Fortran Tools

THE FORTRAN COMPANY

www.fortran.com

Version 7.1

Library of Congress Catalog Card Number

19 18 17 16 15 14 13 12

ISBN

The Fortran Company
3014 South Cook Street
Denver, Colorado 80210 USA
www.fortran.com

Composition by The Fortran Company

Open Office was used to produce this manual.

Table of Contents

1 Installation and Startup

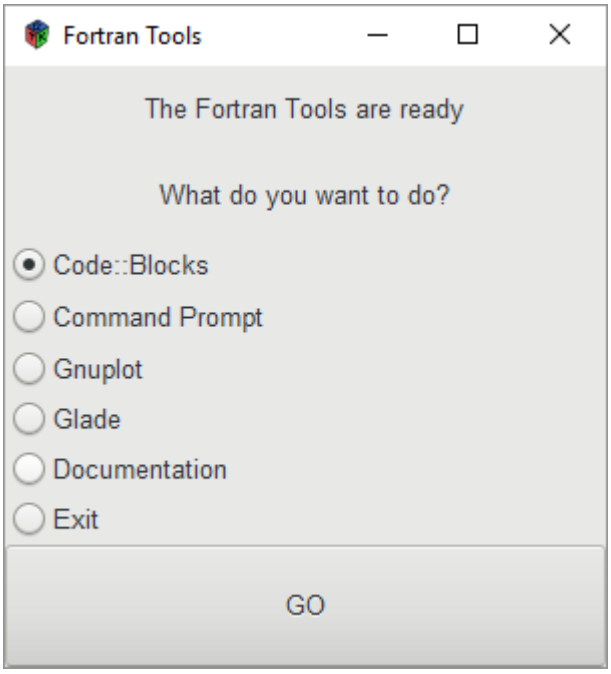
This section describes the installation and how to start execution of the Fortran Tools.

The 32-bit version of the Fortran Tools does not include the GTK GUI builder or the dialog box described below. Please ignore any reference to these features in this manual.

Unfortunately, the coarray Fortran features are not integrated with gfortran, hence also not with the Fortran Tools. Coarray Fortran programs can be run on Windows 10, but just from the command line and only after installing additional software. Procedures for doing this are described in Section 14: Coarrays.

1.1 Starting the Fortran Tools

With the 64-bit version, after the Fortran Tools are installed, simply select the Fortran Tools icon on the Desktop, the icon in the "Apps" (Windows 8 and 10), or execute the file `bin\fortran_tools.exe` in the installation folder and the following dialog will appear. To start Code::Blocks directly, execute the file `codeblocks\codeblocks.exe`. Or start a command prompt window and execute `codeblocks`.



To start the 32-bit version, use a command prompt to edit, compile, and run a program, or execute `codeblocks.exe`.

1.2 Software Agreement

Before installing the software, please read and abide by the Software Agreement in Appendix A. Please note that most of this software is developed by third parties as open source software and is subject to the licensing agreements for the software.

1.3 Implementations Available

The 64-bit implementation is for Windows 7, Windows 8, Windows 10, and Windows Server 64-bit systems. The 32-bit implementation is for Windows XP, Windows 7, and any later 32-bit systems.

1.4 Contents

The software includes a Fortran 03 compiler with many Fortran 08 features, the Code::Blocks development environment, other software (including interval arithmetic, linear algebra, and plotting software), this *Fortran Tools* manual, several books in PDF format, example programs, appropriate licenses, some source code, and documentation. It also includes a C compiler (`gcc`) and other language compilers found in the MinGW collection.

1.5 Installation

To install the Fortran Tools, download and unzip the zip file, execute (click) the resulting setup file, and follow the directions in the installation dialog.

It is a good idea to save the setup file somewhere safe before installing the Fortran Tools.

The steps in Sections 1.6-1.8 are performed during setup. They already should be set as described. The information is included in case you want to change or reset some of these items.

Notes:

1. If you have administrative privileges, the software will be installed for all users; if not, it will be installed just for the user doing the installation.
2. ***Do not use or create a folder (directory) with a space in its name for any purpose related to the Fortran Tools.***
3. The default installation file for the Fortran Tools is `C:\Fortran_Tools`. It may be installed in a different directory, but please note #2 above.
4. You will be asked to allow a *Custom Action* named *Set up Fortran Tools* to be performed. Do it.

5. Acrobat Reader should be installed to read the documentation that is in PDF format. It is available free from adobe.com if it is not already installed on your system. The Microsoft Reader also should work.

To uninstall the Fortran Tools, select *Programs* in the Control Panel, select the Fortran Tools and *Uninstall*. Check the installation folder as the uninstall program does not always remove everything.

1.6 Setting Environment Variables

The installation of the Fortran Tools should change or add the following environment variables. <InstallDir> is the installation directory, which is C:\Fortran_Tools by default.

```
FORTRAN_TOOLS=<InstallDir>
PATH=<InstallDir>\codeblocks
PATH=<InstallDir>\bin
PATH=<InstallDir>\gnuplot\bin
FT_INC=-I<InstallDir>\include
FT_LIB=<InstallDir>\lib\libfortrantools.a
FT=-I<InstallDir>\include <InstallDir>\lib\libfortrantools.a
GTK_LIB=-LC:\<InstallDir>\lib -lgtkfortran -lgtk-3 -lgdk-3 -lgdi32 -limm32
-lshell32 -lole32 -Wl,-luuid -lwinmm -ldwmapi -lsetupapi -lcfgmgr32 -lz
-lpangowin32-1.0 -lpangocairo-1.0 -lpango-1.0 -latk-1.0 -lcairo-gobject -lcairo
-lgdk_pixbuf-2.0 -lgio-2.0 -lgobject-2.0 -lintl -lglib-2.0
GTK=-IC:\<InstallDir>\include -LC:\Fortran_Tools\lib -lfortrantools
-lgtkfortran -lgtk-3 -lgdk-3 -lgdi32 -limm32 -lshell32 -lole32 -Wl,-luuid
-lwinmm -ldwmapi -lsetupapi -lcfgmgr32 -lz -lpangowin32-1.0 -lpangocairo-1.0
-lpango-1.0 -latk-1.0 -lcairo-gobject -lcairo -lgdk_pixbuf-2.0 -lgio-2.0
-lgobject-2.0 -lintl -lglib-2.0
PLOT_LIB=-LC:\<InstallDir>\lib  -lplplotfortran -lplplot -lplfortrandemolib
-ltcl.dll -ltk.dll C:/Windows/System32/gdi32.dll
C:/Windows/System32/comdlg32.dll -lfreetype.dll -lfreetype.dll -lcsirocsa
-lqsastime -LC:/<InstallDir>/lib/gcc/x86_64-w64-mingw32/8.3.0 -lstdc++.dll
-ladvapi32 -lshell32 -luser32 -lkernel32 -lmingw32 -lgcc_s -lgcc -lmoldname
-lmingwex
PLOT=-IC:\<InstallDir>\include C:\Fortran_Tools\lib\libfortrantools.a
-LC:\<InstallDir>\lib  -lplplotfortran -lplplot -lplfortrandemolib -ltcl.dll
-ltk.dll C:/Windows/System32/gdi32.dll C:/Windows/System32/comdlg32.dll
-lfreetype.dll -lfreetype.dll -lcsirocsa -lqsastime
-LC:/Fortran_Tools/lib/gcc/x86_64-w64-mingw32/8.3.0 -lstdc++.dll -ladvapi32
-lshell32 -luser32 -lkernel32 -lmingw32 -lgcc_s -lgcc -lmoldname -lmingwex
PLPLOT_LIB=<InstallDir>\lib
CAF_NUM_IMAGES=4
LAPACK=-LC:<InstallDir>\lib -llapack -lblas
GNUTERM=windows
```

Your path variable can be set as follows. Click *Start* → *Control Panel* → *System and Security* → *System* → *Advanced system settings*→ *Environment Variables* → *System variables*. Scroll down in the *System variables* in the bottom window (the top window is per-user variables if you do not have administrative privileges). Select and edit the environment variable.

1.7 Creating a Shortcut

Installation creates a shortcut labeled "Fortran_Tools", which is just a shortcut to the Fortran Tools startup dialog. In order to create a shortcut to `codeblocks`, `wgnuplot`, or `glade:`

1. Use `Explorer` (the file manager) to locate the Fortran Tools installation directory. Go into the appropriate subfolder.
2. Right click on the appropriate *.exe* file and select *Create Shortcut*.
3. Right click on the shortcut and change the name to whatever you like.
4. Move (drag) the icon to the desktop.

1.8 Setting up Code::Blocks

You should be able to create, compile, and run a Fortran program with Code::Blocks (Section 2) without modifying any settings described in this section. They are here in case you want to modify some settings.

1.8.1 Specifying the Location of Gfortran

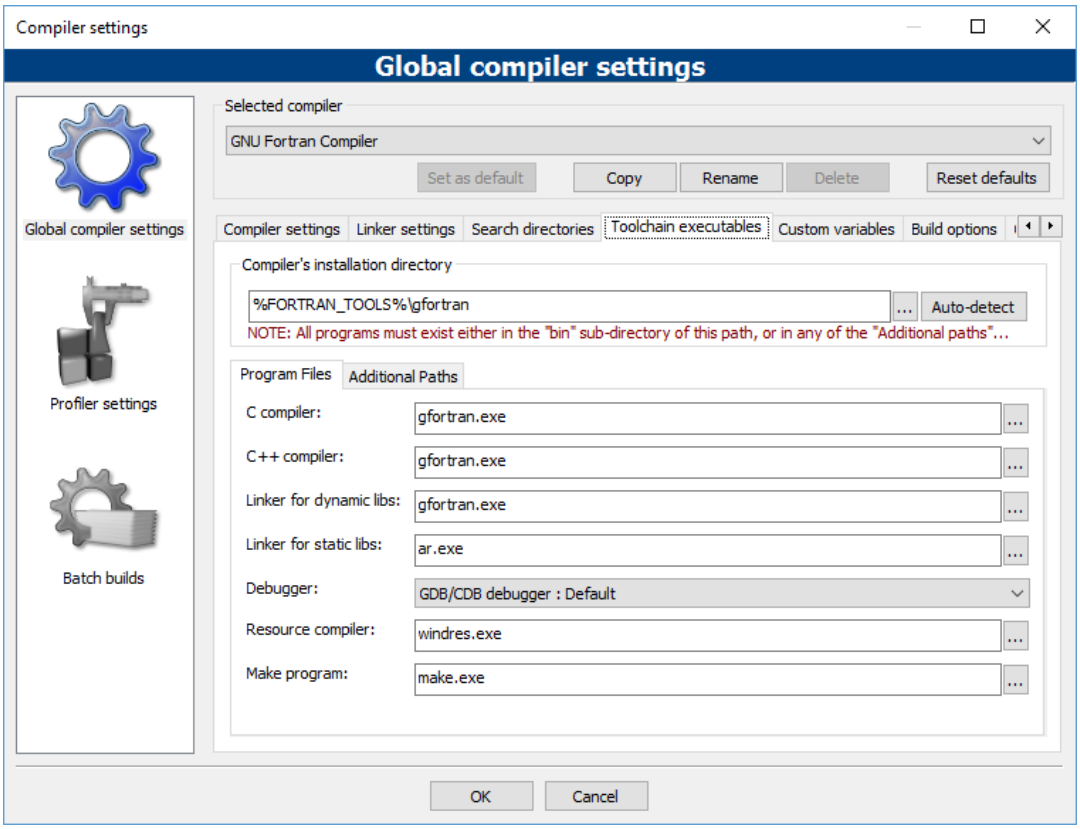
When a build takes place, the location of the compiler `gfortran` and other tools need to be known by Code::Blocks. These are set by default and they should not need to be changed.

