*`' is not in the GNU Fortran language because the use of '`/*`' in a program might suggest to some readers that a block, not trailing, comment is started (and thus ended by '`*/`', not end of line), since that is the meaning of '`/*`' in C.

Also, such readers might think they can use '`//`' to start a trailing comment as an alternative to '`/*`', but '`//`' already denotes concatenation, and such a "comment" might actually result in a program that compiles without error (though it would likely behave incorrectly).

11.3 Debug Line

Use of '`D`' or '`d`' as the first character (column 1) of a source line denotes a debug line.

In turn, a debug line is treated as either a comment line or a normal line, depending on whether debug lines are enabled.

When treated as a comment line, a line beginning with '`D`' or '`d`' is treated as if it the first character was '`C`' or '`c`', respectively. When treated as a normal line, such a line is treated as if the first character was ⟨SPC⟩ (space).

(Currently, `g77` provides no means for treating debug lines as normal lines.)

11.4 Dollar Signs in Symbol Names

Dollar signs ('`$`') are allowed in symbol names (after the first character) when the '`-fdollar-ok`' option is specified.

11.5 Case Sensitivity

GNU Fortran offers the programmer way too much flexibility in deciding how source files are to be treated vis-a-vis uppercase and lowercase characters. There are 66 useful settings that affect case sensitivity, plus 10 settings that are nearly useless, with the remaining 116 settings being either redundant or useless.

None of these settings have any effect on the contents of comments (the text after a '`c`' or '`C`' in Column 1, for example) or of character or Hollerith constants. Note that things like the '`E`' in the statement '`CALL FOO(3.2E10)`' and the '`TO`' in '`ASSIGN 10 TO LAB`' are considered built-in keywords, and so are affected by these settings.

Low-level switches are identified in this section as follows:

A Source Case Conversion:

0 Preserve (see Note 1)

1 Convert to Upper Case

2 Convert to Lower Case

B Built-in Keyword Matching:

0 Match Any Case (per-character basis)

1 Match Upper Case Only

2 Match Lower Case Only

3 Match InitialCaps Only (see tables for spellings)

C Built-in Intrinsic Matching:

0 Match Any Case (per-character basis)

1 Match Upper Case Only

2 Match Lower Case Only

3 Match InitialCaps Only (see tables for spellings)

D User-defined Symbol Possibilities (warnings only):

0 Allow Any Case (per-character basis)

1 Allow Upper Case Only

2 Allow Lower Case Only

3 Allow InitialCaps Only (see Note 2)

Note 1: `g77` eventually will support `NAMELIST` in a manner that is consistent with these source switches—in the sense that input will be expected to meet the same requirements as source code in terms of matching symbol names and keywords (for the exponent letters).

Currently, however, `NAMELIST` is supported by `libf2c`, which uppercases `NAMELIST` input and symbol names for matching. This means not only that `NAMELIST` output currently shows symbol (and keyword) names in uppercase even if lower-case source conversion (option A2) is selected, but that `NAMELIST` cannot be adequately supported when source case preservation (option A0) is selected.

If A0 is selected, a warning message will be output for each `NAMELIST` statement to this effect. The behavior of the program is undefined at run time if two or more symbol names appear in a given `NAMELIST` such that the names are identical when converted to upper case (e.g. '`NAMELIST /X/ VAR, Var, var`'). For complete and total elegance, perhaps there should be a warning when option A2 is selected, since the output of NAMELIST is currently in uppercase but will someday be lowercase (when a `libg77` is written), but that seems to be overkill for a product in beta test.

Note 2: Rules for InitialCaps names are:

- Must be a single uppercase letter, **or**
- Must start with an uppercase letter and contain at least one lowercase letter.

So '`A`', '`Ab`', '`ABc`', '`AbC`', and '`Abc`' are valid InitialCaps names, but '`AB`', '`A2`', and '`ABC`' are not. Note that most, but not all, built-in names meet these requirements—the exceptions are some of the two-letter format specifiers, such as '`BN`' and '`BZ`'.

Here are the names of the corresponding command-line options:

```
A0: -fsource-case-preserve
A1: -fsource-case-upper
A2: -fsource-case-lower

B0: -fmatch-case-any
B1: -fmatch-case-upper
B2: -fmatch-case-lower
B3: -fmatch-case-initcap

C0: -fintrin-case-any
```

```
C1: -fintrin-case-upper
C2: -fintrin-case-lower
C3: -fintrin-case-initcap

D0: -fsymbol-case-any
D1: -fsymbol-case-upper
D2: -fsymbol-case-lower
D3: -fsymbol-case-initcap
```

Useful combinations of the above settings, along with abbreviated option names that set some of these combinations all at once:

```
 1: A0--  B0---  C0---  D0---    -fcase-preserve
 2: A0--  B0---  C0---  D-1--
 3: A0--  B0---  C0---  D--2-
 4: A0--  B0---  C0---  D---3
 5: A0--  B0---  C-1--  D0---
 6: A0--  B0---  C-1--  D-1--
 7: A0--  B0---  C-1--  D--2-
 8: A0--  B0---  C-1--  D---3
 9: A0--  B0---  C--2-  D0---
10: A0--  B0---  C--2-  D-1--
11: A0--  B0---  C--2-  D--2-
12: A0--  B0---  C--2-  D---3
13: A0--  B0---  C---3  D0---
14: A0--  B0---  C---3  D-1--
15: A0--  B0---  C---3  D--2-
16: A0--  B0---  C---3  D---3
17: A0--  B-1--  C0---  D0---
18: A0--  B-1--  C0---  D-1--
19: A0--  B-1--  C0---  D--2-
20: A0--  B-1--  C0---  D---3
21: A0--  B-1--  C-1--  D0---
22: A0--  B-1--  C-1--  D-1--    -fcase-strict-upper
23: A0--  B-1--  C-1--  D--2-
24: A0--  B-1--  C-1--  D---3
25: A0--  B-1--  C--2-  D0---
26: A0--  B-1--  C--2-  D-1--
27: A0--  B-1--  C--2-  D--2-
28: A0--  B-1--  C--2-  D---3
29: A0--  B-1--  C---3  D0---
30: A0--  B-1--  C---3  D-1--
31: A0--  B-1--  C---3  D--2-
32: A0--  B-1--  C---3  D---3
33: A0--  B--2-  C0---  D0---
34: A0--  B--2-  C0---  D-1--
35: A0--  B--2-  C0---  D--2-
36: A0--  B--2-  C0---  D---3
37: A0--  B--2-  C-1--  D0---
38: A0--  B--2-  C-1--  D-1--
```

```
39: A0--  B--2-  C-1--  D--2-
40: A0--  B--2-  C-1--  D---3
41: A0--  B--2-  C--2-  D0---
42: A0--  B--2-  C--2-  D-1--
43: A0--  B--2-  C--2-  D--2-    -fcase-strict-lower
44: A0--  B--2-  C--2-  D---3
45: A0--  B--2-  C---3  D0---
46: A0--  B--2-  C---3  D-1--
47: A0--  B--2-  C---3  D--2-
48: A0--  B--2-  C---3  D---3
49: A0--  B---3  C0---  D0---
50: A0--  B---3  C0---  D-1--
51: A0--  B---3  C0---  D--2-
52: A0--  B---3  C0---  D---3
53: A0--  B---3  C-1--  D0---
54: A0--  B---3  C-1--  D-1--
55: A0--  B---3  C-1--  D--2-
56: A0--  B---3  C-1--  D---3
57: A0--  B---3  C--2-  D0---
58: A0--  B---3  C--2-  D-1--
59: A0--  B---3  C--2-  D--2-
60: A0--  B---3  C--2-  D---3
61: A0--  B---3  C---3  D0---
62: A0--  B---3  C---3  D-1--
63: A0--  B---3  C---3  D--2-
64: A0--  B---3  C---3  D---3    -fcase-initcap
65: A-1-  B01--  C01--  D01--    -fcase-upper
66: A--2  B0-2-  C0-2-  D0-2-    -fcase-lower
```

Number 22 is the "strict" ANSI FORTRAN 77 model wherein all input (except comments, character constants, and Hollerith strings) must be entered in uppercase. Use '`-fcase-strict-upper`' to specify this combination.

Number 43 is like Number 22 except all input must be lowercase. Use '`-fcase-strict-lower`' to specify this combination.

Number 65 is the "classic" ANSI FORTRAN 77 model as implemented on many non-UNIX machines whereby all the source is translated to uppercase. Use '`-fcase-upper`' to specify this combination.

Number 66 is the "canonical" UNIX model whereby all the source is translated to lowercase. Use '`-fcase-lower`' to specify this combination.

There are a few nearly useless combinations:

```
67: A-1-  B01--  C01--  D--2-
68: A-1-  B01--  C01--  D---3
69: A-1-  B01--  C--23  D01--
70: A-1-  B01--  C--23  D--2-
71: A-1-  B01--  C--23  D---3
72: A--2  B01--  C0-2-  D-1--
73: A--2  B01--  C0-2-  D---3
74: A--2  B01--  C-1-3  D0-2-
```

```
75: A--2  B01--  C-1-3  D-1--
76: A--2  B01--  C-1-3  D---3
```

The above allow some programs to be compiled but with restrictions that make most useful programs impossible: Numbers 67 and 72 warn about *any* user-defined symbol names (such as '`SUBROUTINE FOO`'); Numbers 68 and 73 warn about any user-defined symbol names longer than one character that don't have at least one non-alphabetic character after the first; Numbers 69 and 74 disallow any references to intrinsics; and Numbers 70, 71, 75, and 76 are combinations of the restrictions in 67+69, 68+69, 72+74, and 73+74, respectively.

All redundant combinations are shown in the above tables anyplace where more than one setting is shown for a low-level switch. For example, '`B0-2-`' means either setting 0 or 2 is valid for switch B. The "proper" setting in such a case is the one that copies the setting of switch A—any other setting might slightly reduce the speed of the compiler, though possibly to an unmeasurable extent.

All remaining combinations are useless in that they prevent successful compilation of non-null source files (source files with something other than comments).

11.6 VXT Fortran

`g77` supports certain constructs that have different meanings in VXT Fortran than they do in the GNU Fortran language.

Generally, this manual uses the invented term VXT Fortran to refer VAX FORTRAN (circa v4). That compiler offered many popular features, though not necessarily those that are specific to the VAX processor architecture, the VMS operating system, or Digital Equipment Corporation's Fortran product line. (VAX and VMS probably are trademarks of Digital Equipment Corporation.)

An extension offered by a Digital Fortran product that also is offered by several other Fortran products for different kinds of systems is probably going to be considered for inclusion in `g77` someday, and is considered a VXT Fortran feature.

The '`-fvxt`' option generally specifies that, where the meaning of a construct is ambiguous (means one thing in GNU Fortran and another in VXT Fortran), the VXT Fortran meaning is to be assumed.

11.6.1 Meaning of Double Quote

`g77` treats double-quote ('`"`') as beginning an octal constant of `INTEGER(KIND=1)` type when the `-fvxt` option is specified. The form of this octal constant is

"octal-digits

where *octal-digits* is a nonempty string of characters in the set '`01234567`'.

For example, the `-fvxt` option permits this:

```
PRINT *, "20
END
```

The above program would print the value '`16`'.

See Section 10.7.3 [Integer Type], page 87, for information on the preferred construct for integer constants specified using GNU Fortran's octal notation.

(In the GNU Fortran language, the double-quote character (‘"’) delimits a character constant just as does apostrophe (‘’’). There is no way to allow both constructs in the general case, since statements like ‘`PRINT *,"2000 !comment?"`’ would be ambiguous.)

11.6.2 Meaning of Exclamation Point in Column 6

`g77` treats an exclamation point (‘`!`’) in column 6 of a fixed-form source file as a continuation character rather than as the beginning of a comment (as it does in any other column) when the `-fvxt` option is specified.

The following program, when run, prints a message indicating whether it is interpreted according to GNU Fortran (and Fortran 90) rules or VXT Fortran rules:

```
C234567  (This line begins in column 1.)
      I = 0
     !1
      IF (I.EQ.0) PRINT *, ' I am a VXT Fortran program'
      IF (I.EQ.1) PRINT *, ' I am a Fortran 90 program'
      IF (I.LT.0 .OR. I.GT.1) PRINT *, ' I am a HAL 9000 computer'
      END
```

(In the GNU Fortran and Fortran 90 languages, exclamation point is a valid character and, unlike space (⟨SPC⟩) or zero (‘`0`’), marks a line as a continuation line when it appears in column 6.)

11.7 Fortran 90

The GNU Fortran language includes a number of features that are part of Fortran 90, even when the ‘`-ff90`’ option is not specified. The features enabled by ‘`-ff90`’ are intended to be those that, when ‘`-ff90`’ is not specified, would have another meaning to `g77`—usually meaning something invalid in the GNU Fortran language.

So, the purpose of ‘`-ff90`’ is not to specify whether `g77` is to gratuitously reject Fortran 90 constructs. The ‘`-pedantic`’ option specified with ‘`-fno-f90`’ is intended to do that, although its implementation is certainly incomplete at this point.

When ‘`-ff90`’ is specified:

- The type of ‘`REAL(`*expr*`)`’ and ‘`AIMAG(`*expr*`)`’, where *expr* is `COMPLEX` type, is the same type as the real part of *expr*.

  For example, assuming ‘`Z`’ is type `COMPLEX(KIND=2)`, ‘`REAL(Z)`’ would return a value of type `REAL(KIND=2)`, not of type `REAL(KIND=1)`, since ‘`-ff90`’ is specified.

11.8 Pedantic Compilation

The ‘`-fpedantic`’ command-line option specifies that `g77` is to warn about code that is not standard-conforming. This is useful for finding some extensions `g77` accepts that other compilers might not accept. (Note that the ‘`-pedantic`’ and ‘`-pedantic-errors`’ options always imply ‘`-fpedantic`’.)

With ‘`-fno-f90`’ in force, ANSI FORTRAN 77 is used as the standard for conforming code. With ‘`-ff90`’ in force, Fortran 90 is used.

The constructs for which g77 issues diagnostics when '-fpedantic' and '-fno-f90' are in force are:

- Automatic arrays, as in

  ```
  SUBROUTINE X(N)
  REAL A(N)
  ...
  ```

  where 'A' is not listed in any ENTRY statement, and thus is not a dummy argument.

- The commas in 'READ (5), I' and 'WRITE (10), J'.

  These commas are disallowed by FORTRAN 77, but, while strictly superfluous, are syntactically elegant, especially given that commas are required in statements such as 'READ 99, I' and 'PRINT *, J'. Many compilers permit the superfluous commas for this reason.

- DOUBLE COMPLEX, either explicitly or implicitly.

  An explicit use of this type is via a DOUBLE COMPLEX or IMPLICIT DOUBLE COMPLEX statement, for examples.

  An example of an implicit use is the expression 'C*D', where 'C' is COMPLEX(KIND=1) and 'D' is DOUBLE PRECISION. This expression is prohibited by ANSI FORTRAN 77 because the rules of promotion would suggest that it produce a DOUBLE COMPLEX result—a type not provided for by that standard.

- Automatic conversion of numeric expressions to INTEGER(KIND=1) in contexts such as:
  - Array-reference indexes.
  - Alternate-return values.
  - Computed GOTO.
  - FORMAT run-time expressions (not yet supported).
  - Dimension lists in specification statements.
  - Numbers for I/O statements (such as 'READ (UNIT=3.2), I')
  - Sizes of CHARACTER entities in specification statements.
  - Kind types in specification entities (a Fortran 90 feature).
  - Initial, terminal, and incrementation parameters for implied-DO constructs in DATA statements.

- Automatic conversion of LOGICAL expressions to INTEGER in contexts such as arithmetic IF (where COMPLEX expressions are disallowed anyway).

- Zero-size array dimensions, as in:

  ```
  INTEGER I(10,20,4:2)
  ```

- Zero-length CHARACTER entities, as in:

  ```
  PRINT *, ''
  ```

- Substring operators applied to character constants and named constants, as in:

  ```
  PRINT *, 'hello'(3:5)
  ```

- Null arguments passed to statement function, as in:

  ```
  PRINT *, FOO(,3)
  ```

- Disagreement among program units regarding whether a given `COMMON` area is `SAVE`d (for targets where program units in a single source file are "glued" together as they typically are for UNIX development environments).
- Disagreement among program units regarding the size of a named `COMMON` block.
- Specification statements following first `DATA` statement.

  (In the GNU Fortran language, '`DATA I/1/`' may be followed by '`INTEGER J`', but not '`INTEGER I`'. The '`-fpedantic`' option disallows both of these.)
- Semicolon as statement separator, as in:

```
CALL FOO; CALL BAR
```

- Use of '`&`' in column 1 of fixed-form source (to indicate continuation).
- Use of `CHARACTER` constants to initialize numeric entities, and vice versa.
- Expressions having two arithmetic operators in a row, such as '`X*-Y`'.

If '`-fpedantic`' is specified along with '`-ff90`', the following constructs result in diagnostics:

- Use of semicolon as a statement separator on a line that has an `INCLUDE` directive.

11.9 Distensions

The '`-fugly-*`' command-line options determine whether certain features supported by VAX FORTRAN and other such compilers, but considered too ugly to be in code that can be changed to use safer and/or more portable constructs, are accepted. These are humorously referred to as "distensions", extensions that just plain look ugly in the harsh light of day.

Note: The '`-fugly`' option, which currently serves as shorthand to enable all of the distensions below, is likely to be removed in a future version of `g77`. That's because it's likely new distensions will be added that conflict with existing ones in terms of assigning meaning to a given chunk of code. (Also, it's pretty clear that users should not use '`-fugly`' as shorthand when the next release of `g77` might add a distension to that that causes their existing code, when recompiled, to behave differently—perhaps even fail to compile or run correctly.)

11.9.1 Implicit Argument Conversion

The '`-fno-ugly-args`' option disables passing typeless and Hollerith constants as actual arguments in procedure invocations. For example:

```
CALL FOO(4HABCD)
CALL BAR('123'O)
```

These constructs can be too easily used to create non-portable code, but are not considered as "ugly" as others. Further, they are widely used in existing Fortran source code in ways that often are quite portable. Therefore, they are enabled by default.

11.9.2 Ugly Assumed-Size Arrays

The '`-fugly-assumed`' option enables the treatment of any array with a final dimension specified as '`1`' as an assumed-size array, as if '`*`' had been specified instead.

For example, '`DIMENSION X(1)`' is treated as if it had read '`DIMENSION X(*)`' if '`X`' is listed as a dummy argument in a preceding `SUBROUTINE`, `FUNCTION`, or `ENTRY` statement in the same program unit.

Use an explicit lower bound to avoid this interpretation. For example, '`DIMENSION X(1:1)`' is never treated as if it had read '`DIMENSION X(*)`' or '`DIMENSION X(1:*)`'. Nor is '`DIMENSION X(2-1)`' affected by this option, since that kind of expression is unlikely to have been intended to designate an assumed-size array.

This option is used to prevent warnings being issued about apparent out-of-bounds reference such as '`X(2) = 99`'.

It also prevents the array from being used in contexts that disallow assumed-size arrays, such as '`PRINT *,X`'. In such cases, a diagnostic is generated and the source file is not compiled.

The construct affected by this option is used only in old code that pre-exists the widespread acceptance of adjustable and assumed-size arrays in the Fortran community.

Note: This option does not affect how '`DIMENSION X(1)`' is treated if '`X`' is listed as a dummy argument only *after* the `DIMENSION` statement (presumably in an `ENTRY` statement). For example, '`-fugly-assumed`' has no effect on the following program unit:

```
SUBROUTINE X
REAL A(1)
RETURN
ENTRY Y(A)
PRINT *, A
END
```

11.9.3 Ugly Complex Part Extraction

The '`-fugly-complex`' option enables use of the `REAL()` and `AIMAG()` intrinsics with arguments that are `COMPLEX` types other than `COMPLEX(KIND=1)`.

With '`-ff90`' in effect, these intrinsics return the unconverted real and imaginary parts (respectively) of their argument.

With '`-fno-f90`' in effect, these intrinsics convert the real and imaginary parts to `REAL(KIND=1)`, and return the result of that conversion.

Due to this ambiguity, the GNU Fortran language defines these constructs as invalid, except in the specific case where they are entirely and solely passed as an argument to an invocation of the `REAL()` intrinsic. For example,

```
REAL(REAL(Z))
```

is permitted even when '`Z`' is `COMPLEX(KIND=2)` and '`-fno-ugly-complex`' is in effect, because the meaning is clear.

`g77` enforces this restriction, unless '`-fugly-complex`' is specified, in which case the appropriate interpretation is chosen and no diagnostic is issued.

See Section 24.1 [CMPAMBIG], page 313, for information on how to cope with existing code with unclear expectations of `REAL()` and `AIMAG()` with `COMPLEX(KIND=2)` arguments.

See Section 10.11.9.212 [RealPart Intrinsic], page 155, for information on the `REALPART()` intrinsic, used to extract the real part of a complex expression without conversion. See Section 10.11.9.146 [ImagPart Intrinsic], page 137, for information on the `IMAGPART()` intrinsic, used to extract the imaginary part of a complex expression without conversion.

11.9.4 Ugly Null Arguments

The '`-fugly-comma`' option enables use of a single trailing comma to mean "pass an extra trailing null argument" in a list of actual arguments to an external procedure, and use of an empty list of arguments to such a procedure to mean "pass a single null argument".

(Null arguments often are used in some procedure-calling schemes to indicate omitted arguments.)

For example, '`CALL FOO(,)`' means "pass two null arguments", rather than "pass one null argument". Also, '`CALL BAR()`' means "pass one null argument".

This construct is considered "ugly" because it does not provide an elegant way to pass a single null argument that is syntactically distinct from passing no arguments. That is, this construct changes the meaning of code that makes no use of the construct.

So, with '`-fugly-comma`' in force, '`CALL FOO()`' and '`I = JFUNC()`' pass a single null argument, instead of passing no arguments as required by the Fortran 77 and 90 standards.

Note: Many systems gracefully allow the case where a procedure call passes one extra argument that the called procedure does not expect.

So, in practice, there might be no difference in the behavior of a program that does '`CALL FOO()`' or '`I = JFUNC()`' and is compiled with '`-fugly-comma`' in force as compared to its behavior when compiled with the default, '`-fno-ugly-comma`', in force, assuming '`FOO`' and '`JFUNC`' do not expect any arguments to be passed.

11.9.5 Ugly Conversion of Initializers

The constructs disabled by '`-fno-ugly-init`' are:

- Use of Hollerith and typeless constants in contexts where they set initial (compile-time) values for variables, arrays, and named constants—that is, `DATA` and `PARAMETER` statements, plus type-declaration statements specifying initial values.

  Here are some sample initializations that are disabled by the '`-fno-ugly-init`' option:

  ```
  PARAMETER (VAL='9A304FFE'X)
  REAL*8 STRING/8HOUTPUT00/
  DATA VAR/4HABCD/
  ```

- In the same contexts as above, use of character constants to initialize numeric items and vice versa (one constant per item).

  Here are more sample initializations that are disabled by the '`-fno-ugly-init`' option:

  ```
  INTEGER IA
  CHARACTER BELL
  PARAMETER (IA = 'A')
  PARAMETER (BELL = 7)
  ```

- Use of Hollerith and typeless constants on the right-hand side of assignment statements to numeric types, and in other contexts (such as passing arguments in invocations of intrinsic procedures and statement functions) that are treated as assignments to known types (the dummy arguments, in these cases).

  Here are sample statements that are disabled by the '`-fno-ugly-init`' option:

```
IVAR = 4HABCD
PRINT *, IMAX0(2HAB, 2HBA)
```

The above constructs, when used, can tend to result in non-portable code. But, they are widely used in existing Fortran code in ways that often are quite portable. Therefore, they are enabled by default.

11.9.6 Ugly Integer Conversions

The constructs enabled via '`-fugly-logint`' are:

- Automatic conversion between `INTEGER` and `LOGICAL` as dictated by context (typically implies nonportable dependencies on how a particular implementation encodes `.TRUE.` and `.FALSE.`).
- Use of a `LOGICAL` variable in `ASSIGN` and assigned-`GOTO` statements.

The above constructs are disabled by default because use of them tends to lead to non-portable code. Even existing Fortran code that uses that often turns out to be non-portable, if not outright buggy.

Some of this is due to differences among implementations as far as how `.TRUE.` and `.FALSE.` are encoded as `INTEGER` values—Fortran code that assumes a particular coding is likely to use one of the above constructs, and is also likely to not work correctly on implementations using different encodings.

See Section 18.5.5 [Equivalence Versus Equality], page 287, for more information.

11.9.7 Ugly Assigned Labels

The '`-fugly-assign`' option forces `g77` to use the same storage for assigned labels as it would for a normal assignment to the same variable.

For example, consider the following code fragment:

```
I = 3
ASSIGN 10 TO I
```

Normally, for portability and improved diagnostics, `g77` reserves distinct storage for a "sibling" of '`I`', used only for `ASSIGN` statements to that variable (along with the corresponding assigned-`GOTO` and assigned-'`FORMAT`'-I/O statements that reference the variable).

However, some code (that violates the ANSI FORTRAN 77 standard) attempts to copy assigned labels among variables involved with `ASSIGN` statements, as in:

```
ASSIGN 10 TO I
ISTATE(5) = I
...
J = ISTATE(ICUR)
GOTO J
```

Such code doesn't work under `g77` unless '`-fugly-assign`' is specified on the command-line, ensuring that the value of `I` referenced in the second line is whatever value `g77` uses to designate statement label '`10`', so the value may be copied into the '`ISTATE`' array, later retrieved into a variable of the appropriate type ('`J`'), and used as the target of an assigned-`GOTO` statement.

Note: To avoid subtle program bugs, when '`-fugly-assign`' is specified, `g77` requires the type of variables specified in assigned-label contexts *must* be the same type returned by `%LOC()`. On many systems, this type is effectively the same as `INTEGER(KIND=1)`, while, on others, it is effectively the same as `INTEGER(KIND=2)`.

Do *not* depend on `g77` actually writing valid pointers to these variables, however. While `g77` currently chooses that implementation, it might be changed in the future.

See Section 16.12 [Assigned Statement Labels (ASSIGN and GOTO)], page 249, for implementation details on assigned-statement labels.

12 The GNU Fortran Compiler

The GNU Fortran compiler, `g77`, supports programs written in the GNU Fortran language and in some other dialects of Fortran.

Some aspects of how `g77` works are universal regardless of dialect, and yet are not properly part of the GNU Fortran language itself. These are described below.

Note: This portion of the documentation definitely needs a lot of work!

12.1 Compiler Limits

`g77`, as with GNU tools in general, imposes few arbitrary restrictions on lengths of identifiers, number of continuation lines, number of external symbols in a program, and so on.

For example, some other Fortran compiler have an option (such as '`-Nlx`') to increase the limit on the number of continuation lines. Also, some Fortran compilation systems have an option (such as '`-Nxx`') to increase the limit on the number of external symbols.

`g77`, `gcc`, and GNU `ld` (the GNU linker) have no equivalent options, since they do not impose arbitrary limits in these areas.

`g77` does currently limit the number of dimensions in an array to the same degree as do the Fortran standards—seven (7). This restriction might well be lifted in a future version.

12.2 Compiler Types

Fortran implementations have a fair amount of freedom given them by the standard as far as how much storage space is used and how much precision and range is offered by the various types such as `LOGICAL(KIND=1)`, `INTEGER(KIND=1)`, `REAL(KIND=1)`, `REAL(KIND=2)`, `COMPLEX(KIND=1)`, and `CHARACTER`. Further, many compilers offer so-called '`*`*n*' notation, but the interpretation of *n* varies across compilers and target architectures.

The standard requires that `LOGICAL(KIND=1)`, `INTEGER(KIND=1)`, and `REAL(KIND=1)` occupy the same amount of storage space, and that `COMPLEX(KIND=1)` and `REAL(KIND=2)` take twice as much storage space as `REAL(KIND=1)`. Further, it requires that `COMPLEX(KIND=1)` entities be ordered such that when a `COMPLEX(KIND=1)` variable is storage-associated (such as via `EQUIVALENCE`) with a two-element `REAL(KIND=1)` array named '`R`', '`R(1)`' corresponds to the real element and '`R(2)`' to the imaginary element of the `COMPLEX(KIND=1)` variable.

(Few requirements as to precision or ranges of any of these are placed on the implementation, nor is the relationship of storage sizes of these types to the `CHARACTER` type specified, by the standard.)

`g77` follows the above requirements, warning when compiling a program requires placement of items in memory that contradict the requirements of the target architecture. (For example, a program can require placement of a `REAL(KIND=2)` on a boundary that is not an even multiple of its size, but still an even multiple of the size of a `REAL(KIND=1)` variable. On some target architectures, using the canonical mapping of Fortran types to underlying architectural types, such placement is prohibited by the machine definition or the Application Binary Interface (ABI) in force for the configuration defined for building `gcc` and `g77`. `g77` warns about such situations when it encounters them.)

`g77` follows consistent rules for configuring the mapping between Fortran types, including the '`*`*n*' notation, and the underlying architectural types as accessed by a similarly-configured applicable version of the `gcc` compiler. These rules offer a widely portable, consistent Fortran/C environment, although they might well conflict with the expectations of users of Fortran compilers designed and written for particular architectures.

These rules are based on the configuration that is in force for the version of `gcc` built in the same release as `g77` (and which was therefore used to build both the `g77` compiler components and the `libf2c` run-time library):

`REAL(KIND=1)`
: Same as `float` type.

`REAL(KIND=2)`
: Same as whatever floating-point type that is twice the size of a `float`—usually, this is a `double`.

`INTEGER(KIND=1)`
: Same as an integral type that is occupies the same amount of memory storage as `float`—usually, this is either an `int` or a `long int`.

`LOGICAL(KIND=1)`
: Same `gcc` type as `INTEGER(KIND=1)`.

`INTEGER(KIND=2)`
: Twice the size, and usually nearly twice the range, as `INTEGER(KIND=1)`—usually, this is either a `long int` or a `long long int`.

`LOGICAL(KIND=2)`
: Same `gcc` type as `INTEGER(KIND=2)`.

`INTEGER(KIND=3)`
: Same `gcc` type as signed `char`.

`LOGICAL(KIND=3)`
: Same `gcc` type as `INTEGER(KIND=3)`.

`INTEGER(KIND=6)`
: Twice the size, and usually nearly twice the range, as `INTEGER(KIND=3)`—usually, this is a `short`.

`LOGICAL(KIND=6)`
: Same `gcc` type as `INTEGER(KIND=6)`.

`COMPLEX(KIND=1)`
: Two `REAL(KIND=1)` scalars (one for the real part followed by one for the imaginary part).

`COMPLEX(KIND=2)`
: Two `REAL(KIND=2)` scalars.

numeric-type`*`*n*
: (Where *numeric-type* is any type other than `CHARACTER`.) Same as whatever `gcc` type occupies *n* times the storage space of a `gcc char` item.

`DOUBLE PRECISION`
: Same as `REAL(KIND=2)`.

`DOUBLE COMPLEX`
: Same as `COMPLEX(KIND=2)`.

Note that the above are proposed correspondences and might change in future versions of `g77`—avoid writing code depending on them.

Other types supported by `g77` are derived from gcc types such as `char`, `short`, `int`, `long int`, `long long int`, `long double`, and so on. That is, whatever types `gcc` already supports, `g77` supports now or probably will support in a future version. The rules for the '*numeric-type*`*`*n*' notation apply to these types, and new values for '*numeric-type*`(KIND=`*n*`)`' will be assigned in a way that encourages clarity, consistency, and portability.

12.3 Compiler Constants

`g77` strictly assigns types to *all* constants not documented as "typeless" (typeless constants including '`'1'Z`', for example). Many other Fortran compilers attempt to assign types to typed constants based on their context. This results in hard-to-find bugs, nonportable code, and is not in the spirit (though it strictly follows the letter) of the 77 and 90 standards.

`g77` might offer, in a future release, explicit constructs by which a wider variety of typeless constants may be specified, and/or user-requested warnings indicating places where `g77` might differ from how other compilers assign types to constants.

See Section 18.5.4 [Context-Sensitive Constants], page 286, for more information on this issue.

12.4 Compiler Intrinsics

`g77` offers an ever-widening set of intrinsics. Currently these all are procedures (functions and subroutines).

Some of these intrinsics are unimplemented, but their names reserved to reduce future problems with existing code as they are implemented. Others are implemented as part of the GNU Fortran language, while yet others are provided for compatibility with other dialects of Fortran but are not part of the GNU Fortran language.

To manage these distinctions, `g77` provides intrinsic *groups*, a facility that is simply an extension of the intrinsic groups provided by the GNU Fortran language.

12.4.1 Intrinsic Groups

A given specific intrinsic belongs in one or more groups. Each group is deleted, disabled, hidden, or enabled by default or a command-line option. The meaning of each term follows.

Deleted No intrinsics are recognized as belonging to that group.

Disabled Intrinsics are recognized as belonging to the group, but references to them (other than via the `INTRINSIC` statement) are disallowed through that group.

Hidden Intrinsics in that group are recognized and enabled (if implemented) *only* if the first mention of the actual name of an intrinsic in a program unit is in an `INTRINSIC` statement.

Enabled Intrinsics in that group are recognized and enabled (if implemented).

The distinction between deleting and disabling a group is illustrated by the following example. Assume intrinsic '`FOO`' belongs only to group '`FGR`'. If group '`FGR`' is deleted, the following program unit will successfully compile, because '`FOO()`' will be seen as a reference to an external function named '`FOO`':

```
PRINT *, FOO()
END
```

If group '`FGR`' is disabled, compiling the above program will produce diagnostics, either because the '`FOO`' intrinsic is improperly invoked or, if properly invoked, it is not enabled. To change the above program so it references an external function '`FOO`' instead of the disabled '`FOO`' intrinsic, add the following line to the top:

```
EXTERNAL FOO
```

So, deleting a group tells `g77` to pretend as though the intrinsics in that group do not exist at all, whereas disabling it tells `g77` to recognize them as (disabled) intrinsics in intrinsic-like contexts.

Hiding a group is like enabling it, but the intrinsic must be first named in an `INTRINSIC` statement to be considered a reference to the intrinsic rather than to an external procedure. This might be the "safest" way to treat a new group of intrinsics when compiling old code, because it allows the old code to be generally written as if those new intrinsics never existed, but to be changed to use them by inserting `INTRINSIC` statements in the appropriate places. However, it should be the goal of development to use `EXTERNAL` for all names of external procedures that might be intrinsic names.

If an intrinsic is in more than one group, it is enabled if any of its containing groups are enabled; if not so enabled, it is hidden if any of its containing groups are hidden; if not so hidden, it is disabled if any of its containing groups are disabled; if not so disabled, it is deleted. This extra complication is necessary because some intrinsics, such as `IBITS`, belong to more than one group, and hence should be enabled if any of the groups to which they belong are enabled, and so on.

The groups are:

`badu77` UNIX intrinsics having inappropriate forms (usually functions that have intended side effects).

`gnu` Intrinsics the GNU Fortran language supports that are extensions to the Fortran standards (77 and 90).

`f2c` Intrinsics supported by AT&T's `f2c` converter and/or `libf2c`.

`f90` Fortran 90 intrinsics.

`mil` MIL-STD 1753 intrinsics (`MVBITS`, `IAND`, `BTEST`, and so on).

`unix` UNIX intrinsics (`IARGC`, `EXIT`, `ERF`, and so on).

`vxt` VAX/VMS FORTRAN (current as of v4) intrinsics.

12.4.2 Other Intrinsics

`g77` supports intrinsics other than those in the GNU Fortran language proper. This set of intrinsics is described below.

12.4.2.1 ACosD Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL ACosD`' to use this name for an external procedure.

12.4.2.2 AIMax0 Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL AIMax0`' to use this name for an external procedure.

12.4.2.3 AIMin0 Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL AIMin0`' to use this name for an external procedure.

12.4.2.4 AJMax0 Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL AJMax0`' to use this name for an external procedure.

12.4.2.5 AJMin0 Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL AJMin0`' to use this name for an external procedure.

12.4.2.6 ASinD Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL ASinD`' to use this name for an external procedure.

12.4.2.7 ATan2D Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL ATan2D`' to use this name for an external procedure.

12.4.2.8 ATanD Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL ATanD`' to use this name for an external procedure.

12.4.2.9 BITest Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL BITest`' to use this name for an external procedure.

12.4.2.10 BJTest Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL BJTest`' to use this name for an external procedure.

12.4.2.11 CDAbs Intrinsic

`CDAbs(`*A*`)`

CDAbs: `REAL(KIND=2)` function.

A: `COMPLEX(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: `f2c`, `vxt`.

Description:

Archaic form of `ABS()` that is specific to one type for *A*. See Section 10.11.9.2 [Abs Intrinsic], page 98.

12.4.2.12 CDCos Intrinsic

`CDCos(`*X*`)`

CDCos: `COMPLEX(KIND=2)` function.

X: `COMPLEX(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: `f2c`, `vxt`.

Description:

Archaic form of `COS()` that is specific to one type for *X*. See Section 10.11.9.46 [Cos Intrinsic], page 110.