1. Select the *Settings* tab, then *Compiler*, then *Global compiler settings* (if not already selected).
2. Make sure that the *Selected compiler* is *GNU Fortran Compiler*.
3. Select the *Toolchain executables* tab.
4. Set the *Compiler's installation directory* to the Fortran installation folder; at installation, it is `%FORTRAN_TOOLS%` which is the value of the environment variable `FORTRAN_TOOLS`.
5. Select the *Program Files* tab (if not selected).
6. Fill in the table entries as follows:

C compiler:	`gfortran.exe`
C++ compiler:	`gfortran.exe`
Linker for dynamic libs:	`gfortran.exe`
Linker for static libs:	`ar.exe`
Debugger:	`GDB/CDB debugger : Default`
Resource compiler:	`windres.exe`
Make program:	`make.exe`

Note that there is no entry for a Fortran compiler. This is just something left over from the fact that Code::Blocks originally was written for C and C++.

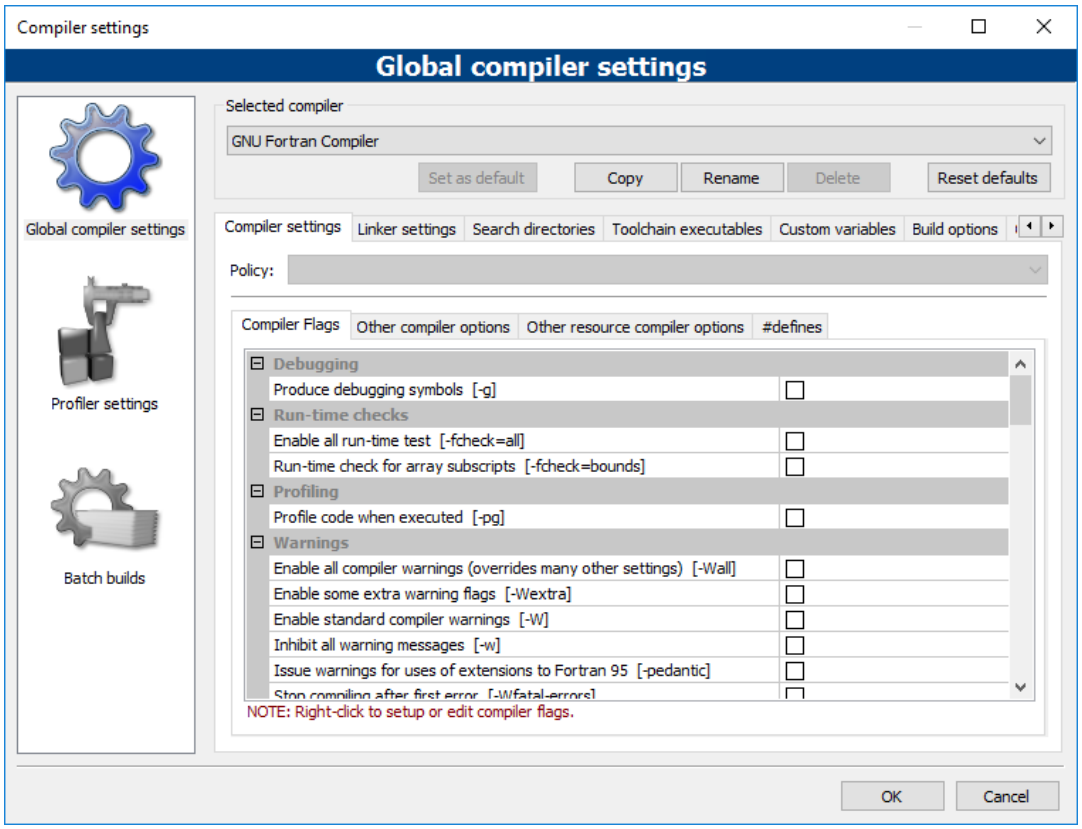
7. Then select OK.



1.8.2 Compiler Options

Some Gfortran compiler options may be set as follows. These options will affect all compilations done with Code::Blocks.

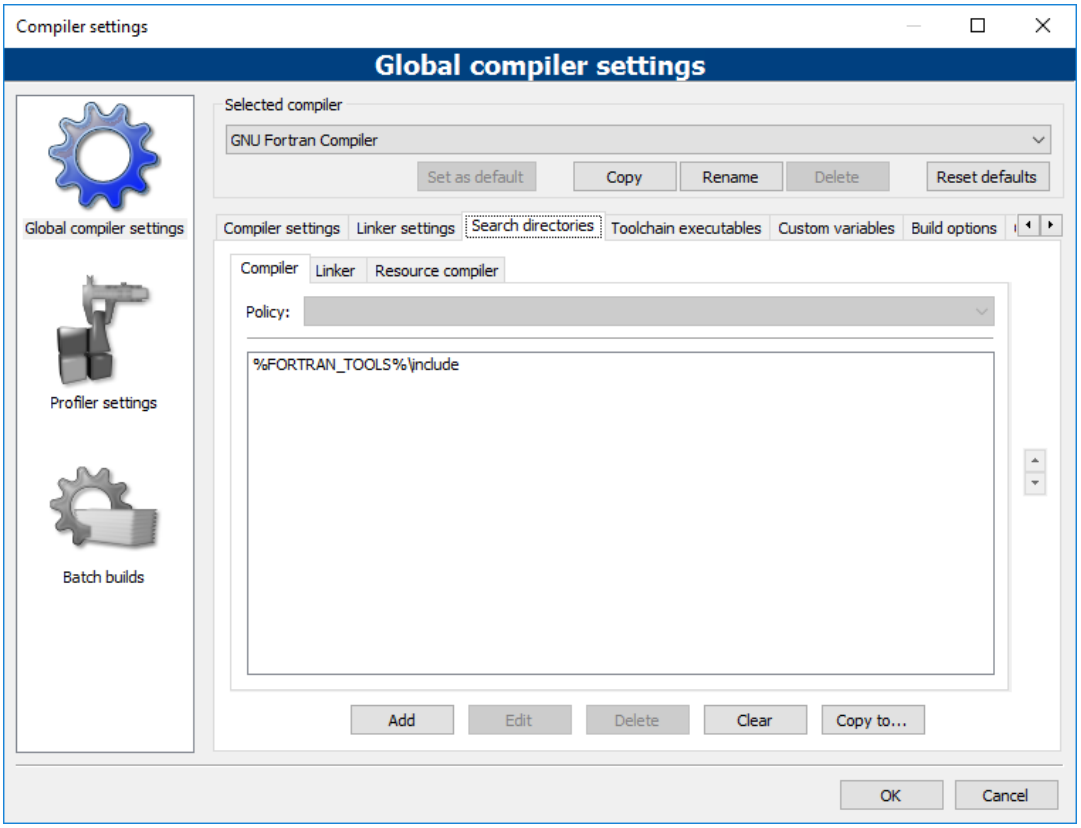
1. Select the *Settings* tab. Then select *Compiler* from the pulldown menu.
2. Select *Global compiler settings* from the window on the left if that window is not already showing.
3. Make sure that the *Selected compiler* is *GNU Fortran Compiler*.
4. No global options are set by default.
5. If you can find an option that will generate code for the architecture of your machine (lower part of the options list), you might set that one.
6. After setting the compiler options, select *OK* (unless you want to change some other settings).



1.8.3 Specifying Search Directories

When using a module provided by the Fortran Tools or when using a module or include file not in the project folder, the compiler must be told where to find the `.mod` or `.inc` file. This is set by default for the "built-in" Fortran Tools modules and it should not need to be changed you want to include other module or library files.

1. Select the *Settings* tab, then *Compiler settings*, then *Global compiler settings*, as above for compiler options.
2. Make sure that the Selected compiler is *GNU Fortran Compiler*.
3. Select the *Search directories* tab.
4. Select the *Compiler* tab.
5. Select *Add* at the bottom of the *Search directories* window.
6. In the *Add library* window, use the browser (box with ... to the right) to find the file and then select *Open* or just type in the name of the folder.



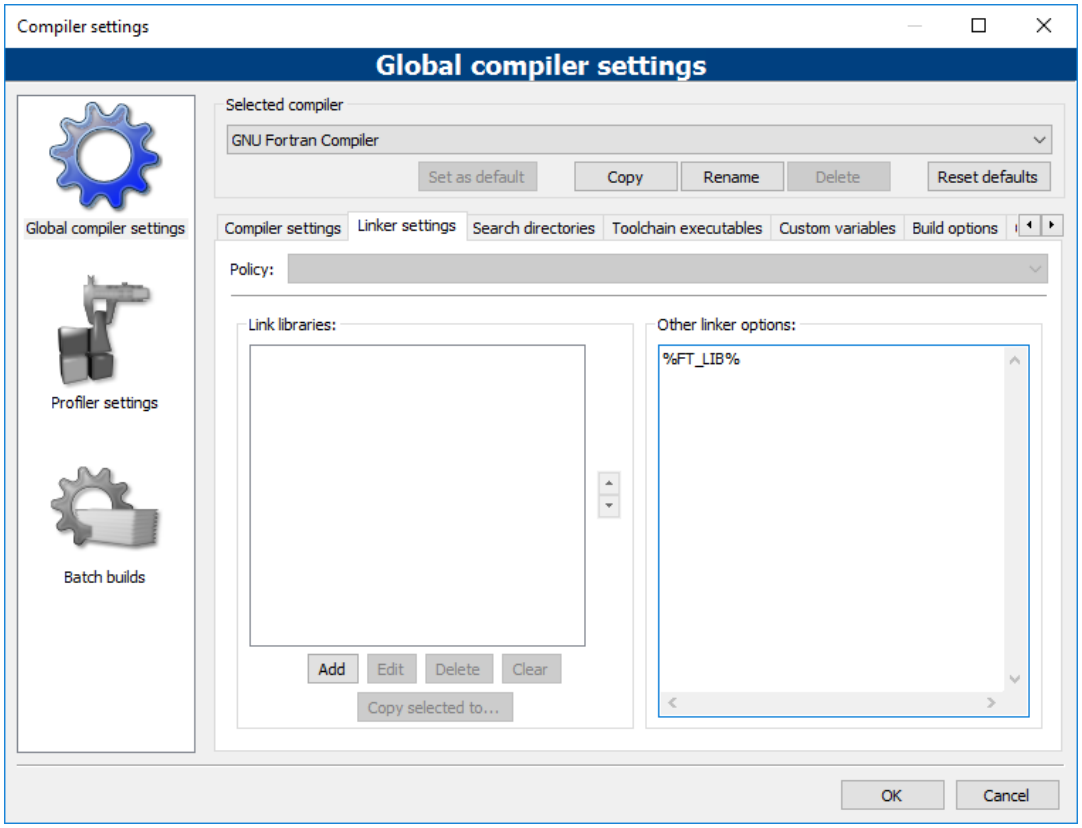
1.8.4 Including Libraries

To run some of the Fortran Tools software, such as the Slatec numerical programs or GTK (Section 16), a library needs to be linked to your program; otherwise, this is not necessary. Or there may be other libraries you may wish to include with your program. Here is how to add the Fortran Tools library, the GTK library, and the Plplot library. The Fortran Tools library is set by default for the Fortran Tools and it should not need to be changed unless you want to include other libraries. The GTK and Plplot libraries are set up not as global settings, but as project settings.

1. Select the *Settings* tab, then *Compiler settings*, then *Global compiler settings*, as above for compiler options.
2. Make sure that the *Selected compiler* is *GNU Fortran Compiler*.
3. Select the *Linker settings* tab.
4. Select *Add* at the bottom of the *Link libraries* window.

5. In the *Add library* window, use the browser (box with ... to the right) to find the file and then select *Open* or just type in the name of the library. Or specify inclusion of libraries in the *Other linker options* panel.

6. Then select *OK*.



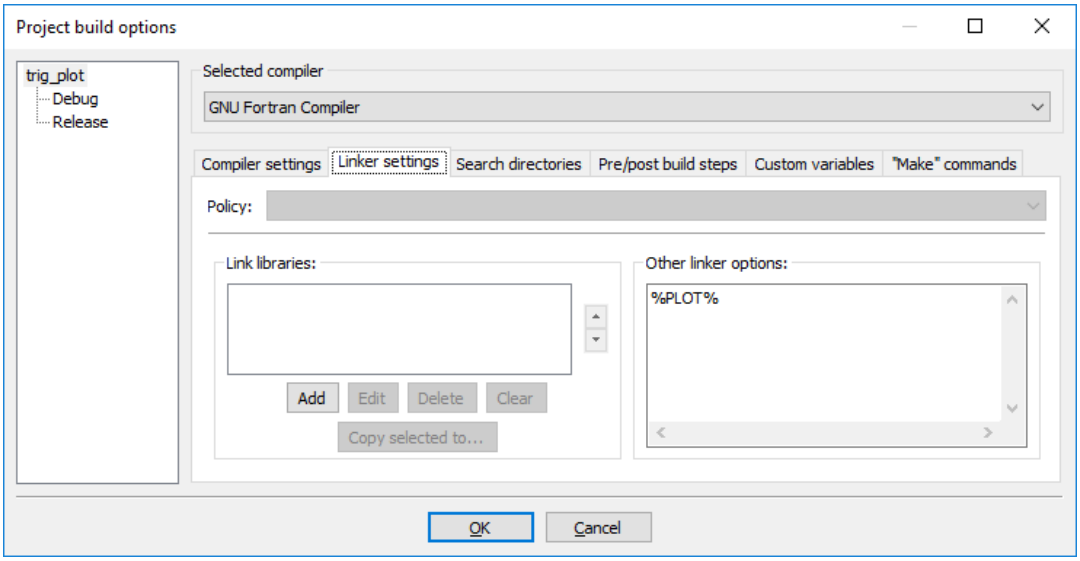
1.8.5 Removing Libraries

If none of the Fortran Tools special software, such as interval arithmetic, is to be used, the entry in *Linker settings* may be removed; this will make the executable program smaller.

1.8.6 Project Settings

The global settings described so far affect all projects started with default “personality”. It is possible to establish some settings so that they affect only one project. This is done by selecting *Project* → *Build Options*. Then select the appropriate configuration (upper lefthand corner) and set the options in the same way that global options are set. The following screen shot shows the *Other linker options* that are set for the `trig_plot` project in the

projects folder of the distribution. This setting (and others) are needed to use the Plplot library.



1.8.7 Removing Coarray Fortran

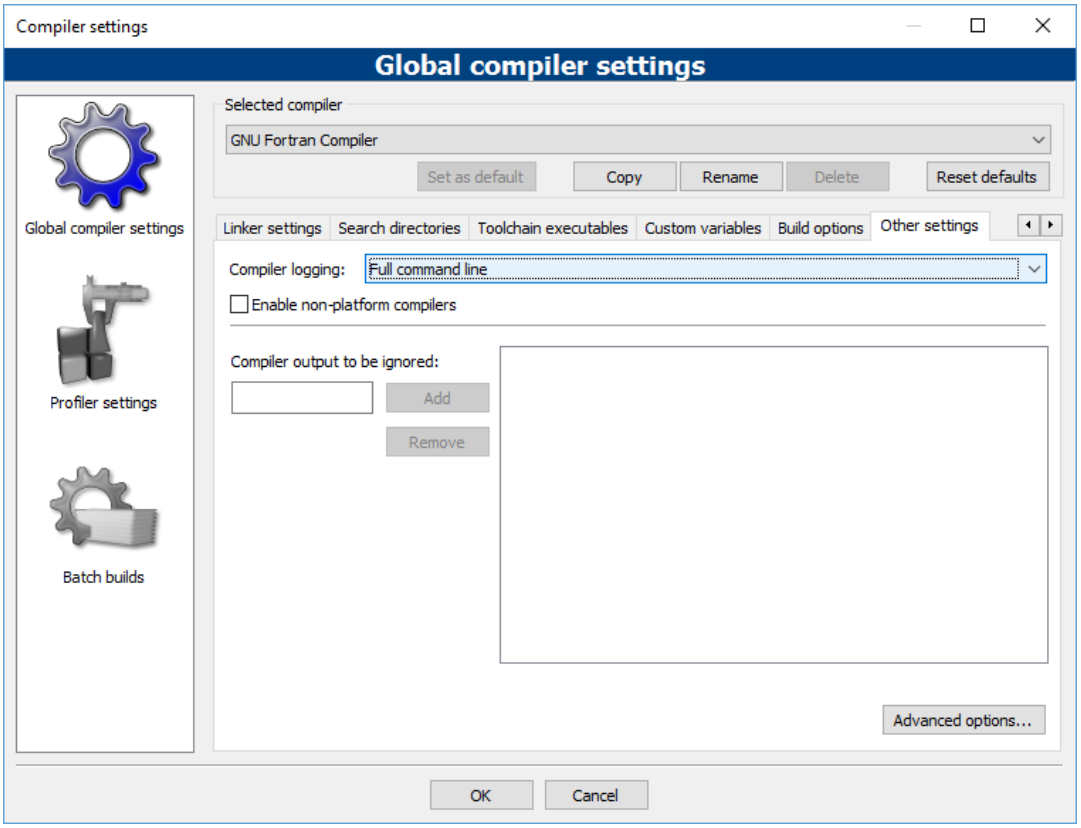
If no Fortran programs will use coarrays, the entire folder `%FORTRAN_TOOLS%\caf` may be removed. This will significantly decrease the size of the installation files.

1.8.8 Displaying Commands Executed

It is often helpful to see which commands are executed when a program is built.

1. Select *Settings* → *Compiler* then *Global compiler settings* as above for compiler options.
2. Select the *Other settings* tab; this is the right-most tab and may not be visible until you move the tabs to the left.
3. For *Computer logging*, select *Full command line* and *OK*.

The commands executed during compilation will appear in the Build log window at the bottom of Code::Blocks.



1.8.9 The Debugger Execution Path

The location of the debugger is set as follows.

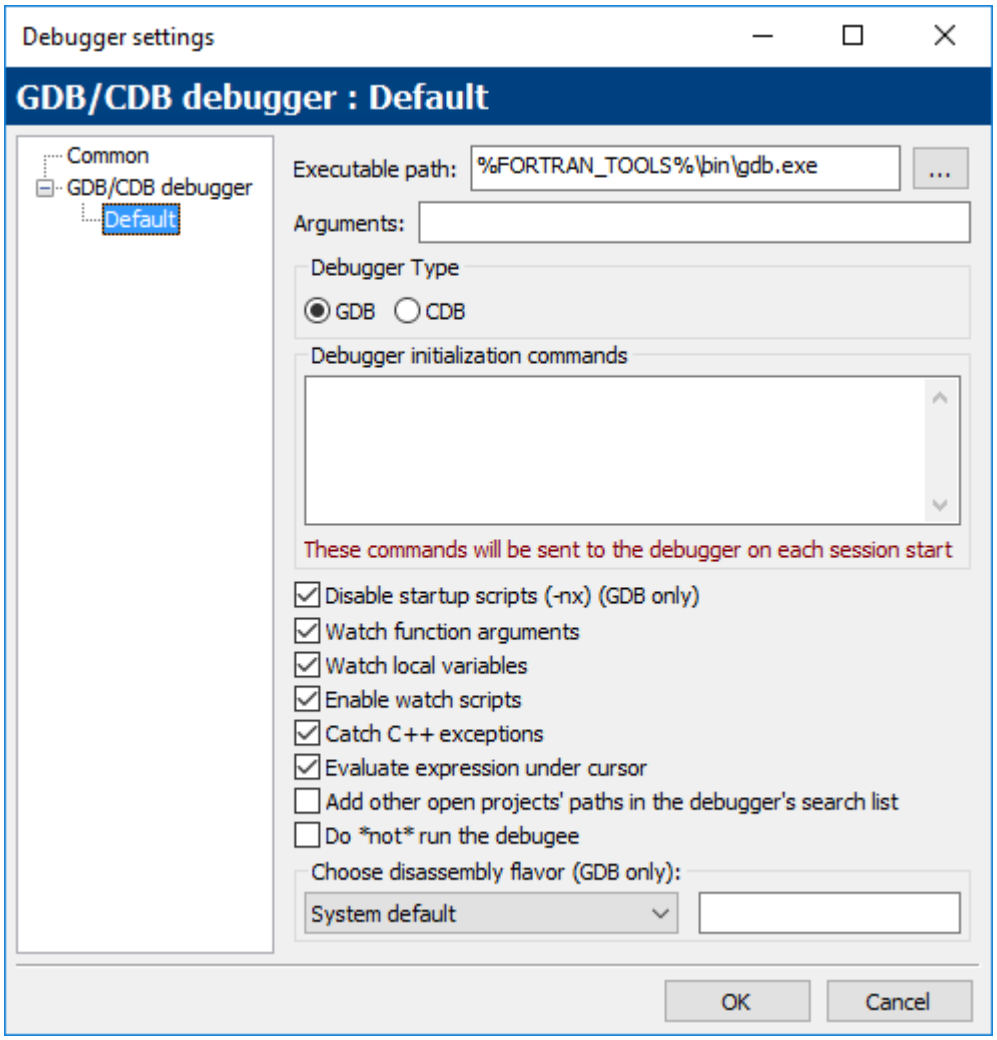
1. Select *Settings* → *Debugger...*
2. *Select GDB debugger* and *Default* in the upper left-hand corner.
3. In the window labeled *Executable path*, enter

   ```
   %FORTRAN_TOOLS%\bin\gdb.exe
   ```

   or, for Code::Blocks with coarrays

   ```
   %FORTRAN_TOOLS%\caf\bin\gdb.exe
   ```

4. Check the box labeled *Evaluate expression under cursor* if not already checked.



1.9 Support

If you have any difficulties or technical queries, please send email to `info@fortran.com`.