12.4.2.13 CDExp Intrinsic

`CDExp(`*X*`)`

CDExp: `COMPLEX(KIND=2)` function.

X: `COMPLEX(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: `f2c`, `vxt`.

Description:

Archaic form of `EXP()` that is specific to one type for *X*. See Section 10.11.9.99 [Exp Intrinsic], page 123.

12.4.2.14 CDLog Intrinsic

`CDLog(`*X*`)`

CDLog: `COMPLEX(KIND=2)` function.

X: `COMPLEX(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: `f2c`, `vxt`.

Description:

Archaic form of `LOG()` that is specific to one type for *X*. See Section 10.11.9.170 [Log Intrinsic], page 145.

12.4.2.15 CDSin Intrinsic

`CDSin(`*X*`)`

CDSin: `COMPLEX(KIND=2)` function.

X: `COMPLEX(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: `f2c`, `vxt`.

Description:

Archaic form of `SIN()` that is specific to one type for *X*. See Section 10.11.9.229 [Sin Intrinsic], page 159.

12.4.2.16 CDSqRt Intrinsic

`CDSqRt(`*X*`)`

CDSqRt: `COMPLEX(KIND=2)` function.

X: `COMPLEX(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: `f2c`, `vxt`.

Description:

Archaic form of `SQRT()` that is specific to one type for *X*. See Section 10.11.9.235 [SqRt Intrinsic], page 160.

12.4.2.17 ChDir Intrinsic (function)

`ChDir(`*Dir*`)`

ChDir: `INTEGER(KIND=1)` function.

Dir: `CHARACTER`; scalar; INTENT(IN).

Intrinsic groups: `badu77`.

Description:

Sets the current working directory to be *Dir*. Returns 0 on success or a non-zero error code. See `chdir(3)`.

Caution: Using this routine during I/O to a unit connected with a non-absolute file name can cause subsequent I/O on such a unit to fail because the I/O library may reopen files by name.

Due to the side effects performed by this intrinsic, the function form is not recommended.

For information on other intrinsics with the same name: See Section 10.11.9.40 [ChDir Intrinsic (subroutine)], page 108.

12.4.2.18 ChMod Intrinsic (function)

`ChMod(`*Name*`, `*Mode*`)`

ChMod: `INTEGER(KIND=1)` function.

Name: `CHARACTER`; scalar; INTENT(IN).

Mode: `CHARACTER`; scalar; INTENT(IN).

Intrinsic groups: `badu77`.

Description:

Changes the access mode of file *Name* according to the specification *Mode*, which is given in the format of `chmod(1)`. A null character ('`CHAR(0)`') marks the end of the name in *Name*—otherwise, trailing blanks in *Name* are ignored. Currently, *Name* must not contain the single quote character.

Returns 0 on success or a non-zero error code otherwise.

Note that this currently works by actually invoking `/bin/chmod` (or the `chmod` found when the library was configured) and so may fail in some circumstances and will, anyway, be slow.

Due to the side effects performed by this intrinsic, the function form is not recommended.

For information on other intrinsics with the same name: See Section 10.11.9.41 [ChMod Intrinsic (subroutine)], page 109.

12.4.2.19 CosD Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL CosD`' to use this name for an external procedure.

12.4.2.20 DACosD Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL DACosD`' to use this name for an external procedure.

12.4.2.21 DASinD Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL DASinD`' to use this name for an external procedure.

12.4.2.22 DATan2D Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL DATan2D`' to use this name for an external procedure.

12.4.2.23 DATanD Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL DATanD`' to use this name for an external procedure.

12.4.2.24 Date Intrinsic

```
CALL Date(Date)
```

Date: `CHARACTER`; scalar; INTENT(OUT).

Intrinsic groups: `vxt`.

Description:

Returns *Date* in the form '*dd-mmm-yy*', representing the numeric day of the month *dd*, a three-character abbreviation of the month name *mmm* and the last two digits of the year *yy*, e.g. '`25-Nov-96`'.

This intrinsic is not recommended, due to the year 2000 approaching. See Section 10.11.9.53 [CTime Intrinsic (subroutine)], page 112, for information on obtaining more digits for the current (or any) date.

12.4.2.25 DbleQ Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL DbleQ`' to use this name for an external procedure.

12.4.2.26 DCmplx Intrinsic

`DCmplx(`*X*`, `*Y*`)`

DCmplx: `COMPLEX(KIND=2)` function.

X: `INTEGER`, `REAL`, or `COMPLEX`; scalar; INTENT(IN).

Y: `INTEGER` or `REAL`; OPTIONAL (must be omitted if *X* is `COMPLEX`); scalar; INTENT(IN).

Intrinsic groups: `f2c`, `vxt`.

Description:

If *X* is not type `COMPLEX`, constructs a value of type `COMPLEX(KIND=2)` from the real and imaginary values specified by *X* and *Y*, respectively. If *Y* is omitted, '`0D0`' is assumed.

If *X* is type `COMPLEX`, converts it to type `COMPLEX(KIND=2)`.

Although this intrinsic is not standard Fortran, it is a popular extension offered by many compilers that support `DOUBLE COMPLEX`, since it offers the easiest way to convert to `DOUBLE COMPLEX` without using Fortran 90 features (such as the '`KIND=`' argument to the `CMPLX()` intrinsic).

('`CMPLX(0D0, 0D0)`' returns a single-precision `COMPLEX` result, as required by standard FORTRAN 77. That's why so many compilers provide `DCMPLX()`, since '`DCMPLX(0D0, 0D0)`' returns a `DOUBLE COMPLEX` result. Still, `DCMPLX()` converts even `REAL*16` arguments to their `REAL*8` equivalents in most dialects of Fortran, so neither it nor `CMPLX()` allow easy construction of arbitrary-precision values without potentially forcing a conversion involving extending or reducing precision. GNU Fortran provides such an intrinsic, called `COMPLEX()`.)

See Section 10.11.9.44 [Complex Intrinsic], page 110, for information on easily constructing a `COMPLEX` value of arbitrary precision from `REAL` arguments.

12.4.2.27 DConjg Intrinsic

`DConjg(`*Z*`)`

DConjg: `COMPLEX(KIND=2)` function.

Z: `COMPLEX(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: `f2c`, `vxt`.

Description:

Archaic form of `CONJG()` that is specific to one type for *Z*. See Section 10.11.9.45 [Conjg Intrinsic], page 110.

12.4.2.28 DCosD Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL DCosD`' to use this name for an external procedure.

12.4.2.29 DFloat Intrinsic

```
DFloat(A)
```

DFloat: `REAL(KIND=2)` function.

A: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: `f2c`, `vxt`.

Description:

Archaic form of `REAL()` that is specific to one type for *A*. See Section 10.11.9.211 [Real Intrinsic], page 154.

12.4.2.30 DFlotI Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL DFlotI`' to use this name for an external procedure.

12.4.2.31 DFlotJ Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL DFlotJ`' to use this name for an external procedure.

12.4.2.32 DImag Intrinsic

```
DImag(Z)
```

DImag: `REAL(KIND=2)` function.

Z: `COMPLEX(KIND=2)`; scalar; INTENT(IN).

Intrinsic groups: `f2c`, `vxt`.

Description:

Archaic form of `AIMAG()` that is specific to one type for *Z*. See Section 10.11.9.8 [AImag Intrinsic], page 100.

12.4.2.33 DReal Intrinsic

```
DReal(A)
```

DReal: `REAL(KIND=2)` function.

A: `INTEGER`, `REAL`, or `COMPLEX`; scalar; INTENT(IN).

Intrinsic groups: `vxt`.

Description:

Converts *A* to `REAL(KIND=2)`.

If *A* is type `COMPLEX`, its real part is converted (if necessary) to `REAL(KIND=2)`, and its imaginary part is disregarded.

Although this intrinsic is not standard Fortran, it is a popular extension offered by many compilers that support `DOUBLE COMPLEX`, since it offers the easiest way to extract the real part of a `DOUBLE COMPLEX` value without using the Fortran 90 `REAL()` intrinsic in a way that produces a return value inconsistent with the way many FORTRAN 77 compilers handle `REAL()` of a `DOUBLE COMPLEX` value.

See Section 10.11.9.212 [RealPart Intrinsic], page 155, for information on a GNU Fortran intrinsic that avoids these areas of confusion.

See Section 10.11.9.67 [Dble Intrinsic], page 115, for information on the standard FORTRAN 77 replacement for `DREAL()`.

See Section 10.11.5 [REAL() and AIMAG() of Complex], page 96, for more information on this issue.

12.4.2.34 DSinD Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL DSinD`' to use this name for an external procedure.

12.4.2.35 DTanD Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL DTanD`' to use this name for an external procedure.

12.4.2.36 Dtime Intrinsic (function)

`Dtime(`*TArray*`)`

Dtime: `REAL(KIND=1)` function.

TArray: `REAL(KIND=1)`; DIMENSION(2); INTENT(OUT).

Intrinsic groups: `badu77`.

Description:

Initially, return the number of seconds of runtime since the start of the process's execution as the function value, and the user and system components of this in '*TArray*`(1)`' and '*TArray*`(2)`' respectively. The functions' value is equal to '*TArray*`(1) +` *TArray*`(2)`'.

Subsequent invocations of '`DTIME()`' return values accumulated since the previous invocation.

Due to the side effects performed by this intrinsic, the function form is not recommended.

For information on other intrinsics with the same name: See Section 10.11.9.91 [Dtime Intrinsic (subroutine)], page 121.

12.4.2.37 FGet Intrinsic (function)

`FGet(`*C*`)`

FGet: `INTEGER(KIND=1)` function.

C: `CHARACTER`; scalar; INTENT(OUT).

Intrinsic groups: `badu77`.

Description:

Reads a single character into *C* in stream mode from unit 5 (by-passing normal formatted input) using `getc(3)`. Returns 0 on success, −1 on end-of-file, and the error code from `ferror(3)` otherwise.

Stream I/O should not be mixed with normal record-oriented (formatted or unformatted) I/O on the same unit; the results are unpredictable.

For information on other intrinsics with the same name: See Section 10.11.9.103 [FGet Intrinsic (subroutine)], page 124.

12.4.2.38 FGetC Intrinsic (function)

`FGetC(`*Unit*`, `*C*`)`

FGetC: `INTEGER(KIND=1)` function.

Unit: `INTEGER`; scalar; INTENT(IN).

C: `CHARACTER`; scalar; INTENT(OUT).

Intrinsic groups: `badu77`.

Description:

Reads a single character into *C* in stream mode from unit *Unit* (by-passing normal formatted output) using `getc(3)`. Returns 0 on success, −1 on end-of-file, and the error code from `ferror(3)` otherwise.

Stream I/O should not be mixed with normal record-oriented (formatted or unformatted) I/O on the same unit; the results are unpredictable.

For information on other intrinsics with the same name: See Section 10.11.9.104 [FGetC Intrinsic (subroutine)], page 125.

12.4.2.39 FloatI Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL FloatI`' to use this name for an external procedure.

12.4.2.40 FloatJ Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL FloatJ`' to use this name for an external procedure.

12.4.2.41 FPut Intrinsic (function)

`FPut(`*C*`)`

FPut: `INTEGER(KIND=1)` function.

C: `CHARACTER`; scalar; INTENT(IN).

Intrinsic groups: `badu77`.

Description:

Writes the single character *C* in stream mode to unit 6 (by-passing normal formatted output) using `getc(3)`. Returns 0 on success, the error code from `ferror(3)` otherwise.

Stream I/O should not be mixed with normal record-oriented (formatted or unformatted) I/O on the same unit; the results are unpredictable.

For information on other intrinsics with the same name: See Section 10.11.9.109 [FPut Intrinsic (subroutine)], page 126.

12.4.2.42 FPutC Intrinsic (function)

`FPutC(`*Unit*`, `*C*`)`

FPutC: `INTEGER(KIND=1)` function.

Unit: `INTEGER`; scalar; INTENT(IN).

C: `CHARACTER`; scalar; INTENT(IN).

Intrinsic groups: `badu77`.

Description:

Writes the single character *C* in stream mode to unit *Unit* (by-passing normal formatted output) using `putc(3)`. Returns 0 on success, the error code from `ferror(3)` otherwise.

Stream I/O should not be mixed with normal record-oriented (formatted or unformatted) I/O on the same unit; the results are unpredictable.

For information on other intrinsics with the same name: See Section 10.11.9.110 [FPutC Intrinsic (subroutine)], page 126.

12.4.2.43 IDate Intrinsic (VXT)

`CALL IDate(`*M*`, `*D*`, `*Y*`)`

M: `INTEGER(KIND=1)`; scalar; INTENT(OUT).

D: `INTEGER(KIND=1)`; scalar; INTENT(OUT).

Y: `INTEGER(KIND=1)`; scalar; INTENT(OUT).

Intrinsic groups: `vxt`.

Description:

Returns the numerical values of the current local time. The month (in the range 1–12) is returned in *M*, the day (in the range 1–7) in *D*, and the year in *Y* (in the range 0–99).

This intrinsic is not recommended, due to the year 2000 approaching.

For information on other intrinsics with the same name: See Section 10.11.9.138 [IDate Intrinsic (UNIX)], page 135.

12.4.2.44 IIAbs Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL IIAbs`' to use this name for an external procedure.

12.4.2.45 IIAnd Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL IIAnd`' to use this name for an external procedure.

12.4.2.46 IIBClr Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL IIBClr`' to use this name for an external procedure.

12.4.2.47 IIBits Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL IIBits`' to use this name for an external procedure.

12.4.2.48 IIBSet Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL IIBSet`' to use this name for an external procedure.

12.4.2.49 IIDiM Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL IIDiM`' to use this name for an external procedure.

12.4.2.50 IIDInt Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL IIDInt`' to use this name for an external procedure.

12.4.2.51 IIDNnt Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL IIDNnt`' to use this name for an external procedure.

12.4.2.52 IIEOr Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL IIEOr`' to use this name for an external procedure.

12.4.2.53 IIFix Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL IIFix`' to use this name for an external procedure.

12.4.2.54 IInt Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL IInt`' to use this name for an external procedure.

12.4.2.55 IIOr Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL IIOr`' to use this name for an external procedure.

12.4.2.56 IIQint Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL IIQint`' to use this name for an external procedure.

12.4.2.57 IIQNnt Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL IIQNnt`' to use this name for an external procedure.

12.4.2.58 IIShftC Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL IIShftC`' to use this name for an external procedure.

12.4.2.59 IISign Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL IISign`' to use this name for an external procedure.

12.4.2.60 IMax0 Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL IMax0`' to use this name for an external procedure.

12.4.2.61 IMax1 Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL IMax1`' to use this name for an external procedure.

12.4.2.62 IMin0 Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL IMin0`' to use this name for an external procedure.

12.4.2.63 IMin1 Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL IMin1`' to use this name for an external procedure.

12.4.2.64 IMod Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL IMod`' to use this name for an external procedure.

12.4.2.65 ININt Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL ININt`' to use this name for an external procedure.

12.4.2.66 INot Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL INot`' to use this name for an external procedure.

12.4.2.67 IZExt Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL IZExt`' to use this name for an external procedure.

12.4.2.68 JIAbs Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL JIAbs`' to use this name for an external procedure.

12.4.2.69 JIAnd Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL JIAnd`' to use this name for an external procedure.

12.4.2.70 JIBClr Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL JIBClr`' to use this name for an external procedure.

12.4.2.71 JIBits Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL JIBits`' to use this name for an external procedure.

12.4.2.72 JIBSet Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL JIBSet`' to use this name for an external procedure.

12.4.2.73 JIDiM Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL JIDiM`' to use this name for an external procedure.

12.4.2.74 JIDInt Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL JIDInt`' to use this name for an external procedure.

12.4.2.75 JIDNnt Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL JIDNnt`' to use this name for an external procedure.

12.4.2.76 JIEOr Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL JIEOr`' to use this name for an external procedure.

12.4.2.77 JIFix Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL JIFix`' to use this name for an external procedure.

12.4.2.78 JInt Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL JInt`' to use this name for an external procedure.

12.4.2.79 JIOr Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL JIOr`' to use this name for an external procedure.

12.4.2.80 JIQint Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL JIQint`' to use this name for an external procedure.

12.4.2.81 JIQNnt Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL JIQNnt`' to use this name for an external procedure.

12.4.2.82 JIShft Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL JIShft`' to use this name for an external procedure.

12.4.2.83 JIShftC Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL JIShftC`' to use this name for an external procedure.

12.4.2.84 JISign Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL JISign`' to use this name for an external procedure.

12.4.2.85 JMax0 Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL JMax0`' to use this name for an external procedure.

12.4.2.86 JMax1 Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL JMax1`' to use this name for an external procedure.

12.4.2.87 JMin0 Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL JMin0`' to use this name for an external procedure.

12.4.2.88 JMin1 Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL JMin1`' to use this name for an external procedure.

12.4.2.89 JMod Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL JMod`' to use this name for an external procedure.

12.4.2.90 JNInt Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL JNInt`' to use this name for an external procedure.

12.4.2.91 JNot Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL JNot`' to use this name for an external procedure.

12.4.2.92 JZExt Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL JZExt`' to use this name for an external procedure.

12.4.2.93 Kill Intrinsic (function)

`Kill(`*Pid*`, `*Signal*`)`

Kill: `INTEGER(KIND=1)` function.

Pid: `INTEGER`; scalar; INTENT(IN).

Signal: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: `badu77`.

Description:

Sends the signal specified by *Signal* to the process *Pid*. Returns 0 on success or a non-zero error code. See `kill(2)`.

Due to the side effects performed by this intrinsic, the function form is not recommended.

For information on other intrinsics with the same name: See Section 10.11.9.158 [Kill Intrinsic (subroutine)], page 141.

12.4.2.94 Link Intrinsic (function)

`Link(`*Path1*`, `*Path2*`)`

Link: `INTEGER(KIND=1)` function.

Path1: `CHARACTER`; scalar; INTENT(IN).

Path2: `CHARACTER`; scalar; INTENT(IN).

Intrinsic groups: `badu77`.

Description:

Makes a (hard) link from file *Path1* to *Path2*. A null character ('`CHAR(0)`') marks the end of the names in *Path1* and *Path2*—otherwise, trailing blanks in *Path1* and *Path2* are ignored. Returns 0 on success or a non-zero error code. See `link(2)`.

Due to the side effects performed by this intrinsic, the function form is not recommended.

For information on other intrinsics with the same name: See Section 10.11.9.165 [Link Intrinsic (subroutine)], page 143.

12.4.2.95 QAbs Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QAbs`' to use this name for an external procedure.

12.4.2.96 QACos Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QACos`' to use this name for an external procedure.

12.4.2.97 QACosD Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QACosD`' to use this name for an external procedure.

12.4.2.98 QASin Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QASin`' to use this name for an external procedure.

12.4.2.99 QASinD Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QASinD`' to use this name for an external procedure.

12.4.2.100 QATan Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QATan`' to use this name for an external procedure.

12.4.2.101 QATan2 Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QATan2`' to use this name for an external procedure.

12.4.2.102 QATan2D Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QATan2D`' to use this name for an external procedure.

12.4.2.103 QATanD Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QATanD`' to use this name for an external procedure.

12.4.2.104 QCos Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QCos`' to use this name for an external procedure.

12.4.2.105 QCosD Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QCosD`' to use this name for an external procedure.

12.4.2.106 QCosH Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QCosH`' to use this name for an external procedure.

12.4.2.107 QDiM Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QDiM`' to use this name for an external procedure.

12.4.2.108 QExp Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QExp`' to use this name for an external procedure.

12.4.2.109 QExt Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QExt`' to use this name for an external procedure.

12.4.2.110 QExtD Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QExtD`' to use this name for an external procedure.

12.4.2.111 QFloat Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QFloat`' to use this name for an external procedure.

12.4.2.112 QInt Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QInt`' to use this name for an external procedure.

12.4.2.113 QLog Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QLog`' to use this name for an external procedure.

12.4.2.114 QLog10 Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QLog10`' to use this name for an external procedure.

12.4.2.115 QMax1 Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QMax1`' to use this name for an external procedure.

12.4.2.116 QMin1 Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QMin1`' to use this name for an external procedure.

12.4.2.117 QMod Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QMod`' to use this name for an external procedure.

12.4.2.118 QNInt Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QNInt`' to use this name for an external procedure.

12.4.2.119 QSin Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QSin`' to use this name for an external procedure.

12.4.2.120 QSinD Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QSinD`' to use this name for an external procedure.

12.4.2.121 QSinH Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QSinH`' to use this name for an external procedure.

12.4.2.122 QSqRt Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QSqRt`' to use this name for an external procedure.

12.4.2.123 QTan Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QTan`' to use this name for an external procedure.

12.4.2.124 QTanD Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QTanD`' to use this name for an external procedure.

12.4.2.125 QTanH Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL QTanH`' to use this name for an external procedure.

12.4.2.126 Rename Intrinsic (function)

`Rename(`*Path1*`, `*Path2*`)`

Rename: `INTEGER(KIND=1)` function.

Path1: `CHARACTER`; scalar; INTENT(IN).

Path2: `CHARACTER`; scalar; INTENT(IN).

Intrinsic groups: `badu77`.

Description:

Renames the file *Path1* to *Path2*. A null character ('`CHAR(0)`') marks the end of the names in *Path1* and *Path2*—otherwise, trailing blanks in *Path1* and *Path2* are ignored. See `rename(2)`. Returns 0 on success or a non-zero error code.

Due to the side effects performed by this intrinsic, the function form is not recommended.

For information on other intrinsics with the same name: See Section 10.11.9.213 [Rename Intrinsic (subroutine)], page 155.

12.4.2.127 Secnds Intrinsic

`Secnds(`*T*`)`

Secnds: `REAL(KIND=1)` function.

T: `REAL(KIND=1)`; scalar; INTENT(IN).

Intrinsic groups: `vxt`.

Description:

Returns the local time in seconds since midnight minus the value *T*.

12.4.2.128 Signal Intrinsic (function)

`Signal(`*Number*`, `*Handler*`)`

Signal: `INTEGER(KIND=7)` function.

Number: `INTEGER`; scalar; INTENT(IN).

Handler: Signal handler (`INTEGER FUNCTION` or `SUBROUTINE`) or dummy/global `INTEGER(KIND=1)` scalar.

Intrinsic groups: `badu77`.

Description:

If *Handler* is a an `EXTERNAL` routine, arranges for it to be invoked with a single integer argument (of system-dependent length) 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)`.

Note that *Handler* will be called using C conventions, so the value of its argument in Fortran terms is obtained by applying `%LOC()` (or *LOC()*) to it.

The value returned by `signal(2)` is returned.

Due to the side effects performed by this intrinsic, the function form is not recommended.

Warning: If the returned value is stored in an `INTEGER(KIND=1)` (default `INTEGER`) argument, truncation of the original return value occurs on some systems (such as Alphas, which have 64-bit pointers but 32-bit default integers), with no warning issued by `g77` under normal circumstances.

Therefore, the following code fragment might silently fail on some systems:

```
INTEGER RTN
EXTERNAL MYHNDL
RTN = SIGNAL(signum, MYHNDL)
```

```
...
! Restore original handler:
RTN = SIGNAL(signum, RTN)
```

The reason for the failure is that 'RTN' might not hold all the information on the original handler for the signal, thus restoring an invalid handler. This bug could manifest itself as a spurious run-time failure at an arbitrary point later during the program's execution, for example.

Warning: Use of the libf2c run-time library function 'signal_' directly (such as via 'EXTERNAL SIGNAL') requires use of the %VAL() construct to pass an INTEGER value (such as 'SIG_IGN' or 'SIG_DFL') for the *Handler* argument.

However, while 'RTN = SIGNAL(*signum*, %VAL(SIG_IGN))' works when 'SIGNAL' is treated as an external procedure (and resolves, at link time, to libf2c's 'signal_' routine), this construct is not valid when 'SIGNAL' is recognized as the intrinsic of that name.

Therefore, for maximum portability and reliability, code such references to the 'SIGNAL' facility as follows:

```
INTRINSIC SIGNAL
...
RTN = SIGNAL(signum, SIG_IGN)
```

g77 will compile such a call correctly, while other compilers will generally either do so as well or reject the 'INTRINSIC SIGNAL' statement via a diagnostic, allowing you to take appropriate action.

For information on other intrinsics with the same name: See Section 10.11.9.228 [Signal Intrinsic (subroutine)], page 158.

12.4.2.129 SinD Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use 'EXTERNAL SinD' to use this name for an external procedure.

12.4.2.130 SnglQ Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use 'EXTERNAL SnglQ' to use this name for an external procedure.

12.4.2.131 SymLnk Intrinsic (function)

```
SymLnk(Path1, Path2)
```

SymLnk: INTEGER(KIND=1) function.

Path1: CHARACTER; scalar; INTENT(IN).

Path2: CHARACTER; scalar; INTENT(IN).

Intrinsic groups: badu77.

Description:

Makes a symbolic link from file *Path1* to *Path2*. A null character ('CHAR(0)') marks the end of the names in *Path1* and *Path2*—otherwise, trailing blanks in *Path1* and *Path2*

are ignored. Returns 0 on success or a non-zero error code (`ENOSYS` if the system does not provide `symlink(2)`).

Due to the side effects performed by this intrinsic, the function form is not recommended.

For information on other intrinsics with the same name: See Section 10.11.9.240 [SymLnk Intrinsic (subroutine)], page 162.

12.4.2.132 System Intrinsic (function)

`System(`*Command*`)`

System: `INTEGER(KIND=1)` function.

Command: `CHARACTER`; scalar; INTENT(IN).

Intrinsic groups: `badu77`.

Description:

Passes the command *Command* to a shell (see `system(3)`). Returns the value returned by `system(3)`, presumably 0 if the shell command succeeded. Note that which shell is used to invoke the command is system-dependent and environment-dependent.

Due to the side effects performed by this intrinsic, the function form is not recommended. However, the function form can be valid in cases where the actual side effects performed by the call are unimportant to the application.

For example, on a UNIX system, '`SAME = SYSTEM('cmp a b')`' does not perform any side effects likely to be important to the program, so the programmer would not care if the actual system call (and invocation of `cmp`) was optimized away in a situation where the return value could be determined otherwise, or was not actually needed ('`SAME`' not actually referenced after the sample assignment statement).

For information on other intrinsics with the same name: See Section 10.11.9.241 [System Intrinsic (subroutine)], page 163.

12.4.2.133 TanD Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL TanD`' to use this name for an external procedure.

12.4.2.134 Time Intrinsic (VXT)

`CALL Time(`*Time*`)`

Time: `CHARACTER*8`; scalar; INTENT(OUT).

Intrinsic groups: `vxt`.

Description:

Returns in *Time* a character representation of the current time as obtained from `ctime(3)`.

See Section 10.11.9.101 [Fdate Intrinsic (subroutine)], page 123 for an equivalent routine.

For information on other intrinsics with the same name: See Section 10.11.9.245 [Time Intrinsic (UNIX)], page 164.

12.4.2.135 UMask Intrinsic (function)

`UMask(`*Mask*`)`

UMask: `INTEGER(KIND=1)` function.

Mask: `INTEGER`; scalar; INTENT(IN).

Intrinsic groups: `badu77`.

Description:

Sets the file creation mask to *Mask* and returns the old value. See `umask(2)`.

Due to the side effects performed by this intrinsic, the function form is not recommended.

For information on other intrinsics with the same name: See Section 10.11.9.254 [UMask Intrinsic (subroutine)], page 166.

12.4.2.136 Unlink Intrinsic (function)

`Unlink(`*File*`)`

Unlink: `INTEGER(KIND=1)` function.

File: `CHARACTER`; scalar; INTENT(IN).

Intrinsic groups: `badu77`.

Description:

Unlink the file *File*. A null character ('`CHAR(0)`') marks the end of the name in *File*—otherwise, trailing blanks in *File* are ignored. Returns 0 on success or a non-zero error code. See `unlink(2)`.

Due to the side effects performed by this intrinsic, the function form is not recommended.

For information on other intrinsics with the same name: See Section 10.11.9.255 [Unlink Intrinsic (subroutine)], page 166.

12.4.2.137 ZExt Intrinsic

This intrinsic is not yet implemented. The name is, however, reserved as an intrinsic. Use '`EXTERNAL ZExt`' to use this name for an external procedure.

13 Other Compilers

An individual Fortran source file can be compiled to an object ('`*.o`') file instead of to the final program executable. This allows several portions of a program to be compiled at different times and linked together whenever a new version of the program is needed. However, it introduces the issue of *object compatibility* across the various object files (and libraries, or '`*.a`' files) that are linked together to produce any particular executable file.

Object compatibility is an issue when combining, in one program, Fortran code compiled by more than one compiler (or more than one configuration of a compiler). If the compilers disagree on how to transform the names of procedures, there will normally be errors when linking such programs. Worse, if the compilers agree on naming, but disagree on issues like how to pass parameters, return arguments, and lay out `COMMON` areas, the earliest detected errors might be the incorrect results produced by the program (and that assumes these errors are detected, which is not always the case).

Normally, `g77` generates code that is object-compatible with code generated by a version of `f2c` configured (with, for example, '`f2c.h`' definitions) to be generally compatible with `g77` as built by `gcc`. (Normally, `f2c` will, by default, conform to the appropriate configuration, but it is possible that older or perhaps even newer versions of `f2c`, or versions having certain configuration changes to `f2c` internals, will produce object files that are incompatible with `g77`.)

For example, a Fortran string subroutine argument will become two arguments on the C side: a `char *` and an `int` length.

Much of this compatibility results from the fact that `g77` uses the same run-time library, `libf2c`, used by `f2c`.

Other compilers might or might not generate code that is object-compatible with `libf2c` and current `g77`, and some might offer such compatibility only when explicitly selected via a command-line option to the compiler.

Note: This portion of the documentation definitely needs a lot of work!

13.1 Dropping `f2c` Compatibility

Specifying '`-fno-f2c`' allows `g77` to generate, in some cases, faster code, by not needing to allow to the possibility of linking with code compiled by `f2c`.

For example, this affects how `REAL(KIND=1)`, `COMPLEX(KIND=1)`, and `COMPLEX(KIND=2)` functions are called. With '`-fno-f2c`', they are compiled as returning the appropriate `gcc` type (`float`, `__complex__ float`, `__complex__ double`, in many configurations).

With '`-ff2c`' in force, they are compiled differently (with perhaps slower run-time performance) to accommodate the restrictions inherent in `f2c`'s use of K&R C as an intermediate language—`REAL(KIND=1)` functions return C's `double` type, while `COMPLEX` functions return `void` and use an extra argument pointing to a place for the functions to return their values.

It is possible that, in some cases, leaving '`-ff2c`' in force might produce faster code than using '`-fno-f2c`'. Feel free to experiment, but remember to experiment with changing the way *entire programs and their Fortran libraries are compiled* at a time, since this sort of

experimentation affects the interface of code generated for a Fortran source file—that is, it affects object compatibility.

Note that `f2c` compatibility is a fairly static target to achieve, though not necessarily perfectly so, since, like `g77`, it is still being improved. However, specifying '`-fno-f2c`' causes `g77` to generate code that will probably be incompatible with code generated by future versions of `g77` when the same option is in force. You should make sure you are always able to recompile complete programs from source code when upgrading to new versions of `g77` or `f2c`, especially when using options such as '`-fno-f2c`'.

Therefore, if you are using `g77` to compile libraries and other object files for possible future use and you don't want to require recompilation for future use with subsequent versions of `g77`, you might want to stick with `f2c` compatibility for now, and carefully watch for any announcements about changes to the `f2c/libf2c` interface that might affect existing programs (thus requiring recompilation).

It is probable that a future version of `g77` will not, by default, generate object files compatible with `f2c`, and that version probably would no longer use `libf2c`. If you expect to depend on this compatibility in the long term, use the options '`-ff2c -ff2c-library`' when compiling all of the applicable code. This should cause future versions of `g77` either to produce compatible code (at the expense of the availability of some features and performance), or at the very least, to produce diagnostics.

13.2 Compilers Other Than `f2c`

On systems with Fortran compilers other than `f2c` and `g77`, code compiled by `g77` is not expected to work well with code compiled by the native compiler. (This is true for `f2c`-compiled objects as well.) Libraries compiled with the native compiler probably will have to be recompiled with `g77` to be used with `g77`-compiled code.

Reasons for such incompatibilities include:

- There might be differences in the way names of Fortran procedures are translated for use in the system's object-file format. For example, the statement '`CALL FOO`' might be compiled by `g77` to call a procedure the linker `ld` sees given the name '`_foo_`', while the apparently corresponding statement '`SUBROUTINE FOO`' might be compiled by the native compiler to define the linker-visible name '`_foo`', or '`_FOO_`', and so on.
- There might be subtle type mismatches which cause subroutine arguments and function return values to get corrupted.

  This is why simply getting `g77` to transform procedure names the same way a native compiler does is not usually a good idea—unless some effort has been made to ensure that, aside from the way the two compilers transform procedure names, everything else about the way they generate code for procedure interfaces is identical.
- Native compilers use libraries of private I/O routines which will not be available at link time unless you have the native compiler—and you would have to explicitly ask for them.

  For example, on the Sun you would have to add '`-L/usr/lang/SCx.x -lF77 -lV77`' to the link command.

14 Other Languages

Note: This portion of the documentation definitely needs a lot of work!

14.1 Tools and advice for interoperating with C and C++

The following discussion assumes that you are running `g77` in `f2c` compatibility mode, i.e. not using '`-fno-f2c`'. It provides some advice about quick and simple techniques for linking Fortran and C (or C++), the most common requirement. For the full story consult the description of code generation. See Chapter 16 [Debugging and Interfacing], page 239.

When linking Fortran and C, it's usually best to use `g77` to do the linking so that the correct libraries are included (including the maths one). If you're linking with C++ you will want to add '`-lstdc++`', '`-lg++`' or whatever. If you need to use another driver program (or `ld` directly), you can find out what linkage options `g77` passes by running '`g77 -v`'.

14.1.1 C Interfacing Tools

Even if you don't actually use it as a compiler, '`f2c`' from `ftp://ftp.netlib.org/f2c/src`, can be a useful tool when you're interfacing (linking) Fortran and C. See Section 14.1.3 [Generating Skeletons and Prototypes with `f2c`], page 211.

To use `f2c` for this purpose you only need retrieve and build the '`src`' directory from the distribution, consult the '`README`' instructions there for machine-specifics, and install the `f2c` program on your path.

Something else that might be useful is '`cfortran.h`' from `ftp://zebra/desy.de/cfortran`. This is a fairly general tool which can be used to generate interfaces for calling in both directions between Fortran and C. It can be used in `f2c` mode with `g77`—consult its documentation for details.

14.1.2 Accessing Type Information in C

Generally, C code written to link with `g77` code—calling and/or being called from Fortran—should '`#include <f2c.h>`' to define the C versions of the Fortran types. Don't assume Fortran `INTEGER` types correspond to C '`int`'s, for instance; instead, declare them as `integer`, a type defined by '`f2c.h`'. '`f2c.h`' is installed where `gcc` will find it by default, assuming you use a copy of `gcc` compatible with `g77`, probably built at the same time as `g77`.

14.1.3 Generating Skeletons and Prototypes with `f2c`

A simple and foolproof way to write `g77`-callable C routines—e.g. to interface with an existing library—is to write a file (named, for example, '`fred.f`') of dummy Fortran skeletons comprising just the declaration of the routine(s) and dummy arguments plus '`END`' statements. Then run `f2c` on file '`fred.f`' to produce '`fred.c`' into which you can edit useful code, confident the calling sequence is correct, at least. (There are some errors otherwise commonly made in generating C interfaces with f2c conventions, such as not using `doublereal` as the return type of a `REAL FUNCTION`.)

`f2c` also can help with calling Fortran from C, using its '`-P`' option to generate C prototypes appropriate for calling the Fortran.[1] If the Fortran code containing any routines to be called from C is in file '`joe.f`', use the command *`f2c -P joe.f`* to generate the file '`joe.P`' containing prototype information. `#include` this in the C which has to call the Fortran routines to make sure you get it right.

See Section 16.8 [Arrays (DIMENSION], page 244, for information on the differences between the way Fortran (including compilers like `g77`) and C handle arrays.

14.1.4 C++ Considerations

`f2c` can be used to generate suitable code for compilation with a C++ system using the '`-C++`' option. The important thing about linking `g77`-compiled code with C++ is that the prototypes for the `g77` routines must specify C linkage to avoid name mangling. So, use an '`extern "C"`' declaration. `f2c`'s '`-C++`' option will take care of this when generating skeletons or prototype files as above, and also avoid clashes with C++ reserved words in addition to those in C.

14.1.5 Startup Code

Unlike with some runtime systems, it shouldn't be necessary (unless there are bugs) to use a Fortran main program to ensure the runtime—specifically the i/o system—is initialized. However, to use the `g77` intrinsics `GETARG()` and `IARGC()` the `main()` routine from the '`libf2c`' library must be used, either explicitly or implicitly by using a Fortran main program. This `main()` program calls `MAIN__()` (where the names are C-type `extern` names, i.e. not mangled). You need to provide this nullary procedure as the entry point for your C code if using '`libf2c`''s `main`. In some cases it might be necessary to provide a dummy version of this to avoid linkers complaining about failure to resolve `MAIN__()` if linking against '`libf2c`' and not using `main()` from it.