1.10 What is Installed

Code::Blocks can be started by clicking on the Fortran Tools shortcut icon and selecting Code::Blocks in the dialog box that appears. It also can be started by clicking on `..\Fortran_Tools\codeblocks\codeblocks.exe`.

The Fortran compiler is located in `..\Fortran_Tools\bin`. It can be run from Code::Blocks (Section 2) or executed as a command (Section 3).

The directory `..\Fortran_Tools\include` contains include files and module files needed when using some of the special software described in this manual.

The directory `..\Fortran_Tools\lib` contains modules and libraries needed when using some of the special software described in this manual.

The directory `..\Fortran_Tools\bin` contains executables for the Fortran Tools.

The directory `..\Fortran_Tools\gnuplot` contains software for Gnuplot.

The directory `..\Fortran_Tools\examples` contains many examples; the subfolders `gtk` and `plplot` contain examples for these programs.

The directory `..\Fortran_Tools\projects` contains several Code::Blocks projects.

Also in the distribution directory is some Fortran Tools documentation (`doc`), source code for some of the software (`src`), and copies of some of the relevant software licenses (`lic`).

1.11 Licenses

1.11.1 Code::Blocks

Code::Blocks is subject to the GNU Public License 3, a copy of which is in the `lic` folder of the Fortran Tools distribution.

1.11.2 Gfortran

Gfortran also is subject to the GPL 3 license.

1.11.3 GTK

The GTK software is licensed under the LGPL license, a copy of which is in the `lic` folder of the Fortran Tools distribution.

1.11.4 Matran

The copyright and license information for Matran are found in the file `Matran.html` in the `doc` directory of the Fortran Tools distribution.

1.11.5 Slatec

This software library is distributed without restrictions by Netlib: `http://www.netlib.org/`.

1.11.6 Gnuplot

The Gnuplot software is subject to the license in the file `gnuplot_license.txt` in the `lic` directory.

1.11.7 Plplot

The Plplot software is subject to the LPGL license.

1.11.8 Other Software

Other software in the Fortran Tools may be controlled by notices in the software or documentation for that software.

1.12 Source Code

1.12.1 Code::Blocks

Source files for Code::Blocks may be found at `http://www.codeblocks.org/`.

1.12.2 Gfortran

The source files for Gfortran may be found at https://gcc.gnu.org/mirrors.html; binaries may be found at http://gcc.gnu.org/wiki/GFortranBinaries.

1.12.3 GTK

The source code for GTK may be obtained at `www.gtk.org`. The source code for GTK-Fortran is in the `src\gtk` directory of the Fortran Tools distribution.

1.12.4 Matran

Source code for Matran may be found at http://www.cs.umd.edu/~stewart/matran/Matran.html.

1.12.5 Slatec

Source code for this software library is found at http://www.netlib.org/.

1.12.6 Plotting Software

The Gnuplot and Plplot source code is at http://www.sourceforge.net.

1.12.7 Other Software

Source code for some of the other software is in the `src` directory of the distribution.

1.13 Documentation

The doc directory copied from the distribution contains the following documentation.

- The complete Fortran Tools manual, of which this is the first section. This manual describes how to run Fortran Tools and contains information about the software provided by The Fortran Company. A PDF version and an Open Office version are provided. The current version of this manual can be found at http://www.fortran.com/Fortran_Tools.pdf.

- Gfortran information may be found in the file `gfortran.pdf` in the `doc` folder and `gcc.pdf` describes other related information, such as linking options. The most recent version of this document may be found at `https://gcc.gnu.org/onlinedocs/gfortran`.
- A Code:Blocks manual is `code_blocks_user_manual.pdf`. There are links to other information at the web site `http://wiki.codeblocks.org`.
- Matran is described in *MatranWriteup* in PDF format in the `doc` directory of the distribution.
- Documentation for Gnuplot is in the file `Gnuplot.pdf` in the `doc` directory of the distribution.
- Documentation for Plplot is in the file `Plplot.pdf` in the `doc` directory of the distribution.
- A GTK manual is in the doc directory of the distribution. More information about GTK may be found at `www.gtk.org/documentation.php`.
- Copies of several books, including *Programmers Guide to Fortran 95 Using F*, *Key Features of Fortran 95*, and *Fortran Array and Pointer Techniques*.
- Documentation of several other programs provided with the Fortran Tools is in the `doc` directory of the distribution.

For additional information about Fortran, visit

`http://www.fortran.com/`

or contact

The Fortran Company
3014 South Cook Street
Denver, Colorado 80210 USA
+1-877-355-6640 (voice & fax)
info@fortran.com

2 Code::Blocks

Code::Blocks is a graphical interface for Fortran. It may be used to edit, compile, run, and debug Fortran programs. It is open source software.

2.1 Introduction to Using Code::Blocks

You can always run Fortran programs from the command line (Section 3), but if you want to use the Code::Blocks graphical interface to edit, compile, run, and debug Fortran programs, follow the instructions in this section.

When using Code::Blocks, your code is organized into *workspaces* and *projects*. A project usually will contain the code for one complete program, consisting of a main program and possibly some modules; a project also may consist of a static or dynamic library (Section 12). These files, and others used by the Code::Blocks system, usually are stored in one directory whose name is the name of the project. A workspace consists of projects; it uses a directory whose name is the name of the workspace to store the project directories. A workspace might contain only one project.

To use Code::Blocks for Fortran programs, you create a project, which is part of some workspace. You then add source code to the project, by modifying the simple sample provided, by copying existing files into the project, or by creating new source files and typing in the code. Then, the project can be built, run, and debugged.

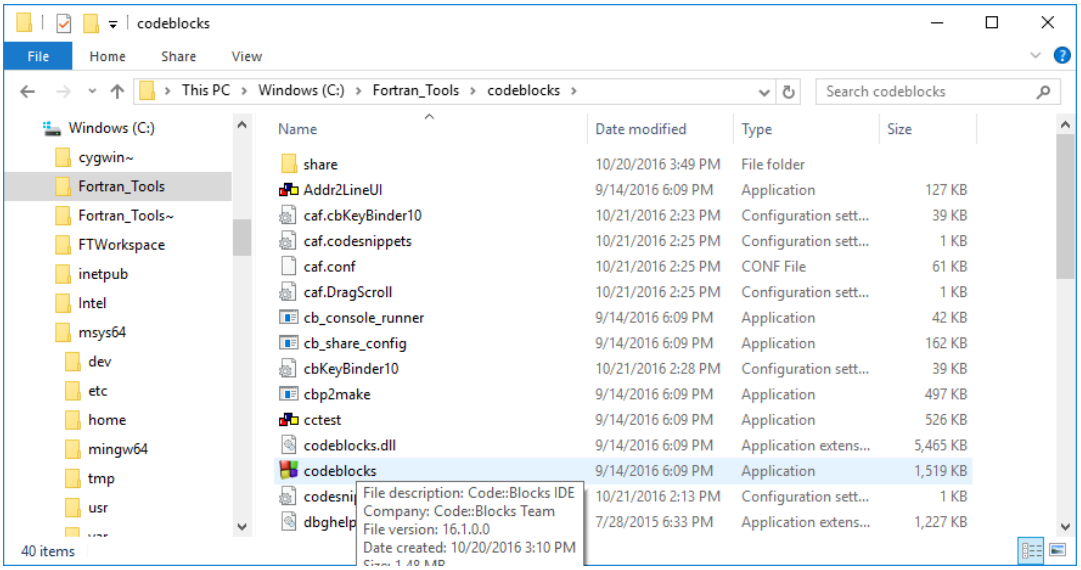
2.2 Starting Code::Blocks

1. You can start Code::Blocks using the yellow Fortran Tools icon. Or, if you created a shortcut (1.7) to Code::Blocks, select it. Otherwise, use `Explorer` to find `codeblocks.exe` in the `codeblocks` subfolder of the Fortran Tools distribution folder and select it for execution. It is the file with the four colored blocks as its icon. Or, you can type `codeblocks` in a command prompt.

2. The easiest way to start is to open an existing project. In this case, the project is opened by selecting the `.cbp` (Code::Blocks project) file and Open. Files with extension `.cbp` are located in the project folder for the project. There are several examples of existing projects in the `projects` folder of the Fortran Tools distribution.

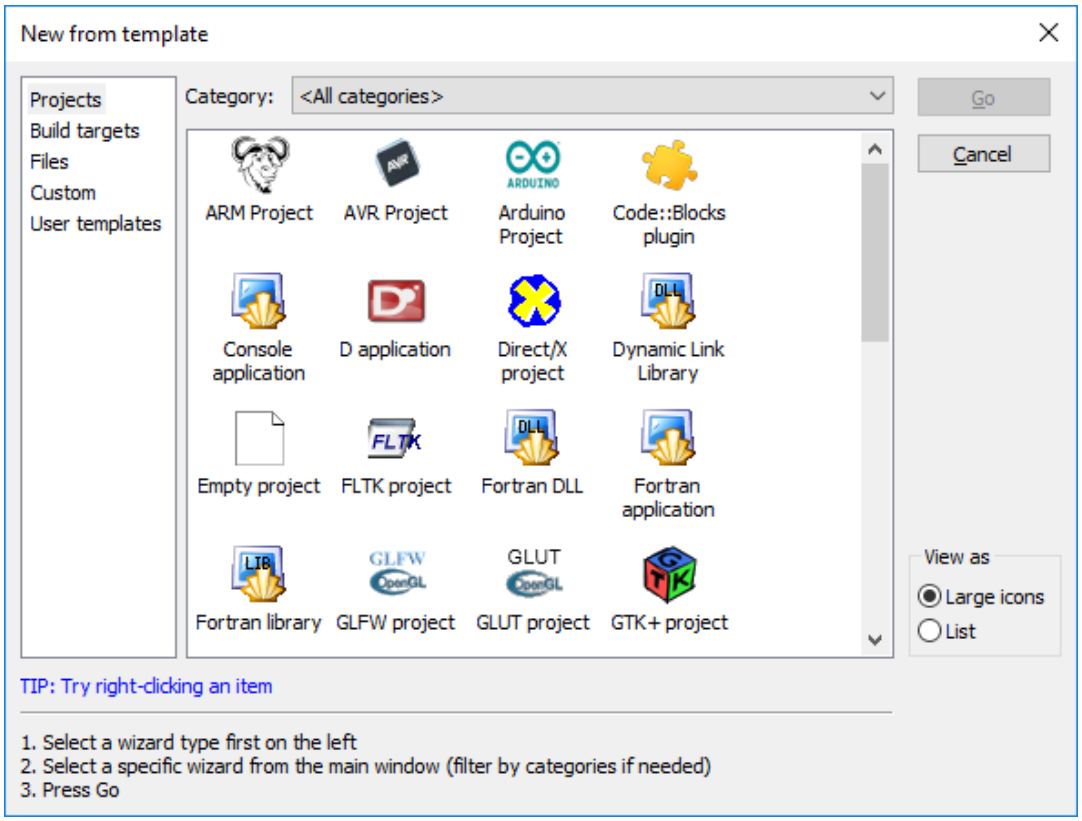
3. Or, to create a new project, on the screen that appears, select *Create a new project*.

4. For a new project, select *Project* and the Fortran application icon from the *New from template* window. Then follow the instructions in Section 2.3.

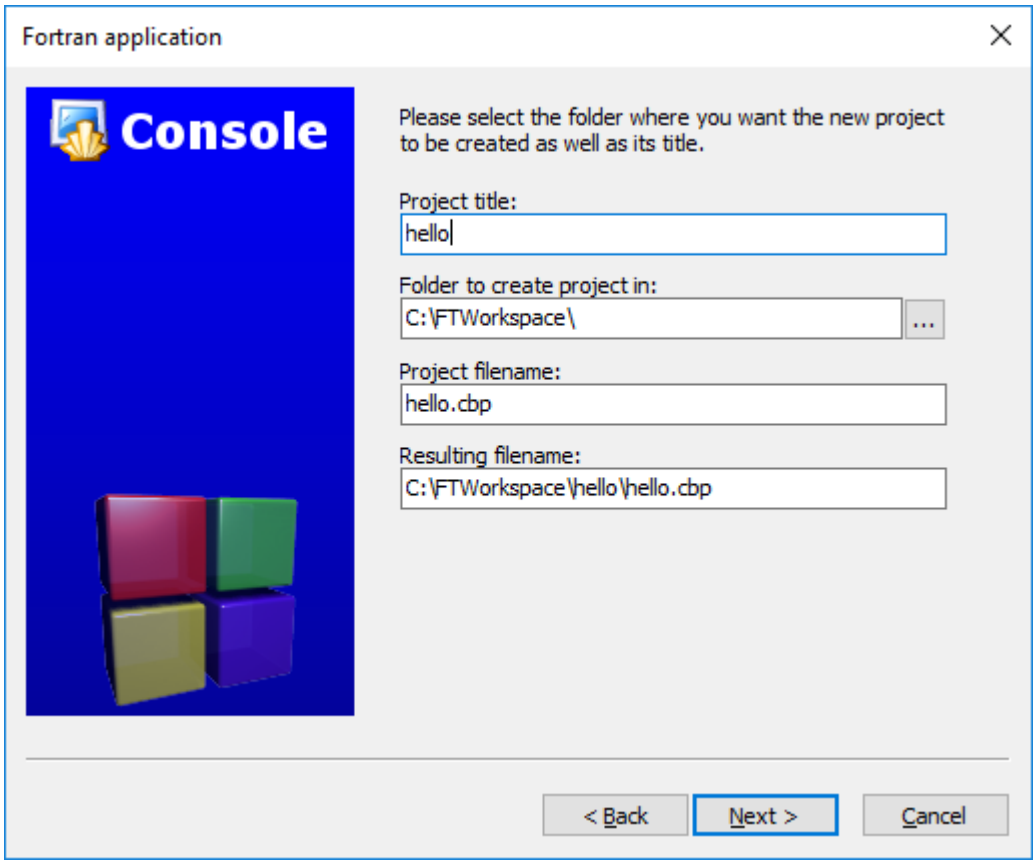


2.3 Creating a New Project

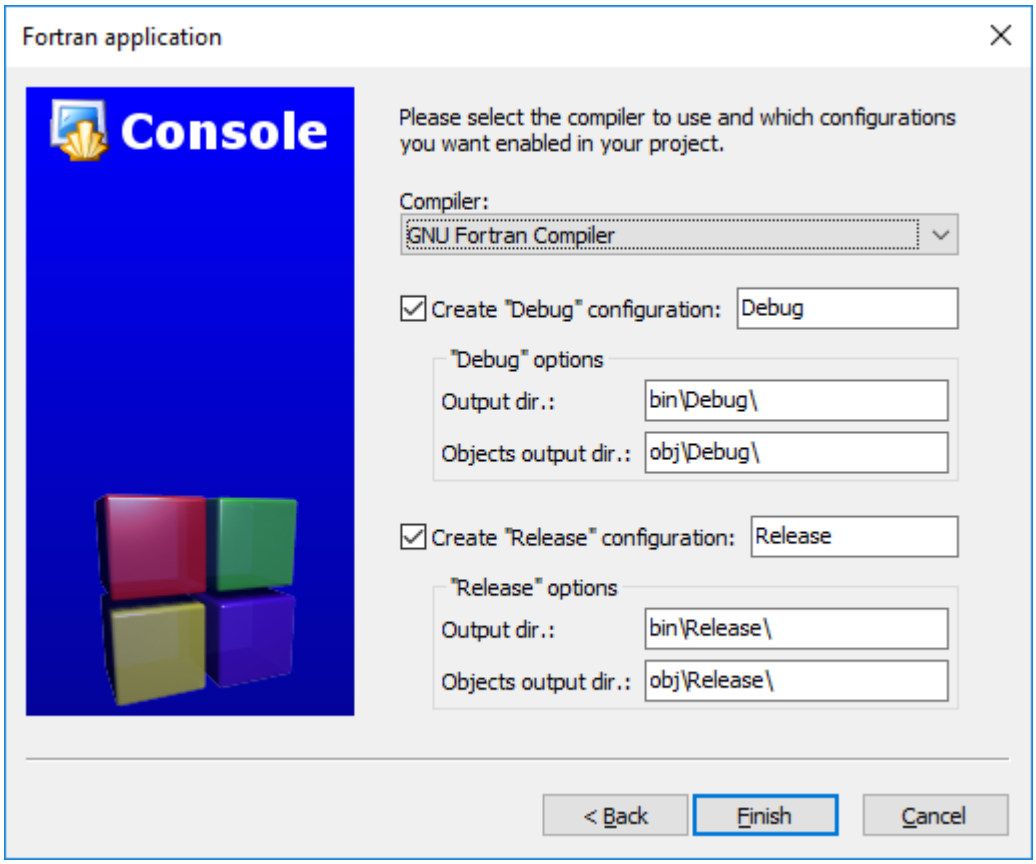
1. When Code::Blocks is started, a new project may be created by selecting *Create a new project* as described above. When Code::Blocks is running, a new project may be created by selecting the *File* tab (upper left corner), then *New*, then *Project*.
2. Select the *Fortran application* icon and then *Go*.



3. Select *Next* to get the Fortran application window. In this window, type a *Project title*, such as `hello`. Also type in a workspace, which probably should not be a subfolder of the Fortran Tools distribution, in case you want to remove the distribution and reinstall it later. A reasonable choice might be something like `C:\FTWorkspace`. The remaining two entries will be filled in for you. Then select *Next*.

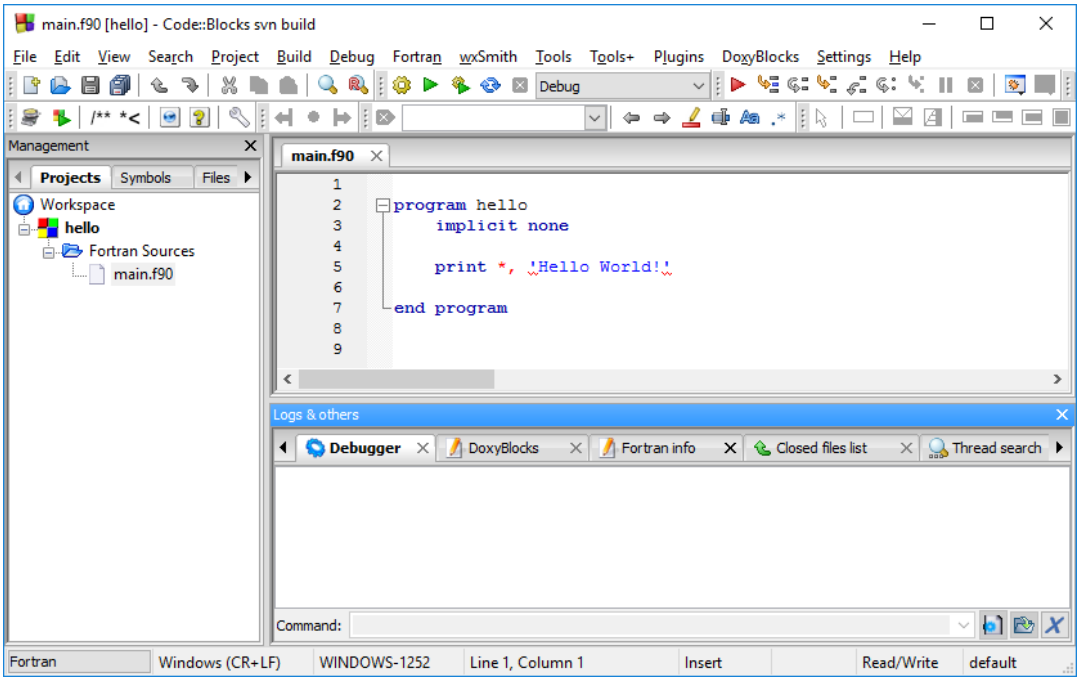


4. In the next window, be sure to select the Compiler *GNU Fortran Compiler*. Set it as the default compiler if it is not already. The other fields should be OK as is; a Release version is not necessary unless you plan to make a production version of the code later. Then select *Finish*.