---

[1] The files generated like this can also be used for inter-unit consistency checking of dummy and actual arguments, although the '`ftnchek`' tool from `ftp://ftp.netlib.org/fortran` or `ftp://ftp.dsm.fordham.edu` is probably better for this purpose.

15 Installing GNU Fortran

The following information describes how to install g77.

The information in this file generally pertains to dealing with *source* distributions of g77 and gcc. It is possible that some of this information will be applicable to some *binary* distributions of these products—however, since these distributions are not made by the maintainers of g77, responsibility for binary distributions rests with whoever built and first distributed them.

Nevertheless, efforts to make g77 easier to both build and install from source and package up as a binary distribution are ongoing.

15.1 Prerequisites

The procedures described to unpack, configure, build, and install g77 assume your system has certain programs already installed.

The following prerequisites should be met by your system before you follow the g77 installation instructions:

gzip
: To unpack the gcc and g77 distributions, you'll need the gunzip utility in the gzip distribution. Most UNIX systems already have gzip installed. If yours doesn't, you can get it from the FSF.

  Note that you'll need tar and other utilities as well, but all UNIX systems have these. There are GNU versions of all these available—in fact, a complete GNU UNIX system can be put together on most systems, if desired.

  The version of GNU gzip used to package this release is 1.2.4. (The version of GNU tar used to package this release is 1.12.)

'gcc-2.7.2.3.tar.gz'
: You need to have this, or some other applicable, version of gcc on your system. The version should be an exact copy of a distribution from the FSF. Its size is approximately 7.1MB.

  If you've already unpacked 'gcc-2.7.2.3.tar.gz' into a directory (named 'gcc-2.7.2.3') called the *source tree* for gcc, you can delete the distribution itself, but you'll need to remember to skip any instructions to unpack this distribution.

  Without an applicable gcc source tree, you cannot build g77. You can obtain an FSF distribution of gcc from the FSF.

'g77-0.5.22.tar.gz'
: You probably have already unpacked this package, or you are reading an advance copy of these installation instructions, which are contained in this distribution. The size of this package is approximately 1.5MB.

  You can obtain an FSF distribution of g77 from the FSF, the same way you obtained gcc.

Enough disk space

The amount of disk space needed to unpack, build, install, and use `g77` depends on the type of system you're using, how you build `g77`, and how much of it you install (primarily, which languages you install).

The sizes shown below assume all languages distributed in `gcc-gcc`, plus `g77`, will be built and installed. These sizes are indicative of GNU/Linux systems on Intel x86 running COFF and on Digital Alpha (AXP) systems running ELF. These should be fairly representative of 32-bit and 64-bit systems, respectively.

Note that all sizes are approximate and subject to change without notice! They are based on preliminary releases of g77 made shortly before the public beta release.

— `gcc` and `g77` distributions occupy 8.6MB packed, 35MB unpacked. These consist of the source code and documentation, plus some derived files (mostly documentation), for `gcc` and `g77`. Any deviations from these numbers for different kinds of systems are likely to be very minor.

— A "bootstrap" build requires an additional 67.3MB for a total of 102MB on an ix86, and an additional 98MB for a total of 165MB on an Alpha.

— Removing '`gcc/stage1`' after the build recovers 10.7MB for a total of 91MB on an ix86, and recovers ??MB for a total of ??MB on an Alpha.

After doing this, the integrity of the build can still be verified via '`make compare`', and the `gcc` compiler modified and used to build itself for testing fairly quickly, using the copy of the compiler kept in `gcc/stage2`.

— Removing '`gcc/stage2`' after the build further recovers 27.3MB for a total of 64.3MB, and recovers ??MB for a total of ??MB on an Alpha.

After doing this, the compiler can still be installed, especially if GNU `make` is used to avoid gratuitous rebuilds (or, the installation can be done by hand).

— Installing `gcc` and `g77` copies 14.9MB onto the '`--prefix`' disk for a total of 79.2MB on an ix86, and copies ??MB onto the '`--prefix`' disk for a total of ??MB on an Alpha.

After installation, if no further modifications and builds of `gcc` or `g77` are planned, the source and build directory may be removed, leaving the total impact on a system's disk storage as that of the amount copied during installation.

Systems with the appropriate version of `gcc` installed don't require the complete bootstrap build. Doing a "straight build" requires about as much space as does a bootstrap build followed by removing both the '`gcc/stage1`' and '`gcc/stage2`' directories.

Installing `gcc` and `g77` over existing versions might require less *new* disk space, but note that, unlike many products, `gcc` installs itself in a way that avoids overwriting other installed versions of itself, so that other versions may easily be invoked (via '`gcc -V` *version*').

So, the amount of space saved as a result of having an existing version of `gcc` and `g77` already installed is not much—typically only the command drivers (`gcc`,

`g77`, `g++`, and so on, which are small) and the documentation is overwritten by the new installation. The rest of the new installation is done without replacing existing installed versions (assuming they have different version numbers).

`patch`
: Although you can do everything `patch` does yourself, by hand, without much trouble, having `patch` installed makes installation of new versions of GNU utilities such as `g77` so much easier that it is worth getting. You can obtain `patch` the same way you obtained `gcc` and `g77`.

  In any case, you can apply patches by hand—patch files are designed for humans to read them.

  The version of GNU `patch` used to develop this release is 2.5.

`make`
: Your system must have `make`, and you will probably save yourself a lot of trouble if it is GNU `make` (sometimes referred to as `gmake`).

  The version of GNU `make` used to develop this release is 3.76.1.

`cc`
: Your system must have a working C compiler.

  See section "Installing GNU CC" in *Using and Porting GNU CC*, for more information on prerequisites for installing `gcc`.

`bison`
: If you do not have `bison` installed, you can usually work around any need for it, since `g77` itself does not use it, and `gcc` normally includes all files generated by running it in its distribution. You can obtain `bison` the same way you obtained `gcc` and `g77`.

  The version of GNU `bison` used to develop this release is 1.25.

  See Section 15.5.12 [Missing bison?], page 235, for information on how to work around not having `bison`.

`makeinfo`
: If you are missing `makeinfo`, you can usually work around any need for it. You can obtain `makeinfo` the same way you obtained `gcc` and `g77`.

  The version of GNU `makeinfo` used to develop this release is 1.68, from GNU `texinfo` version 3.11.

  See Section 15.5.13 [Missing makeinfo?], page 236, for information on getting around the lack of `makeinfo`.

`sed`
: All UNIX systems have `sed`, but some have a broken version that cannot handle configuring, building, or installing `gcc` or `g77`.

  The version of GNU `sed` used to develop this release is 2.05. (Note that GNU `sed` version 3.0 was withdrawn by the FSF—if you happen to have this version installed, replace it with version 2.05 immediately. See a GNU distribution site for further explanation.)

`root` access or equivalent
: To perform the complete installation procedures on a system, you need to have `root` access to that system, or equivalent access to the '`--prefix`' directory tree specified on the `configure` command line.

  Portions of the procedure (such as configuring and building `g77`) can be performed by any user with enough disk space and virtual memory.

However, these instructions are oriented towards less-experienced users who want to install `g77` on their own personal systems.

System administrators with more experience will want to determine for themselves how they want to modify the procedures described below to suit the needs of their installation.

15.2 Problems Installing

This is a list of problems (and some apparent problems which don't really mean anything is wrong) that show up when configuring, building, installing, or porting GNU Fortran.

See section "Installation Problems" in *Using and Porting GNU CC*, for more information on installation problems that can afflict either `gcc` or `g77`.

15.2.1 General Problems

These problems can occur on most or all systems.

15.2.1.1 GNU C Required

Compiling `g77` requires GNU C, not just ANSI C. Fixing this wouldn't be very hard (just tedious), but the code using GNU extensions to the C language is expected to be rewritten for 0.6 anyway, so there are no plans for an interim fix.

This requirement does not mean you must already have `gcc` installed to build `g77`. As long as you have a working C compiler, you can use a bootstrap build to automate the process of first building `gcc` using the working C compiler you have, then building `g77` and rebuilding `gcc` using that just-built `gcc`, and so on.

15.2.1.2 Patching GNU CC Necessary

`g77` currently requires application of a patch file to the gcc compiler tree. The necessary patches should be folded in to the mainline gcc distribution.

Some combinations of versions of `g77` and `gcc` might actually *require* no patches, but the patch files will be provided anyway as long as there are more changes expected in subsequent releases. These patch files might contain unnecessary, but possibly helpful, patches. As a result, it is possible this issue might never be resolved, except by eliminating the need for the person configuring `g77` to apply a patch by hand, by going to a more automated approach (such as configure-time patching).

15.2.1.3 Building GNU CC Necessary

It should be possible to build the runtime without building `cc1` and other non-Fortran items, but, for now, an easy way to do that is not yet established.

15.2.1.4 Missing strtoul

On SunOS4 systems, linking the `f771` program produces an error message concerning an undefined symbol named '`_strtoul`'.

This is not a `g77` bug. See Section 15.5.5 [Patching GNU Fortran], page 229, for information on a workaround provided by `g77`.

The proper fix is either to upgrade your system to one that provides a complete ANSI C environment, or improve `gcc` so that it provides one for all the languages and configurations it supports.

Note: In earlier versions of `g77`, an automated workaround for this problem was attempted. It worked for systems without '`_strtoul`', substituting the incomplete-yet-sufficient version supplied with `g77` for those systems. However, the automated workaround failed mysteriously for systems that appeared to have conforming ANSI C environments, and it was decided that, lacking resources to more fully investigate the problem, it was better to not punish users of those systems either by requiring them to work around the problem by hand or by always substituting an incomplete `strtoul()` implementation when their systems had a complete, working one. Unfortunately, this meant inconveniencing users of systems not having `strtoul()`, but they're using obsolete (and generally unsupported) systems anyway.

15.2.1.5 Object File Differences

A comparison of object files after building Stage 3 during a bootstrap build will result in '`gcc/f/zzz.o`' being flagged as different from the Stage 2 version. That is because it contains a string with an expansion of the `__TIME__` macro, which expands to the current time of day. It is nothing to worry about, since '`gcc/f/zzz.c`' doesn't contain any actual code. It does allow you to override its use of `__DATE__` and `__TIME__` by defining macros for the compilation—see the source code for details.

15.2.1.6 Cleanup Kills Stage Directories

It'd be helpful if `g77`'s '`Makefile.in`' or '`Make-lang.in`' would create the various '`stage`*n*' directories and their subdirectories, so developers and expert installers wouldn't have to reconfigure after cleaning up.

15.2.1.7 Missing `gperf`?

If a build aborts trying to invoke `gperf`, that strongly suggests an improper method was used to create the `gcc` source directory, such as the UNIX '`cp -r`' command instead of '`cp -pr`', since this problem very likely indicates that the date-time-modified information on the `gcc` source files is incorrect.

The proper solution is to recreate the `gcc` source directory from a `gcc` distribution known to be provided by the FSF.

It is possible you might be able to temporarily work around the problem, however, by trying these commands:

```
sh# cd gcc
sh# touch c-gperf.h
sh#
```

These commands update the date-time-modified information for the file produced by the invocation of `gperf` in the current versions of `gcc`, so that `make` no longer believes it needs to update it. This file should already exist in a `gcc` distribution, but mistakes made when copying the `gcc` directory can leave the modification information set such that the `gperf` input files look more "recent" than the corresponding output files.

If the above does not work, definitely start from scratch and avoid copying the `gcc` using any method that does not reliably preserve date-time-modified information, such as the UNIX '`cp -r`' command (use '`cp -pr`' instead).

15.2.2 System-specific Problems

If your system is based on a Digital Alpha (AXP) architecture and employs a 64-bit operating system (such as GNU/Linux), you might consider using `egcs` instead of versions of `g77` based on versions of `gcc` prior to 2.8. `http://www.cygnus.com/egcs` for information on `egcs`, or obtain a copy from `ftp://egcs.cygnus.com/pub/egcs`.

If your system is Irix 6, to obtain a working version of `gcc`, `http://reality.sgi.com/knobi/gcc-2.7.2.x-`

15.2.3 Cross-compiler Problems

`g77` has been in alpha testing since September of 1992, and in public beta testing since February of 1995. Alpha testing was done by a small number of people worldwide on a fairly wide variety of machines, involving self-compilation in most or all cases. Beta testing has been done primarily via self-compilation, but in more and more cases, cross-compilation (and "criss-cross compilation", where a version of a compiler is built on one machine to run on a second and generate code that runs on a third) has been tried and has succeeded, to varying extents.

Generally, `g77` can be ported to any configuration to which `gcc`, `f2c`, and `libf2c` can be ported and made to work together, aside from the known problems described in this manual. If you want to port `g77` to a particular configuration, you should first make sure `gcc` and `libf2c` can be ported to that configuration before focusing on `g77`, because `g77` is so dependent on them.

Even for cases where `gcc` and `libf2c` work, you might run into problems with cross-compilation on certain machines, for several reasons.

- There is one known bug (a design bug to be fixed in 0.6) that prevents configuration of `g77` as a cross-compiler in some cases, though there are assumptions made during configuration that probably make doing non-self-hosting builds a hassle, requiring manual intervention.
- `gcc` might still have some trouble being configured for certain combinations of machines. For example, it might not know how to handle floating-point constants.
- Improvements to the way `libf2c` is built could make building `g77` as a cross-compiler easier—for example, passing and using '`$(LD)`' and '`$(AR)`' in the appropriate ways.

- There are still some challenges putting together the right run-time libraries (needed by `libf2c`) for a target system, depending on the systems involved in the configuration. (This is a general problem with cross-compilation, and with `gcc` in particular.)

15.3 Changing Settings Before Building

Here are some internal `g77` settings that can be changed by editing source files in '`gcc/f/`' before building.

This information, and perhaps even these settings, represent stop-gap solutions to problems people doing various ports of `g77` have encountered. As such, none of the following information is expected to be pertinent in future versions of `g77`.

15.3.1 Larger File Unit Numbers

As distributed, whether as part of `f2c` or `g77`, `libf2c` accepts file unit numbers only in the range 0 through 99. For example, a statement such as '`WRITE (UNIT=100)`' causes a run-time crash in `libf2c`, because the unit number, 100, is out of range.

If you know that Fortran programs at your installation require the use of unit numbers higher than 99, you can change the value of the '`MXUNIT`' macro, which represents the maximum unit number, to an appropriately higher value.

To do this, edit the file '`f/runtime/libI77/fio.h`' in your `g77` source tree, changing the following line:

```
#define MXUNIT 100
```

Change the line so that the value of '`MXUNIT`' is defined to be at least one *greater* than the maximum unit number used by the Fortran programs on your system.

(For example, a program that does '`WRITE (UNIT=255)`' would require '`MXUNIT`' set to at least 256 to avoid crashing.)

Then build or rebuild `g77` as appropriate.

Note: Changing this macro has *no* effect on other limits your system might place on the number of files open at the same time. That is, the macro might allow a program to do '`WRITE (UNIT=100)`', but the library and operating system underlying `libf2c` might disallow it if many other files have already been opened (via `OPEN` or implicitly via `READ`, `WRITE`, and so on). Information on how to increase these other limits should be found in your system's documentation.

15.3.2 Always Flush Output

Some Fortran programs require output (writes) to be flushed to the operating system (under UNIX, via the `fflush()` library call) so that errors, such as disk full, are immediately flagged via the relevant `ERR=` and `IOSTAT=` mechanism, instead of such errors being flagged later as subsequent writes occur, forcing the previously written data to disk, or when the file is closed.

Essentially, the difference can be viewed as synchronous error reporting (immediate flagging of errors during writes) versus asynchronous, or, more precisely, buffered error reporting (detection of errors might be delayed).

`libf2c` supports flagging write errors immediately when it is built with the '`ALWAYS_FLUSH`' macro defined. This results in a `libf2c` that runs slower, sometimes quite a bit slower, under certain circumstances—for example, accessing files via the networked file system NFS—but the effect can be more reliable, robust file I/O.

If you know that Fortran programs requiring this level of precision of error reporting are to be compiled using the version of `g77` you are building, you might wish to modify the `g77` source tree so that the version of `libf2c` is built with the '`ALWAYS_FLUSH`' macro defined, enabling this behavior.

To do this, find this line in '`f/runtime/configure.in`' in your `g77` source tree:

```
dnl AC_DEFINE(ALWAYS_FLUSH)
```

Remove the leading '`dnl `', so the line begins with '`AC_DEFINE(`', and run `autoconf` in that file's directory. (Or, if you don't have `autoconf`, you can modify '`f2c.h.in`' in the same directory to include the line '`#define ALWAYS_FLUSH`' after '`#define F2C_INCLUDE`'.)

Then build or rebuild `g77` as appropriate.

15.3.3 Maximum Stackable Size

`g77`, on most machines, puts many variables and arrays on the stack where possible, and can be configured (by changing '`FFECOM_sizeMAXSTACKITEM`' in '`gcc/f/com.c`') to force smaller-sized entities into static storage (saving on stack space) or permit larger-sized entities to be put on the stack (which can improve run-time performance, as it presents more opportunities for the GBE to optimize the generated code).

Note: Putting more variables and arrays on the stack might cause problems due to system-dependent limits on stack size. Also, the value of '`FFECOM_sizeMAXSTACKITEM`' has no effect on automatic variables and arrays. See Section 18.1 [But-bugs], page 265, for more information.

15.3.4 Floating-point Bit Patterns

The `g77` build will crash if an attempt is made to build it as a cross-compiler for a target when `g77` cannot reliably determine the bit pattern of floating-point constants for the target. Planned improvements for version 0.6 of `g77` will give it the capabilities it needs to not have to crash the build but rather generate correct code for the target. (Currently, `g77` would generate bad code under such circumstances if it didn't crash during the build, e.g. when compiling a source file that does something like '`EQUIVALENCE (I,R)`' and '`DATA R/9.43578/`'.)

15.3.5 Initialization of Large Aggregate Areas

A warning message is issued when `g77` sees code that provides initial values (e.g. via `DATA`) to an aggregate area (`COMMON` or `EQUIVALENCE`, or even a large enough array or `CHARACTER` variable) that is large enough to increase `g77`'s compile time by roughly a factor of 10.

This size currently is quite small, since `g77` currently has a known bug requiring too much memory and time to handle such cases. In '`gcc/f/data.c`', the macro '`FFEDATA_sizeTOO_BIG_INIT_`'

is defined to the minimum size for the warning to appear. The size is specified in storage units, which can be bytes, words, or whatever, on a case-by-case basis.

After changing this macro definition, you must (of course) rebuild and reinstall `g77` for the change to take effect.

Note that, as of version 0.5.18, improvements have reduced the scope of the problem for *sparse* initialization of large arrays, especially those with large, contiguous uninitialized areas. However, the warning is issued at a point prior to when `g77` knows whether the initialization is sparse, and delaying the warning could mean it is produced too late to be helpful.

Therefore, the macro definition should not be adjusted to reflect sparse cases. Instead, adjust it to generate the warning when densely initialized arrays begin to cause responses noticeably slower than linear performance would suggest.

15.3.6 Alpha Problems Fixed

`g77` used to warn when it was used to compile Fortran code for a target configuration that is not basically a 32-bit machine (such as an Alpha, which is a 64-bit machine, especially if it has a 64-bit operating system running on it). That was because `g77` was known to not work properly on such configurations.

As of version 0.5.20, `g77` is believed to work well enough on such systems. So, the warning is no longer needed or provided.

However, support for 64-bit systems, especially in areas such as cross-compilation and handling of intrinsics, is still incomplete. The symptoms are believed to be compile-time diagnostics rather than the generation of bad code. It is hoped that version 0.6 will completely support 64-bit systems.

15.4 Quick Start

This procedure configures, builds, and installs `g77` "out of the box" and works on most UNIX systems. Each command is identified by a unique number, used in the explanatory text that follows. For the most part, the output of each command is not shown, though indications of the types of responses are given in a few cases.

To perform this procedure, the installer must be logged in as user `root`. Much of it can be done while not logged in as `root`, and users experienced with UNIX administration should be able to modify the procedure properly to do so.

Following traditional UNIX conventions, it is assumed that the source trees for `g77` and `gcc` will be placed in '`/usr/src`'. It also is assumed that the source distributions themselves already reside in '`/usr/FSF`', a naming convention used by the author of `g77` on his own system:

```
/usr/FSF/gcc-2.7.2.3.tar.gz
/usr/FSF/g77-0.5.22.tar.gz
```

Users of the following systems should not blindly follow these quick-start instructions, because of problems their systems have coping with straightforward installation of `g77`:

- SunOS4

Instead, see Section 15.5 [Complete Installation], page 225, for detailed information on how to configure, build, and install g77 for your particular system. Also, see Chapter 18 [Known Causes of Trouble with GNU Fortran], page 265, for information on bugs and other problems known to afflict the installation process, and how to report newly discovered ones.

If your system is *not* on the above list, and *is* a UNIX system or one of its variants, you should be able to follow the instructions below. If you vary *any* of the steps below, you might run into trouble, including possibly breaking existing programs for other users of your system. Before doing so, it is wise to review the explanations of some of the steps. These explanations follow this list of steps.

```
sh[ 1]# cd /usr/src
sh[ 2]# gunzip -c < /usr/FSF/gcc-2.7.2.3.tar.gz | tar xf -
[Might say "Broken pipe"...that is normal on some systems.]
sh[ 3]# gunzip -c < /usr/FSF/g77-0.5.22.tar.gz | tar xf -
["Broken pipe" again possible.]
sh[ 4]# ln -s gcc-2.7.2.3 gcc
sh[ 5]# ln -s g77-0.5.22 g77
sh[ 6]# mv -i g77/* gcc
[No questions should be asked by mv here; or, you made a mistake.]
sh[ 7]# patch -p1 -E -V t -d gcc < gcc/f/gbe/2.7.2.3.diff
[Unless patch complains about rejected patches, this step worked.]
sh[ 8]# cd gcc
sh[ 9]# touch f77-install-ok
[Do not do the above if your system already has an f77
command, unless you've checked that overwriting it
is okay.]
sh[10]# touch f2c-install-ok
[Do not do the above if your system already has an f2c
command, unless you've checked that overwriting it
is okay.  Else, touch f2c-exists-ok.]
sh[11]# ./configure --prefix=/usr
[Do not do the above if gcc is not installed in /usr/bin.
You might need a different --prefix=..., as
described below.]
sh[12]# make bootstrap
[This takes a long time, and is where most problems occur.]
sh[13]# make compare
[This verifies that the compiler is 'sane'.  Only
the file 'f/zzz.o' (aka 'tmp-foo1' and 'tmp-foo2')
should be in the list of object files this command
prints as having different contents.  If other files
are printed, you have likely found a g77 bug.]
sh[14]# rm -fr stage1
sh[15]# make -k install
[The actual installation.]
sh[16]# g77 -v
[Verify that g77 is installed, obtain version info.]
sh[17]#
```

See Section 15.5.11 [Updating Your Info Directory], page 235, for information on how to update your system's top-level `info` directory to contain a reference to this manual, so that users of `g77` can easily find documentation instead of having to ask you for it.

Elaborations of many of the above steps follows:

Step 1: *`cd /usr/src`*

You can build `g77` pretty much anyplace. By convention, this manual assumes '`/usr/src`'. It might be helpful if other users on your system knew where to look for the source code for the installed version of `g77` and `gcc` in any case.

Step 3: *`gunzip -d < /usr/FSF/g77-0.5.22.tar.gz | tar xf -`*

It is not always necessary to obtain the latest version of `g77` as a complete '`.tar.gz`' file if you have a complete, earlier distribution of `g77`. If appropriate, you can unpack that earlier version of `g77`, and then apply the appropriate patches to achieve the same result—a source tree containing version 0.5.22 of `g77`.

Step 4: *`ln -s gcc-2.7.2.3 gcc`*

Step 5: *`ln -s g77-0.5.22 g77`*

These commands mainly help reduce typing, and help reduce visual clutter in examples in this manual showing what to type to install `g77`.

See Section 15.5.1 [Unpacking], page 225, for information on using distributions of `g77` made by organizations other than the FSF.

Step 6: *`mv -i g77/* gcc`*

After doing this, you can, if you like, type '`rm g77`' and '`rmdir g77-0.5.22`' to remove the empty directory and the symbol link to it. But, it might be helpful to leave them around as quick reminders of which version(s) of `g77` are installed on your system.

See Section 15.5.1 [Unpacking], page 225, for information on the contents of the '`g77`' directory (as merged into the '`gcc`' directory).

Step 7: *`patch -p1 ...`*

If you are using GNU `patch` version 2.5 or later, this should produce a list of files patched. (Other versions of `patch` might not work properly.)

If messages about "fuzz", "offset", or especially "reject files" are printed, it might mean you applied the wrong patch file. If you believe this is the case, it is best to restart the sequence after deleting (or at least renaming to unused names) the top-level directories for `g77` and `gcc` and their symbolic links.

After this command finishes, the `gcc` directory might have old versions of several files as saved by `patch`. To remove these, after *`cd gcc`*, type *`rm -i *.~*~`*.

See Section 15.5.2 [Merging Distributions], page 226, for more information.

Note: `gcc` versions circa 2.7.2.2 and 2.7.2.3 are known to have slightly differing versions of the `gcc/ChangeLog` file, depending on how they are obtained. You can safely ignore diagnostics `patch` reports when patching this particular file, since it is purely a documentation file for implementors. See '`gcc/f/gbe/2.7.2.3.diff`' for more information.

Step 9: *touch f77-install-ok*

Don't do this if you don't want to overwrite an existing version of `f77` (such as a native compiler, or a script that invokes `f2c`). Otherwise, installation will overwrite the `f77` command and the `f77` man pages with copies of the corresponding `g77` material.

See Section 15.5.3 [Installing `f77`], page 228, for more information.

Step 10: *touch f2c-install-ok*

Don't do this if you don't want to overwrite an existing installation of `libf2c` (though, chances are, you do). Instead, *touch f2c-exists-ok* to allow the installation to continue without any error messages about '`/usr/lib/libf2c.a`' already existing.

See Section 15.5.4 [Installing `f2c`], page 228, for more information.

Step 11: *./configure --prefix=/usr*

This is where you specify that the '`g77`' executable is to be installed in '`/usr/bin/`', the '`libf2c.a`' library is to be installed in '`/usr/lib/`', and so on.

You should ensure that any existing installation of the '`gcc`' executable is in '`/usr/bin/`'. Otherwise, installing `g77` so that it does not fully replace the existing installation of `gcc` is likely to result in the inability to compile Fortran programs.

See Section 15.5.6 [Where in the World Does Fortran (and GNU CC) Go?], page 230, for more information on determining where to install `g77`. See Section 15.5.7 [Configuring gcc], page 231, for more information on the configuration process triggered by invoking the '`./configure`' script.

Step 12: *make bootstrap*

See section "Installing GNU CC" in *Using and Porting GNU CC*, for information on the kinds of diagnostics you should expect during this procedure.

See Section 15.5.8 [Building gcc], page 231, for complete `g77`-specific information on this step.

Step 13: *make compare*

See Section 20.2 [Where to Port Bugs], page 295, for information on where to report that you observed more than '`f/zzz.o`' having different contents during this phase.

See Section 20.3 [How to Report Bugs], page 296, for information on *how* to report bugs like this.

Step 14: *rm -fr stage1*

You don't need to do this, but it frees up disk space.

Step 15: *make -k install*

If this doesn't seem to work, try:

```
make -k install install-libf77 install-f2c-all
```

See Section 15.5.10 [Installation of Binaries], page 234, for more information.

See Section 15.5.11 [Updating Your Info Directory], page 235, for information on entering this manual into your system's list of texinfo manuals.

Step 16: *g77 -v*

If this command prints approximately 25 lines of output, including the GNU Fortran Front End version number (which should be the same as the version number for the version of `g77` you just built and installed) and the version numbers for the three parts of the `libf2c` library (`libF77`, `libI77`, `libU77`), and those version numbers are all in agreement, then there is a high likelihood that the installation has been successfully completed.

You might consider doing further testing. For example, log in as a non-privileged user, then create a small Fortran program, such as:

```
      PROGRAM SMTEST
      DO 10 I=1, 10
         PRINT *, 'Hello World #', I
10    CONTINUE
      END
```

Compile, link, and run the above program, and, assuming you named the source file '`smtest.f`', the session should look like this:

```
sh# g77 -o smtest smtest.f
sh# ./smtest
 Hello World # 1
 Hello World # 2
 Hello World # 3
 Hello World # 4
 Hello World # 5
 Hello World # 6
 Hello World # 7
 Hello World # 8
 Hello World # 9
 Hello World # 10
sh#
```

After proper installation, you don't need to keep your gcc and g77 source and build directories around anymore. Removing them can free up a lot of disk space.

15.5 Complete Installation

Here is the complete `g77`-specific information on how to configure, build, and install `g77`.

15.5.1 Unpacking

The `gcc` source distribution is a stand-alone distribution. It is designed to be unpacked (producing the `gcc` source tree) and built as is, assuming certain prerequisites are met (including the availability of compatible UNIX programs such as `make`, `cc`, and so on).

However, before building `gcc`, you will want to unpack and merge the `g77` distribution in with it, so that you build a Fortran-capable version of `gcc`, which includes the `g77` command, the necessary run-time libraries, and this manual.

Unlike gcc, the g77 source distribution is *not* a stand-alone distribution. It is designed to be unpacked and, afterwards, immediately merged into an applicable gcc source tree. That is, the g77 distribution *augments* a gcc distribution—without gcc, generally only the documentation is immediately usable.

A sequence of commands typically used to unpack gcc and g77 is:

```
sh# cd /usr/src
sh# gunzip -c /usr/FSF/gcc-2.7.2.3.tar.gz | tar xf -
sh# gunzip -c /usr/FSF/g77-0.5.22.tar.gz | tar xf -
sh# ln -s gcc-2.7.2.3 gcc
sh# ln -s g77-0.5.22 g77
sh# mv -i g77/* gcc
```

Notes: The commands beginning with 'gunzip...' might print 'Broken pipe...' as they complete. That is nothing to worry about, unless you actually *hear* a pipe breaking. The ln commands are helpful in reducing typing and clutter in installation examples in this manual. Hereafter, the top level of gcc source tree is referred to as 'gcc', and the top level of just the g77 source tree (prior to issuing the mv command, above) is referred to as 'g77'.

There are three top-level names in a g77 distribution:

```
g77/COPYING.g77
g77/README.g77
g77/f
```

All three entries should be moved (or copied) into a gcc source tree (typically named after its version number and as it appears in the FSF distributions—e.g. 'gcc-2.7.2.3').

'g77/f' is the subdirectory containing all of the code, documentation, and other information that is specific to g77. The other two files exist to provide information on g77 to someone encountering a gcc source tree with g77 already present, who has not yet read these installation instructions and thus needs help understanding that the source tree they are looking at does not come from a single FSF distribution. They also help people encountering an unmerged g77 source tree for the first time.

Note: Please use **only** gcc and g77 source trees as distributed by the FSF. Use of modified versions, such as the Pentium-specific-optimization port of gcc, is likely to result in problems that appear to be in the g77 code but, in fact, are not. Do not use such modified versions unless you understand all the differences between them and the versions the FSF distributes—in which case you should be able to modify the g77 (or gcc) source trees appropriately so g77 and gcc can coexist as they do in the stock FSF distributions.

15.5.2 Merging Distributions

After merging the g77 source tree into the gcc source tree, the final merge step is done by applying the pertinent patches the g77 distribution provides for the gcc source tree.

Read the file 'gcc/f/gbe/README', and apply the appropriate patch file for the version of the GNU CC compiler you have, if that exists. If the directory exists but the appropriate file does not exist, you are using either an old, unsupported version, or a release one that is newer than the newest gcc version supported by the version of g77 you have.

As of version 0.5.18, g77 modifies the version number of gcc via the pertinent patches. This is done because the resulting version of gcc is deemed sufficiently different from the

vanilla distribution to make it worthwhile to present, to the user, information signaling the fact that there are some differences.

GNU version numbers make it easy to figure out whether a particular version of a distribution is newer or older than some other version of that distribution. The format is, generally, *major.minor.patch*, with each field being a decimal number. (You can safely ignore leading zeros; for example, 1.5.3 is the same as 1.5.03.) The *major* field only increases with time. The other two fields are reset to 0 when the field to their left is incremented; otherwise, they, too, only increase with time. So, version 2.6.2 is newer than version 2.5.8, and version 3.0 is newer than both. (Trailing '`.0`' fields often are omitted in announcements and in names for distributions and the directories they create.)

If your version of `gcc` is older than the oldest version supported by `g77` (as casually determined by listing the contents of '`gcc/f/gbe/`'), you should obtain a newer, supported version of `gcc`. (You could instead obtain an older version of `g77`, or try and get your `g77` to work with the old `gcc`, but neither approach is recommended, and you shouldn't bother reporting any bugs you find if you take either approach, because they're probably already fixed in the newer versions you're not using.)

If your version of `gcc` is newer than the newest version supported by `g77`, it is possible that your `g77` will work with it anyway. If the version number for `gcc` differs only in the *patch* field, you might as well try applying the `g77` patch that is for the newest version of `gcc` having the same *major* and *minor* fields, as this is likely to work.

So, for example, if a particular version of `g77` has support for `gcc` versions 2.7.0 and 2.7.1, it is likely that '`gcc-2.7.2`' would work well with `g77` by using the '`2.7.1.diff`' patch file provided with `g77` (aside from some offsets reported by `patch`, which usually are harmless).

However, '`gcc-2.8.0`' would almost certainly not work with that version of `g77` no matter which patch file was used, so a new version of `g77` would be needed (and you should wait for it rather than bothering the maintainers—see Chapter 9 [User-Visible Changes], page 65).

This complexity is the result of `gcc` and `g77` being separate distributions. By keeping them separate, each product is able to be independently improved and distributed to its user base more frequently.

However, `g77` often requires changes to contemporary versions of `gcc`. Also, the GBE interface defined by `gcc` typically undergoes some incompatible changes at least every time the *minor* field of the version number is incremented, and such changes require corresponding changes to the `g77` front end (FFE).

It is hoped that the GBE interface, and the `gcc` and `g77` products in general, will stabilize sufficiently for the need for hand-patching to disappear.

If you are using GNU `patch` version 2.5 or later, this should produce a list of files patched. (Other versions of `patch` might not work properly.)

If messages about "fuzz", "offset", or especially "reject files" are printed, it might mean you applied the wrong patch file. If you believe this is the case, it is best to restart the sequence after deleting (or at least renaming to unused names) the top-level directories for `g77` and `gcc` and their symbolic links. That is because `patch` might have partially patched

some `gcc` source files, so reapplying the correct patch file might result in the correct patches being applied incorrectly (due to the way `patch` necessarily works).

After `patch` finishes, the `gcc` directory might have old versions of several files as saved by `patch`. To remove these, after *`cd gcc`*, type *`rm -i *.~*~`*.

Note: `gcc` versions circa 2.7.2.2 and 2.7.2.3 are known to have slightly differing versions of the `gcc/ChangeLog` file, depending on how they are obtained. You can safely ignore diagnostics `patch` reports when patching this particular file, since it is purely a documentation file for implementors. See '`gcc/f/gbe/2.7.2.3.diff`' for more information.

Note: `g77`'s configuration file '`gcc/f/config-lang.in`' ensures that the source code for the version of `gcc` being configured has at least one indication of being patched as required specifically by `g77`. This configuration-time checking should catch failure to apply the correct patch and, if so caught, should abort the configuration with an explanation. *Please* do not try to disable the check, otherwise `g77` might well appear to build and install correctly, and even appear to compile correctly, but could easily produce broken code.

'`LC_ALL=C TZ=UTC0 diff -rcp2N`' is used to create the patch files in '`gcc/f/gbe/`'.

15.5.3 Installing `f77`

You should decide whether you want installation of `g77` to also install an `f77` command. On systems with a native `f77`, this is not normally desired, so `g77` does not do this by default.

If you want `f77` installed, create the file '`f77-install-ok`' (e.g. via the UNIX command '`touch f77-install-ok`') in the source or build top-level directory (the same directory in which the `g77` '`f`' directory resides, not the '`f`' directory itself), or edit '`gcc/f/Make-lang.in`' and change the definition of the '`F77_INSTALL_FLAG`' macro appropriately.

Usually, this means that, after typing '`cd gcc`', you would type '`touch f77-install-ok`'.

When you enable installation of `f77`, either a link to or a direct copy of the `g77` command is made. Similarly, '`f77.1`' is installed as a man page.

(The `uninstall` target in the '`gcc/Makefile`' also tests this macro and file, when invoked, to determine whether to delete the installed copies of `f77` and '`f77.1`'.)

Note: No attempt is yet made to install a program (like a shell script) that provides compatibility with any other `f77` programs. Only the most rudimentary invocations of `f77` will work the same way with `g77`.

15.5.4 Installing `f2c`

Currently, `g77` does not include `f2c` itself in its distribution. However, it does include a modified version of the `libf2c`. This version is normally compatible with `f2c`, but has been modified to meet the needs of `g77` in ways that might possibly be incompatible with some versions or configurations of `f2c`.

Decide how installation of `g77` should affect any existing installation of `f2c` on your system.