5. In the *Projects* window (to the left of your screen), the name of the project just created will appear. (Expand the + by *Workspace* and the name of the project if necessary). Under *Fortran Sources* the name of a Fortran program, `main.f90` will appear. This program may be examined by clicking on it. It may be compiled and run by selecting the *Build* tab and then *Build and run*. More details on compiling and running a program appear later.

   A similar project `hello` is in the `projects` folder of the Fortran Tools distribution.

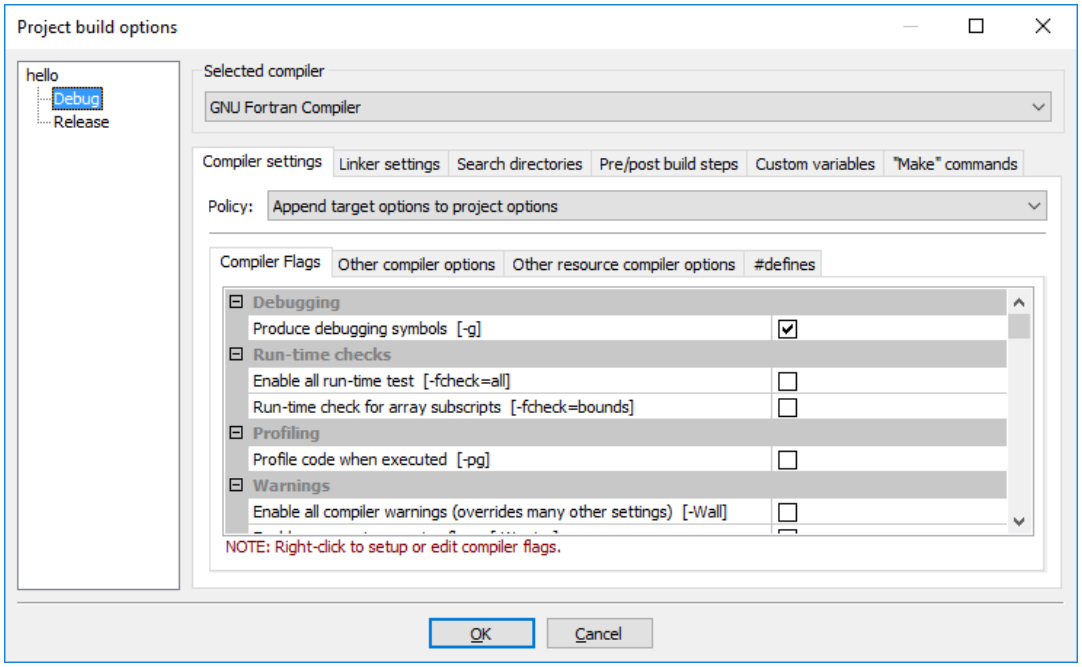


2.4 Compiler Options

Setting global compiler options was discussed in Section 1.8.2 Additional options may be set that are specific to either the Debug or the Release version of a specific project. In many cases, the defaults should be fine and this section can be skipped.

Select the *Project* tab at the top of the Code::Blocks screen and then *Build options*. Make sure the *Selected compiler* is `GNU Fortran Compiler`. Select the *Compiler settings* tab. Also select *Compiler options* and *All categories*. Select *Debug* in the pane to the left of your screen to see the options set for the Debug version. The option *-g* should be selected. It is recommended that the options *-fcheck=all*, *-Wall*, *-std=f2008*, and *-finit-real=nan* also be checked for debugging. This last option is helpful in catching undefined variables (those referenced without being given a value); it must be set by typing it into the window *Project* → *Build options* → *Other options*. To build an executable to be executed on another computer, use the *-static* option so that all necessary libraries are included; this needs to be specified by typing it into the *Other linker* options under the *Linker* settings tab under either the global compiler options or the project settings. This does not include the Fortran Tools libraries.

Select *Release* in the left-hand pane. The options *-s* and *-O2* should be selected, if they are not already.



2.5 Library, Module, and Include Files

For a particular project, additional library, module and include files may be specified on the same Project build options window accessed above. Select *Linker* settings to add a library (1.8.4). Select *Search directories* to indicate where to find a module or include file.

2.6 Importing Existing Files

There are many useful programs provided by the Fortran Tools in the `examples` subfolder of the distribution folder. You may have other Fortran source files you wish to include in a project. Here is how to include such a file.

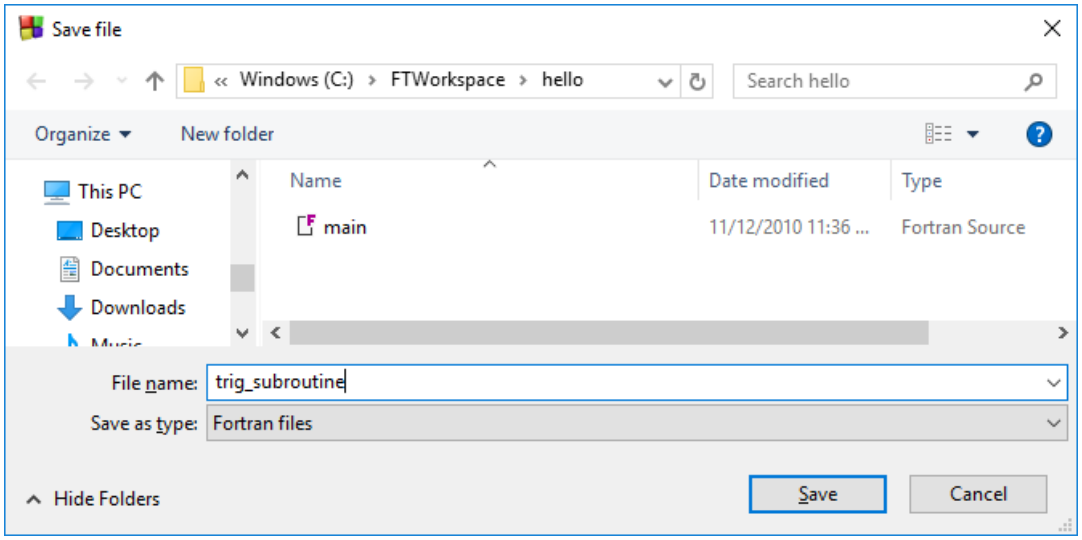
1. Select the *Project* tab, then *Add files*.
2. Use the Explorer-like window that appears to navigate to the file you want. For examples, go to `...\Fortran_Tools\examples`. Select the file you want (or select multiple files using *Control-Click*) and select *Open*.
3. Hit *OK* on the next screen.
4. The file will appear as part of the project with the complete path where it is located.

Any modifications you make to the file will change the original. The file in the project is just a reference to the file you added to the project. If you delete the file from the project, it will not destroy the file on disk.

Another solution is to copy the file to be added into the project workspace (e.g., `C:\FTWorkspace\hello`) and then add the copy to the project as described above.

2.7 Creating a New Source File

1. Select *File* in the upper left corner of your screen, then *New*, then *Empty file*.
2. Answer *Yes* on the next window.
3. In the *Save file* window, navigate to the place where you want to keep the new file. This often will be the workspace containing the current project (e.g., `C:\FTWorkspace\hello`).
4. Type in the *File name*, e.g., `trig_subroutine`. The extension `.f90` will be provided for you.
5. On the *Save as Type* line, select *Fortran files* from the pulldown menu. Select *Save*.



6. In the *Multiple Selection* window, select at least the *Debug* target and also the *Release* target if you want. Select *OK*. The empty file will be added to the project and will appear in the edit window. You can then type in the Fortran statements.

2.8 Removing a Source File

To remove a file from a project, highlight the file name in the *Projects* window, click with the right mouse button, and select *Remove file from project*. The file will no longer be part of the project, but will still be saved on the disk.

2.9 Editing a Source File

1. Double click on the name of the file in the *Projects* window. This will display the contents of that file in the editor view, which occupies the upper central portion of the screen.

2. If you make a change to the file (this will be indicated by an asterisk by the file name just above the edit view), save the file by selecting the save icon (floppy disk) near the upper left corner of your screen. When a changed Fortran file is compiled, it is saved for you.

To provide a simple example, create a new project called `sine` and a new empty file `sine.f90`. Then enter the following program:

```
program sine
   implicit none
   print *, "The sine of 0.5 is", sin(0.5)
end program sine
```

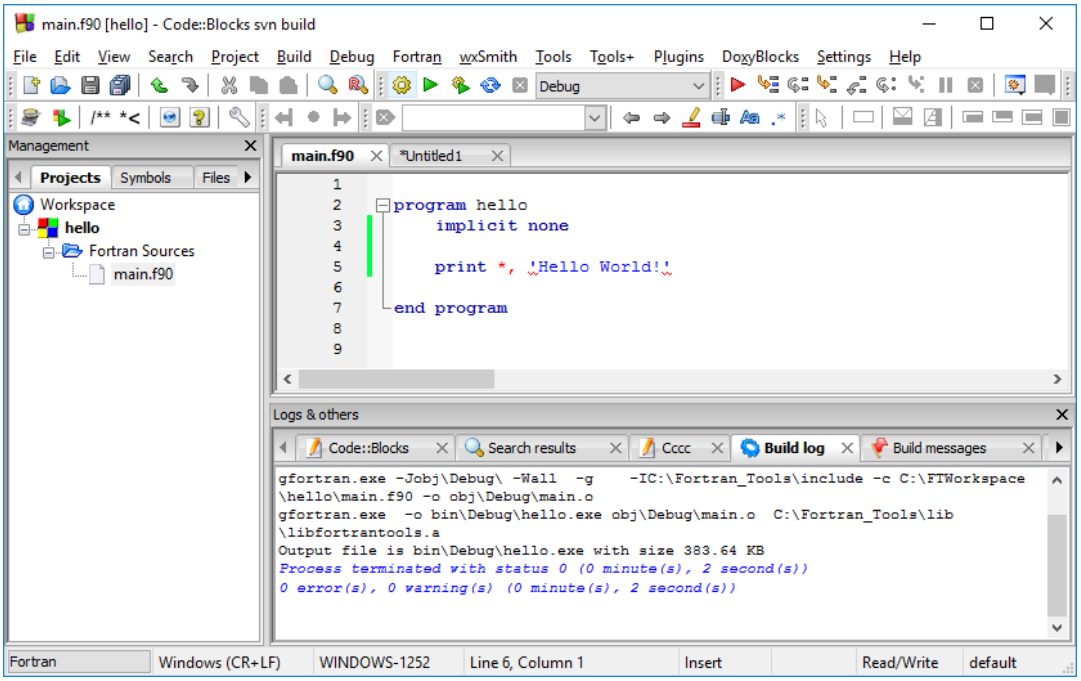
Note the syntax highlighting of Fortran code by Code::Blocks. Comments, character strings, and keywords appear in different colors so that they may be identified readily.