If you do not have `f2c` on your system (e.g. no '`/usr/bin/f2c`', no '`/usr/include/f2c.h`', and no '`/usr/lib/libf2c.a`', '`/usr/lib/libF77.a`', or '`/usr/lib/libI77.a`'), you don't need to be concerned with this item.

If you do have `f2c` on your system, you need to decide how users of `f2c` will be affected by your installing `g77`. Since `g77` is currently designed to be object-code-compatible with `f2c` (with very few, clear exceptions), users of `f2c` might want to combine `f2c`-compiled object files with `g77`-compiled object files in a single executable.

To do this, users of `f2c` should use the same copies of '`f2c.h`' and '`libf2c.a`' that `g77` uses (and that get built as part of `g77`).

If you do nothing here, the `g77` installation process will not overwrite the '`include/f2c.h`' and '`lib/libf2c.a`' files with its own versions, and in fact will not even install '`libf2c.a`' for use with the newly installed versions of `gcc` and `g77` if it sees that '`lib/libf2c.a`' exists—instead, it will print an explanatory message and skip this part of the installation.

To install `g77`'s versions of '`f2c.h`' and '`libf2c.a`' in the appropriate places, create the file '`f2c-install-ok`' (e.g. via the UNIX command '`touch f2c-install-ok`') in the source or build top-level directory (the same directory in which the `g77` '`f`' directory resides, not the '`f`' directory itself), or edit '`gcc/f/Make-lang.in`' and change the definition of the '`F2C_INSTALL_FLAG`' macro appropriately.

Usually, this means that, after typing '`cd gcc`', you would type '`touch f2c-install-ok`'.

Make sure that when you enable the overwriting of '`f2c.h`' and '`libf2c.a`' as used by `f2c`, you have a recent and properly configured version of '`bin/f2c`' so that it generates code that is compatible with `g77`.

If you don't want installation of `g77` to overwrite `f2c`'s existing installation, but you do want `g77` installation to proceed with installation of its own versions of '`f2c.h`' and '`libf2c.a`' in places where `g77` will pick them up (even when linking `f2c`-compiled object files—which might lead to incompatibilities), create the file '`f2c-exists-ok`' (e.g. via the UNIX command '`touch f2c-exists-ok`') in the source or build top-level directory, or edit '`gcc/f/Make-lang.in`' and change the definition of the '`F2CLIBOK`' macro appropriately.

15.5.5 Patching GNU Fortran

If you're using a SunOS4 system, you'll need to make the following change to '`gcc/f/proj.h`': edit the line reading

```
#define FFEPROJ_STRTOUL 1  ...
```

by replacing the '`1`' with '`0`'. Or, you can avoid editing the source by adding

```
CFLAGS='-DFFEPROJ_STRTOUL=0 -g -O'
```

to the command line for `make` when you invoke it. ('`-g`' is the default for '`CFLAGS`'.)

This causes a minimal version of `strtoul()` provided as part of the `g77` distribution to be compiled and linked into whatever `g77` programs need it, since some systems (like SunOS4 with only the bundled compiler and its runtime) do not provide this function in their system libraries.

Similarly, a minimal version of `bsearch()` is available and can be enabled by editing a line similar to the one for `strtoul()` above in '`gcc/f/proj.h`', if your system libraries lack `bsearch()`. The method of overriding '`X_CFLAGS`' may also be used.

These are not problems with g77, which requires an ANSI C environment. You should upgrade your system to one that provides a full ANSI C environment, or encourage the maintainers of gcc to provide one to all gcc-based compilers in future gcc distributions.

See Section 15.2 [Problems Installing], page 216, for more information on why strtoul() comes up missing and on approaches to dealing with this problem that have already been tried.

15.5.6 Where in the World Does Fortran (and GNU CC) Go?

Before configuring, you should make sure you know where you want the g77 and gcc binaries to be installed after they're built, because this information is given to the configuration tool and used during the build itself.

A g77 installation necessarily requires installation of a g77-aware version of gcc, so that the gcc command recognizes Fortran source files and knows how to compile them.

For this to work, the version of gcc that you will be building as part of g77 **must** be installed as the "active" version of gcc on the system.

Sometimes people make the mistake of installing gcc as '/usr/local/bin/gcc', leaving an older, non-Fortran-aware version in '/usr/bin/gcc'. (Or, the opposite happens.) This can result in g77 being unable to compile Fortran source files, because when it calls on gcc to do the actual compilation, gcc complains that it does not recognize the language, or the file name suffix.

So, determine whether gcc already is installed on your system, and, if so, *where* it is installed, and prepare to configure the new version of gcc you'll be building so that it installs over the existing version of gcc.

You might want to back up your existing copy of 'bin/gcc', and the entire 'lib/' directory, before you perform the actual installation (as described in this manual).

Existing gcc installations typically are found in '/usr' or '/usr/local'. If you aren't certain where the currently installed version of gcc and its related programs reside, look at the output of this command:

```
gcc -v -o /tmp/delete-me -xc /dev/null -xnone
```

All sorts of interesting information on the locations of various gcc-related programs and data files should be visible in the output of the above command. (The output also is likely to include a diagnostic from the linker, since there's no 'main_()' function.) However, you do have to sift through it yourself; gcc currently provides no easy way to ask it where it is installed and where it looks for the various programs and data files it calls on to do its work.

Just *building* g77 should not overwrite any installed programs—but, usually, after you build g77, you will want to install it, so backing up anything it might overwrite is a good idea. (This is true for any package, not just g77, though in this case it is intentional that g77 overwrites gcc if it is already installed—it is unusual that the installation process for one distribution intentionally overwrites a program or file installed by another distribution.)

Another reason to back up the existing version first, or make sure you can restore it easily, is that it might be an older version on which other users have come to depend for certain behaviors. However, even the new version of gcc you install will offer users

the ability to specify an older version of the actual compilation programs if desired, and these older versions need not include any `g77` components. See section "Specifying Target Machine and Compiler Version" in *Using and Porting GNU CC*, for information on the '`-V`' option of `gcc`.

15.5.7 Configuring GNU CC

`g77` is configured automatically when you configure `gcc`. There are two parts of `g77` that are configured in two different ways—`g77`, which "camps on" to the `gcc` configuration mechanism, and `libf2c`, which uses a variation of the GNU `autoconf` configuration system.

Generally, you shouldn't have to be concerned with either `g77` or `libf2c` configuration, unless you're configuring `g77` as a cross-compiler. In this case, the `libf2c` configuration, and possibly the `g77` and `gcc` configurations as well, might need special attention. (This also might be the case if you're porting `gcc` to a whole new system—even if it is just a new operating system on an existing, supported CPU.)

To configure the system, see section "Installing GNU CC" in *Using and Porting GNU CC*, following the instructions for running '`./configure`'. Pay special attention to the '`--prefix=`' option, which you almost certainly will need to specify.

(Note that `gcc` installation information is provided as a straight text file in '`gcc/INSTALL`'.)

The information printed by the invocation of '`./configure`' should show that the '`f`' directory (the Fortran language) has been configured. If it does not, there is a problem.

Note: Configuring with the '`--srcdir`' argument is known to work with GNU `make`, but it is not known to work with other variants of `make`. Irix5.2 and SunOS4.1 versions of `make` definitely won't work outside the source directory at present. `g77`'s portion of the '`configure`' script issues a warning message about this when you configure for building binaries outside the source directory.

15.5.8 Building GNU CC

Building `g77` requires building enough of `gcc` that these instructions assume you're going to build all of `gcc`, including `g++`, `protoize`, and so on. You can save a little time and disk space by changes the '`LANGUAGES`' macro definition in `gcc/Makefile.in` or `gcc/Makefile`, but if you do that, you're on your own. One change is almost *certainly* going to cause failures: removing '`c`' or '`f77`' from the definition of the '`LANGUAGES`' macro.

After configuring `gcc`, which configures `g77` and `libf2c` automatically, you're ready to start the actual build by invoking `make`.

Note: You **must** have run '`./configure`' before you run `make`, even if you're using an already existing `gcc` development directory, because '`./configure`' does the work to recognize that you've added `g77` to the configuration.

There are two general approaches to building GNU CC from scratch:

bootstrap This method uses minimal native system facilities to build a barebones, unoptimized `gcc`, that is then used to compile ("bootstrap") the entire system.

straight This method assumes a more complete native system exists, and uses that just once to build the entire system.

On all systems without a recent version of `gcc` already installed, the *bootstrap* method must be used. In particular, `g77` uses extensions to the C language offered, apparently, only by `gcc`.

On most systems with a recent version of `gcc` already installed, the *straight* method can be used. This is an advantage, because it takes less CPU time and disk space for the build. However, it does require that the system have fairly recent versions of many GNU programs and other programs, which are not enumerated here.

15.5.8.1 Bootstrap Build

A complete bootstrap build is done by issuing a command beginning with '`make bootstrap` ...', as described in section "Installing GNU CC" in *Using and Porting GNU CC*. This is the most reliable form of build, but it does require the most disk space and CPU time, since the complete system is built twice (in Stages 2 and 3), after an initial build (during Stage 1) of a minimal `gcc` compiler using the native compiler and libraries.

You might have to, or want to, control the way a bootstrap build is done by entering the `make` commands to build each stage one at a time, as described in the `gcc` manual. For example, to save time or disk space, you might want to not bother doing the Stage 3 build, in which case you are assuming that the `gcc` compiler you have built is basically sound (because you are giving up the opportunity to compare a large number of object files to ensure they're identical).

To save some disk space during installation, after Stage 2 is built, you can type '`rm -fr stage1`' to remove the binaries built during Stage 1.

Note: See Section 15.2.1.5 [Object File Differences], page 217, for information on expected differences in object files produced during Stage 2 and Stage 3 of a bootstrap build. These differences will be encountered as a result of using the '`make compare`' or similar command sequence recommended by the GNU CC installation documentation.

Also, See section "Installing GNU CC" in *Using and Porting GNU CC*, for important information on building `gcc` that is not described in this `g77` manual. For example, explanations of diagnostic messages and whether they're expected, or indicate trouble, are found there.

15.5.8.2 Straight Build

If you have a recent version of `gcc` already installed on your system, and if you're reasonably certain it produces code that is object-compatible with the version of `gcc` you want to build as part of building `g77`, you can save time and disk space by doing a straight build.

To build just the C and Fortran compilers and the necessary run-time libraries, issue the following command:

```
make -k CC=gcc LANGUAGES=f77 all g77
```

(The '`g77`' target is necessary because the `gcc` build procedures apparently do not automatically build command drivers for languages in subdirectories. It's the '`all`' target that triggers building everything except, apparently, the `g77` command itself.)

If you run into problems using this method, you have two options:

- Abandon this approach and do a bootstrap build.
- Try to make this approach work by diagnosing the problems you're running into and retrying.

Especially if you do the latter, you might consider submitting any solutions as bug/fix reports. See Chapter 18 [Known Causes of Trouble with GNU Fortran], page 265.

However, understand that many problems preventing a straight build from working are not g77 problems, and, in such cases, are not likely to be addressed in future versions of g77.

15.5.9 Pre-installation Checks

Before installing the system, which includes installing gcc, you might want to do some minimum checking to ensure that some basic things work.

Here are some commands you can try, and output typically printed by them when they work:

```
sh# cd /usr/src/gcc
sh# ./g77 --driver=./xgcc -B./ -v
g77 version 0.5.22
 ./xgcc -B./ -v -fnull-version -o /tmp/gfa18047 ...
Reading specs from ./specs
gcc version 2.7.2.3.f.2
 ./cpp -lang-c -v -isystem ./include -undef ...
GNU CPP version 2.7.2.3.f.2 (Linux/Alpha)
#include "..." search starts here:
#include <...> search starts here:
 ./include
 /usr/local/include
 /usr/alpha-unknown-linux/include
 /usr/lib/gcc-lib/alpha-unknown-linux/2.7.2.3.f.2/include
 /usr/include
End of search list.
 ./f771 /tmp/cca18048.i -fset-g77-defaults -quiet -dumpbase ...
GNU F77 version 2.7.2.3.f.2 (Linux/Alpha) compiled ...
GNU Fortran Front End version 0.5.22 compiled: ...
 as -nocpp -o /tmp/cca180481.o /tmp/cca18048.s
 ld -G 8 -O1 -o /tmp/gfa18047 /usr/lib/crt0.o -L. ...
__G77_LIBF77_VERSION__: 0.5.22
@(#)LIBF77 VERSION 19970404
__G77_LIBI77_VERSION__: 0.5.22
@(#) LIBI77 VERSION pjw,dmg-mods 19970816
__G77_LIBU77_VERSION__: 0.5.22
@(#) LIBU77 VERSION 19970609
sh# ./xgcc -B./ -v -o /tmp/delete-me -xc /dev/null -xnone
Reading specs from ./specs
gcc version 2.7.2.3.f.2
 ./cpp -lang-c -v -isystem ./include -undef ...
GNU CPP version 2.7.2.3.f.2 (Linux/Alpha)
```

```
#include "..." search starts here:
#include <...> search starts here:
 ./include
 /usr/local/include
 /usr/alpha-unknown-linux/include
 /usr/lib/gcc-lib/alpha-unknown-linux/2.7.2.3.f.2/include
 /usr/include
End of search list.
 ./cc1 /tmp/cca18063.i -quiet -dumpbase null.c -version ...
GNU C version 2.7.2.3.f.2 (Linux/Alpha) compiled ...
 as -nocpp -o /tmp/cca180631.o /tmp/cca18063.s
 ld -G 8 -O1 -o /tmp/delete-me /usr/lib/crt0.o -L. ...
/usr/lib/crt0.o: In function `__start':
crt0.S:110: undefined reference to `main'
/usr/lib/crt0.o(.lita+0x28): undefined reference to `main'
sh#
```

(Note that long lines have been truncated, and '...' used to indicate such truncations.)

The above two commands test whether `g77` and `gcc`, respectively, are able to compile empty (null) source files, whether invocation of the C preprocessor works, whether libraries can be linked, and so on.

If the output you get from either of the above two commands is noticeably different, especially if it is shorter or longer in ways that do not look consistent with the above sample output, you probably should not install `gcc` and `g77` until you have investigated further.

For example, you could try compiling actual applications and seeing how that works. (You might want to do that anyway, even if the above tests work.)

To compile using the not-yet-installed versions of `gcc` and `g77`, use the following commands to invoke them.

To invoke `g77`, type:

```
/usr/src/gcc/g77 --driver=/usr/src/gcc/xgcc -B/usr/src/gcc/ ...
```

To invoke `gcc`, type:

```
/usr/src/gcc/xgcc -B/usr/src/gcc/ ...
```

15.5.10 Installation of Binaries

After configuring, building, and testing `g77` and `gcc`, when you are ready to install them on your system, type:

```
make -k CC=gcc LANGUAGES=f77 install
```

As described in section "Installing GNU CC" in *Using and Porting GNU CC*, the values for the '`CC`' and '`LANGUAGES`' macros should be the same as those you supplied for the build itself.

So, the details of the above command might vary if you used a bootstrap build (where you might be able to omit both definitions, or might have to supply the same definitions you used when building the final stage) or if you deviated from the instructions for a straight build.

If the above command does not install '`libf2c.a`' as expected, try this:

```
make -k ... install install-libf77 install-f2c-all
```

We don't know why some non-GNU versions of `make` sometimes require this alternate command, but they do. (Remember to supply the appropriate definitions for '`CC`' and '`LANGUAGES`' where you see '...' in the above command.)

Note that using the '`-k`' option tells `make` to continue after some installation problems, like not having `makeinfo` installed on your system. It might not be necessary for your system.

15.5.11 Updating Your Info Directory

As part of installing `g77`, you should make sure users of `info` can easily access this manual on-line. Do this by making sure a line such as the following exists in '`/usr/info/dir`', or in whatever file is the top-level file in the `info` directory on your system (perhaps '`/usr/local/info/dir`':

```
* g77: (g77).            The GNU Fortran programming language.
```

If the menu in '`dir`' is organized into sections, `g77` probably belongs in a section with a name such as one of the following:

- Fortran Programming
- Writing Programs
- Programming Languages
- Languages Other Than C
- Scientific/Engineering Tools
- GNU Compilers

15.5.12 Missing `bison`?

If you cannot install `bison`, make sure you have started with a *fresh* distribution of `gcc`, do *not* do '`make maintainer-clean`' (in other versions of `gcc`, this was called '`make realclean`'), and, to ensure that `bison` is not invoked by `make` during the build, type these commands:

```
sh# cd gcc
sh# touch bi-parser.c bi-parser.h c-parse.c c-parse.h cexp.c
sh# touch cp/parse.c cp/parse.h objc-parse.c
sh#
```

These commands update the date-time-modified information for all the files produced by the various invocations of `bison` in the current versions of `gcc`, so that `make` no longer believes it needs to update them. All of these files should already exist in a `gcc` distribution, but the application of patches to upgrade to a newer version can leave the modification information set such that the `bison` input files look more "recent" than the corresponding output files.

Note: New versions of `gcc` might change the set of files it generates by invoking `bison`—if you cannot figure out for yourself how to handle such a situation, try an older version of `gcc` until you find someone who can (or until you obtain and install `bison`).

15.5.13 Missing `makeinfo`?

If you cannot install `makeinfo`, either use the `-k` option when invoking make to specify any of the '`install`' or related targets, or specify '`MAKEINFO=echo`' on the `make` command line.

If you fail to do one of these things, some files, like '`libf2c.a`', might not be installed, because the failed attempt by `make` to invoke `makeinfo` causes it to cancel any further processing.

15.6 Distributing Binaries

If you are building `g77` for distribution to others in binary form, first make sure you are aware of your legal responsibilities (read the file '`gcc/COPYING`' thoroughly).

Then, consider your target audience and decide where `g77` should be installed.

For systems like GNU/Linux that have no native Fortran compiler (or where `g77` could be considered the native compiler for Fortran and `gcc` for C, etc.), you should definitely configure `g77` for installation in '`/usr/bin`' instead of '`/usr/local/bin`'. Specify the '`--prefix=/usr`' option when running '`./configure`'. You might also want to set up the distribution so the `f77` command is a link to `g77`—just make an empty file named '`f77-install-ok`' in the source or build directory (the one in which the '`f`' directory resides, not the '`f`' directory itself) when you specify one of the '`install`' or '`uninstall`' targets in a `make` command.

For a system that might already have `f2c` installed, you definitely will want to make another empty file (in the same directory) named either '`f2c-exists-ok`' or '`f2c-install-ok`'. Use the former if you don't want your distribution to overwrite `f2c`-related files in existing systems; use the latter if you want to improve the likelihood that users will be able to use both `f2c` and `g77` to compile code for a single program without encountering link-time or run-time incompatibilities.

(Make sure you clearly document, in the "advertising" for your distribution, how installation of your distribution will affect existing installations of `gcc`, `f2c`, `f77`, '`libf2c.a`', and so on. Similarly, you should clearly document any requirements you assume are met by users of your distribution.)

For other systems with native `f77` (and `cc`) compilers, configure `g77` as you (or most of your audience) would configure `gcc` for their installations. Typically this is for installation in '`/usr/local`', and would not include a copy of `g77` named `f77`, so users could still use the native `f77`.

In any case, for `g77` to work properly, you **must** ensure that the binaries you distribute include:

'`bin/g77`' This is the command most users use to compile Fortran.

'`bin/gcc`' This is the command all users use to compile Fortran, either directly or indirectly via the `g77` command. The '`bin/gcc`' executable file must have been built from a `gcc` source tree into which a `g77` source tree was merged and configured, or it will not know how to compile Fortran programs.

'`bin/f77`' In installations with no non-GNU native Fortran compiler, this is the same as '`bin/g77`'. Otherwise, it should be omitted from the distribution, so the one on already on a particular system does not get overwritten.

'`info/g77.info*`'
This is the documentation for `g77`. If it is not included, users will have trouble understanding diagnostics messages and other such things, and will send you a lot of email asking questions.

Please edit this documentation (by editing '`gcc/f/*.tex`' and doing '`make doc`' from the '`/usr/src/gcc`' directory) to reflect any changes you've made to `g77`, or at least to encourage users of your binary distribution to report bugs to you first.

Also, whether you distribute binaries or install `g77` on your own system, it might be helpful for everyone to add a line listing this manual by name and topic to the top-level `info` node in '`/usr/info/dir`'. That way, users can find `g77` documentation more easily. See Section 15.5.11 [Updating Your Info Directory], page 235.

'`man/man1/g77.1`'
This is the short man page for `g77`. It is out of date, but you might as well include it for people who really like man pages.

'`man/man1/f77.1`'
In installations where `f77` is the same as `g77`, this is the same as '`man/man1/g77.1`'. Otherwise, it should be omitted from the distribution, so the one already on a particular system does not get overwritten.

'`lib/gcc-lib/.../f771`'
This is the actual Fortran compiler.

'`lib/gcc-lib/.../libf2c.a`'
This is the run-time library for `g77`-compiled programs.

Whether you want to include the slightly updated (and possibly improved) versions of `cc1`, `cc1plus`, and whatever other binaries get rebuilt with the changes the GNU Fortran distribution makes to the GNU back end, is up to you. These changes are highly unlikely to break any compilers, and it is possible they'll fix back-end bugs that can be demonstrated using front ends other than GNU Fortran's.

Please assure users that unless they have a specific need for their existing, older versions of `gcc` command, they are unlikely to experience any problems by overwriting it with your version—though they could certainly protect themselves by making backup copies first! Otherwise, users might try and install your binaries in a "safe" place, find they cannot compile Fortran programs with your distribution (because, perhaps, they're picking up their old version of the `gcc` command, which does not recognize Fortran programs), and assume that your binaries (or, more generally, GNU Fortran distributions in general) are broken, at least for their system.

Finally, **please** ask for bug reports to go to you first, at least until you're sure your distribution is widely used and has been well tested. This especially goes for those of you making any changes to the `g77` sources to port `g77`, e.g. to OS/2. `fortran@gnu.org` has

received a fair number of bug reports that turned out to be problems with other peoples' ports and distributions, about which nothing could be done for the user. Once you are quite certain a bug report does not involve your efforts, you can forward it to us.

16 Debugging and Interfacing

GNU Fortran currently generates code that is object-compatible with the `f2c` converter. Also, it avoids limitations in the current GBE, such as the inability to generate a procedure with multiple entry points, by generating code that is structured differently (in terms of procedure names, scopes, arguments, and so on) than might be expected.

As a result, writing code in other languages that calls on, is called by, or shares in-memory data with `g77`-compiled code generally requires some understanding of the way `g77` compiles code for various constructs.

Similarly, using a debugger to debug `g77`-compiled code, even if that debugger supports native Fortran debugging, generally requires this sort of information.

This section describes some of the basic information on how `g77` compiles code for constructs involving interfaces to other languages and to debuggers.

Caution: Much or all of this information pertains to only the current release of `g77`, sometimes even to using certain compiler options with `g77` (such as '`-fno-f2c`'). Do not write code that depends on this information without clearly marking said code as non-portable and subject to review for every new release of `g77`. This information is provided primarily to make debugging of code generated by this particular release of `g77` easier for the user, and partly to make writing (generally nonportable) interface code easier. Both of these activities require tracking changes in new version of `g77` as they are installed, because new versions can change the behaviors described in this section.

16.1 Main Program Unit (PROGRAM)

When `g77` compiles a main program unit, it gives it the public procedure name '`MAIN__`'. The `libf2c` library has the actual `main()` procedure as is typical of C-based environments, and it is this procedure that performs some initial start-up activity and then calls '`MAIN__`'.

Generally, `g77` and `libf2c` are designed so that you need not include a main program unit written in Fortran in your program—it can be written in C or some other language. Especially for I/O handling, this is the case, although `g77` version 0.5.16 includes a bug fix for `libf2c` that solved a problem with using the `OPEN` statement as the first Fortran I/O activity in a program without a Fortran main program unit.

However, if you don't intend to use `g77` (or `f2c`) to compile your main program unit—that is, if you intend to compile a `main()` procedure using some other language—you should carefully examine the code for `main()` in `libf2c`, found in the source file '`gcc/f/runtime/libF77/main.c`', to see what kinds of things might need to be done by your `main()` in order to provide the Fortran environment your Fortran code is expecting.

For example, `libf2c`'s `main()` sets up the information used by the `IARGC` and `GETARG` intrinsics. Bypassing `libf2c`'s `main()` without providing a substitute for this activity would mean that invoking `IARGC` and `GETARG` would produce undefined results.

When debugging, one implication of the fact that `main()`, which is the place where the debugged program "starts" from the debugger's point of view, is in `libf2c` is that you won't be starting your Fortran program at a point you recognize as your Fortran code.

The standard way to get around this problem is to set a break point (a one-time, or temporary, break point will do) at the entrance to '`MAIN__`', and then run the program. A convenient way to do so is to add the `gdb` command

```
tbreak MAIN__
```

to the file '`.gdbinit`' in the directory in which you're debugging (using `gdb`).

After doing this, the debugger will see the current execution point of the program as at the beginning of the main program unit of your program.

Of course, if you really want to set a break point at some other place in your program and just start the program running, without first breaking at '`MAIN__`', that should work fine.

16.2 Procedures (SUBROUTINE and FUNCTION)

Currently, `g77` passes arguments via reference—specifically, by passing a pointer to the location in memory of a variable, array, array element, a temporary location that holds the result of evaluating an expression, or a temporary or permanent location that holds the value of a constant.

Procedures that accept `CHARACTER` arguments are implemented by `g77` so that each `CHARACTER` argument has two actual arguments.

The first argument occupies the expected position in the argument list and has the user-specified name. This argument is a pointer to an array of characters, passed by the caller.

The second argument is appended to the end of the user-specified calling sequence and is named '`__g77_length_`*x*', where *x* is the user-specified name. This argument is of the C type `ftnlen` (see '`gcc/f/runtime/f2c.h.in`' for information on that type) and is the number of characters the caller has allocated in the array pointed to by the first argument.

A procedure will ignore the length argument if '`X`' is not declared `CHARACTER*(*)`, because for other declarations, it knows the length. Not all callers necessarily "know" this, however, which is why they all pass the extra argument.

The contents of the `CHARACTER` argument are specified by the address passed in the first argument (named after it). The procedure can read or write these contents as appropriate.

When more than one `CHARACTER` argument is present in the argument list, the length arguments are appended in the order the original arguments appear. So '`CALL FOO('HI','THERE')`' is implemented in C as '`foo("hi","there",2,5);`', ignoring the fact that `g77` does not provide the trailing null bytes on the constant strings (`f2c` does provide them, but they are unnecessary in a Fortran environment, and you should not expect them to be there).

Note that the above information applies to `CHARACTER` variables and arrays **only**. It does **not** apply to external `CHARACTER` functions or to intrinsic `CHARACTER` functions. That is, no second length argument is passed to '`FOO`' in this case:

```
CHARACTER X
EXTERNAL X
CALL FOO(X)
```

Nor does '`FOO`' expect such an argument in this case:

```
SUBROUTINE FOO(X)
CHARACTER X
EXTERNAL X
```

Because of this implementation detail, if a program has a bug such that there is disagreement as to whether an argument is a procedure, and the type of the argument is `CHARACTER`, subtle symptoms might appear.

16.3 Functions (FUNCTION and RETURN)

`g77` handles in a special way functions that return the following types:

- `CHARACTER`
- `COMPLEX`
- `REAL(KIND=1)`

For `CHARACTER`, `g77` implements a subroutine (a C function returning `void`) with two arguments prepended: '`__g77_result`', which the caller passes as a pointer to a `char` array expected to hold the return value, and '`__g77_length`', which the caller passes as an `ftnlen` value specifying the length of the return value as declared in the calling program. For `CHARACTER*(*)`, the called function uses '`__g77_length`' to determine the size of the array that '`__g77_result`' points to; otherwise, it ignores that argument.

For `COMPLEX`, when '`-ff2c`' is in force, `g77` implements a subroutine with one argument prepended: '`__g77_result`', which the caller passes as a pointer to a variable of the type of the function. The called function writes the return value into this variable instead of returning it as a function value. When '`-fno-f2c`' is in force, `g77` implements a `COMPLEX` function as `gcc`'s '`__complex__ float`' or '`__complex__ double`' function (or an emulation thereof, when '`-femulate-complex`' is in effect), returning the result of the function in the same way as `gcc` would.

For `REAL(KIND=1)`, when '`-ff2c`' is in force, `g77` implements a function that actually returns `REAL(KIND=2)` (typically C's `double` type). When '`-fno-f2c`' is in force, `REAL(KIND=1)` functions return `float`.

16.4 Names

Fortran permits each implementation to decide how to represent names as far as how they're seen in other contexts, such as debuggers and when interfacing to other languages, and especially as far as how casing is handled.

External names—names of entities that are public, or "accessible", to all modules in a program—normally have an underscore ('`_`') appended by `g77`, to generate code that is compatible with f2c. External names include names of Fortran things like common blocks, external procedures (subroutines and functions, but not including statement functions, which are internal procedures), and entry point names.

However, use of the '`-fno-underscoring`' option disables this kind of transformation of external names (though inhibiting the transformation certainly improves the chances of colliding with incompatible externals written in other languages—but that might be intentional.

When '`-funderscoring`' is in force, any name (external or local) that already has at least one underscore in it is implemented by `g77` by appending two underscores. (This second underscore can be disabled via the '`-fno-second-underscore`' option.) External names are changed this way for `f2c` compatibility. Local names are changed this way to avoid collisions with external names that are different in the source code—`f2c` does the same thing, but there's no compatibility issue there except for user expectations while debugging.

For example:

```
Max_Cost = 0
```

Here, a user would, in the debugger, refer to this variable using the name '`max_cost__`' (or '`MAX_COST__`' or '`Max_Cost__`', as described below). (We hope to improve `g77` in this regard in the future—don't write scripts depending on this behavior! Also, consider experimenting with the '`-fno-underscoring`' option to try out debugging without having to massage names by hand like this.)

`g77` provides a number of command-line options that allow the user to control how case mapping is handled for source files. The default is the traditional UNIX model for Fortran compilers—names are mapped to lower case. Other command-line options can be specified to map names to upper case, or to leave them exactly as written in the source file.

For example:

```
Foo = 9.436
```

Here, it is normally the case that the variable assigned will be named '`foo`'. This would be the name to enter when using a debugger to access the variable.

However, depending on the command-line options specified, the name implemented by `g77` might instead be '`FOO`' or even '`Foo`', thus affecting how debugging is done.

Also:

```
Call Foo
```

This would normally call a procedure that, if it were in a separate C program, be defined starting with the line:

```
void foo_()
```

However, `g77` command-line options could be used to change the casing of names, resulting in the name '`FOO_`' or '`Foo_`' being given to the procedure instead of '`foo_`', and the '`-fno-underscoring`' option could be used to inhibit the appending of the underscore to the name.

16.5 Common Blocks (COMMON)

`g77` names and lays out `COMMON` areas the same way f2c does, for compatibility with f2c.

Currently, `g77` does not emit "true" debugging information for members of a `COMMON` area, due to an apparent bug in the GBE.

(As of Version 0.5.19, `g77` emits debugging information for such members in the form of a constant string specifying the base name of the aggregate area and the offset of the member in bytes from the start of the area. Use the '`-fdebug-kludge`' option to enable this behavior. In `gdb`, use '`set language c`' before printing the value of the member, then

'`set language fortran`' to restore the default language, since `gdb` doesn't provide a way to print a readable version of a character string in Fortran language mode.

This kludge will be removed in a future version of `g77` that, in conjunction with a contemporary version of `gdb`, properly supports Fortran-language debugging, including access to members of `COMMON` areas.)

See Section 7.10 [Options for Code Generation Conventions], page 41, for information on the '`-fdebug-kludge`' option.

Moreover, `g77` currently implements a `COMMON` area such that its type is an array of the C `char` data type.

So, when debugging, you must know the offset into a `COMMON` area for a particular item in that area, and you have to take into account the appropriate multiplier for the respective sizes of the types (as declared in your code) for the items preceding the item in question as compared to the size of the `char` type.

For example, using default implicit typing, the statement

```
COMMON I(15), R(20), T
```

results in a public 144-byte `char` array named '`_BLNK__`' with '`I`' placed at '`_BLNK__[0]`', '`R`' at '`_BLNK__[60]`', and '`T`' at '`_BLNK__[140]`'. (This is assuming that the target machine for the compilation has 4-byte `INTEGER(KIND=1)` and `REAL(KIND=1)` types.)

16.6 Local Equivalence Areas (EQUIVALENCE)

`g77` treats storage-associated areas involving a `COMMON` block as explained in the section on common blocks.

A local `EQUIVALENCE` area is a collection of variables and arrays connected to each other in any way via `EQUIVALENCE`, none of which are listed in a `COMMON` statement.

Currently, `g77` does not emit "true" debugging information for members in a local `EQUIVALENCE` area, due to an apparent bug in the GBE.

(As of Version 0.5.19, `g77` does emit debugging information for such members in the form of a constant string specifying the base name of the aggregate area and the offset of the member in bytes from the start of the area. Use the '`-fdebug-kludge`' option to enable this behavior. In `gdb`, use '`set language c`' before printing the value of the member, then '`set language fortran`' to restore the default language, since `gdb` doesn't provide a way to print a readable version of a character string in Fortran language mode.

This kludge will be removed in a future version of `g77` that, in conjunction with a contemporary version of `gdb`, properly supports Fortran-language debugging, including access to members of `EQUIVALENCE` areas.)

See Section 7.10 [Options for Code Generation Conventions], page 41, for information on the '`-fdebug-kludge`' option.

Moreover, `g77` implements a local `EQUIVALENCE` area such that its type is an array of the C `char` data type.

The name `g77` gives this array of `char` type is '`__g77_equiv_`*x*', where *x* is the name of the item that is placed at the beginning (offset 0) of this array. If more than one such

item is placed at the beginning, *x* is the name that sorts to the top in an alphabetical sort of the list of such items.

When debugging, you must therefore access members of `EQUIVALENCE` areas by specifying the appropriate '`__g77_equiv_x`' array section with the appropriate offset. See the explanation of debugging `COMMON` blocks for info applicable to debugging local `EQUIVALENCE` areas.

(*Note:* `g77` version 0.5.18 and earlier chose the name for *x* using a different method when more than one name was in the list of names of entities placed at the beginning of the array. Though the documentation specified that the first name listed in the `EQUIVALENCE` statements was chosen for *x*, `g77` in fact chose the name using a method that was so complicated, it seemed easier to change it to an alphabetical sort than to describe the previous method in the documentation.)

16.7 Complex Variables (COMPLEX)

As of 0.5.20, `g77` defaults to handling `COMPLEX` types (and related intrinsics, constants, functions, and so on) in a manner that makes direct debugging involving these types in Fortran language mode difficult.

Essentially, `g77` implements these types using an internal construct similar to C's `struct`, at least as seen by the `gcc` back end.

Currently, the back end, when outputting debugging info with the compiled code for the assembler to digest, does not detect these `struct` types as being substitutes for Fortran complex. As a result, the Fortran language modes of debuggers such as `gdb` see these types as C `struct` types, which they might or might not support.

Until this is fixed, switch to C language mode to work with entities of `COMPLEX` type and then switch back to Fortran language mode afterward. (In `gdb`, this is accomplished via '`set lang c`' and either '`set lang fortran`' or '`set lang auto`'.)

Note: Compiling with the '`-fno-emulate-complex`' option avoids the debugging problem, but is known to cause other problems like compiler crashes and generation of incorrect code, so it is not recommended.

16.8 Arrays (DIMENSION)

Fortran uses "column-major ordering" in its arrays. This differs from other languages, such as C, which use "row-major ordering". The difference is that, with Fortran, array elements adjacent to each other in memory differ in the *first* subscript instead of the last; '`A(5,10,20)`' immediately follows '`A(4,10,20)`', whereas with row-major ordering it would follow '`A(5,10,19)`'.

This consideration affects not only interfacing with and debugging Fortran code, it can greatly affect how code is designed and written, especially when code speed and size is a concern.

Fortran also differs from C, a popular language for interfacing and to support directly in debuggers, in the way arrays are treated. In C, arrays are single-dimensional and have interesting relationships to pointers, neither of which is true for Fortran. As a result, dealing with Fortran arrays from within an environment limited to C concepts can be challenging.

For example, accessing the array element '`A(5,10,20)`' is easy enough in Fortran (use '`A(5,10,20)`'), but in C some difficult machinations are needed. First, C would treat the A array as a single-dimension array. Second, C does not understand low bounds for arrays as does Fortran. Third, C assumes a low bound of zero (0), while Fortran defaults to a low bound of one (1) and can supports an arbitrary low bound. Therefore, calculations must be done to determine what the C equivalent of '`A(5,10,20)`' would be, and these calculations require knowing the dimensions of '`A`'.

For '`DIMENSION A(2:11,21,0:29)`', the calculation of the offset of '`A(5,10,20)`' would be:

```
   (5-2)
+ (10-1)*(11-2+1)
+ (20-0)*(11-2+1)*(21-1+1)
= 4293
```

So the C equivalent in this case would be '`a[4293]`'.

When using a debugger directly on Fortran code, the C equivalent might not work, because some debuggers cannot understand the notion of low bounds other than zero. However, unlike `f2c`, `g77` does inform the GBE that a multi-dimensional array (like '`A`' in the above example) is really multi-dimensional, rather than a single-dimensional array, so at least the dimensionality of the array is preserved.

Debuggers that understand Fortran should have no trouble with non-zero low bounds, but for non-Fortran debuggers, especially C debuggers, the above example might have a C equivalent of '`a[4305]`'. This calculation is arrived at by eliminating the subtraction of the lower bound in the first parenthesized expression on each line—that is, for '`(5-2)`' substitute '`(5)`', for '`(10-1)`' substitute '`(10)`', and for '`(20-0)`' substitute '`(20)`'. Actually, the implication of this can be that the expression '`*(&a[2][1][0] + 4293)`' works fine, but that '`a[20][10][5]`' produces the equivalent of '`*(&a[0][0][0] + 4305)`' because of the missing lower bounds.

Come to think of it, perhaps the behavior is due to the debugger internally compensating for the lower bounds by offsetting the base address of '`a`', leaving '`&a`' set lower, in this case, than '`&a[2][1][0]`' (the address of its first element as identified by subscripts equal to the corresponding lower bounds).

You know, maybe nobody really needs to use arrays.

16.9 Adjustable Arrays (DIMENSION)

Adjustable and automatic arrays in Fortran require the implementation (in this case, the `g77` compiler) to "memorize" the expressions that dimension the arrays each time the procedure is invoked. This is so that subsequent changes to variables used in those expressions, made during execution of the procedure, do not have any effect on the dimensions of those arrays.

For example:

```
REAL ARRAY(5)
DATA ARRAY/5*2/
CALL X(ARRAY, 5)
```

```
      END
      SUBROUTINE X(A, N)
      DIMENSION A(N)
      N = 20
      PRINT *, N, A
      END
```

Here, the implementation should, when running the program, print something like:

```
      20   2.  2.  2.  2.  2.
```

Note that this shows that while the value of 'N' was successfully changed, the size of the 'A' array remained at 5 elements.

To support this, g77 generates code that executes before any user code (and before the internally generated computed GOTO to handle alternate entry points, as described below) that evaluates each (nonconstant) expression in the list of subscripts for an array, and saves the result of each such evaluation to be used when determining the size of the array (instead of re-evaluating the expressions).

So, in the above example, when 'X' is first invoked, code is executed that copies the value of 'N' to a temporary. And that same temporary serves as the actual high bound for the single dimension of the 'A' array (the low bound being the constant 1). Since the user program cannot (legitimately) change the value of the temporary during execution of the procedure, the size of the array remains constant during each invocation.

For alternate entry points, the code g77 generates takes into account the possibility that a dummy adjustable array is not actually passed to the actual entry point being invoked at that time. In that case, the public procedure implementing the entry point passes to the master private procedure implementing all the code for the entry points a NULL pointer where a pointer to that adjustable array would be expected. The g77-generated code doesn't attempt to evaluate any of the expressions in the subscripts for an array if the pointer to that array is NULL at run time in such cases. (Don't depend on this particular implementation by writing code that purposely passes NULL pointers where the callee expects adjustable arrays, even if you know the callee won't reference the arrays—nor should you pass NULL pointers for any dummy arguments used in calculating the bounds of such arrays or leave undefined any values used for that purpose in COMMON—because the way g77 implements these things might change in the future!)

16.10 Alternate Entry Points (ENTRY)

The GBE does not understand the general concept of alternate entry points as Fortran provides via the ENTRY statement. g77 gets around this by using an approach to compiling procedures having at least one ENTRY statement that is almost identical to the approach used by f2c. (An alternate approach could be used that would probably generate faster, but larger, code that would also be a bit easier to debug.)

Information on how g77 implements ENTRY is provided for those trying to debug such code. The choice of implementation seems unlikely to affect code (compiled in other languages) that interfaces to such code.

g77 compiles exactly one public procedure for the primary entry point of a procedure plus each ENTRY point it specifies, as usual. That is, in terms of the public interface, there is no difference between

```
SUBROUTINE X
END
SUBROUTINE Y
END
```

and:

```
SUBROUTINE X
ENTRY Y
END
```

The difference between the above two cases lies in the code compiled for the 'X' and 'Y' procedures themselves, plus the fact that, for the second case, an extra internal procedure is compiled.

For every Fortran procedure with at least one ENTRY statement, g77 compiles an extra procedure named '__g77_masterfun_*x*', where *x* is the name of the primary entry point (which, in the above case, using the standard compiler options, would be 'x_' in C).

This extra procedure is compiled as a private procedure—that is, a procedure not accessible by name to separately compiled modules. It contains all the code in the program unit, including the code for the primary entry point plus for every entry point. (The code for each public procedure is quite short, and explained later.)

The extra procedure has some other interesting characteristics.

The argument list for this procedure is invented by g77. It contains a single integer argument named '__g77_which_entrypoint', passed by value (as in Fortran's '%VAL()' intrinsic), specifying the entry point index—0 for the primary entry point, 1 for the first entry point (the first ENTRY statement encountered), 2 for the second entry point, and so on.

It also contains, for functions returning CHARACTER and (when '-ff2c' is in effect) COMPLEX functions, and for functions returning different types among the ENTRY statements (e.g. 'REAL FUNCTION R()' containing 'ENTRY I()'), an argument named '__g77_result' that is expected at run time to contain a pointer to where to store the result of the entry point. For CHARACTER functions, this storage area is an array of the appropriate number of characters; for COMPLEX functions, it is the appropriate area for the return type; for multiple-return-type functions, it is a union of all the supported return types (which cannot include CHARACTER, since combining CHARACTER and non-CHARACTER return types via ENTRY in a single function is not supported by g77).

For CHARACTER functions, the '__g77_result' argument is followed by yet another argument named '__g77_length' that, at run time, specifies the caller's expected length of the returned value. Note that only CHARACTER*(*) functions and entry points actually make use of this argument, even though it is always passed by all callers of public CHARACTER functions (since the caller does not generally know whether such a function is CHARACTER*(*) or whether there are any other callers that don't have that information).

The rest of the argument list is the union of all the arguments specified for all the entry points (in their usual forms, e.g. CHARACTER arguments have extra length arguments, all appended at the end of this list). This is considered the "master list" of arguments.

The code for this procedure has, before the code for the first executable statement, code much like that for the following Fortran statement:

```
      GOTO (100000,100001,100002), __g77_which_entrypoint
100000 ...code for primary entry point...
100001 ...code immediately following first ENTRY statement...
100002 ...code immediately following second ENTRY statement...
```

(Note that invalid Fortran statement labels and variable names are used in the above example to highlight the fact that it represents code generated by the `g77` internals, not code to be written by the user.)

It is this code that, when the procedure is called, picks which entry point to start executing.

Getting back to the public procedures ('`X`' and '`Y`' in the original example), those procedures are fairly simple. Their interfaces are just like they would be if they were self-contained procedures (without `ENTRY`), of course, since that is what the callers expect. Their code consists of simply calling the private procedure, described above, with the appropriate extra arguments (the entry point index, and perhaps a pointer to a multiple-type- return variable, local to the public procedure, that contains all the supported returnable non-character types). For arguments that are not listed for a given entry point that are listed for other entry points, and therefore that are in the "master list" for the private procedure, null pointers (in C, the `NULL` macro) are passed. Also, for entry points that are part of a multiple-type-returning function, code is compiled after the call of the private procedure to extract from the multi-type union the appropriate result, depending on the type of the entry point in question, returning that result to the original caller.

When debugging a procedure containing alternate entry points, you can either set a break point on the public procedure itself (e.g. a break point on '`X`' or '`Y`') or on the private procedure that contains most of the pertinent code (e.g. '`__g77_masterfun_`*x*'). If you do the former, you should use the debugger's command to "step into" the called procedure to get to the actual code; with the latter approach, the break point leaves you right at the actual code, skipping over the public entry point and its call to the private procedure (unless you have set a break point there as well, of course).

Further, the list of dummy arguments that is visible when the private procedure is active is going to be the expanded version of the list for whichever particular entry point is active, as explained above, and the way in which return values are handled might well be different from how they would be handled for an equivalent single-entry function.

16.11 Alternate Returns (SUBROUTINE and RETURN)

Subroutines with alternate returns (e.g. '`SUBROUTINE X(*)`' and '`CALL X(*50)`') are implemented by `g77` as functions returning the C `int` type. The actual alternate-return arguments are omitted from the calling sequence. Instead, the caller uses the return value to do a rough equivalent of the Fortran computed-`GOTO` statement, as in '`GOTO (50), X()`' in the example above (where '`X`' is quietly declared as an `INTEGER(KIND=1)` function), and the callee just returns whatever integer is specified in the `RETURN` statement for the subroutine For example, '`RETURN 1`' is implemented as '`X = 1`' followed by '`RETURN`' in C, and '`RETURN`' by itself is '`X = 0`' and '`RETURN`').

16.12 Assigned Statement Labels (ASSIGN and GOTO)

For portability to machines where a pointer (such as to a label, which is how `g77` implements `ASSIGN` and its relatives, the assigned-`GOTO` and assigned-`FORMAT`-I/O statements) is wider (bitwise) than an `INTEGER(KIND=1)`, `g77` uses a different memory location to hold the `ASSIGN`ed value of a variable than it does the numerical value in that variable, unless the variable is wide enough (can hold enough bits).

In particular, while `g77` implements

```
I = 10
```

as, in C notation, '`i = 10;`', it implements

```
ASSIGN 10 TO I
```

as, in GNU's extended C notation (for the label syntax), '`__g77_ASSIGN_I = &&L10;`' (where '`L10`' is just a massaging of the Fortran label '`10`' to make the syntax C-like; `g77` doesn't actually generate the name '`L10`' or any other name like that, since debuggers cannot access labels anyway).

While this currently means that an `ASSIGN` statement does not overwrite the numeric contents of its target variable, *do not* write any code depending on this feature. `g77` has already changed this implementation across versions and might do so in the future. This information is provided only to make debugging Fortran programs compiled with the current version of `g77` somewhat easier. If there's no debugger-visible variable named '`__g77_ASSIGN_I`' in a program unit that does '`ASSIGN 10 TO I`', that probably means `g77` has decided it can store the pointer to the label directly into '`I`' itself.

See Section 11.9.7 [Ugly Assigned Labels], page 181, for information on a command-line option to force `g77` to use the same storage for both normal and assigned-label uses of a variable.

16.13 Run-time Library Errors

The `libf2c` library currently has the following table to relate error code numbers, returned in `IOSTAT=` variables, to messages. This information should, in future versions of this document, be expanded upon to include detailed descriptions of each message.

In line with good coding practices, any of the numbers in the list below should *not* be directly written into Fortran code you write. Instead, make a separate `INCLUDE` file that defines `PARAMETER` names for them, and use those in your code, so you can more easily change the actual numbers in the future.

The information below is culled from the definition of '`F_err`' in '`f/runtime/libI77/err.c`' in the `g77` source tree.

```
100: "error in format"
101: "illegal unit number"
102: "formatted io not allowed"
103: "unformatted io not allowed"
104: "direct io not allowed"
105: "sequential io not allowed"
106: "can't backspace file"
107: "null file name"
```

```
108: "can't stat file"
109: "unit not connected"
110: "off end of record"
111: "truncation failed in endfile"
112: "incomprehensible list input"
113: "out of free space"
114: "unit not connected"
115: "read unexpected character"
116: "bad logical input field"
117: "bad variable type"
118: "bad namelist name"
119: "variable not in namelist"
120: "no end record"
121: "variable count incorrect"
122: "subscript for scalar variable"
123: "invalid array section"
124: "substring out of bounds"
125: "subscript out of bounds"
126: "can't read file"
127: "can't write file"
128: "'new' file exists"
129: "can't append to file"
130: "non-positive record number"
131: "I/O started while already doing I/O"
```

17 Collected Fortran Wisdom

Most users of g77 can be divided into two camps:

- Those writing new Fortran code to be compiled by g77.
- Those using g77 to compile existing, "legacy" code.

Users writing new code generally understand most of the necessary aspects of Fortran to write "mainstream" code, but often need help deciding how to handle problems, such as the construction of libraries containing BLOCK DATA.

Users dealing with "legacy" code sometimes don't have much experience with Fortran, but believe that the code they're compiling already works when compiled by other compilers (and might not understand why, as is sometimes the case, it doesn't work when compiled by g77).

The following information is designed to help users do a better job coping with existing, "legacy" Fortran code, and with writing new code as well.

17.1 Advantages Over f2c

Without f2c, g77 would have taken much longer to do and probably not been as good for quite a while. Sometimes people who notice how much g77 depends on, and documents encouragement to use, f2c ask why g77 was created if f2c already existed.

This section gives some basic answers to these questions, though it is not intended to be comprehensive.

17.1.1 Language Extensions

g77 offers several extensions to the Fortran language that f2c doesn't.

However, f2c offers a few that g77 doesn't, like fairly complete support for INTEGER*2. It is expected that g77 will offer some or all of these missing features at some time in the future. (Version 0.5.18 of g77 offers some rudimentary support for some of these features.)

17.1.2 Compiler Options

g77 offers a whole bunch of compiler options that f2c doesn't.

However, f2c offers a few that g77 doesn't, like an option to generate code to check array subscripts at run time. It is expected that g77 will offer some or all of these missing options at some time in the future.

17.1.3 Compiler Speed

Saving the steps of writing and then rereading C code is a big reason why g77 should be able to compile code much faster than using f2c in conjunction with the equivalent invocation of gcc.

However, due to g77's youth, lots of self-checking is still being performed. As a result, this improvement is as yet unrealized (though the potential seems to be there for quite a big speedup in the future). It is possible that, as of version 0.5.18, g77 is noticeably faster compiling many Fortran source files than using f2c in conjunction with gcc.

17.1.4 Program Speed

`g77` has the potential to better optimize code than `f2c`, even when `gcc` is used to compile the output of `f2c`, because `f2c` must necessarily translate Fortran into a somewhat lower-level language (C) that cannot preserve all the information that is potentially useful for optimization, while `g77` can gather, preserve, and transmit that information directly to the GBE.

For example, `g77` implements `ASSIGN` and assigned `GOTO` using direct assignment of pointers to labels and direct jumps to labels, whereas `f2c` maps the assigned labels to integer values and then uses a C `switch` statement to encode the assigned `GOTO` statements.

However, as is typical, theory and reality don't quite match, at least not in all cases, so it is still the case that `f2c` plus `gcc` can generate code that is faster than `g77`.

Version 0.5.18 of `g77` offered default settings and options, via patches to the `gcc` back end, that allow for better program speed, though some of these improvements also affected the performance of programs translated by `f2c` and then compiled by `g77`'s version of `gcc`.

Version 0.5.20 of `g77` offers further performance improvements, at least one of which (alias analysis) is not generally applicable to `f2c` (though `f2c` could presumably be changed to also take advantage of this new capability of the `gcc` back end, assuming this is made available in an upcoming release of `gcc`).

17.1.5 Ease of Debugging

Because `g77` compiles directly to assembler code like `gcc`, instead of translating to an intermediate language (C) as does `f2c`, support for debugging can be better for `g77` than `f2c`.

However, although `g77` might be somewhat more "native" in terms of debugging support than `f2c` plus `gcc`, there still are a lot of things "not quite right". Many of the important ones should be resolved in the near future.

For example, `g77` doesn't have to worry about reserved names like `f2c` does. Given '`FOR = WHILE`', `f2c` must necessarily translate this to something *other* than '`for = while;`', because C reserves those words.

However, `g77` does still uses things like an extra level of indirection for `ENTRY`-laden procedures—in this case, because the back end doesn't yet support multiple entry points.

Another example is that, given

```
COMMON A, B
EQUIVALENCE (B, C)
```

the `g77` user should be able to access the variables directly, by name, without having to traverse C-like structures and unions, while `f2c` is unlikely to ever offer this ability (due to limitations in the C language).

However, due to apparent bugs in the back end, `g77` currently doesn't take advantage of this facility at all—it doesn't emit any debugging information for `COMMON` and `EQUIVALENCE` areas, other than information on the array of `char` it creates (and, in the case of local `EQUIVALENCE`, names) for each such area.

Yet another example is arrays. g77 represents them to the debugger using the same "dimensionality" as in the source code, while f2c must necessarily convert them all to one-dimensional arrays to fit into the confines of the C language. However, the level of support offered by debuggers for interactive Fortran-style access to arrays as compiled by g77 can vary widely. In some cases, it can actually be an advantage that f2c converts everything to widely supported C semantics.

In fairness, g77 could do many of the things f2c does to get things working at least as well as f2c—for now, the developers prefer making g77 work the way they think it is supposed to, and finding help improving the other products (the back end of gcc; gdb; and so on) to get things working properly.

17.1.6 Character and Hollerith Constants

To avoid the extensive hassle that would be needed to avoid this, f2c uses C character constants to encode character and Hollerith constants. That means a constant like '`''HELLO''`' is translated to '`"hello"`' in C, which further means that an extra null byte is present at the end of the constant. This null byte is superfluous.

g77 does not generate such null bytes. This represents significant savings of resources, such as on systems where '`/dev/null`' or '`/dev/zero`' represent bottlenecks in the systems' performance, because g77 simply asks for fewer zeros from the operating system than f2c.

17.2 Block Data and Libraries

To ensure that block data program units are linked, especially a concern when they are put into libraries, give each one a name (as in '`BLOCK DATA FOO`') and make sure there is an '`EXTERNAL FOO`' statement in every program unit that uses any common block initialized by the corresponding `BLOCK DATA`. g77 currently compiles a `BLOCK DATA` as if it were a `SUBROUTINE`, that is, it generates an actual procedure having the appropriate name. The procedure does nothing but return immediately if it happens to be called. For '`EXTERNAL FOO`', where '`FOO`' is not otherwise referenced in the same program unit, g77 assumes there exists a '`BLOCK DATA FOO`' in the program and ensures that by generating a reference to it so the linker will make sure it is present. (Specifically, g77 outputs in the data section a static pointer to the external name '`FOO`'.)

The implementation g77 currently uses to make this work is one of the few things not compatible with f2c as currently shipped. f2c currently does nothing with '`EXTERNAL FOO`' except issue a warning that '`FOO`' is not otherwise referenced, and for '`BLOCK DATA FOO`', f2c doesn't generate a dummy procedure with the name '`FOO`'. The upshot is that you shouldn't mix f2c and g77 in this particular case. If you use f2c to compile '`BLOCK DATA FOO`', then any g77-compiled program unit that says '`EXTERNAL FOO`' will result in an unresolved reference when linked. If you do the opposite, then '`FOO`' might not be linked in under various circumstances (such as when '`FOO`' is in a library, or you're using a "clever" linker—so clever, it produces a broken program with little or no warning by omitting initializations of global data because they are contained in unreferenced procedures).

The changes you make to your code to make g77 handle this situation, however, appear to be a widely portable way to handle it. That is, many systems permit it (as they should, since

the FORTRAN 77 standard permits '`EXTERNAL FOO`' when '`FOO`' is a block data program unit), and of the ones that might not link '`BLOCK DATA FOO`' under some circumstances, most of them appear to do so once '`EXTERNAL FOO`' is present in the appropriate program units.

Here is the recommended approach to modifying a program containing a program unit such as the following:

```
BLOCK DATA FOO
COMMON /VARS/ X, Y, Z
DATA X, Y, Z / 3., 4., 5. /
END
```

If the above program unit might be placed in a library module, then ensure that every program unit in every program that references that particular `COMMON` area uses the `EXTERNAL` statement to force the area to be initialized.

For example, change a program unit that starts with

```
INTEGER FUNCTION CURX()
COMMON /VARS/ X, Y, Z
CURX = X
END
```

so that it uses the `EXTERNAL` statement, as in:

```
INTEGER FUNCTION CURX()
COMMON /VARS/ X, Y, Z
EXTERNAL FOO
CURX = X
END
```

That way, '`CURX`' is compiled by `g77` (and many other compilers) so that the linker knows it must include '`FOO`', the `BLOCK DATA` program unit that sets the initial values for the variables in '`VAR`', in the executable program.

17.3 Loops

The meaning of a `DO` loop in Fortran is precisely specified in the Fortran standard...and is quite different from what many programmers might expect.

In particular, Fortran `DO` loops are implemented as if the number of trips through the loop is calculated *before* the loop is entered.

The number of trips for a loop is calculated from the *start*, *end*, and *increment* values specified in a statement such as:

```
DO iter = start, end, increment
```

The trip count is evaluated using a fairly simple formula based on the three values following the '`=`' in the statement, and it is that trip count that is effectively decremented during each iteration of the loop. If, at the beginning of an iteration of the loop, the trip count is zero or negative, the loop terminates. The per-loop-iteration modifications to *iter* are not related to determining whether to terminate the loop.

There are two important things to remember about the trip count:

- It can be *negative*, in which case it is treated as if it was zero—meaning the loop is not executed at all.

- The type used to *calculate* the trip count is the same type as *iter*, but the final calculation, and thus the type of the trip count itself, always is `INTEGER(KIND=1)`.

These two items mean that there are loops that cannot be written in straightforward fashion using the Fortran `DO`.

For example, on a system with the canonical 32-bit two's-complement implementation of `INTEGER(KIND=1)`, the following loop will not work:

```
DO I = -2000000000, 2000000000
```

Although the *start* and *end* values are well within the range of `INTEGER(KIND=1)`, the *trip count* is not. The expected trip count is 40000000001, which is outside the range of `INTEGER(KIND=1)` on many systems.

Instead, the above loop should be constructed this way:

```
I = -2000000000
DO
  IF (I .GT. 2000000000) EXIT
  ...
  I = I + 1
END DO
```

The simple `DO` construct and the `EXIT` statement (used to leave the innermost loop) are F90 features that `g77` supports.

Some Fortran compilers have buggy implementations of `DO`, in that they don't follow the standard. They implement `DO` as a straightforward translation to what, in C, would be a `for` statement. Instead of creating a temporary variable to hold the trip count as calculated at run time, these compilers use the iteration variable *iter* to control whether the loop continues at each iteration.

The bug in such an implementation shows up when the trip count is within the range of the type of *iter*, but the magnitude of '`ABS(`*end*`) + ABS(`*incr*`)`' exceeds that range. For example:

```
DO I = 2147483600, 2147483647
```

A loop started by the above statement will work as implemented by `g77`, but the use, by some compilers, of a more C-like implementation akin to

```
for (i = 2147483600; i <= 2147483647; ++i)
```

produces a loop that does not terminate, because '`i`' can never be greater than 2147483647, since incrementing it beyond that value overflows '`i`', setting it to -2147483648. This is a large, negative number that still is less than 2147483647.

Another example of unexpected behavior of `DO` involves using a nonintegral iteration variable *iter*, that is, a `REAL` variable. Consider the following program:

```
      DATA BEGIN, END, STEP /.1, .31, .007/
      DO 10 R = BEGIN, END, STEP
         IF (R .GT. END) PRINT *, R, ' .GT. ', END, '!!'
         PRINT *,R
10    CONTINUE
      PRINT *,'LAST = ',R
      IF (R .LE. END) PRINT *, R, ' .LE. ', END, '!!'
      END
```

A C-like view of DO would hold that the two "exclamatory" PRINT statements are never executed. However, this is the output of running the above program as compiled by g77 on a GNU/Linux ix86 system:

```
 .100000001
 .107000001
 .114
 .120999999
...
 .289000005
 .296000004
 .303000003
LAST =   .310000002
 .310000002 .LE.   .310000002!!
```

Note that one of the two checks in the program turned up an apparent violation of the programmer's expectation—yet, the loop is correctly implemented by g77, in that it has 30 iterations. This trip count of 30 is correct when evaluated using the floating-point representations for the *begin*, *end*, and *incr* values (.1, .31, .007) on GNU/Linux ix86 are used. On other systems, an apparently more accurate trip count of 31 might result, but, nevertheless, g77 is faithfully following the Fortran standard, and the result is not what the author of the sample program above apparently expected. (Such other systems might, for different values in the DATA statement, violate the other programmer's expectation, for example.)

Due to this combination of imprecise representation of floating-point values and the often-misunderstood interpretation of DO by standard-conforming compilers such as g77, use of DO loops with REAL iteration variables is not recommended. Such use can be caught by specifying '-Wsurprising'. See Section 7.5 [Warning Options], page 35, for more information on this option.

17.4 Working Programs

Getting Fortran programs to work in the first place can be quite a challenge—even when the programs already work on other systems, or when using other compilers.

g77 offers some facilities that might be useful for tracking down bugs in such programs.

17.4.1 Not My Type

A fruitful source of bugs in Fortran source code is use, or mis-use, of Fortran's implicit-typing feature, whereby the type of a variable, array, or function is determined by the first character of its name.

Simple cases of this include statements like 'LOGX=9.227', without a statement such as 'REAL LOGX'. In this case, 'LOGX' is implicitly given INTEGER(KIND=1) type, with the result of the assignment being that it is given the value '9'.

More involved cases include a function that is defined starting with a statement like 'DOUBLE PRECISION FUNCTION IPS(...)'. Any caller of this function that does not also declare 'IPS' as type DOUBLE PRECISION (or, in GNU Fortran, REAL(KIND=2)) is likely to

assume it returns `INTEGER`, or some other type, leading to invalid results or even program crashes.

The '`-Wimplicit`' option might catch failures to properly specify the types of variables, arrays, and functions in the code.

However, in code that makes heavy use of Fortran's implicit-typing facility, this option might produce so many warnings about cases that are working, it would be hard to find the one or two that represent bugs. This is why so many experienced Fortran programmers strongly recommend widespread use of the `IMPLICIT NONE` statement, despite it not being standard FORTRAN 77, to completely turn off implicit typing. (`g77` supports `IMPLICIT NONE`, as do almost all FORTRAN 77 compilers.)

Note that '`-Wimplicit`' catches only implicit typing of *names*. It does not catch implicit typing of expressions such as '`X**(2/3)`'. Such expressions can be buggy as well—in fact, '`X**(2/3)`' is equivalent to '`X**0`', due to the way Fortran expressions are given types and then evaluated. (In this particular case, the programmer probably wanted '`X**(2./3.)`'.)

17.4.2 Variables Assumed To Be Zero

Many Fortran programs were developed on systems that provided automatic initialization of all, or some, variables and arrays to zero. As a result, many of these programs depend, sometimes inadvertently, on this behavior, though to do so violates the Fortran standards.

You can ask `g77` for this behavior by specifying the '`-finit-local-zero`' option when compiling Fortran code. (You might want to specify '`-fno-automatic`' as well, to avoid code-size inflation for non-optimized compilations.)

Note that a program that works better when compiled with the '`-finit-local-zero`' option is almost certainly depending on a particular system's, or compiler's, tendency to initialize some variables to zero. It might be worthwhile finding such cases and fixing them, using techniques such as compiling with the '`-O -Wuninitialized`' options using `g77`.

17.4.3 Variables Assumed To Be Saved

Many Fortran programs were developed on systems that saved the values of all, or some, variables and arrays across procedure calls. As a result, many of these programs depend, sometimes inadvertently, on being able to assign a value to a variable, perform a `RETURN` to a calling procedure, and, upon subsequent invocation, reference the previously assigned variable to obtain the value.

They expect this despite not using the `SAVE` statement to specify that the value in a variable is expected to survive procedure returns and calls. Depending on variables and arrays to retain values across procedure calls without using `SAVE` to require it violates the Fortran standards.

You can ask `g77` to assume `SAVE` is specified for all relevant (local) variables and arrays by using the '`-fno-automatic`' option.

Note that a program that works better when compiled with the '`-fno-automatic`' option is almost certainly depending on not having to use the `SAVE` statement as required by the Fortran standard. It might be worthwhile finding such cases and fixing them, using techniques such as compiling with the '`-O -Wuninitialized`' options using `g77`.

17.4.4 Unwanted Variables

The '`-Wunused`' option can find bugs involving implicit typing, sometimes more easily than using '`-Wimplicit`' in code that makes heavy use of implicit typing. An unused variable or array might indicate that the spelling for its declaration is different from that of its intended uses.

Other than cases involving typos, unused variables rarely indicate actual bugs in a program. However, investigating such cases thoroughly has, on occasion, led to the discovery of code that had not been completely written—where the programmer wrote declarations as needed for the whole algorithm, wrote some or even most of the code for that algorithm, then got distracted and forgot that the job was not complete.

17.4.5 Unused Arguments

As with unused variables, It is possible that unused arguments to a procedure might indicate a bug. Compile with '`-W -Wunused`' option to catch cases of unused arguments.

Note that '`-W`' also enables warnings regarding overflow of floating-point constants under certain circumstances.

17.4.6 Surprising Interpretations of Code

The '`-Wsuprising`' option can help find bugs involving expression evaluation or in the way `DO` loops with non-integral iteration variables are handled. Cases found by this option might indicate a difference of interpretation between the author of the code involved, and a standard-conforming compiler such as `g77`. Such a difference might produce actual bugs.

In any case, changing the code to explicitly do what the programmer might have expected it to do, so `g77` and other compilers are more likely to follow the programmer's expectations, might be worthwhile, especially if such changes make the program work better.

17.4.7 Aliasing Assumed To Work

The '`-falias-check`', '`-fargument-alias`', '`-fargument-noalias`', and '`-fno-argument-noalias-global`' options, introduced in version 0.5.20 and `g77`'s version 2.7.2.2.f.2 of `gcc`, control the assumptions regarding aliasing (overlapping) of writes and reads to main memory (core) made by the `gcc` back end.

They are effective only when compiling with '`-O`' (specifying any level other than '`-O0`') or with '`-falias-check`'.

The default for Fortran code is '`-fargument-noalias-global`'. (The default for C code and code written in other C-based languages is '`-fargument-alias`'. These defaults apply regardless of whether you use `g77` or `gcc` to compile your code.)

Note that, on some systems, compiling with '`-fforce-addr`' in effect can produce more optimal code when the default aliasing options are in effect (and when optimization is enabled).

If your program is not working when compiled with optimization, it is possible it is violating the Fortran standards (77 and 90) by relying on the ability to "safely" modify

variables and arrays that are aliased, via procedure calls, to other variables and arrays, without using `EQUIVALENCE` to explicitly set up this kind of aliasing.

(The FORTRAN 77 standard's prohibition of this sort of overlap, generally referred to therein as "storage assocation", appears in Sections 15.9.3.6. This prohibition allows implementations, such as `g77`, to, for example, implement the passing of procedures and even values in `COMMON` via copy operations into local, perhaps more efficiently accessed temporaries at entry to a procedure, and, where appropriate, via copy operations back out to their original locations in memory at exit from that procedure, without having to take into consideration the order in which the local copies are updated by the code, among other things.)

To test this hypothesis, try compiling your program with the '`-fargument-alias`' option, which causes the compiler to revert to assumptions essentially the same as made by versions of `g77` prior to 0.5.20.

If the program works using this option, that strongly suggests that the bug is in your program. Finding and fixing the bug(s) should result in a program that is more standard-conforming and that can be compiled by `g77` in a way that results in a faster executable.

(You might want to try compiling with '`-fargument-noalias`', a kind of half-way point, to see if the problem is limited to aliasing between dummy arguments and `COMMON` variables—this option assumes that such aliasing is not done, while still allowing aliasing among dummy arguments.)

An example of aliasing that is invalid according to the standards is shown in the following program, which might *not* produce the expected results when executed:

```
I = 1
CALL FOO(I, I)
PRINT *, I
END

SUBROUTINE FOO(J, K)
J = J + K
K = J * K
PRINT *, J, K
END
```

The above program attempts to use the temporary aliasing of the '`J`' and '`K`' arguments in '`FOO`' to effect a pathological behavior—the simultaneous changing of the values of *both* '`J`' and '`K`' when either one of them is written.

The programmer likely expects the program to print these values:

```
2  4
4
```

However, since the program is not standard-conforming, an implementation's behavior when running it is undefined, because subroutine '`FOO`' modifies at least one of the arguments, and they are aliased with each other. (Even if one of the assignment statements was deleted, the program would still violate these rules. This kind of on-the-fly aliasing is permitted by the standard only when none of the aliased items are defined, or written, while the aliasing is in effect.)

As a practical example, an optimizing compiler might schedule the '`J =`' part of the second line of '`FOO`' *after* the reading of '`J`' and '`K`' for the '`J * K`' expression, resulting in the following output:

```
2  2
2
```

Essentially, compilers are promised (by the standard and, therefore, by programmers who write code they claim to be standard-conforming) that if they cannot detect aliasing via static analysis of a single program unit's `EQUIVALENCE` and `COMMON` statements, no such aliasing exists. In such cases, compilers are free to assume that an assignment to one variable will not change the value of another variable, allowing it to avoid generating code to re-read the value of the other variable, to re-schedule reads and writes, and so on, to produce a faster executable.

The same promise holds true for arrays (as seen by the called procedure)—an element of one dummy array cannot be aliased with, or overlap, any element of another dummy array or be in a `COMMON` area known to the procedure.

(These restrictions apply only when the procedure defines, or writes to, one of the aliased variables or arrays.)

Unfortunately, there is no way to find *all* possible cases of violations of the prohibitions against aliasing in Fortran code. Static analysis is certainly imperfect, as is run-time analysis, since neither can catch all violations. (Static analysis can catch all likely violations, and some that might never actually happen, while run-time analysis can catch only those violations that actually happen during a particular run. Neither approach can cope with programs mixing Fortran code with routines written in other languages, however.)

Currently, `g77` provides neither static nor run-time facilities to detect any cases of this problem, although other products might. Run-time facilities are more likely to be offered by future versions of `g77`, though patches improving `g77` so that it provides either form of detection are welcome.

17.4.8 Output Assumed To Flush

For several versions prior to 0.5.20, `g77` configured its version of the `libf2c` run-time library so that one of its configuration macros, '`ALWAYS_FLUSH`', was defined.

This was done as a result of a belief that many programs expected output to be flushed to the operating system (under UNIX, via the `fflush()` library call) with the result that errors, such as disk full, would be immediately flagged via the relevant `ERR=` and `IOSTAT=` mechanism.

Because of the adverse effects this approach had on the performance of many programs, `g77` no longer configures `libf2c` to always flush output.

If your program depends on this behavior, either insert the appropriate '`CALL FLUSH`' statements, or modify the sources to the `libf2c`, rebuild and reinstall `g77`, and relink your programs with the modified library.

(Ideally, `libf2c` would offer the choice at run-time, so that a compile-time option to `g77` or `f2c` could result in generating the appropriate calls to flushing or non-flushing library routines.)

See Section 15.3.2 [Always Flush Output], page 219, for information on how to modify the `g77` source tree so that a version of `libf2c` can be built and installed with the '`ALWAYS_FLUSH`' macro defined.

17.4.9 Large File Unit Numbers

If your program crashes at run time with a message including the text '`illegal unit number`', that probably is a message from the run-time library, `libf2c`, used, and distributed with, `g77`.

The message means that your program has attempted to use a file unit number that is out of the range accepted by `libf2c`. Normally, this range is 0 through 99, and the high end of the range is controlled by a `libf2c` source-file macro named '`MXUNIT`'.

If you can easily change your program to use unit numbers in the range 0 through 99, you should do so.

Otherwise, see Section 15.3.1 [Larger File Unit Numbers], page 219, for information on how to change '`MXUNIT`' in `libf2c` so you can build and install a new version of `libf2c` that supports the larger unit numbers you need.

Note: While `libf2c` places a limit on the range of Fortran file-unit numbers, the underlying library and operating system might impose different kinds of limits. For example, some systems limit the number of files simultaneously open by a running program. Information on how to increase these limits should be found in your system's documentation.

17.5 Overly Convenient Command-line Options

These options should be used only as a quick-and-dirty way to determine how well your program will run under different compilation models without having to change the source. Some are more problematic than others, depending on how portable and maintainable you want the program to be (and, of course, whether you are allowed to change it at all is crucial).

You should not continue to use these command-line options to compile a given program, but rather should make changes to the source code:

`-finit-local-zero`

> (This option specifies that any uninitialized local variables and arrays have default initialization to binary zeros.)
>
> Many other compilers do this automatically, which means lots of Fortran code developed with those compilers depends on it.
>
> It is safer (and probably would produce a faster program) to find the variables and arrays that need such initialization and provide it explicitly via `DATA`, so that '`-finit-local-zero`' is not needed.
>
> Consider using '`-Wuninitialized`' (which requires '`-O`') to find likely candidates, but do not specify '`-finit-local-zero`' or '`-fno-automatic`', or this technique won't work.

`-fno-automatic`
(This option specifies that all local variables and arrays are to be treated as if they were named in `SAVE` statements.)

Many other compilers do this automatically, which means lots of Fortran code developed with those compilers depends on it.

The effect of this is that all non-automatic variables and arrays are made static, that is, not placed on the stack or in heap storage. This might cause a buggy program to appear to work better. If so, rather than relying on this command-line option (and hoping all compilers provide the equivalent one), add `SAVE` statements to some or all program unit sources, as appropriate. Consider using '`-Wuninitialized`' (which requires '`-O`') to find likely candidates, but do not specify '`-finit-local-zero`' or '`-fno-automatic`', or this technique won't work.

The default is '`-fautomatic`', which tells `g77` to try and put variables and arrays on the stack (or in fast registers) where possible and reasonable. This tends to make programs faster.