Line numbers appear in the edit window (this can be turned off), and the line number and character position within the line of the cursor are displayed at the bottom of the Code::Blocks window.

Here is another nice feature of Code::Blocks. If you want to comment a whole block of statements, it is necessary to put the comment symbol (`!`) at the beginning of each line. To do this using Code::Blocks, select the lines to be commented (or uncommented), click on the *Edit* tab and select *Comment* or *Uncomment*.

2.10 Building a Project

To build the program, select the project name in the Projects window. Select the *Debug* option on the *Build* icon bar; later, the *Release* option may be used to generate a production version of the program. Then select *Build* from the Build pull-down menu. Or you can select the little wheel icon on the Build icon bar.



Messages related to the compilation should appear in the *Logs & others* window near the bottom of your screen. The *Build messages* window will show errors (double click on the error and a red box will appear to the left of the statement with the error). The *Build log* window will show the commands used to do the compilation and linking.

Any files modified but not saved will be saved before compilation takes place.

Whenever a project is built, only those files that have been changed since the last build and those files that depend on them (`use` and `include`) are compiled. If there is ever doubt that all the necessary files are being recompiled do *Build* → *Rebuild* to recompile everything.

2.11 Running a Program

To run the program, select the *Build* tab and then *Run* from the pulldown menu. Or you can select the little right-facing green arrow icon. For input and output, a console window will appear.

Sometimes it is desirable to run a program in a folder other than the default. For example, the program may require several input files. To put the executable file in another folder and run the program in that folder, select *Project* → *Properties* → *Build targets* → *Execution working dir* and then either type in the folder or browse for it with the elipses (…) button. Then build the program.

2.12 Closing a Project or Workspace

To close a project or workspace, select the *File* tab and then select the appropriate *Close...* message from the pulldown menu. To remove the files from disk, use Explorer.

2.13 Debugging

A program can be debugged using the same Code::Blocks interface that is used to edit, build, and run the program. The debugger has a lot of features, some of which take some effort to learn, but if all you use it for is a replacement for debugging by inserting `print` statements, learning just the simplest features to do that will be well worth the effort.

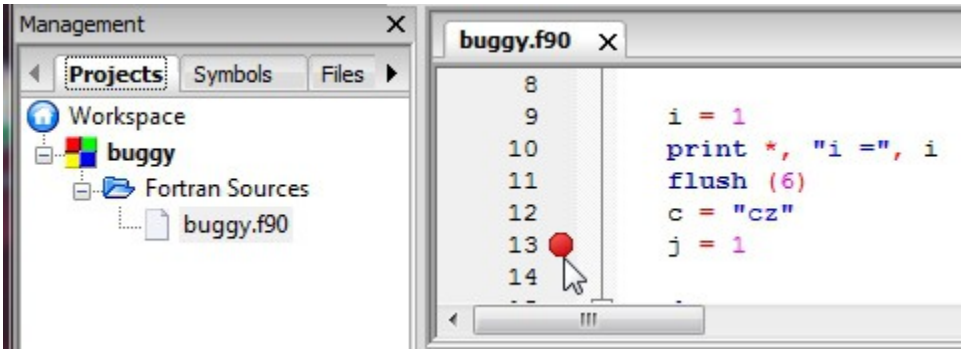
To use the debugger, Fortran programs must be compiled with the `-g` option (1.8.2); this is the default for the Debug release.

Let's explore some of the features with an example.

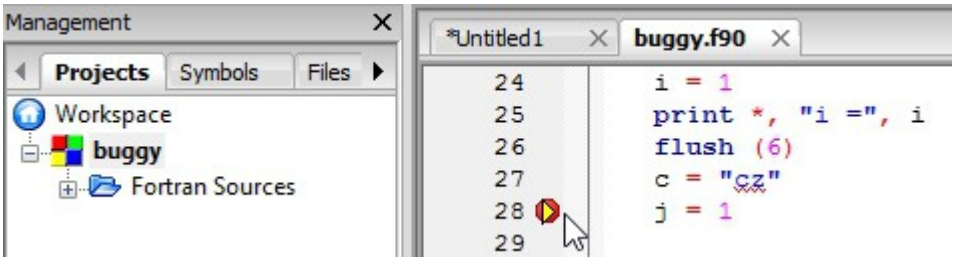
1. Create a new project named `buggy` in your workspace or open the project buggy in the projects folder of the Fortran Tools distribution (and skip to step 4).
2. If the project contains the source file `main.f90`, remove it from the project (2.8).
3. Add to the project (2.6) the file `buggy.f90` in the `examples` folder of the Fortran Tools distribution. Copy it to the workspace folder if you don't want to risk modifying the original in the `examples` folder. Take a look at it if you like.
4. Build the Debug version of the project.
5. Run the program. There appears to be a problem; if you can figure it out, great, but if not, we need to do some debugging.
6. Set a breakpoint: with the source file `buggy.f90` in the edit window. Place the cursor in the left margin of the edit window to the left of the statement

   ```
   j = 1
   ```

   and to the right of the line number. Click or right click and select *Add breakpoint*. Notice that a red circle appears in the margin to indicate the presence of the breakpoint.



7. Select the project name; select the debug icon (the little red arrow); or select *Debug*, then *Start*. You will be asked if you want to save the perspective; answer *Yes.*

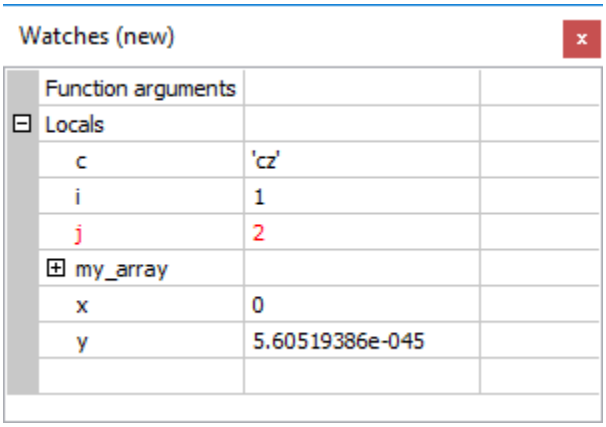


8. The program is suspended at the breakpoint as indicated by the little yellow arrow in the left margin.

9. To determine the problem, we want to execute a few statements and then see how things look. One way to do this is to use the icons to the right of the Debug icon. Move your cursor over them to see what they do. *Run to cursor* treats the location of the cursor as a temporary breakpoint and runs to that point in the program. *Next line* executes one statement. *Next instruction* lets you see the next assembler instruction. *Step into* executes one statement, and stops inside a procedure if one is invoked. *Step out* terminates execution of a procedure if one is being executed. *Stop debugger* (the red circle with the X) stops the program.

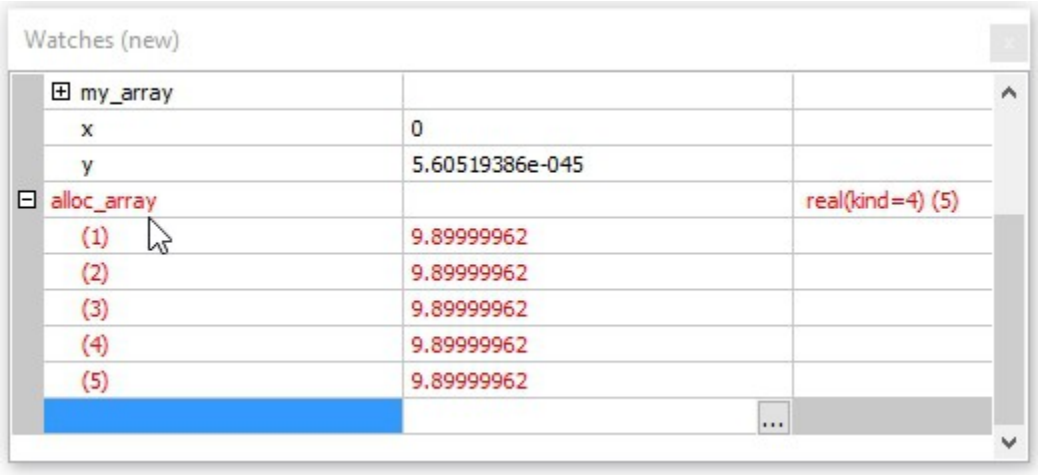
10. The simplest way to examine the value of a variable when the program is stopped at a breakpoint is to select the variable (anywhere in the program). Select it by moving the cursor over it with the left mouse button down, and double click on it with the left mouse button. (Sometimes, just hovering over the variable works.) The type and value of the variable show up in a little box.

11. Either click on the *Debugging* windows icon (on the debug bar) or select *Debug* then *Debugging windows*, then *Watches*. This will create a little window inside which is displayed the values of variables in the program.



12. Use *Next line* to run to the first `if` statement. check the value of the variables `i` and `j` by examining the *Watches* window. Expand *Locals*, if necessary. The value of `j` is red because it was the last thing changed.

13. Note that the values for the module array `alloc_array` do not appear as they are not local. However, those values my be observed using the cursor as described above. They also may be observed by typing the name of the array in the left portion of the first blank line after the line containing `my_array`.



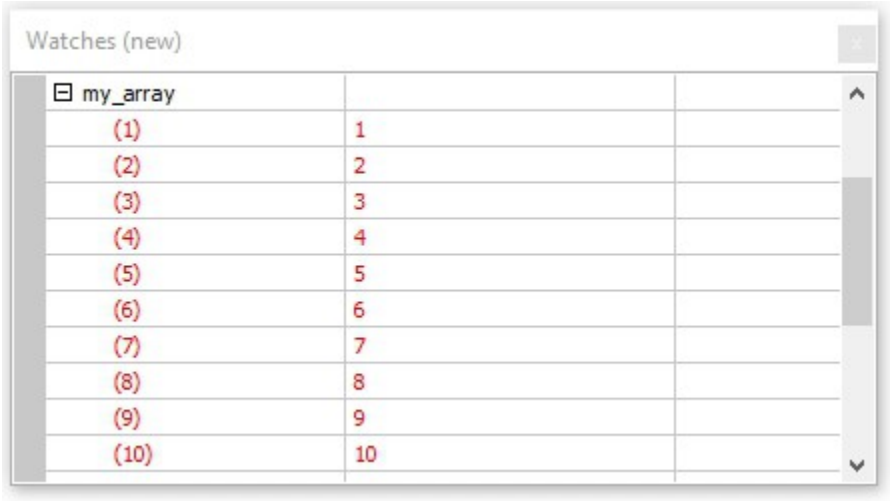
14. Perform *Next line* several times to watch the loop get executed three or four times. Look at the *Watches* window and notice that each time `j` changes, it turns red. In fact, since the loop exits only when `i > n`, and `i` never changes during the loop, that explains the problem. Fix it by changing the test to use `j` instead of `i`. Simply selecting the *Debug/Continue* icon will cause the changed program to be recompiled, then executed.