Note: Automatic variables and arrays are not affected by this option. These are variables and arrays that are *necessarily* automatic, either due to explicit statements, or due to the way they are declared. Examples include local variables and arrays not given the `SAVE` attribute in procedures declared `RECURSIVE`, and local arrays declared with non-constant bounds (automatic arrays). Currently, `g77` supports only automatic arrays, not `RECURSIVE` procedures or other means of explicitly specifying that variables or arrays are automatic.

`-fugly` Fix the source code so that '`-fno-ugly`' will work. Note that, for many programs, it is difficult to practically avoid using the features enabled via '`-fugly-init`', and these features pose the lowest risk of writing nonportable code, among the various "ugly" features.

`-f`*group*`-intrinsics-hide`
Change the source code to use `EXTERNAL` for any external procedure that might be the name of an intrinsic. It is easy to find these using '`-f`*group*`-intrinsics-disable`'.

17.6 Faster Programs

Aside from the usual `gcc` options, such as '`-O`', '`-ffast-math`', and so on, consider trying some of the following approaches to speed up your program (once you get it working).

17.6.1 Aligned Data

On some systems, such as those with Pentium Pro CPUs, programs that make heavy use of `REAL(KIND=2)` (`DOUBLE PRECISION`) might run much slower than possible due to the compiler not aligning these 64-bit values to 64-bit boundaries in memory. (The effect also is present, though to a lesser extent, on the 586 (Pentium) architecture.)

The Intel x86 architecture generally ensures that these programs will work on all its implementations, but particular implementations (such as Pentium Pro) perform better

with more strict alignment. (Such behavior isn't unique to the Intel x86 architecture.) Other architectures might *demand* 64-bit alignment of 64-bit data.

There are a variety of approaches to use to address this problem:

- Order your `COMMON` and `EQUIVALENCE` areas such that the variables and arrays with the widest alignment guidelines come first.

  For example, on most systems, this would mean placing `COMPLEX(KIND=2)`, `REAL(KIND=2)`, and `INTEGER(KIND=2)` entities first, followed by `REAL(KIND=1)`, `INTEGER(KIND=1)`, and `LOGICAL(KIND=1)` entities, then `INTEGER(KIND=6)` entities, and finally `CHARACTER` and `INTEGER(KIND=3)` entities.

  The reason to use such placement is it makes it more likely that your data will be aligned properly, without requiring you to do detailed analysis of each aggregate (`COMMON` and `EQUIVALENCE`) area.

  Specifically, on systems where the above guidelines are appropriate, placing `CHARACTER` entities before `REAL(KIND=2)` entities can work just as well, but only if the number of bytes occupied by the `CHARACTER` entities is divisible by the recommended alignment for `REAL(KIND=2)`.

  By ordering the placement of entities in aggregate areas according to the simple guidelines above, you avoid having to carefully count the number of bytes occupied by each entity to determine whether the actual alignment of each subsequent entity meets the alignment guidelines for the type of that entity.

  If you don't ensure correct alignment of `COMMON` elements, the compiler may be forced by some systems to violate the Fortran semantics by adding padding to get `DOUBLE PRECISION` data properly aligned. If the unfortunate practice is employed of overlaying different types of data in the `COMMON` block, the different variants of this block may become misaligned with respect to each other. Even if your platform doesn't require strict alignment, `COMMON` should be laid out as above for portability. (Unfortunately the FORTRAN 77 standard didn't anticipate this possible requirement, which is compiler-independent on a given platform.)
- Use the (x86-specific) '`-malign-double`' option when compiling programs for the Pentium and Pentium Pro architectures (called 586 and 686 in the `gcc` configuration subsystem). The warning about this in the `gcc` manual isn't generally relevant to Fortran, but using it will force `COMMON` to be padded if necessary to align `DOUBLE PRECISION` data.
- Ensure that '`crt0.o`' or '`crt1.o`' on your system guarantees a 64-bit aligned stack for `main()`. The recent one from GNU (`glibc2`) will do this on x86 systems, but we don't know of any other x86 setups where it will be right. Read your system's documentation to determine if it is appropriate to upgrade to a more recent version to obtain the optimal alignment.

Progress is being made on making this work "out of the box" on future versions of `g77`, `gcc`, and some of the relevant operating systems (such as GNU/Linux).

17.6.2 Prefer Automatic Uninitialized Variables

If you're using '`-fno-automatic`' already, you probably should change your code to allow compilation with '`-fautomatic`' (the default), to allow the program to run faster.

Similarly, you should be able to use '`-fno-init-local-zero`' (the default) instead of '`-finit-local-zero`'. This is because it is rare that every variable affected by these options in a given program actually needs to be so affected.

For example, '`-fno-automatic`', which effectively `SAVE`s every local non-automatic variable and array, affects even things like `DO` iteration variables, which rarely need to be `SAVE`d, and this often reduces run-time performances. Similarly, '`-fno-init-local-zero`' forces such variables to be initialized to zero—when `SAVE`d (such as when '`-fno-automatic`'), this by itself generally affects only startup time for a program, but when not `SAVE`d, it can slow down the procedure every time it is called.

See Section 17.5 [Overly Convenient Command-Line Options], page 261, for information on the '`-fno-automatic`' and '`-finit-local-zero`' options and how to convert their use into selective changes in your own code.

17.6.3 Avoid f2c Compatibility

If you aren't linking with any code compiled using `f2c`, try using the '`-fno-f2c`' option when compiling *all* the code in your program. (Note that `libf2c` is *not* an example of code that is compiled using `f2c`—it is compiled by a C compiler, typically `gcc`.)

17.6.4 Use Submodel Options

Using an appropriate '`-m`' option to generate specific code for your CPU may be worthwhile, though it may mean the executable won't run on other versions of the CPU that don't support the same instruction set. See section "Hardware Models and Configurations" in *Using and Porting GNU CC*.

For recent CPUs that don't have explicit support in the released version of `gcc`, it may still be possible to get improvements. For instance, the flags recommended for 586/686 (Pentium(Pro)) chips for building the Linux kernel are:

```
-m486 -malign-loops=2 -malign-jumps=2 -malign-functions=2
-fomit-frame-pointer
```

'`-fomit-frame-pointer`' will, however, inhibit debugging on x86 systems.

18 Known Causes of Trouble with GNU Fortran

This section describes known problems that affect users of GNU Fortran. Most of these are not GNU Fortran bugs per se—if they were, we would fix them. But the result for a user might be like the result of a bug.

Some of these problems are due to bugs in other software, some are missing features that are too much work to add, and some are places where people's opinions differ as to what is best.

Information on bugs that show up when configuring, porting, building, or installing `g77` is not provided here. See Section 15.2 [Problems Installing], page 216.

To find out about major bugs discovered in the current release and possible workarounds for them, retrieve `ftp://alpha.gnu.org/g77.plan`.

(Note that some of this portion of the manual is lifted directly from the `gcc` manual, with minor modifications to tailor it to users of `g77`. Anytime a bug seems to have more to do with the `gcc` portion of `g77`, See section "Known Causes of Trouble with GNU CC" in *Using and Porting GNU CC*.)

18.1 Bugs Not In GNU Fortran

These are bugs to which the maintainers often have to reply, "but that isn't a bug in `g77`...". Some of these already are fixed in new versions of other software; some still need to be fixed; some are problems with how `g77` is installed or is being used; some are the result of bad hardware that causes software to misbehave in sometimes bizarre ways; some just cannot be addressed at this time until more is known about the problem.

Please don't re-report these bugs to the `g77` maintainers—if you must remind someone how important it is to you that the problem be fixed, talk to the people responsible for the other products identified below, but preferably only after you've tried the latest versions of those products. The `g77` maintainers have their hands full working on just fixing and improving `g77`, without serving as a clearinghouse for all bugs that happen to affect `g77` users.

See Chapter 17 [Collected Fortran Wisdom], page 251, for information on behavior of Fortran programs, and the programs that compile them, that might be *thought* to indicate bugs.

18.1.1 Signal 11 and Friends

A whole variety of strange behaviors can occur when the software, or the way you are using the software, stresses the hardware in a way that triggers hardware bugs. This might seem hard to believe, but it happens frequently enough that there exist documents explaining in detail what the various causes of the problems are, what typical symptoms look like, and so on.

Generally these problems are referred to in this document as "signal 11" crashes, because the Linux kernel, running on the most popular hardware (the Intel x86 line), often stresses the hardware more than other popular operating systems. When hardware problems do

occur under GNU/Linux on x86 systems, these often manifest themselves as "signal 11" problems, as illustrated by the following diagnostic:

```
sh# g77 myprog.f
gcc: Internal compiler error: program f771 got fatal signal 11
sh#
```

It is *very* important to remember that the above message is *not* the only one that indicates a hardware problem, nor does it always indicate a hardware problem.

In particular, on systems other than those running the Linux kernel, the message might appear somewhat or very different, as it will if the error manifests itself while running a program other than the `g77` compiler. For example, it will appear somewhat different when running your program, when running Emacs, and so on.

How to cope with such problems is well beyond the scope of this manual.

However, users of Linux-based systems (such as GNU/Linux) should review `http://www.bitwizard.nl/sig` a source of detailed information on diagnosing hardware problems, by recognizing their common symptoms.

Users of other operating systems and hardware might find this reference useful as well. If you know of similar material for another hardware/software combination, please let us know so we can consider including a reference to it in future versions of this manual.

18.1.2 Cannot Link Fortran Programs

On some systems, perhaps just those with out-of-date (shared?) libraries, unresolved-reference errors happen when linking `g77`-compiled programs (which should be done using `g77`).

If this happens to you, try appending '`-lc`' to the command you use to link the program, e.g. '`g77 foo.f -lc`'. `g77` already specifies '`-lf2c -lm`' when it calls the linker, but it cannot also specify '`-lc`' because not all systems have a file named '`libc.a`'.

It is unclear at this point whether there are legitimately installed systems where '`-lf2c -lm`' is insufficient to resolve code produced by `g77`.

If your program doesn't link due to unresolved references to names like '`_main`', make sure you're using the `g77` command to do the link, since this command ensures that the necessary libraries are loaded by specifying '`-lf2c -lm`' when it invokes the `gcc` command to do the actual link. (Use the '`-v`' option to discover more about what actually happens when you use the `g77` and `gcc` commands.)

Also, try specifying '`-lc`' as the last item on the `g77` command line, in case that helps.

18.1.3 Large Common Blocks

On some older GNU/Linux systems, programs with common blocks larger than 16MB cannot be linked without some kind of error message being produced.

This is a bug in older versions of `ld`, fixed in more recent versions of `binutils`, such as version 2.6.

18.1.4 Debugger Problems

There are some known problems when using `gdb` on code compiled by `g77`. Inadequate investigation as of the release of 0.5.16 results in not knowing which products are the culprit, but '`gdb-4.14`' definitely crashes when, for example, an attempt is made to print the contents of a `COMPLEX(KIND=2)` dummy array, on at least some GNU/Linux machines, plus some others.

18.1.5 NeXTStep Problems

Developers of Fortran code on NeXTStep (all architectures) have to watch out for the following problem when writing programs with large, statically allocated (i.e. non-stack based) data structures (common blocks, saved arrays).

Due to the way the native loader ('`/bin/ld`') lays out data structures in virtual memory, it is very easy to create an executable wherein the '`__DATA`' segment overlaps (has addresses in common) with the '`UNIX STACK`' segment.

This leads to all sorts of trouble, from the executable simply not executing, to bus errors. The NeXTStep command line tool `ebadexec` points to the problem as follows:

```
% /bin/ebadexec a.out
/bin/ebadexec: __LINKEDIT segment (truncated address = 0x3de000
rounded size = 0x2a000) of executable file: a.out overlaps with UNIX
STACK segment (truncated address = 0x400000 rounded size =
0x3c00000) of executable file: a.out
```

(In the above case, it is the '`__LINKEDIT`' segment that overlaps the stack segment.)

This can be cured by assigning the '`__DATA`' segment (virtual) addresses beyond the stack segment. A conservative estimate for this is from address 6000000 (hexadecimal) onwards—this has always worked for me [Toon Moene]:

```
% g77 -segaddr __DATA 6000000 test.f
% ebadexec a.out
ebadexec: file: a.out appears to be executable
%
```

Browsing through '`gcc/f/Makefile.in`', you will find that the `f771` program itself also has to be linked with these flags—it has large statically allocated data structures. (Version 0.5.18 reduces this somewhat, but probably not enough.)

(The above item was contributed by Toon Moene (`toon@moene.indiv.nluug.nl`).)

18.1.6 Stack Overflow

`g77` code might fail at runtime (probably with a "segmentation violation") due to overflowing the stack. This happens most often on systems with an environment that provides substantially more heap space (for use when arbitrarily allocating and freeing memory) than stack space.

Often this can be cured by increasing or removing your shell's limit on stack usage, typically using *`limit stacksize`* (in `csh` and derivatives) or *`ulimit -s`* (in `sh` and derivatives).

Increasing the allowed stack size might, however, require changing some operating system or system configuration parameters.

You might be able to work around the problem by compiling with the '`-fno-automatic`' option to reduce stack usage, probably at the expense of speed.

See Section 15.3.3 [Maximum Stackable Size], page 220, for information on patching `g77` to use different criteria for placing local non-automatic variables and arrays on the stack.

However, if your program uses large automatic arrays (for example, has declarations like '`REAL A(N)`' where '`A`' is a local array and '`N`' is a dummy or `COMMON` variable that can have a large value), neither use of '`-fno-automatic`', nor changing the cut-off point for `g77` for using the stack, will solve the problem by changing the placement of these large arrays, as they are *necessarily* automatic.

`g77` currently provides no means to specify that automatic arrays are to be allocated on the heap instead of the stack. So, other than increasing the stack size, your best bet is to change your source code to avoid large automatic arrays. Methods for doing this currently are outside the scope of this document.

(*Note:* If your system puts stack and heap space in the same memory area, such that they are effectively combined, then a stack overflow probably indicates a program that is either simply too large for the system, or buggy.)

18.1.7 Nothing Happens

It is occasionally reported that a "simple" program, such as a "Hello, World!" program, does nothing when it is run, even though the compiler reported no errors, despite the program containing nothing other than a simple `PRINT` statement.

This most often happens because the program has been compiled and linked on a UNIX system and named '`test`', though other names can lead to similarly unexpected run-time behavior on various systems.

Essentially this problem boils down to giving your program a name that is already known to the shell you are using to identify some other program, which the shell continues to execute instead of your program when you invoke it via, for example:

```
sh# test
sh#
```

Under UNIX and many other system, a simple command name invokes a searching mechanism that might well not choose the program located in the current working directory if there is another alternative (such as the `test` command commonly installed on UNIX systems).

The reliable way to invoke a program you just linked in the current directory under UNIX is to specify it using an explicit pathname, as in:

```
sh# ./test
 Hello, World!
sh#
```

Users who encounter this problem should take the time to read up on how their shell searches for commands, how to set their search path, and so on. The relevant UNIX commands to learn about include `man`, `info` (on GNU systems), `setenv` (or `set` and `env`), `which`, and `find`.

18.1.8 Strange Behavior at Run Time

`g77` code might fail at runtime with "segmentation violation", "bus error", or even something as subtle as a procedure call overwriting a variable or array element that it is not supposed to touch.

These can be symptoms of a wide variety of actual bugs that occurred earlier during the program's run, but manifested themselves as *visible* problems some time later.

Overflowing the bounds of an array—usually by writing beyond the end of it—is one of two kinds of bug that often occurs in Fortran code.

The other kind of bug is a mismatch between the actual arguments passed to a procedure and the dummy arguments as declared by that procedure.

Both of these kinds of bugs, and some others as well, can be difficult to track down, because the bug can change its behavior, or even appear to not occur, when using a debugger.

That is, these bugs can be quite sensitive to data, including data representing the placement of other data in memory (that is, pointers, such as the placement of stack frames in memory).

Plans call for improving `g77` so that it can offer the ability to catch and report some of these problems at compile, link, or run time, such as by generating code to detect references to beyond the bounds of an array, or checking for agreement between calling and called procedures.

In the meantime, finding and fixing the programming bugs that lead to these behaviors is, ultimately, the user's responsibility, as difficult as that task can sometimes be.

One runtime problem that has been observed might have a simple solution. If a formatted `WRITE` produces an endless stream of spaces, check that your program is linked against the correct version of the C library. The configuration process takes care to account for your system's normal '`libc`' not being ANSI-standard, which will otherwise cause this behaviour. If your system's default library is ANSI-standard and you subsequently link against a non-ANSI one, there might be problems such as this one.

Specifically, on Solaris2 systems, avoid picking up the `BSD` library from '`/usr/ucblib`'.

18.1.9 Floating-point Errors

Some programs appear to produce inconsistent floating-point results compiled by `g77` versus by other compilers.

Often the reason for this behavior is the fact that floating-point values are represented on almost all Fortran systems by *approximations*, and these approximations are inexact even for apparently simple values like 0.1, 0.2, 0.3, 0.4, 0.6, 0.7, 0.8, 0.9, 1.1, and so on. Most Fortran systems, including all current ports of `g77`, use binary arithmetic to represent these approximations.

Therefore, the exact value of any floating-point approximation as manipulated by `g77`-compiled code is representable by adding some combination of the values 1.0, 0.5, 0.25, 0.125, and so on (just keep dividing by two) through the precision of the fraction (typically around 23 bits for `REAL(KIND=1)`, 52 for `REAL(KIND=2)`), then multiplying the sum by a integral power of two (in Fortran, by '`2**N`') that typically is between -127 and +128 for

`REAL(KIND=1)` and -1023 and +1024 for `REAL(KIND=2)`, then multiplying by -1 if the number is negative.

So, a value like 0.2 is exactly represented in decimal—since it is a fraction, '`2/10`', with a denomenator that is compatible with the base of the number system (base 10). However, '`2/10`' cannot be represented by any finite number of sums of any of 1.0, 0.5, 0.25, and so on, so 0.2 cannot be exactly represented in binary notation.

(On the other hand, decimal notation can represent any binary number in a finite number of digits. Decimal notation cannot do so with ternary, or base-3, notation, which would represent floating-point numbers as sums of any of '`1/1`', '`1/3`', '`1/9`', and so on. After all, no finite number of decimal digits can exactly represent '`1/3`'. Fortunately, few systems use ternary notation.)

Moreover, differences in the way run-time I/O libraries convert between these approximations and the decimal representation often used by programmers and the programs they write can result in apparent differences between results that do not actually exist, or exist to such a small degree that they usually are not worth worrying about.

For example, consider the following program:

```
PRINT *, 0.2
END
```

When compiled by `g77`, the above program might output '`0.20000003`', while another compiler might produce a executable that outputs '`0.2`'.

This particular difference is due to the fact that, currently, conversion of floating-point values by the `libf2c` library, used by `g77`, handles only double-precision values.

Since '`0.2`' in the program is a single-precision value, it is converted to double precision (still in binary notation) before being converted back to decimal. The conversion to binary appends _binary_ zero digits to the original value—which, again, is an inexact approximation of 0.2—resulting in an approximation that is much less exact than is connoted by the use of double precision.

(The appending of binary zero digits has essentially the same effect as taking a particular decimal approximation of '`1/3`', such as '`0.3333333`', and appending decimal zeros to it, producing '`0.33333330000000000`'. Treating the resulting decimal approximation as if it really had 18 or so digits of valid precision would make it seem a very poor approximation of '`1/3`'.)

As a result of converting the single-precision approximation to double precision by appending binary zeros, the conversion of the resulting double-precision value to decimal produces what looks like an incorrect result, when in fact the result is *inexact*, and is probably no less inaccurate or imprecise an approximation of 0.2 than is produced by other compilers that happen to output the converted value as "exactly" '`0.2`'. (Some compilers behave in a way that can make them appear to retain more accuracy across a conversion of a single-precision constant to double precision. See Section 18.5.4 [Context-Sensitive Constants], page 286, to see why this practice is illusory and even dangerous.)

Note that a more exact approximation of the constant is computed when the program is changed to specify a double-precision constant:

```
PRINT *, 0.2D0
END
```

Future versions of `g77` and/or `libf2c` might convert single-precision values directly to decimal, instead of converting them to double precision first. This would tend to result in output that is more consistent with that produced by some other Fortran implementations.

18.2 Actual Bugs We Haven't Fixed Yet

This section identifies bugs that `g77` *users* might run into. This includes bugs that are actually in the `gcc` back end (GBE) or in `libf2c`, because those sets of code are at least somewhat under the control of (and necessarily intertwined with) `g77`, so it isn't worth separating them out.

For information on bugs that might afflict people who configure, port, build, and install `g77`, Section 15.2 [Problems Installing], page 216.

- `g77`'s version of `gcc`, and probably `g77` itself, cannot be reliably used with the '`-O2`' option (or higher) on Digital Semiconductor Alpha AXP machines. The problem is most immediately noticed in differences discovered by *`make compare`* following a bootstrap build using '`-O2`'. It also manifests itself as a failure to compile '`DATA`' statements such as '`DATA R/7./`' correctly; in this case, '`R`' might be initialized to '`4.0`'.

  Until this bug is fixed, use only '`-O1`' or no optimization.

- Something about `g77`'s straightforward handling of label references and definitions sometimes prevents the GBE from unrolling loops. Until this is solved, try inserting or removing `CONTINUE` statements as the terminal statement, using the `END DO` form instead, and so on. (Probably improved, but not wholly fixed, in 0.5.21.)

- The `g77` command itself should more faithfully process options the way the `gcc` command does. For example, `gcc` accepts abbreviated forms of long options, `g77` generally doesn't.

- Some confusion in diagnostics concerning failing `INCLUDE` statements from within `INCLUDE`'d or `#include`'d files.

- `g77` assumes that `INTEGER(KIND=1)` constants range from '`-2**31`' to '`2**31-1`' (the range for two's-complement 32-bit values), instead of determining their range from the actual range of the type for the configuration (and, someday, for the constant).

  Further, it generally doesn't implement the handling of constants very well in that it makes assumptions about the configuration that it no longer makes regarding variables (types).

  Included with this item is the fact that `g77` doesn't recognize that, on IEEE-754/854-compliant systems, '`0./0.`' should produce a NaN and no warning instead of the value '`0.`' and a warning. This is to be fixed in version 0.6, when `g77` will use the `gcc` back end's constant-handling mechanisms to replace its own.

- `g77` uses way too much memory and CPU time to process large aggregate areas having any initialized elements.

  For example, '`REAL A(1000000)`' followed by '`DATA A(1)/1/`' takes up way too much time and space, including the size of the generated assembler file. This is to be mitigated somewhat in version 0.6.

Version 0.5.18 improves cases like this—specifically, cases of *sparse* initialization that leave large, contiguous areas uninitialized—significantly. However, even with the improvements, these cases still require too much memory and CPU time.

(Version 0.5.18 also improves cases where the initial values are zero to a much greater degree, so if the above example ends with '`DATA A(1)/0/`', the compile-time performance will be about as good as it will ever get, aside from unrelated improvements to the compiler.)

Note that `g77` does display a warning message to notify the user before the compiler appears to hang. See Section 15.3.5 [Initialization of Large Aggregate Areas], page 220, for information on how to change the point at which `g77` decides to issue this warning.

- `g77` doesn't emit variable and array members of common blocks for use with a debugger (the '`-g`' command-line option). The code is present to do this, but doesn't work with at least one debug format—perhaps it works with others. And it turns out there's a similar bug for local equivalence areas, so that has been disabled as well.

  As of Version 0.5.19, a temporary kludge solution is provided whereby some rudimentary information on a member is written as a string that is the member's value as a character string.

  See Section 7.10 [Options for Code Generation Conventions], page 41, for information on the '`-fdebug-kludge`' option.

- When debugging, after starting up the debugger but before being able to see the source code for the main program unit, the user must currently set a breakpoint at '`MAIN__`' (or '`MAIN___`' or '`MAIN_`' if '`MAIN__`' doesn't exist) and run the program until it hits the breakpoint. At that point, the main program unit is activated and about to execute its first executable statement, but that's the state in which the debugger should start up, as is the case for languages like C.

- Debugging `g77`-compiled code using debuggers other than `gdb` is likely not to work.

  Getting `g77` and `gdb` to work together is a known problem—getting `g77` to work properly with other debuggers, for which source code often is unavailable to `g77` developers, seems like a much larger, unknown problem, and is a lower priority than making `g77` and `gdb` work together properly.

  On the other hand, information about problems other debuggers have with `g77` output might make it easier to properly fix `g77`, and perhaps even improve `gdb`, so it is definitely welcome. Such information might even lead to all relevant products working together properly sooner.

- `g77` currently inserts needless padding for things like '`COMMON A,IPAD`' where '`A`' is `CHARACTER*1` and '`IPAD`' is `INTEGER(KIND=1)` on machines like x86, because the back end insists that '`IPAD`' be aligned to a 4-byte boundary, but the processor has no such requirement (though it's good for performance).

  It is possible that this is not a real bug, and could be considered a performance feature, but it might be important to provide the ability to Fortran code to specify minimum padding for aggregate areas such as common blocks—and, certainly, there is the potential, with the current setup, for interface differences in the way such areas are laid out between `g77` and other compilers.

- g77 doesn't work perfectly on 64-bit configurations such as the Alpha. This problem is expected to be largely resolved as of version 0.5.20, and further addressed by 0.5.21. Version 0.6 should solve most or all related problems (such as 64-bit machines other than Digital Semiconductor ("DEC") Alphas).

  One known bug that causes a compile-time crash occurs when compiling code such as the following with optimization:

  ```
  SUBROUTINE CRASH (TEMP)
  INTEGER*2 HALF(2)
  REAL TEMP
  HALF(1) = NINT (TEMP)
  END
  ```

  It is expected that a future version of g77 will have a fix for this problem, almost certainly by the time g77 supports the forthcoming version 2.8.0 of gcc.

- Maintainers of gcc report that the back end definitely has "broken" support for COMPLEX types. Based on their input, it seems many of the problems affect only the more-general facilities for gcc's __complex__ type, such as __complex__ int (where the real and imaginary parts are integers) that GNU Fortran does not use.

  Version 0.5.20 of g77 works around this problem by not using the back end's support for COMPLEX. The new option '-fno-emulate-complex' avoids the work-around, reverting to using the same "broken" mechanism as that used by versions of g77 prior to 0.5.20.

- There seem to be some problems with passing constants, and perhaps general expressions (other than simple variables/arrays), to procedures when compiling on some systems (such as i386) with '-fPIC', as in when compiling for ELF targets. The symptom is that the assembler complains about invalid opcodes. This bug is in the gcc back end, and it apparently occurs only when compiling sufficiently complicated functions *without* the '-O' option.

18.3 Missing Features

This section lists features we know are missing from g77, and which we want to add someday. (There is no priority implied in the ordering below.)

18.3.1 Better Source Model

g77 needs to provide, as the default source-line model, a "pure visual" mode, where the interpretation of a source program in this mode can be accurately determined by a user looking at a traditionally displayed rendition of the program (assuming the user knows whether the program is fixed or free form).

The design should assume the user cannot tell tabs from spaces and cannot see trailing spaces on lines, but has canonical tab stops and, for fixed-form source, has the ability to always know exactly where column 72 is (since the Fortran standard itself requires this for fixed-form source).

This would change the default treatment of fixed-form source to not treat lines with tabs as if they were infinitely long—instead, they would end at column 72 just as if the tabs were replaced by spaces in the canonical way.

As part of this, provide common alternate models (Digital, `f2c`, and so on) via command-line options. This includes allowing arbitrarily long lines for free-form source as well as fixed-form source and providing various limits and diagnostics as appropriate.

Also, `g77` should offer, perhaps even default to, warnings when characters beyond the last valid column are anything other than spaces. This would mean code with "sequence numbers" in columns 73 through 80 would be rejected, and there's a lot of that kind of code around, but one of the most frequent bugs encountered by new users is accidentally writing fixed-form source code into and beyond column 73. So, maybe the users of old code would be able to more easily handle having to specify, say, a `-Wno-col73to80` option.

18.3.2 Fortran 90 Support

`g77` does not support many of the features that distinguish Fortran 90 (and, now, Fortran 95) from ANSI FORTRAN 77.

Some Fortran 90 features are supported, because they make sense to offer even to die-hard users of F77. For example, many of them codify various ways F77 has been extended to meet users' needs during its tenure, so `g77` might as well offer them as the primary way to meet those same needs, even if it offers compatibility with one or more of the ways those needs were met by other F77 compilers in the industry.

Still, many important F90 features are not supported, because no attempt has been made to research each and every feature and assess its viability in `g77`. In the meantime, users who need those features must use Fortran 90 compilers anyway, and the best approach to adding some F90 features to GNU Fortran might well be to fund a comprehensive project to create GNU Fortran 95.

18.3.3 Intrinsics in `PARAMETER` Statements

`g77` doesn't allow intrinsics in `PARAMETER` statements. This feature is considered to be absolutely vital, even though it is not standard-conforming, and is scheduled for version 0.6.

Related to this, `g77` doesn't allow non-integral exponentiation in `PARAMETER` statements, such as '`PARAMETER (R=2**.25)`'. It is unlikely `g77` will ever support this feature, as doing it properly requires complete emulation of a target computer's floating-point facilities when building `g77` as a cross-compiler. But, if the `gcc` back end is enhanced to provide such a facility, `g77` will likely use that facility in implementing this feature soon afterwards.

18.3.4 `SELECT CASE` on `CHARACTER` Type

Character-type selector/cases for `SELECT CASE` currently are not supported.

18.3.5 `RECURSIVE` Keyword

`g77` doesn't support the `RECURSIVE` keyword that F90 compilers do. Nor does it provide any means for compiling procedures designed to do recursion.

All recursive code can be rewritten to not use recursion, but the result is not pretty.

18.3.6 Increasing Precision/Range

Some compilers, such as `f2c`, have an option ('`-r8`' or similar) that provides automatic treatment of `REAL` entities such that they have twice the storage size, and a corresponding increase in the range and precision, of what would normally be the `REAL(KIND=1)` (default `REAL`) type. (This affects `COMPLEX` the same way.)

They also typically offer another option ('`-i8`') to increase `INTEGER` entities so they are twice as large (with roughly twice as much range).

(There are potential pitfalls in using these options.)

`g77` does not yet offer any option that performs these kinds of transformations. Part of the problem is the lack of detailed specifications regarding exactly how these options affect the interpretation of constants, intrinsics, and so on.

Until `g77` addresses this need, programmers could improve the portability of their code by modifying it to not require compile-time options to produce correct results. Some free tools are available which may help, specifically in Toolpack (which one would expect to be sound) and the '`fortran`' section of the Netlib repository.

Use of preprocessors can provide a fairly portable means to work around the lack of widely portable methods in the Fortran language itself (though increasing acceptance of Fortran 90 would alleviate this problem).

18.3.7 Popular Non-standard Types

`g77` doesn't fully support `INTEGER*2`, `LOGICAL*1`, and similar. Version 0.6 will provide full support for this very popular set of features. In the meantime, version 0.5.18 provides rudimentary support for them.

18.3.8 Full Support for Compiler Types

`g77` doesn't support `INTEGER`, `REAL`, and `COMPLEX` equivalents for *all* applicable back-end-supported types (`char`, `short int`, `int`, `long int`, `long long int`, and `long double`). This means providing intrinsic support, and maybe constant support (using F90 syntax) as well, and, for most machines will result in automatic support of `INTEGER*1`, `INTEGER*2`, `INTEGER*8`, maybe even `REAL*16`, and so on. This is scheduled for version 0.6.

18.3.9 Array Bounds Expressions

`g77` doesn't support more general expressions to dimension arrays, such as array element references, function references, etc.

For example, `g77` currently does not accept the following:

```
SUBROUTINE X(M, N)
INTEGER N(10), M(N(2), N(1))
```

18.3.10 POINTER Statements

`g77` doesn't support pointers or allocatable objects (other than automatic arrays). This set of features is probably considered just behind intrinsics in `PARAMETER` statements on the list of large, important things to add to `g77`.

In the meantime, consider using the `INTEGER(KIND=7)` declaration to specify that a variable must be able to hold a pointer. This construct is not portable to other non-GNU compilers, but it is portable to all machines GNU Fortran supports when `g77` is used.

See Section 10.11 [Functions and Subroutines], page 91, for information on `%VAL()`, `%REF()`, and `%DESCR()` constructs, which are useful for passing pointers to procedures written in languages other than Fortran.

18.3.11 Sensible Non-standard Constructs

`g77` rejects things other compilers accept, like '`INTRINSIC SQRT,SQRT`'. As time permits in the future, some of these things that are easy for humans to read and write and unlikely to be intended to mean something else will be accepted by `g77` (though '`-fpedantic`' should trigger warnings about such non-standard constructs).

Until `g77` no longer gratuitously rejects sensible code, you might as well fix your code to be more standard-conforming and portable.

The kind of case that is important to except from the recommendation to change your code is one where following good coding rules would force you to write non-standard code that nevertheless has a clear meaning.

For example, when writing an `INCLUDE` file that defines a common block, it might be appropriate to include a `SAVE` statement for the common block (such as '`SAVE /CBLOCK/`'), so that variables defined in the common block retain their values even when all procedures declaring the common block become inactive (return to their callers).

However, putting `SAVE` statements in an `INCLUDE` file would prevent otherwise standard-conforming code from also specifying the `SAVE` statement, by itself, to indicate that all local variables and arrays are to have the `SAVE` attribute.

For this reason, `g77` already has been changed to allow this combination, because although the general problem of gratuitously rejecting unambiguous and "safe" constructs still exists in `g77`, this particular construct was deemed useful enough that it was worth fixing `g77` for just this case.

So, while there is no need to change your code to avoid using this particular construct, there might be other, equally appropriate but non-standard constructs, that you shouldn't have to stop using just because `g77` (or any other compiler) gratuitously rejects it.

Until the general problem is solved, if you have any such construct you believe is worthwhile using (e.g. not just an arbitrary, redundant specification of an attribute), please submit a bug report with an explanation, so we can consider fixing `g77` just for cases like yours.

18.3.12 `FLUSH` Statement

`g77` could perhaps use a `FLUSH` statement that does what '`CALL FLUSH`' does, but that supports '`*`' as the unit designator (same unit as for `PRINT`) and accepts `ERR=` and/or `IOSTAT=` specifiers.

18.3.13 Expressions in FORMAT Statements

g77 doesn't support 'FORMAT(I<J>)' and the like. Supporting this requires a significant redesign or replacement of libf2c.

However, g77 does support this construct when the expression is constant (as of version 0.5.22). For example:

```
      PARAMETER (IWIDTH = 12)
10    FORMAT (I<IWIDTH>)
```

Otherwise, at least for output (PRINT and WRITE), Fortran code making use of this feature can be rewritten to avoid it by constructing the FORMAT string in a CHARACTER variable or array, then using that variable or array in place of the FORMAT statement label to do the original PRINT or WRITE.

Many uses of this feature on input can be rewritten this way as well, but not all can. For example, this can be rewritten:

```
      READ 20, I
20    FORMAT (I<J>)
```

However, this cannot, in general, be rewritten, especially when ERR= and END= constructs are employed:

```
      READ 30, J, I
30    FORMAT (I<J>)
```

18.3.14 Explicit Assembler Code

g77 needs to provide some way, a la gcc, for g77 code to specify explicit assembler code.

18.3.15 Q Edit Descriptor

The Q edit descriptor in FORMATs isn't supported. (This is meant to get the number of characters remaining in an input record.) Supporting this requires a significant redesign or replacement of libf2c.

A workaround might be using internal I/O or the stream-based intrinsics. See Section 10.11.9.104 [FGetC Intrinsic (subroutine)], page 125.

18.3.16 Old-style PARAMETER Statements

g77 doesn't accept 'PARAMETER I=1'. Supporting this obsolete form of the PARAMETER statement would not be particularly hard, as most of the parsing code is already in place and working.

Until time/money is spent implementing it, you might as well fix your code to use the standard form, 'PARAMETER (I=1)' (possibly needing 'INTEGER I' preceding the PARAMETER statement as well, otherwise, in the obsolete form of PARAMETER, the type of the variable is set from the type of the constant being assigned to it).

18.3.17 TYPE and ACCEPT I/O Statements

g77 doesn't support the I/O statements TYPE and ACCEPT. These are common extensions that should be easy to support, but also are fairly easy to work around in user code.

Generally, any 'TYPE fmt,list' I/O statement can be replaced by 'PRINT fmt,list'. And, any 'ACCEPT fmt,list' statement can be replaced by 'READ fmt,list'.

18.3.18 STRUCTURE, UNION, RECORD, MAP

g77 doesn't support STRUCTURE, UNION, RECORD, MAP. This set of extensions is quite a bit lower on the list of large, important things to add to g77, partly because it requires a great deal of work either upgrading or replacing libf2c.

18.3.19 OPEN, CLOSE, and INQUIRE Keywords

g77 doesn't have support for keywords such as DISP='DELETE' in the OPEN, CLOSE, and INQUIRE statements. These extensions are easy to add to g77 itself, but require much more work on libf2c.

18.3.20 ENCODE and DECODE

g77 doesn't support ENCODE or DECODE.

These statements are best replaced by READ and WRITE statements involving internal files (CHARACTER variables and arrays).

For example, replace a code fragment like

```
      INTEGER*1 LINE(80)
...
      DECODE (80, 9000, LINE) A, B, C
...
9000  FORMAT (1X, 3(F10.5))
```

with:

```
      CHARACTER*80 LINE
...
      READ (UNIT=LINE, FMT=9000) A, B, C
...
9000  FORMAT (1X, 3(F10.5))
```

Similarly, replace a code fragment like

```
      INTEGER*1 LINE(80)
...
      ENCODE (80, 9000, LINE) A, B, C
...
9000  FORMAT (1X, 'OUTPUT IS ', 3(F10.5))
```

with:

```
      CHARACTER*80 LINE
...
      WRITE (UNIT=LINE, FMT=9000) A, B, C
```

```
...
9000  FORMAT (1X, 'OUTPUT IS ', 3(F10.5))
```

It is entirely possible that `ENCODE` and `DECODE` will be supported by a future version of `g77`.

18.3.21 Suppressing Space Padding of Source Lines

`g77` should offer VXT-Fortran-style suppression of virtual spaces at the end of a source line if an appropriate command-line option is specified.

This affects cases where a character constant is continued onto the next line in a fixed-form source file, as in the following example:

```
10    PRINT *,'HOW MANY
     1 SPACES?'
```

`g77`, and many other compilers, virtually extend the continued line through column 72 with spaces that become part of the character constant, but Digital Fortran normally didn't, leaving only one space between '`MANY`' and '`SPACES?`' in the output of the above statement.

Fairly recently, at least one version of Digital Fortran was enhanced to provide the other behavior when a command-line option is specified, apparently due to demand from readers of the USENET group '`comp.lang.fortran`' to offer conformance to this widespread practice in the industry. `g77` should return the favor by offering conformance to Digital's approach to handling the above example.

18.3.22 Fortran Preprocessor

`g77` should offer a preprocessor designed specifically for Fortran to replace '`cpp -traditional`'. There are several out there worth evaluating, at least.

Such a preprocessor would recognize Hollerith constants, properly parse comments and character constants, and so on. It might also recognize, process, and thus preprocess files included via the `INCLUDE` directive.

18.3.23 Bit Operations on Floating-point Data

`g77` does not allow `REAL` and other non-integral types for arguments to intrinsics like `AND`, `OR`, and `SHIFT`.

For example, this program is rejected by `g77`, because the intrinsic `IAND` does not accept `REAL` arguments:

```
DATA A/7.54/, B/9.112/
PRINT *, IAND(A, B)
END
```

18.3.24 `POSIX` Standard

`g77` should support the POSIX standard for Fortran.

18.3.25 Floating-point Exception Handling

The `gcc` backend and, consequently, `g77`, currently provides no control over whether or not floating-point exceptions are trapped or ignored. (Ignoring them typically results in NaN values being propagated in systems that conform to IEEE 754.) The behaviour is inherited from the system-dependent startup code.

Most systems provide some C-callable mechanism to change this; this can be invoked at startup using `gcc`'s `constructor` attribute. For example, just compiling and linking the following C code with your program will turn on exception trapping for the "common" exceptions on an x86-based GNU system:

```
#include <fpu_control.h>
void __attribute__ ((constructor))
trapfpe () {
  (void) __setfpucw (_FPU_DEFAULT &
                     ~(_FPU_MASK_IM | _FPU_MASK_ZM | _FPU_MASK_OM));
}
```

18.3.26 Nonportable Conversions

`g77` doesn't accept some particularly nonportable, silent data-type conversions such as `LOGICAL` to `REAL` (as in '`A=.FALSE.`', where '`A`' is type `REAL`), that other compilers might quietly accept.

Some of these conversions are accepted by `g77` when the '`-fugly`' option is specified. Perhaps it should accept more or all of them.

18.3.27 Large Automatic Arrays

Currently, automatic arrays always are allocated on the stack. For situations where the stack cannot be made large enough, `g77` should offer a compiler option that specifies allocation of automatic arrays in heap storage.

18.3.28 Support for Threads

Neither the code produced by `g77` nor the `libf2c` library are thread-safe, nor does `g77` have support for parallel processing (other than the instruction-level parallelism available on some processors). A package such as PVM might help here.

18.3.29 Gracefully Handle Sensible Bad Code

`g77` generally should continue processing for warnings and recoverable (user) errors whenever possible—that is, it shouldn't gratuitously make bad or useless code.

For example:

```
INTRINSIC ZABS
CALL FOO(ZABS)
END
```

When compiling the above with '`-ff2c-intrinsics-disable`', `g77` should indeed complain about passing `ZABS`, but it still should compile, instead of rejecting the entire `CALL`

statement. (Some of this is related to improving the compiler internals to improve how statements are analyzed.)

18.3.30 Non-standard Conversions

'`-Wconversion`' and related should flag places where non-standard conversions are found. Perhaps much of this would be part of '`-Wugly*`'.

18.3.31 Non-standard Intrinsics

`g77` needs a new option, like '`-Wintrinsics`', to warn about use of non-standard intrinsics without explicit `INTRINSIC` statements for them. This would help find code that might fail silently when ported to another compiler.

18.3.32 Modifying `DO` Variable

`g77` should warn about modifying `DO` variables via `EQUIVALENCE`. (The internal information gathered to produce this warning might also be useful in setting the internal "doiter" flag for a variable or even array reference within a loop, since that might produce faster code someday.)

For example, this code is invalid, so `g77` should warn about the invalid assignment to '`NOTHER`':

```
EQUIVALENCE (I, NOTHER)
DO I = 1, 100
   IF (I.EQ. 10) NOTHER = 20
END DO
```

18.3.33 Better Pedantic Compilation

`g77` needs to support '`-fpedantic`' more thoroughly, and use it only to generate warnings instead of rejecting constructs outright. Have it warn: if a variable that dimensions an array is not a dummy or placed explicitly in `COMMON` (F77 does not allow it to be placed in `COMMON` via `EQUIVALENCE`); if specification statements follow statement-function-definition statements; about all sorts of syntactic extensions.

18.3.34 Warn About Implicit Conversions

`g77` needs a '`-Wpromotions`' option to warn if source code appears to expect automatic, silent, and somewhat dangerous compiler-assisted conversion of `REAL(KIND=1)` constants to `REAL(KIND=2)` based on context.

For example, it would warn about cases like this:

```
DOUBLE PRECISION FOO
PARAMETER (TZPHI = 9.435784839284958)
FOO = TZPHI * 3D0
```

18.3.35 Invalid Use of Hollerith Constant

g77 should disallow statements like 'RETURN 2HAB', which are invalid in both source forms (unlike 'RETURN (2HAB)', which probably still makes no sense but at least can be reliably parsed). Fixed-form processing rejects it, but not free-form, except in a way that is a bit difficult to understand.

18.3.36 Dummy Array Without Dimensioning Dummy

g77 should complain when a list of dummy arguments containing an adjustable dummy array does not also contain every variable listed in the dimension list of the adjustable array.

Currently, g77 does complain about a variable that dimensions an array but doesn't appear in any dummy list or COMMON area, but this needs to be extended to catch cases where it doesn't appear in every dummy list that also lists any arrays it dimensions.

For example, g77 should warn about the entry point 'ALT' below, since it includes 'ARRAY' but not 'ISIZE' in its list of arguments:

```
SUBROUTINE PRIMARY(ARRAY, ISIZE)
REAL ARRAY(ISIZE)
ENTRY ALT(ARRAY)
```

18.3.37 Invalid FORMAT Specifiers

g77 should check FORMAT specifiers for validity as it does FORMAT statements.

For example, a diagnostic would be produced for:

```
PRINT 'HI THERE!'  !User meant PRINT *, 'HI THERE!'
```

18.3.38 Ambiguous Dialects

g77 needs a set of options such as '-Wugly*', '-Wautomatic', '-Wvxt', '-Wf90', and so on. These would warn about places in the user's source where ambiguities are found, helpful in resolving ambiguities in the program's dialect or dialects.

18.3.39 Unused Labels

g77 should warn about unused labels when '-Wunused' is in effect.

18.3.40 Informational Messages

g77 needs an option to suppress information messages (notes). '-w' does this but also suppresses warnings. The default should be to suppress info messages.

Perhaps info messages should simply be eliminated.

18.3.41 Uninitialized Variables at Run Time

`g77` needs an option to initialize everything (not otherwise explicitly initialized) to "weird" (machine-dependent) values, e.g. NaNs, bad (non-`NULL`) pointers, and largest-magnitude integers, would help track down references to some kinds of uninitialized variables at run time.

Note that use of the options '`-O -Wuninitialized`' can catch many such bugs at compile time.

18.3.42 Bounds Checking at Run Time

`g77` should offer run-time bounds-checking of array/subscript references in a fashion similar to `f2c`.

Note that `g77` already warns about references to out-of-bounds elements of arrays when it detects these at compile time.

18.3.43 Labels Visible to Debugger

`g77` should output debugging information for statements labels, for use by debuggers that know how to support them. Same with weirder things like construct names. It is not yet known if any debug formats or debuggers support these.

18.4 Disappointments and Misunderstandings

These problems are perhaps regrettable, but we don't know any practical way around them for now.

18.4.1 Mangling of Names in Source Code

The current external-interface design, which includes naming of external procedures, COMMON blocks, and the library interface, has various usability problems, including things like adding underscores where not really necessary (and preventing easier inter-language operability) and yet not providing complete namespace freedom for user C code linked with Fortran apps (due to the naming of functions in the library, among other things).

Project GNU should at least get all this "right" for systems it fully controls, such as the Hurd, and provide defaults and options for compatibility with existing systems and interoperability with popular existing compilers.

18.4.2 Multiple Definitions of External Names

`g77` doesn't allow a common block and an external procedure or `BLOCK DATA` to have the same name. Some systems allow this, but `g77` does not, to be compatible with `f2c`.

`g77` could special-case the way it handles `BLOCK DATA`, since it is not compatible with `f2c` in this particular area (necessarily, since `g77` offers an important feature here), but it is likely that such special-casing would be very annoying to people with programs that use '`EXTERNAL FOO`', with no other mention of '`FOO`' in the same program unit, to refer to

external procedures, since the result would be that `g77` would treat these references as requests to force-load BLOCK DATA program units.

In that case, if `g77` modified names of `BLOCK DATA` so they could have the same names as `COMMON`, users would find that their programs wouldn't link because the '`FOO`' procedure didn't have its name translated the same way.

(Strictly speaking, `g77` could emit a null-but-externally-satisfying definition of '`FOO`' with its name transformed as if it had been a `BLOCK DATA`, but that probably invites more trouble than it's worth.)

18.4.3 Limitation on Implicit Declarations

`g77` disallows `IMPLICIT CHARACTER*(*)`. This is not standard-conforming.

18.5 Certain Changes We Don't Want to Make

This section lists changes that people frequently request, but which we do not make because we think GNU Fortran is better without them.

18.5.1 Backslash in Constants

In the opinion of many experienced Fortran users, '`-fno-backslash`' should be the default, not '`-fbackslash`', as currently set by `g77`.

First of all, you can always specify '`-fno-backslash`' to turn off this processing.

Despite not being within the spirit (though apparently within the letter) of the ANSI FORTRAN 77 standard, `g77` defaults to '`-fbackslash`' because that is what most UNIX `f77` commands default to, and apparently lots of code depends on this feature.

This is a particularly troubling issue. The use of a C construct in the midst of Fortran code is bad enough, worse when it makes existing Fortran programs stop working (as happens when programs written for non-UNIX systems are ported to UNIX systems with compilers that provide the '`-fbackslash`' feature as the default—sometimes with no option to turn it off).

The author of GNU Fortran wished, for reasons of linguistic purity, to make '`-fno-backslash`' the default for GNU Fortran and thus require users of UNIX `f77` and `f2c` to specify '`-fbackslash`' to get the UNIX behavior.

However, the realization that `g77` is intended as a replacement for *UNIX* `f77`, caused the author to choose to make `g77` as compatible with `f77` as feasible, which meant making '`-fbackslash`' the default.

The primary focus on compatibility is at the source-code level, and the question became "What will users expect a replacement for `f77` to do, by default?" Although at least one UNIX `f77` does not provide '`-fbackslash`' as a default, it appears that the majority of them do, which suggests that the majority of code that is compiled by UNIX `f77` compilers expects '`-fbackslash`' to be the default.

It is probably the case that more code exists that would *not* work with '`-fbackslash`' in force than code that requires it be in force.

However, most of *that* code is not being compiled with `f77`, and when it is, new build procedures (shell scripts, makefiles, and so on) must be set up anyway so that they work under UNIX. That makes a much more natural and safe opportunity for non-UNIX users to adapt their build procedures for `g77`'s default of '`-fbackslash`' than would exist for the majority of UNIX `f77` users who would have to modify existing, working build procedures to explicitly specify '`-fbackslash`' if that was not the default.

One suggestion has been to configure the default for '`-fbackslash`' (and perhaps other options as well) based on the configuration of `g77`.

This is technically quite straightforward, but will be avoided even in cases where not configuring defaults to be dependent on a particular configuration greatly inconveniences some users of legacy code.

Many users appreciate the GNU compilers because they provide an environment that is uniform across machines. These users would be inconvenienced if the compiler treated things like the format of the source code differently on certain machines.

Occasionally users write programs intended only for a particular machine type. On these occasions, the users would benefit if the GNU Fortran compiler were to support by default the same dialect as the other compilers on that machine. But such applications are rare. And users writing a program to run on more than one type of machine cannot possibly benefit from this kind of compatibility. (This is consistent with the design goals for `gcc`. To change them for `g77`, you must first change them for `gcc`. Do not ask the maintainers of `g77` to do this for you, or to disassociate `g77` from the widely understood, if not widely agreed-upon, goals for GNU compilers in general.)

This is why GNU Fortran does and will treat backslashes in the same fashion on all types of machines (by default). See Section 10.1 [Direction of Language Development], page 71, for more information on this overall philosophy guiding the development of the GNU Fortran language.

Of course, users strongly concerned about portability should indicate explicitly in their build procedures which options are expected by their source code, or write source code that has as few such expectations as possible.

For example, avoid writing code that depends on backslash ('\') being interpreted either way in particular, such as by starting a program unit with:

```
CHARACTER BACKSL
PARAMETER (BACKSL = '\\')
```

Then, use concatenation of '`BACKSL`' anyplace a backslash is desired. In this way, users can write programs which have the same meaning in many Fortran dialects.

(However, this technique does not work for Hollerith constants—which is just as well, since the only generally portable uses for Hollerith constants are in places where character constants can and should be used instead, for readability.)

18.5.2 Initializing Before Specifying

`g77` does not allow '`DATA VAR/1/`' to appear in the source code before '`COMMON VAR`', '`DIMENSION VAR(10)`', '`INTEGER VAR`', and so on. In general, `g77` requires initialization of a variable or array to be specified *after* all other specifications of attributes (type, size,

placement, and so on) of that variable or array are specified (though *confirmation* of data type is permitted).

It is *possible* g77 will someday allow all of this, even though it is not allowed by the FORTRAN 77 standard.

Then again, maybe it is better to have g77 always require placement of DATA so that it can possibly immediately write constants to the output file, thus saving time and space.

That is, 'DATA A/1000000*1/' should perhaps always be immediately writable to canonical assembler, unless it's already known to be in a COMMON area following as-yet-uninitialized stuff, and to do this it cannot be followed by 'COMMON A'.

18.5.3 Context-Sensitive Intrinsicness

g77 treats procedure references to *possible* intrinsic names as always enabling their intrinsic nature, regardless of whether the *form* of the reference is valid for that intrinsic.

For example, 'CALL SQRT' is interpreted by g77 as an invalid reference to the SQRT intrinsic function, because the reference is a subroutine invocation.

First, g77 recognizes the statement 'CALL SQRT' as a reference to a *procedure* named 'SQRT', not to a *variable* with that name (as it would for a statement such as 'V = SQRT').

Next, g77 establishes that, in the program unit being compiled, SQRT is an intrinsic—not a subroutine that happens to have the same name as an intrinsic (as would be the case if, for example, 'EXTERNAL SQRT' was present).

Finally, g77 recognizes that the *form* of the reference is invalid for that particular intrinsic. That is, it recognizes that it is invalid for an intrinsic *function*, such as SQRT, to be invoked as a *subroutine*.

At that point, g77 issues a diagnostic.

Some users claim that it is "obvious" that 'CALL SQRT' references an external subroutine of their own, not an intrinsic function.

However, g77 knows about intrinsic subroutines, not just functions, and is able to support both having the same names, for example.

As a result of this, g77 rejects calls to intrinsics that are not subroutines, and function invocations of intrinsics that are not functions, just as it (and most compilers) rejects invocations of intrinsics with the wrong number (or types) of arguments.

So, use the 'EXTERNAL SQRT' statement in a program unit that calls a user-written subroutine named 'SQRT'.

18.5.4 Context-Sensitive Constants

g77 does not use context to determine the types of constants or named constants (PARAMETER), except for (non-standard) typeless constants such as ''123'O'.

For example, consider the following statement:

```
PRINT *, 9.435784839284958 * 2D0
```

g77 will interpret the (truncated) constant '9.435784839284958' as a REAL(KIND=1), not REAL(KIND=2), constant, because the suffix D0 is not specified.

As a result, the output of the above statement when compiled by `g77` will appear to have "less precision" than when compiled by other compilers.

In these and other cases, some compilers detect the fact that a single-precision constant is used in a double-precision context and therefore interpret the single-precision constant as if it was *explicitly* specified as a double-precision constant. (This has the effect of appending *decimal*, not *binary*, zeros to the fractional part of the number—producing different computational results.)

The reason this misfeature is dangerous is that a slight, apparently innocuous change to the source code can change the computational results. Consider:

```
REAL ALMOST, CLOSE
DOUBLE PRECISION FIVE
PARAMETER (ALMOST = 5.000000000001)
FIVE = 5
CLOSE = 5.000000000001
PRINT *, 5.000000000001 - FIVE
PRINT *, ALMOST - FIVE
PRINT *, CLOSE - FIVE
END
```

Running the above program should result in the same value being printed three times. With `g77` as the compiler, it does.

However, compiled by many other compilers, running the above program would print two or three distinct values, because in two or three of the statements, the constant '`5.000000000001`', which on most systems is exactly equal to '`5.`' when interpreted as a single-precision constant, is instead interpreted as a double-precision constant, preserving the represented precision. However, this "clever" promotion of type does not extend to variables or, in some compilers, to named constants.

Since programmers often are encouraged to replace manifest constants or permanently-assigned variables with named constants (`PARAMETER` in Fortran), and might need to replace some constants with variables having the same values for pertinent portions of code, it is important that compilers treat code so modified in the same way so that the results of such programs are the same. `g77` helps in this regard by treating constants just the same as variables in terms of determining their types in a context-independent way.

Still, there is a lot of existing Fortran code that has been written to depend on the way other compilers freely interpret constants' types based on context, so anything `g77` can do to help flag cases of this in such code could be very helpful.

18.5.5 Equivalence Versus Equality

Use of `.EQ.` and `.NE.` on `LOGICAL` operands is not supported, except via '`-fugly`', which is not recommended except for legacy code (where the behavior expected by the *code* is assumed).

Legacy code should be changed, as resources permit, to use `.EQV.` and `.NEQV.` instead, as these are permitted by the various Fortran standards.

New code should never be written expecting `.EQ.` or `.NE.` to work if either of its operands is `LOGICAL`.

The problem with supporting this "feature" is that there is unlikely to be consensus on how it works, as illustrated by the following sample program:

```
LOGICAL L,M,N
DATA L,M,N /3*.FALSE./
IF (L.AND.M.EQ.N) PRINT *,'L.AND.M.EQ.N'
END
```

The issue raised by the above sample program is: what is the precedence of `.EQ.` (and `.NE.`) when applied to `LOGICAL` operands?

Some programmers will argue that it is the same as the precedence for `.EQ.` when applied to numeric (such as `INTEGER`) operands. By this interpretation, the subexpression '`M.EQ.N`' must be evaluated first in the above program, resulting in a program that, when run, does not execute the `PRINT` statement.

Other programmers will argue that the precedence is the same as the precedence for `.EQV.`, which is restricted by the standards to `LOGICAL` operands. By this interpretation, the subexpression '`L.AND.M`' must be evaluated first, resulting in a program that *does* execute the `PRINT` statement.

Assigning arbitrary semantic interpretations to syntactic expressions that might legitimately have more than one "obvious" interpretation is generally unwise.

The creators of the various Fortran standards have done a good job in this case, requiring a distinct set of operators (which have their own distinct precedence) to compare `LOGICAL` operands. This requirement results in expression syntax with more certain precedence (without requiring substantial context), making it easier for programmers to read existing code. `g77` will avoid muddying up elements of the Fortran language that were well-designed in the first place.

(Ask C programmers about the precedence of expressions such as '`(a) & (b)`' and '`(a) - (b)`'—they cannot even tell you, without knowing more context, whether the '`&`' and '`-`' operators are infix (binary) or unary!)

18.5.6 Order of Side Effects

`g77` does not necessarily produce code that, when run, performs side effects (such as those performed by function invocations) in the same order as in some other compiler—or even in the same order as another version, port, or invocation (using different command-line options) of `g77`.

It is never safe to depend on the order of evaluation of side effects. For example, an expression like this may very well behave differently from one compiler to another:

```
J = IFUNC() - IFUNC()
```

There is no guarantee that '`IFUNC`' will be evaluated in any particular order. Either invocation might happen first. If '`IFUNC`' returns 5 the first time it is invoked, and returns 12 the second time, '`J`' might end up with the value '`7`', or it might end up with '`-7`'.

Generally, in Fortran, procedures with side-effects intended to be visible to the caller are best designed as *subroutines*, not functions. Examples of such side-effects include:

- The generation of random numbers that are intended to influence return values.
- Performing I/O (other than internal I/O to local variables).

- Updating information in common blocks.

An example of a side-effect that is not intended to be visible to the caller is a function that maintains a cache of recently calculated results, intended solely to speed repeated invocations of the function with identical arguments. Such a function can be safely used in expressions, because if the compiler optimizes away one or more calls to the function, operation of the program is unaffected (aside from being speeded up).

18.6 Warning Messages and Error Messages

The GNU compiler can produce two kinds of diagnostics: errors and warnings. Each kind has a different purpose:

Errors report problems that make it impossible to compile your program. GNU Fortran reports errors with the source file name, line number, and column within the line where the problem is apparent.

Warnings report other unusual conditions in your code that *might* indicate a problem, although compilation can (and does) proceed. Warning messages also report the source file name, line number, and column information, but include the text '`warning:`' to distinguish them from error messages.

Warnings might indicate danger points where you should check to make sure that your program really does what you intend; or the use of obsolete features; or the use of nonstandard features of GNU Fortran. Many warnings are issued only if you ask for them, with one of the '`-W`' options (for instance, '`-Wall`' requests a variety of useful warnings).

Note: Currently, the text of the line and a pointer to the column is printed in most `g77` diagnostics. Probably, as of version 0.6, `g77` will no longer print the text of the source line, instead printing the column number following the file name and line number in a form that GNU Emacs recognizes. This change is expected to speed up and reduce the memory usage of the `g77` compiler.

See Section 7.5 [Options to Request or Suppress Warnings], page 35, for more detail on these and related command-line options.

19 Open Questions

Please consider offering useful answers to these questions!

- How do system administrators and users manage multiple incompatible Fortran compilers on their systems? How can `g77` contribute to this, or at least avoiding intefering with it?

  Currently, `g77` provides rudimentary ways to choose whether to overwrite portions of other Fortran compilation systems (such as the `f77` command and the `libf2c` library). Is this sufficient? What happens when users choose not to overwrite these—does `g77` work properly in all such installations, picking up its own versions, or does it pick up the existing "alien" versions it didn't overwrite with its own, possibly leading to subtle bugs?

- `LOC()` and other intrinsics are probably somewhat misclassified. Is the a need for more precise classification of intrinsics, and if so, what are the appropriate groupings? Is there a need to individually enable/disable/delete/hide intrinsics from the command line?

20 Reporting Bugs

Your bug reports play an essential role in making GNU Fortran reliable.

When you encounter a problem, the first thing to do is to see if it is already known. See Chapter 18 [Trouble], page 265. If it isn't known, then you should report the problem.

Reporting a bug might help you by bringing a solution to your problem, or it might not. (If it does not, look in the service directory; see Chapter 21 [Service], page 303.) In any case, the principal function of a bug report is to help the entire community by making the next version of GNU Fortran work better. Bug reports are your contribution to the maintenance of GNU Fortran.

Since the maintainers are very overloaded, we cannot respond to every bug report. However, if the bug has not been fixed, we are likely to send you a patch and ask you to tell us whether it works.

In order for a bug report to serve its purpose, you must include the information that makes for fixing the bug.

See Chapter 18 [Known Causes of Trouble with GNU Fortran], page 265, for information on problems we already know about.

See Chapter 21 [How To Get Help with GNU Fortran], page 303, for information on where to ask for help.

20.1 Have You Found a Bug?

If you are not sure whether you have found a bug, here are some guidelines:

- If the compiler gets a fatal signal, for any input whatever, that is a compiler bug. Reliable compilers never crash—they just remain obsolete.
- If the compiler produces invalid assembly code, for any input whatever, that is a compiler bug, unless the compiler reports errors (not just warnings) which would ordinarily prevent the assembler from being run.
- If the compiler produces valid assembly code that does not correctly execute the input source code, that is a compiler bug.

  However, you must double-check to make sure, because you might have run into an incompatibility between GNU Fortran and traditional Fortran. These incompatibilities might be considered bugs, but they are inescapable consequences of valuable features.

  Or you might have a program whose behavior is undefined, which happened by chance to give the desired results with another Fortran compiler. It is best to check the relevant Fortran standard thoroughly if it is possible that the program indeed does something undefined.

  After you have localized the error to a single source line, it should be easy to check for these things. If your program is correct and well defined, you have found a compiler bug.

  It might help if, in your submission, you identified the specific language in the relevant Fortran standard that specifies the desired behavior, if it isn't likely to be obvious and agreed-upon by all Fortran users.

- If the compiler produces an error message for valid input, that is a compiler bug.
- If the compiler does not produce an error message for invalid input, that is a compiler bug. However, you should note that your idea of "invalid input" might be someone else's idea of "an extension" or "support for traditional practice".
- If you are an experienced user of Fortran compilers, your suggestions for improvement of GNU Fortran are welcome in any case.

Many, perhaps most, bug reports against `g77` turn out to be bugs in the user's code. While we find such bug reports educational, they sometimes take a considerable amount of time to track down or at least respond to—time we could be spending making `g77`, not some user's code, better.

Some steps you can take to verify that the bug is not certainly in the code you're compiling with `g77`:

- Compile your code using the `g77` options '`-W -Wall -O`'. These options enable many useful warning; the '`-O`' option enables flow analysis that enables the uninitialized-variable warning.

  If you investigate the warnings and find evidence of possible bugs in your code, fix them first and retry `g77`.
- Compile your code using the `g77` options '`-finit-local-zero`', '`-fno-automatic`', '`-ffloat-store`', and various combinations thereof.

  If your code works with any of these combinations, that is not proof that the bug isn't in `g77`—a `g77` bug exposed by your code might simply be avoided, or have a different, more subtle effect, when different options are used—but it can be a strong indicator that your code is making unawarranted assumptions about the Fortran dialect and/or underlying machine it is being compiled and run on.

  See Section 17.5 [Overly Convenient Command-Line Options], page 261, for information on the '`-fno-automatic`' and '`-finit-local-zero`' options and how to convert their use into selective changes in your own code.
- Validate your code with `ftnchek` or a similar code-checking tool. `ftncheck` can be found at `ftp://ftp.netlib.org/fortran` or `ftp://ftp.dsm.fordham.edu`.

  Here are some sample '`Makefile`' rules using `ftnchek` "project" files to do cross-file checking and `sfmakedepend` (from `ftp://ahab.rutgers.edu/pub/perl/sfmakedepend`) to maintain dependencies automatically. These assume the use of GNU `make`.

```
# Dummy suffix for ftnchek targets:
.SUFFIXES: .chek
.PHONY: chekall

# How to compile .f files (for implicit rule):
FC = g77
# Assume `include' directory:
FFLAGS = -Iinclude -g -O -Wall

# Flags for ftnchek:
CHEK1 = -array=0 -include=includes -noarray
CHEK2 = -nonovice -usage=1 -notruncation
```

```
CHEKFLAGS = $(CHEK1) $(CHEK2)

# Run ftnchek with all the .prj files except the one corresponding
# to the target's root:
%.chek : %.f ; \
  ftnchek $(filter-out $*.prj,$(PRJS)) $(CHEKFLAGS) \
    -noextern -library $<

# Derive a project file from a source file:
%.prj : %.f ; \
  ftnchek $(CHEKFLAGS) -noextern -project -library $<

# The list of objects is assumed to be in variable OBJS.
# Sources corresponding to the objects:
SRCS = $(OBJS:%.o=%.f)
# ftnchek project files:
PRJS = $(OBJS:%.o=%.prj)

# Build the program
prog: $(OBJS) ; \
  $(FC) -o $ $(OBJS)

chekall: $(PRJS) ; \
  ftnchek $(CHEKFLAGS) $(PRJS)

prjs: $(PRJS)

# For Emacs M-x find-tag:
TAGS: $(SRCS) ; \
  etags $(SRCS)

# Rebuild dependencies:
depend: ; \
  sfmakedepend -I $(PLTLIBDIR) -I includes -a prj $(SRCS1)
```

- Try your code out using other Fortran compilers, such as `f2c`. If it does not work on at least one other compiler (assuming the compiler supports the features the code needs), that is a strong indicator of a bug in the code.

  However, even if your code works on many compilers *except* `g77`, that does *not* mean the bug is in `g77`. It might mean the bug is in your code, and that `g77` simply exposes it more readily than other compilers.

20.2 Where to Report Bugs

Send bug reports for GNU Fortran to `fortran@gnu.org`.

Often people think of posting bug reports to a newsgroup instead of mailing them. This sometimes appears to work, but it has one problem which can be crucial: a newsgroup posting does not contain a mail path back to the sender. Thus, if maintainers need more

information, they might be unable to reach you. For this reason, you should always send bug reports by mail to the proper mailing list.

As a last resort, send bug reports on paper to:

```
GNU Compiler Bugs
Free Software Foundation
59 Temple Place - Suite 330
Boston, MA 02111-1307, USA
```

20.3 How to Report Bugs

The fundamental principle of reporting bugs usefully is this: **report all the facts**. If you are not sure whether to state a fact or leave it out, state it!

Often people omit facts because they think they know what causes the problem and they conclude that some details don't matter. Thus, you might assume that the name of the variable you use in an example does not matter. Well, probably it doesn't, but one cannot be sure. Perhaps the bug is a stray memory reference which happens to fetch from the location where that name is stored in memory; perhaps, if the name were different, the contents of that location would fool the compiler into doing the right thing despite the bug. Play it safe and give a specific, complete example. That is the easiest thing for you to do, and the most helpful.

Keep in mind that the purpose of a bug report is to enable someone to fix the bug if it is not known. It isn't very important what happens if the bug is already known. Therefore, always write your bug reports on the assumption that the bug is not known.

Sometimes people give a few sketchy facts and ask, "Does this ring a bell?" This cannot help us fix a bug, so it is rarely helpful. We respond by asking for enough details to enable us to investigate. You might as well expedite matters by sending them to begin with. (Besides, there are enough bells ringing around here as it is.)

Try to make your bug report self-contained. If we have to ask you for more information, it is best if you include all the previous information in your response, as well as the information that was missing.

Please report each bug in a separate message. This makes it easier for us to track which bugs have been fixed and to forward your bugs reports to the appropriate maintainer.

Do not compress and encode any part of your bug report using programs such as '`uuencode`'. If you do so it will slow down the processing of your bug. If you must submit multiple large files, use '`shar`', which allows us to read your message without having to run any decompression programs.

(As a special exception for GNU Fortran bug-reporting, at least for now, if you are sending more than a few lines of code, if your program's source file format contains "interesting" things like trailing spaces or strange characters, or if you need to include binary data files, it is acceptable to put all the files together in a `tar` archive, and, whether you need to do that, it is acceptable to then compress the single file (`tar` archive or source file) using `gzip` and encode it via `uuencode`. Do not use any MIME stuff—the current maintainer can't decode this. Using `compress` instead of `gzip` is acceptable, assuming you have licensed the use of the patented algorithm in `compress` from Unisys.)

To enable someone to investigate the bug, you should include all these things:

- The version of GNU Fortran. You can get this by running `g77` with the '`-v`' option. (Ignore any error messages that might be displayed when the linker is run.)

  Without this, we won't know whether there is any point in looking for the bug in the current version of GNU Fortran.
- A complete input file that will reproduce the bug. If the bug is in the compiler proper ('`f771`') and you are using the C preprocessor, run your source file through the C preprocessor by doing '`g77 -E` *sourcefile* `>` *outfile*', then include the contents of *outfile* in the bug report. (When you do this, use the same '`-I`', '`-D`' or '`-U`' options that you used in actual compilation.)

  A single statement is not enough of an example. In order to compile it, it must be embedded in a complete file of compiler input; and the bug might depend on the details of how this is done.

  Without a real example one can compile, all anyone can do about your bug report is wish you luck. It would be futile to try to guess how to provoke the bug. For example, bugs in register allocation and reloading frequently depend on every little detail of the function they happen in.
- Note that you should include with your bug report any files included by the source file (via the `#include` or `INCLUDE` directive) that you send, and any files they include, and so on.

  It is not necessary to replace the `#include` and `INCLUDE` directives with the actual files in the version of the source file that you send, but it might make submitting the bug report easier in the end. However, be sure to *reproduce* the bug using the *exact* version of the source material you submit, to avoid wild-goose chases.
- The command arguments you gave GNU Fortran to compile that example and observe the bug. For example, did you use '`-O`'? To guarantee you won't omit something important, list all the options.

  If we were to try to guess the arguments, we would probably guess wrong and then we would not encounter the bug.
- The type of machine you are using, and the operating system name and version number. (Much of this information is printed by '`g77 -v`'—if you include that, send along any additional info you have that you don't see clearly represented in that output.)
- The operands you gave to the `configure` command when you installed the compiler.
- A complete list of any modifications you have made to the compiler source. (We don't promise to investigate the bug unless it happens in an unmodified compiler. But if you've made modifications and don't tell us, then you are sending us on a wild-goose chase.)

  Be precise about these changes. A description in English is not enough—send a context diff for them.

  Adding files of your own (such as a machine description for a machine we don't support) is a modification of the compiler source.
- Details of any other deviations from the standard procedure for installing GNU Fortran.
- A description of what behavior you observe that you believe is incorrect. For example, "The compiler gets a fatal signal," or, "The assembler instruction at line 208 in the output is incorrect."

Of course, if the bug is that the compiler gets a fatal signal, then one can't miss it. But if the bug is incorrect output, the maintainer might not notice unless it is glaringly wrong. None of us has time to study all the assembler code from a 50-line Fortran program just on the chance that one instruction might be wrong. We need *you* to do this part!

Even if the problem you experience is a fatal signal, you should still say so explicitly. Suppose something strange is going on, such as, your copy of the compiler is out of synch, or you have encountered a bug in the C library on your system. (This has happened!) Your copy might crash and the copy here would not. If you *said* to expect a crash, then when the compiler here fails to crash, we would know that the bug was not happening. If you don't say to expect a crash, then we would not know whether the bug was happening. We would not be able to draw any conclusion from our observations.

If the problem is a diagnostic when building GNU Fortran with some other compiler, say whether it is a warning or an error.

Often the observed symptom is incorrect output when your program is run. Sad to say, this is not enough information unless the program is short and simple. None of us has time to study a large program to figure out how it would work if compiled correctly, much less which line of it was compiled wrong. So you will have to do that. Tell us which source line it is, and what incorrect result happens when that line is executed. A person who understands the program can find this as easily as finding a bug in the program itself.

- If you send examples of assembler code output from GNU Fortran, please use '`-g`' when you make them. The debugging information includes source line numbers which are essential for correlating the output with the input.
- If you wish to mention something in the GNU Fortran source, refer to it by context, not by line number.

  The line numbers in the development sources don't match those in your sources. Your line numbers would convey no convenient information to the maintainers.
- Additional information from a debugger might enable someone to find a problem on a machine which he does not have available. However, you need to think when you collect this information if you want it to have any chance of being useful.

  For example, many people send just a backtrace, but that is never useful by itself. A simple backtrace with arguments conveys little about GNU Fortran because the compiler is largely data-driven; the same functions are called over and over for different RTL insns, doing different things depending on the details of the insn.

  Most of the arguments listed in the backtrace are useless because they are pointers to RTL list structure. The numeric values of the pointers, which the debugger prints in the backtrace, have no significance whatever; all that matters is the contents of the objects they point to (and most of the contents are other such pointers).

  In addition, most compiler passes consist of one or more loops that scan the RTL insn sequence. The most vital piece of information about such a loop—which insn it has reached—is usually in a local variable, not in an argument.

  What you need to provide in addition to a backtrace are the values of the local variables for several stack frames up. When a local variable or an argument is an RTX, first

print its value and then use the GDB command `pr` to print the RTL expression that it points to. (If GDB doesn't run on your machine, use your debugger to call the function `debug_rtx` with the RTX as an argument.) In general, whenever a variable is a pointer, its value is no use without the data it points to.

Here are some things that are not necessary:

- A description of the envelope of the bug.

  Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it.

  This is often time consuming and not very useful, because the way we will find the bug is by running a single example under the debugger with breakpoints, not by pure deduction from a series of examples. You might as well save your time for something else.

  Of course, if you can find a simpler example to report *instead* of the original one, that is a convenience. Errors in the output will be easier to spot, running under the debugger will take less time, etc. Most GNU Fortran bugs involve just one function, so the most straightforward way to simplify an example is to delete all the function definitions except the one where the bug occurs. Those earlier in the file may be replaced by external declarations if the crucial function depends on them. (Exception: inline functions might affect compilation of functions defined later in the file.)

  However, simplification is not vital; if you don't want to do this, report the bug anyway and send the entire test case you used.

- In particular, some people insert conditionals '`#ifdef BUG`' around a statement which, if removed, makes the bug not happen. These are just clutter; we won't pay any attention to them anyway. Besides, you should send us preprocessor output, and that can't have conditionals.

- A patch for the bug.

  A patch for the bug is useful if it is a good one. But don't omit the necessary information, such as the test case, on the assumption that a patch is all we need. We might see problems with your patch and decide to fix the problem another way, or we might not understand it at all.

  Sometimes with a program as complicated as GNU Fortran it is very hard to construct an example that will make the program follow a certain path through the code. If you don't send the example, we won't be able to construct one, so we won't be able to verify that the bug is fixed.

  And if we can't understand what bug you are trying to fix, or why your patch should be an improvement, we won't install it. A test case will help us to understand.

  See Section 20.4 [Sending Patches], page 300, for guidelines on how to make it easy for us to understand and install your patches.

- A guess about what the bug is or what it depends on.

  Such guesses are usually wrong. Even the maintainer can't guess right about such things without first using the debugger to find the facts.

- A core dump file.

  We have no way of examining a core dump for your type of machine unless we have an identical system—and if we do have one, we should be able to reproduce the crash ourselves.

20.4 Sending Patches for GNU Fortran

If you would like to write bug fixes or improvements for the GNU Fortran compiler, that is very helpful. Send suggested fixes to the bug report mailing list, `fortran@gnu.org`.

Please follow these guidelines so we can study your patches efficiently. If you don't follow these guidelines, your information might still be useful, but using it will take extra work. Maintaining GNU Fortran is a lot of work in the best of circumstances, and we can't keep up unless you do your best to help.

- Send an explanation with your changes of what problem they fix or what improvement they bring about. For a bug fix, just include a copy of the bug report, and explain why the change fixes the bug.

  (Referring to a bug report is not as good as including it, because then we will have to look it up, and we have probably already deleted it if we've already fixed the bug.)
- Always include a proper bug report for the problem you think you have fixed. We need to convince ourselves that the change is right before installing it. Even if it is right, we might have trouble judging it if we don't have a way to reproduce the problem.
- Include all the comments that are appropriate to help people reading the source in the future understand why this change was needed.
- Don't mix together changes made for different reasons. Send them *individually*.

  If you make two changes for separate reasons, then we might not want to install them both. We might want to install just one. If you send them all jumbled together in a single set of diffs, we have to do extra work to disentangle them—to figure out which parts of the change serve which purpose. If we don't have time for this, we might have to ignore your changes entirely.

  If you send each change as soon as you have written it, with its own explanation, then the two changes never get tangled up, and we can consider each one properly without any extra work to disentangle them.

  Ideally, each change you send should be impossible to subdivide into parts that we might want to consider separately, because each of its parts gets its motivation from the other parts.
- Send each change as soon as that change is finished. Sometimes people think they are helping us by accumulating many changes to send them all together. As explained above, this is absolutely the worst thing you could do.

  Since you should send each change separately, you might as well send it right away. That gives us the option of installing it immediately if it is important.
- Use '`diff -c`' to make your diffs. Diffs without context are hard for us to install reliably. More than that, they make it hard for us to study the diffs to decide whether we want to install them. Unidiff format is better than contextless diffs, but not as easy to read as '`-c`' format.

If you have GNU `diff`, use '`diff -cp`', which shows the name of the function that each change occurs in. (The maintainer of GNU Fortran currently uses '`diff -rcp2N`'.)

- Write the change log entries for your changes. We get lots of changes, and we don't have time to do all the change log writing ourselves.

  Read the '`ChangeLog`' file to see what sorts of information to put in, and to learn the style that we use. The purpose of the change log is to show people where to find what was changed. So you need to be specific about what functions you changed; in large functions, it's often helpful to indicate where within the function the change was.

  On the other hand, once you have shown people where to find the change, you need not explain its purpose. Thus, if you add a new function, all you need to say about it is that it is new. If you feel that the purpose needs explaining, it probably does—but the explanation will be much more useful if you put it in comments in the code.

  If you would like your name to appear in the header line for who made the change, send us the header line.

- When you write the fix, keep in mind that we can't install a change that would break other systems.

  People often suggest fixing a problem by changing machine-independent files such as '`toplev.c`' to do something special that a particular system needs. Sometimes it is totally obvious that such changes would break GNU Fortran for almost all users. We can't possibly make a change like that. At best it might tell us how to write another patch that would solve the problem acceptably.

  Sometimes people send fixes that *might* be an improvement in general—but it is hard to be sure of this. It's hard to install such changes because we have to study them very carefully. Of course, a good explanation of the reasoning by which you concluded the change was correct can help convince us.

  The safest changes are changes to the configuration files for a particular machine. These are safe because they can't create new bugs on other machines.

  Please help us keep up with the workload by designing the patch in a form that is good to install.

21 How To Get Help with GNU Fortran

If you need help installing, using or changing GNU Fortran, there are two ways to find it:

- Look in the service directory for someone who might help you for a fee. The service directory is found in the file named '`SERVICE`' in the GNU CC distribution.
- Send a message to `fortran@gnu.org`.

22 Adding Options

To add a new command-line option to `g77`, first decide what kind of option you wish to add. Search the `g77` and `gcc` documentation for one or more options that is most closely like the one you want to add (in terms of what kind of effect it has, and so on) to help clarify its nature.

- *Fortran options* are options that apply only when compiling Fortran programs. They are accepted by `g77` and `gcc`, but they apply only when compiling Fortran programs.
- *Compiler options* are options that apply when compiling most any kind of program.

Fortran options are listed in the file '`gcc/f/lang-options.h`', which is used during the build of `gcc` to build a list of all options that are accepted by at least one language's compiler. This list goes into the '`lang_options`' array in '`gcc/toplev.c`', which uses this array to determine whether a particular option should be offered to the linked-in front end for processing by calling '`lang_option_decode`', which, for `g77`, is in '`gcc/f/com.c`' and just calls '`ffe_decode_option`'.

If the linked-in front end "rejects" a particular option passed to it, '`toplev.c`' just ignores the option, because *some* language's compiler is willing to accept it.

This allows commands like '`gcc -fno-asm foo.c bar.f`' to work, even though Fortran compilation does not currently support the '`-fno-asm`' option; even though the `f771` version of '`lang_decode_option`' rejects '`-fno-asm`', '`toplev.c`' doesn't produce a diagnostic because some other language (C) does accept it.

This also means that commands like '`g77 -fno-asm foo.f`' yield no diagnostics, despite the fact that no phase of the command was able to recognize and process '`-fno-asm`'—perhaps a warning about this would be helpful if it were possible.

Code that processes Fortran options is found in '`gcc/f/top.c`', function '`ffe_decode_option`'. This code needs to check positive and negative forms of each option.

The defaults for Fortran options are set in their global definitions, also found in '`gcc/f/top.c`'. Many of these defaults are actually macros defined in '`gcc/f/target.h`', since they might be machine-specific. However, since, in practice, GNU compilers should behave the same way on all configurations (especially when it comes to language constructs), the practice of setting defaults in '`target.h`' is likely to be deprecated and, ultimately, stopped in future versions of `g77`.

Accessor macros for Fortran options, used by code in the `g77` FFE, are defined in '`gcc/f/top.h`'.

Compiler options are listed in '`gcc/toplev.c`' in the array '`f_options`'. An option not listed in '`lang_options`' is looked up in '`f_options`' and handled from there.

The defaults for compiler options are set in the global definitions for the corresponding variables, some of which are in '`gcc/toplev.c`'.

You can set different defaults for *Fortran-oriented* or *Fortran-reticent* compiler options by changing the way `f771` handles the '`-fset-g77-defaults`' option, which is always provided as the first option when called by `g77` or `gcc`.

This code is in '`ffe_decode_options`' in '`gcc/f/top.c`'. Have it change just the variables that you want to default to a different setting for Fortran compiles compared to compiles of other languages.

The '`-fset-g77-defaults`' option is passed to `f771` automatically because of the specification information kept in '`gcc/f/lang-specs.h`'. This file tells the `gcc` command how to recognize, in this case, Fortran source files (those to be preprocessed, and those that are not), and further, how to invoke the appropriate programs (including `f771`) to process those source files.

It is in '`gcc/f/lang-specs.h`' that '`-fset-g77-defaults`', '`-fversion`', and other options are passed, as appropriate, even when the user has not explicitly specified them. Other "internal" options such as '`-quiet`' also are passed via this mechanism.

23 Projects

If you want to contribute to g77 by doing research, design, specification, documentation, coding, or testing, the following information should give you some ideas.

23.1 Improve Efficiency

Don't bother doing any performance analysis until most of the following items are taken care of, because there's no question they represent serious space/time problems, although some of them show up only given certain kinds of (popular) input.

- Improve 'malloc' package and its uses to specify more info about memory pools and, where feasible, use obstacks to implement them.
- Skip over uninitialized portions of aggregate areas (arrays, COMMON areas, EQUIVALENCE areas) so zeros need not be output. This would reduce memory usage for large initialized aggregate areas, even ones with only one initialized element.

  As of version 0.5.18, a portion of this item has already been accomplished.
- Prescan the statement (in 'sta.c') so that the nature of the statement is determined as much as possible by looking entirely at its form, and not looking at any context (previous statements, including types of symbols). This would allow ripping out of the statement-confirmation, symbol retraction/confirmation, and diagnostic inhibition mechanisms. Plus, it would result in much-improved diagnostics. For example, 'CALL some-intrinsic(...)', where the intrinsic is not a subroutine intrinsic, would result actual error instead of the unimplemented-statement catch-all.
- Throughout g77, don't pass line/column pairs where a simple 'ffewhere' type, which points to the error as much as is desired by the configuration, will do, and don't pass 'ffelexToken' types where a simple 'ffewhere' type will do. Then, allow new default configuration of 'ffewhere' such that the source line text is not preserved, and leave it to things like Emacs' next-error function to point to them (now that 'next-error' supports column, or, perhaps, character-offset, numbers). The change in calling sequences should improve performance somewhat, as should not having to save source lines. (Whether this whole item will improve performance is questionable, but it should improve maintainability.)
- Handle 'DATA (A(I),I=1,1000000)/1000000*2/' more efficiently, especially as regards the assembly output. Some of this might require improving the back end, but lots of improvement in space/time required in g77 itself can be fairly easily obtained without touching the back end. Maybe type-conversion, where necessary, can be speeded up as well in cases like the one shown (converting the '2' into '2.').
- If analysis shows it to be worthwhile, optimize 'lex.c'.
- Consider redesigning 'lex.c' to not need any feedback during tokenization, by keeping track of enough parse state on its own.

23.2 Better Optimization

Much of this work should be put off until after `g77` has all the features necessary for its widespread acceptance as a useful F77 compiler. However, perhaps this work can be done in parallel during the feature-adding work.

- Do the equivalent of the trick of putting '`extern inline`' in front of every function definition in `libf2c` and #include'ing the resulting file in `f2c+gcc`—that is, inline all run-time-library functions that are at all worth inlining. (Some of this has already been done, such as for integral exponentiation.)
- When doing '`CHAR_VAR = CHAR_FUNC(...)`', and it's clear that types line up and '`CHAR_VAR`' is addressable or not a '`VAR_DECL`', make '`CHAR_VAR`', not a temporary, be the receiver for '`CHAR_FUNC`'. (This is now done for `COMPLEX` variables.)
- Design and implement Fortran-specific optimizations that don't really belong in the back end, or where the front end needs to give the back end more info than it currently does.
- Design and implement a new run-time library interface, with the code going into `libgcc` so no special linking is required to link Fortran programs using standard language features. This library would speed up lots of things, from I/O (using precompiled formats, doing just one, or, at most, very few, calls for arrays or array sections, and so on) to general computing (array/section implementations of various intrinsics, implementation of commonly performed loops that aren't likely to be optimally compiled otherwise, etc.).

  Among the important things the library would do are:

  - Be a one-stop-shop-type library, hence shareable and usable by all, in that what are now library-build-time options in `libf2c` would be moved at least to the `g77` compile phase, if not to finer grains (such as choosing how list-directed I/O formatting is done by default at `OPEN` time, for preconnected units via options or even statements in the main program unit, maybe even on a per-I/O basis with appropriate pragma-like devices).
- Probably requiring the new library design, change interface to normally have `COMPLEX` functions return their values in the way `gcc` would if they were declared `__complex__ float`, rather than using the mechanism currently used by `CHARACTER` functions (whereby the functions are compiled as returning void and their first arg is a pointer to where to store the result). (Don't append underscores to external names for `COMPLEX` functions in some cases once `g77` uses `gcc` rather than `f2c` calling conventions.)
- Do something useful with '`doiter`' references where possible. For example, '`CALL FOO(I)`' cannot modify '`I`' if within a `DO` loop that uses '`I`' as the iteration variable, and the back end might find that info useful in determining whether it needs to read '`I`' back into a register after the call. (It normally has to do that, unless it knows '`FOO`' never modifies its passed-by-reference argument, which is rarely the case for Fortran-77 code.)

23.3 Simplify Porting

Making `g77` easier to configure, port, build, and install, either as a single-system compiler or as a cross-compiler, would be very useful.

- A new library (replacing `libf2c`) should improve portability as well as produce more optimal code. Further, `g77` and the new library should conspire to simplify naming of externals, such as by removing unnecessarily added underscores, and to reduce/eliminate the possibility of naming conflicts, while making debugger more straightforward.

  Also, it should make multi-language applications more feasible, such as by providing Fortran intrinsics that get Fortran unit numbers given C `FILE *` descriptors.
- Possibly related to a new library, `g77` should produce the equivalent of a `gcc` '`main(argc, argv)`' function when it compiles a main program unit, instead of compiling something that must be called by a library implementation of `main()`.

  This would do many useful things such as provide more flexibility in terms of setting up exception handling, not requiring programmers to start their debugging sessions with ***`breakpoint MAIN__`*** followed by ***`run`***, and so on.
- The GBE needs to understand the difference between alignment requirements and desires. For example, on Intel x86 machines, `g77` currently imposes overly strict alignment requirements, due to the back end, but it would be useful for Fortran and C programmers to be able to override these *recommendations* as long as they don't violate the actual processor *requirements*.

23.4 More Extensions

These extensions are not the sort of things users ask for "by name", but they might improve the usability of `g77`, and Fortran in general, in the long run. Some of these items really pertain to improving `g77` internals so that some popular extensions can be more easily supported.

- Look through all the documentation on the GNU Fortran language, dialects, compiler, missing features, bugs, and so on. Many mentions of incomplete or missing features are sprinkled throughout. It is not worth repeating them here.
- Support arbitrary operands for concatenation, even in contexts where run-time allocation is required.
- Consider adding a `NUMERIC` type to designate typeless numeric constants, named and unnamed. The idea is to provide a forward-looking, effective replacement for things like the old-style `PARAMETER` statement when people really need typelessness in a maintainable, portable, clearly documented way. Maybe `TYPELESS` would include `CHARACTER`, `POINTER`, and whatever else might come along. (This is not really a call for polymorphism per se, just an ability to express limited, syntactic polymorphism.)
- Support '`OPEN(...,KEY=(...),...)`'.
- Support arbitrary file unit numbers, instead of limiting them to 0 through '`MXUNIT-1`'. (This is a `libf2c` issue.)
- '`OPEN(NOSPANBLOCKS,...)`' is treated as '`OPEN(UNIT=NOSPANBLOCKS,...)`', so a later `UNIT=` in the first example is invalid. Make sure this is what users of this feature would expect.
- Currently `g77` disallows '`READ(1'10)`' since it is an obnoxious syntax, but supporting it might be pretty easy if needed. More details are needed, such as whether general

expressions separated by an apostrophe are supported, or maybe the record number can be a general expression, and so on.

- Support `STRUCTURE`, `UNION`, `MAP`, and `RECORD` fully. Currently there is no support at all for `%FILL` in `STRUCTURE` and related syntax, whereas the rest of the stuff has at least some parsing support. This requires either major changes to `libf2c` or its replacement.
- F90 and `g77` probably disagree about label scoping relative to `INTERFACE` and `END INTERFACE`, and their contained procedure interface bodies (blocks?).
- `ENTRY` doesn't support F90 `RESULT()` yet, since that was added after S8.112.
- Empty-statement handling (10 ;;CONTINUE;;) probably isn't consistent with the final form of the standard (it was vague at S8.112).
- It seems to be an "open" question whether a file, immediately after being `OPEN`ed,is positioned at the beginning, the end, or wherever—it might be nice to offer an option of opening to "undefined" status, requiring an explicit absolute-positioning operation to be performed before any other (besides `CLOSE`) to assist in making applications port to systems (some IBM?) that `OPEN` to the end of a file or some such thing.

23.5 Machine Model

This items pertain to generalizing `g77`'s view of the machine model to more fully accept whatever the GBE provides it via its configuration.

- Switch to using '`REAL_VALUE_TYPE`' to represent floating-point constants exclusively so the target float format need not be required. This means changing the way `g77` handles initialization of aggregate areas having more than one type, such as `REAL` and `INTEGER`, because currently it initializes them as if they were arrays of `char` and uses the bit patterns of the constants of the various types in them to determine what to stuff in elements of the arrays.
- Rely more and more on back-end info and capabilities, especially in the area of constants (where having the `g77` front-end's IL just store the appropriate tree nodes containing constants might be best).
- Suite of C and Fortran programs that a user/administrator can run on a machine to help determine the configuration for `g77` before building and help determine if the compiler works (especially with whatever libraries are installed) after building.

23.6 Internals Documentation

Better info on how `g77` works and how to port it is needed. Much of this should be done only after the redesign planned for 0.6 is complete.

23.7 Internals Improvements

Some more items that would make `g77` more reliable and easier to maintain:

- Generally make expression handling focus more on critical syntax stuff, leaving semantics to callers. For example, anything a caller can check, semantically, let it do so, rather than having '`expr.c`' do it. (Exceptions might include things like diagnosing

'FOO(I--K:)=BAR' where 'FOO' is a PARAMETER—if it seems important to preserve the left-to-right-in-source order of production of diagnostics.)

- Come up with better naming conventions for '-D' to establish requirements to achieve desired implementation dialect via 'proj.h'.
- Clean up used tokens and 'ffewhere's in 'ffeglobal_terminate_1'.
- Replace 'sta.c' 'outpooldisp' mechanism with 'malloc_pool_use'.
- Check for 'opANY' in more places in 'com.c', 'std.c', and 'ste.c', and get rid of the 'opCONVERT(opANY)' kludge (after determining if there is indeed no real need for it).
- Utility to read and check 'bad.def' messages and their references in the code, to make sure calls are consistent with message templates.
- Search and fix '&ffe...' and similar so that 'ffe...ptr...' macros are available instead (a good argument for wishing this could have written all this stuff in C++, perhaps). On the other hand, it's questionable whether this sort of improvement is really necessary, given the availability of tools such as Emacs and Perl, which make finding any address-taking of structure members easy enough?
- Some modules truly export the member names of their structures (and the structures themselves), maybe fix this, and fix other modules that just appear to as well (by appending '_', though it'd be ugly and probably not worth the time).
- Implement C macros 'RETURNS(value)' and 'SETS(something,value)' in 'proj.h' and use them throughout g77 source code (especially in the definitions of access macros in '.h' files) so they can be tailored to catch code writing into a 'RETURNS()' or reading from a 'SETS()'.
- Decorate throughout with const and other such stuff.
- All F90 notational derivations in the source code are still based on the S8.112 version of the draft standard. Probably should update to the official standard, or put documentation of the rules as used in the code...uh...in the code.
- Some 'ffebld_new' calls (those outside of 'ffeexpr.c' or inside but invoked via paths not involving 'ffeexpr_lhs' or 'ffeexpr_rhs') might be creating things in improper pools, leading to such things staying around too long or (doubtful, but possible and dangerous) not long enough.
- Some 'ffebld_list_new' (or whatever) calls might not be matched by 'ffebld_list_bottom' (or whatever) calls, which might someday matter. (It definitely is not a problem just yet.)
- Probably not doing clean things when we fail to EQUIVALENCE something due to alignment/mismatch or other problems—they end up without 'ffestorag' objects, so maybe the backend (and other parts of the front end) can notice that and handle like an 'opANY' (do what it wants, just don't complain or crash). Most of this seems to have been addressed by now, but a code review wouldn't hurt.

23.8 Better Diagnostics

These are things users might not ask about, or that need to be looked into, before worrying about. Also here are items that involve reducing unnecessary diagnostic clutter.

- When `FUNCTION` and `ENTRY` point types disagree (`CHARACTER` lengths, type classes, and so on), '`ANY`'-ize the offending `ENTRY` point and any *new* dummies it specifies.
- Speed up and improve error handling for data when repeat-count is specified. For example, don't output 20 unnecessary messages after the first necessary one for:

```
INTEGER X(20)
CONTINUE
DATA (X(I), J= 1, 20) /20*5/
END
```

  (The `CONTINUE` statement ensures the `DATA` statement is processed in the context of executable, not specification, statements.)

24 Diagnostics

Some diagnostics produced by `g77` require sufficient explanation that the explanations are given below, and the diagnostics themselves identify the appropriate explanation.

Identification uses the GNU Info format—specifically, the `info` command that displays the explanation is given within square brackets in the diagnostic. For example:

```
foo.f:5: Invalid statement [info -f g77 M FOOEY]
```

More details about the above diagnostic is found in the `g77` Info documentation, menu item '`M`', submenu item '`FOOEY`', which is displayed by typing the UNIX command '`info -f g77 M FOOEY`'.

Other Info readers, such as EMACS, may be just as easily used to display the pertinent node. In the above example, '`g77`' is the Info document name, '`M`' is the top-level menu item to select, and, in that node (named '`Diagnostics`', the name of this chapter, which is the very text you're reading now), '`FOOEY`' is the menu item to select.

In this printed version of the `g77` manual, the above example points to a section, below, entitled '`FOOEY`'—though, of course, as the above is just a sample, no such section exists.

24.1 CMPAMBIG

`Ambiguous use of intrinsic` *intrinsic* ...

The type of the argument to the invocation of the *intrinsic* intrinsic is a `COMPLEX` type other than `COMPLEX(KIND=1)`. Typically, it is `COMPLEX(KIND=2)`, also known as `DOUBLE COMPLEX`.

The interpretation of this invocation depends on the particular dialect of Fortran for which the code was written. Some dialects convert the real part of the argument to `REAL(KIND=1)`, thus losing precision; other dialects, and Fortran 90, do no such conversion.

So, GNU Fortran rejects such invocations except under certain circumstances, to avoid making an incorrect assumption that results in generating the wrong code.

To determine the dialect of the program unit, perhaps even whether that particular invocation is properly coded, determine how the result of the intrinsic is used.

The result of *intrinsic* is expected (by the original programmer) to be `REAL(KIND=1)` (the non-Fortran-90 interpretation) if:

- It is passed as an argument to a procedure that explicitly or implicitly declares that argument `REAL(KIND=1)`.

  For example, a procedure with no `DOUBLE PRECISION` or `IMPLICIT DOUBLE PRECISION` statement specifying the dummy argument corresponding to an actual argument of '`REAL(Z)`', where '`Z`' is declared `DOUBLE COMPLEX`, strongly suggests that the programmer expected '`REAL(Z)`' to return `REAL(KIND=1)` instead of `REAL(KIND=2)`.

- It is used in a context that would otherwise not include any `REAL(KIND=2)` but where treating the *intrinsic* invocation as `REAL(KIND=2)` would result in unnecessary promotions and (typically) more expensive operations on the wider type.

  For example:

```
DOUBLE COMPLEX Z
...
R(1) = T * REAL(Z)
```

The above example suggests the programmer expected the real part of 'Z' to be converted to REAL(KIND=1) before being multiplied by 'T' (presumed, along with 'R' above, to be type REAL(KIND=1)).

Otherwise, the conversion would have to be delayed until after the multiplication, requiring not only an extra conversion (of 'T' to REAL(KIND=2)), but a (typically) more expensive multiplication (a double-precision multiplication instead of a single-precision one).

The result of *intrinsic* is expected (by the original programmer) to be REAL(KIND=2) (the Fortran 90 interpretation) if:

- It is passed as an argument to a procedure that explicitly or implicitly declares that argument REAL(KIND=2).

  For example, a procedure specifying a DOUBLE PRECISION dummy argument corresponding to an actual argument of 'REAL(Z)', where 'Z' is declared DOUBLE COMPLEX, strongly suggests that the programmer expected 'REAL(Z)' to return REAL(KIND=2) instead of REAL(KIND=1).

- It is used in an expression context that includes other REAL(KIND=2) operands, or is assigned to a REAL(KIND=2) variable or array element.

  For example:

  ```
  DOUBLE COMPLEX Z
  DOUBLE PRECISION R, T
  ...
  R(1) = T * REAL(Z)
  ```

  The above example suggests the programmer expected the real part of 'Z' to *not* be converted to REAL(KIND=1) by the REAL() intrinsic.

  Otherwise, the conversion would have to be immediately followed by a conversion back to REAL(KIND=2), losing the original, full precision of the real part of Z, before being multiplied by 'T'.

Once you have determined whether a particular invocation of *intrinsic* expects the Fortran 90 interpretation, you can:

- Change it to 'DBLE(*expr*)' (if *intrinsic* is 'REAL') or 'DIMAG(*expr*)' (if *intrinsic* is 'AIMAG') if it expected the Fortran 90 interpretation.

  This assumes *expr* is COMPLEX(KIND=2)—if it is some other type, such as COMPLEX*32, you should use the appropriate intrinsic, such as the one to convert to REAL*16 (perhaps DBLEQ() in place of DBLE(), and QIMAG() in place of DIMAG()).

- Change it to 'REAL(*intrinsic*(*expr*))', otherwise. This converts to REAL(KIND=1) in all working Fortran compilers.

If you don't want to change the code, and you are certain that all ambiguous invocations of *intrinsic* in the source file have the same expectation regarding interpretation, you can:

- Compile with the g77 option '-ff90', to enable the Fortran 90 interpretation.

- Compile with the `g77` options '`-fno-f90 -fugly-complex`', to enable the non-Fortran-90 interpretations.

See Section 10.11.5 [REAL() and AIMAG() of Complex], page 96, for more information on this issue.

Note: If the above suggestions don't produce enough evidence as to whether a particular program expects the Fortran 90 interpretation of this ambiguous invocation of *intrinsic*, there is one more thing you can try.

If you have access to most or all the compilers used on the program to create successfully tested and deployed executables, read the documentation for, and *also* test out, each compiler to determine how it treats the *intrinsic* intrinsic in this case. (If all the compilers don't agree on an interpretation, there might be lurking bugs in the deployed versions of the program.)

The following sample program might help:

```
      PROGRAM JCB003
C
C Written by James Craig Burley 1997-02-23.
C Contact via Internet email: burley@gnu.org
C
C Determine how compilers handle non-standard REAL
C and AIMAG on DOUBLE COMPLEX operands.
C
      DOUBLE COMPLEX Z
      REAL R
      Z = (3.3D0, 4.4D0)
      R = Z
      CALL DUMDUM(Z, R)
      R = REAL(Z) - R
      IF (R .NE. 0.) PRINT *, 'REAL() is Fortran 90'
      IF (R .EQ. 0.) PRINT *, 'REAL() is not Fortran 90'
      R = 4.4D0
      CALL DUMDUM(Z, R)
      R = AIMAG(Z) - R
      IF (R .NE. 0.) PRINT *, 'AIMAG() is Fortran 90'
      IF (R .EQ. 0.) PRINT *, 'AIMAG() is not Fortran 90'
      END
C
C Just to make sure compiler doesn't use naive flow
C analysis to optimize away careful work above,
C which might invalidate results....
C
      SUBROUTINE DUMDUM(Z, R)
      DOUBLE COMPLEX Z
      REAL R
      END
```

If the above program prints contradictory results on a particular compiler, run away!

24.2 EXPIMP

`Intrinsic` *intrinsic* `referenced ...`

The *intrinsic* is explicitly declared in one program unit in the source file and implicitly used as an intrinsic in another program unit in the same source file.

This diagnostic is designed to catch cases where a program might depend on using the name *intrinsic* as an intrinsic in one program unit and as a global name (such as the name of a subroutine or function) in another, but `g77` recognizes the name as an intrinsic in both cases.

After verifying that the program unit making implicit use of the intrinsic is indeed written expecting the intrinsic, add an '`INTRINSIC` *intrinsic*' statement to that program unit to prevent this warning.

This and related warnings are disabled by using the '`-Wno-globals`' option when compiling.

Note that this warning is not issued for standard intrinsics. Standard intrinsics include those described in the FORTRAN 77 standard and, if '`-ff90`' is specified, those described in the Fortran 90 standard. Such intrinsics are not as likely to be confused with user procedures as intrinsics provided as extensions to the standard by `g77`.

24.3 INTGLOB

`Same name '`*intrinsic*`' given ...`

The name *intrinsic* is used for a global entity (a common block or a program unit) in one program unit and implicitly used as an intrinsic in another program unit.

This diagnostic is designed to catch cases where a program intends to use a name entirely as a global name, but `g77` recognizes the name as an intrinsic in the program unit that references the name, a situation that would likely produce incorrect code.

For example:

```
INTEGER FUNCTION TIME()
...
END
...
PROGRAM SAMP
INTEGER TIME
PRINT *, 'Time is ', TIME()
END
```

The above example defines a program unit named '`TIME`', but the reference to '`TIME`' in the main program unit '`SAMP`' is normally treated by `g77` as a reference to the intrinsic `TIME()` (unless a command-line option that prevents such treatment has been specified).

As a result, the program '`SAMP`' will *not* invoke the '`TIME`' function in the same source file.

Since `g77` recognizes `libU77` procedures as intrinsics, and since some existing code uses the same names for its own procedures as used by some `libU77` procedures, this situation is expected to arise often enough to make this sort of warning worth issuing.

After verifying that the program unit making implicit use of the intrinsic is indeed written expecting the intrinsic, add an '`INTRINSIC` *intrinsic*' statement to that program unit to prevent this warning.

Or, if you believe the program unit is designed to invoke the program-defined procedure instead of the intrinsic (as recognized by `g77`), add an '`EXTERNAL` *intrinsic*' statement to the program unit that references the name to prevent this warning.

This and related warnings are disabled by using the '`-Wno-globals`' option when compiling.

Note that this warning is not issued for standard intrinsics. Standard intrinsics include those described in the FORTRAN 77 standard and, if '`-ff90`' is specified, those described in the Fortran 90 standard. Such intrinsics are not as likely to be confused with user procedures as intrinsics provided as extensions to the standard by `g77`.

24.4 LEX

```
Unrecognized character ...
Invalid first character ...
Line too long ...
Non-numeric character ...
Continuation indicator ...
Label at ... invalid with continuation line indicator ...
Character constant ...
Continuation line ...
Statement at ... begins with invalid token
```

Although the diagnostics identify specific problems, they can be produced when general problems such as the following occur:

- The source file contains something other than Fortran code.

  If the code in the file does not look like many of the examples elsewhere in this document, it might not be Fortran code. (Note that Fortran code often is written in lower case letters, while the examples in this document use upper case letters, for stylistic reasons.)

  For example, if the file contains lots of strange-looking characters, it might be APL source code; if it contains lots of parentheses, it might be Lisp source code; if it contains lots of bugs, it might be C++ source code.

- The source file contains free-form Fortran code, but '`-ffree-form`' was not specified on the command line to compile it.

  Free form is a newer form for Fortran code. The older, classic form is called fixed form.

  Fixed-form code is visually fairly distinctive, because numerical labels and comments are all that appear in the first five columns of a line, the sixth column is reserved to denote continuation lines, and actual statements start at or beyond column 7. Spaces generally are not significant, so if you see statements such as '`REALX,Y`' and '`DO10I=1,100`', you are looking at fixed-form code. Comment lines are indicated by the letter '`C`' or the symbol '`*`' in column 1. (Some code uses '`!`' or '`/*`' to begin in-line comments, which many compilers support.)

Free-form code is distinguished from fixed-form source primarily by the fact that statements may start anywhere. (If lots of statements start in columns 1 through 6, that's a strong indicator of free-form source.) Consecutive keywords must be separated by spaces, so '`REALX,Y`' is not valid, while '`REAL X,Y`' is. There are no comment lines per se, but '`!`' starts a comment anywhere in a line (other than within a character or hollerith constant).

See Section 11.1 [Source Form], page 169, for more information.

- The source file is in fixed form and has been edited without sensitivity to the column requirements.

  Statements in fixed-form code must be entirely contained within columns 7 through 72 on a given line. Starting them "early" is more likely to result in diagnostics than finishing them "late", though both kinds of errors are often caught at compile time.

  For example, if the following code fragment is edited by following the commented instructions literally, the result, shown afterward, would produce a diagnostic when compiled:

  ```
  C On XYZZY systems, remove "C" on next line:
  C     CALL XYZZY_RESET
  ```

  The result of editing the above line might be:

  ```
  C On XYZZY systems, remove "C" on next line:
       CALL XYZZY_RESET
  ```

  However, that leaves the first '`C`' in the '`CALL`' statement in column 6, making it a comment line, which is not really what the author intended, and which is likely to result in one of the above-listed diagnostics.

  Replacing the '`C`' in column 1 with a space is the proper change to make, to ensure the '`CALL`' keyword starts in or after column 7.

  Another common mistake like this is to forget that fixed-form source lines are significant through only column 72, and that, normally, any text beyond column 72 is ignored or is diagnosed at compile time.

  See Section 11.1 [Source Form], page 169, for more information.

- The source file requires preprocessing, and the preprocessing is not being specified at compile time.

  A source file containing lines beginning with `#define`, `#include`, `#if`, and so on is likely one that requires preprocessing.

  If the file's suffix is '`.f`' or '`.for`', the file will normally be compiled *without* preprocessing by `g77`.

  Change the file's suffix from '`.f`' to '`.F`' (or, on systems with case-insensitive file names, to '`.fpp`') or from '`.for`' to '`.fpp`'. `g77` compiles files with such names *with* preprocessing.

  Or, learn how to use `gcc`'s '`-x`' option to specify the language '`f77-cpp-input`' for Fortran files that require preprocessing. See Section 7.2 [gcc], page 27.

- The source file is preprocessed, and the results of preprocessing result in syntactic errors that are not necessarily obvious to someone examining the source file itself.

Examples of errors resulting from preprocessor macro expansion include exceeding the line-length limit, improperly starting, terminating, or incorporating the apostrophe or double-quote in a character constant, improperly forming a hollerith constant, and so on.

See Section 7.2 [Options Controlling the Kind of Output], page 27, for suggestions about how to use, and not use, preprocessing for Fortran code.

24.5 GLOBALS

```
Global name name defined at ... already defined...
Global name name at ... has different type...
Too many arguments passed to name at ...
Too few arguments passed to name at ...
Argument #n of name is ...
```

These messages all identify disagreements about the global procedure named *name* among different program units (usually including *name* itself).

These disagreements, if not diagnosed, could result in a compiler crash if the compiler attempted to inline a reference to *name* within a calling program unit that disagreed with the *name* program unit regarding whether the procedure is a subroutine or function, the type of the return value of the procedure (if it is a function), the number of arguments the procedure accepts, or the type of each argument.

Such disagreements *should* be fixed in the Fortran code itself. However, if that is not immediately practical, and the code has been working for some time, it is possible it will work when compiled by `g77` with the '`-fno-globals`' option.

The '`-fno-globals`' option disables these diagnostics, and also disables all inlining of references to global procedures to avoid compiler crashes. The diagnostics are actually produced, but as warnings, unless the '`-Wno-globals`' option also is specified.

After using '`-fno-globals`' to work around these problems, it is wise to stop using that option and address them by fixing the Fortran code, because such problems, while they might not actually result in bugs on some systems, indicate that the code is not as portable as it could be. In particular, the code might appear to work on a particular system, but have bugs that affect the reliability of the data without exhibiting any other outward manifestations of the bugs.

Index

.

/

‘

_

5

6

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Short Contents

Table of Contents