15. This bug would be pretty easy to figure out by setting the compiler option (2.4)

```
-fbounds-check
```

16. We have fixed the bug, but let's try a few more things with the debugger to see how they work. After rebuilding the project, set a breakpoint at the statement that prints the value of `sum(my_array)` and start the debugger again.

17. When the program stops at the breakpoint, look in the Watches window. `j` is 11, as it should be. Display all the values of `my_array` by selecting + by the array name, if necessary. Also, check the values of `ptr_array` to ensure that it is an alias of `my_array`.

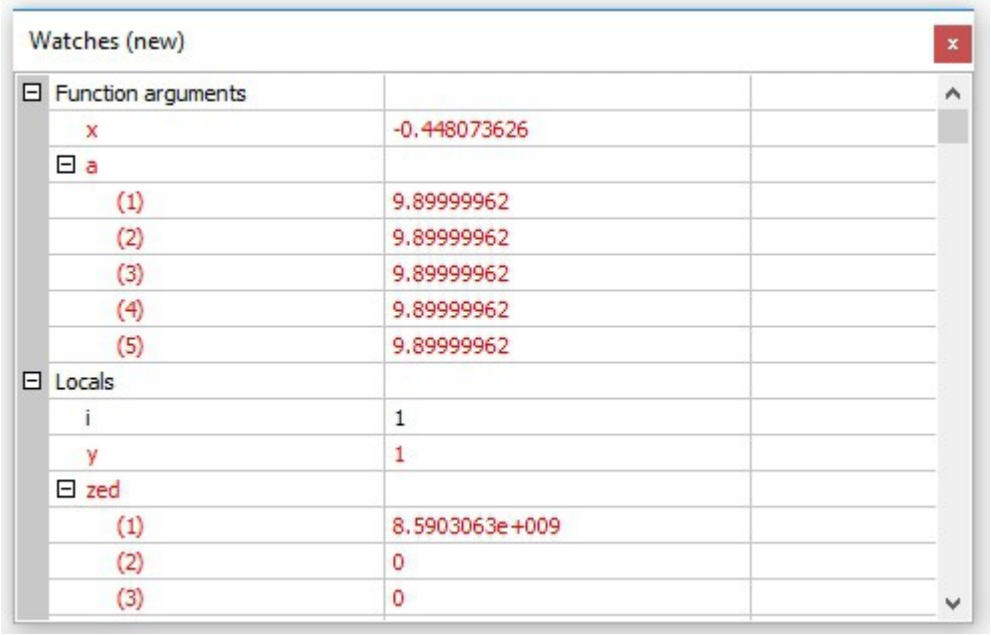


18. Use *Step into* until you get to the `call` statement. Be sure to use *Step into* again (maybe twice) to enter the subroutine `SubA`.

19. Place a breakpoint on the line.

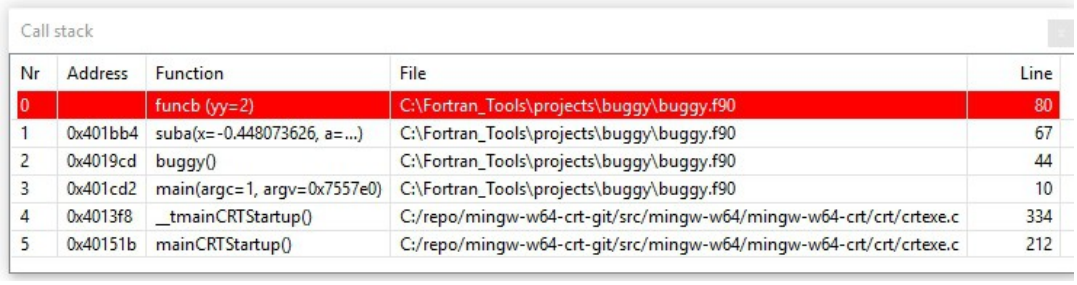
```
zed(i) = FuncB(y)
```

and select *Debug/Continue* to run to the breakpoint. Note that the variables local to the subroutine now appear in the Watches window. The variable `i` is the one declared local to the subroutine, not the one declared in the main program. Also, the values of the subroutine arguments `x` and `a` appear under *Function Arguments* (C has no subroutines). Note that values of the assumed-shape dummy argument `a` are available.



20. *Step into* `FuncB`. Notice that the variables local to the function (e.g., `xx` and `B_result`) now appear in the *Watches* window and the value of the function argument `yy` appears under Functions (expand the + if necessary).

21. To see the chain of calls that led to the execution of `FuncB`, select the *Debug* tab, then *Debugging windows* (or select the *Debugging windows* icon from the Debug icon bar), then select *Call stack*. This shows that `FuncB` was called from `SubA`, which was called from the main program `buggy`. The arguments passed to `FuncB` and `SubA` also are displayed. The other three lines show system routines used to set up the execution of the program `buggy`.



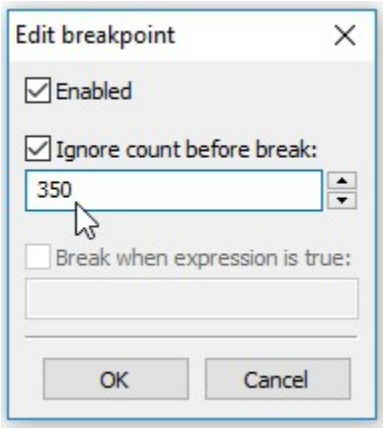
22. Suppose we think all is OK in `FuncB`. *Step out* to complete execution of `FuncB` and go back to `SubA`.

23. Now suppose we suspect that something goes wrong during the last iteration or two of the `do` loop in `SubA`. It would be tedious step through the loop more than 300 times. Instead we can set a *conditional breakpoint*. Set a breakpoint at the line

    ```
    y = i
    ```

    Then right click on the red circle and select *Edit breakpoint*. Select *Ignore count before break* and in the field, type

    ```
    350
    ```

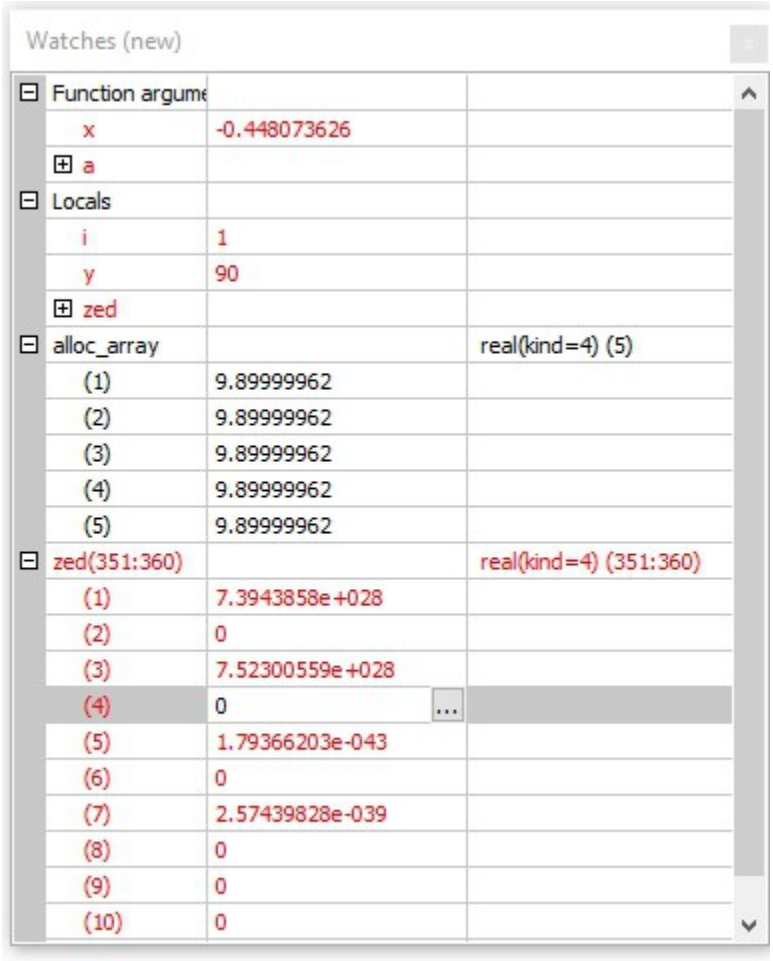
    

    Another option would be to type a logical expression in the *Break when condition is true* field so that the breakpoint would be passed until the condition is true, when the program is stopped. The expression `i .gt. 355` works, but alas, the expression `i > 355` does not.

    Select *OK*.

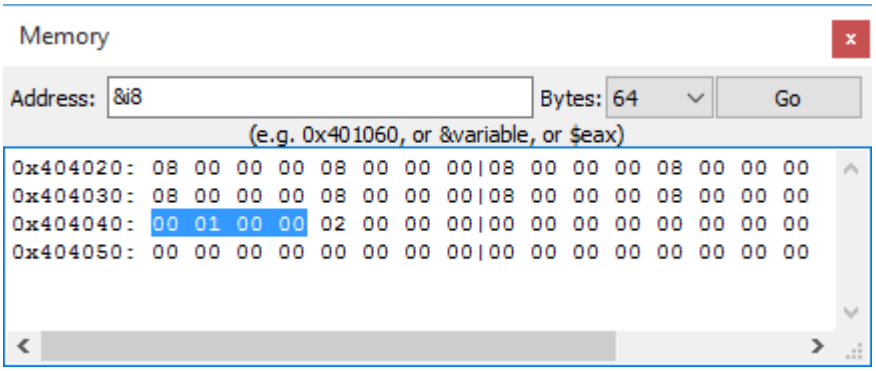
    Remove the breakpoint at the `print` statement and start the debugging process over again

24. To look at the last few values of the array `zed`, select the *Debug* tab, then *Debugging windows,* then *Watches.* Below the last in the list of Function arguments, in the left-most column, enter `zed(351:360)`. The values of `zed(351:360)` are displayed in the Watches window, but with indices 1, 2, ... instead of 351, 352, …

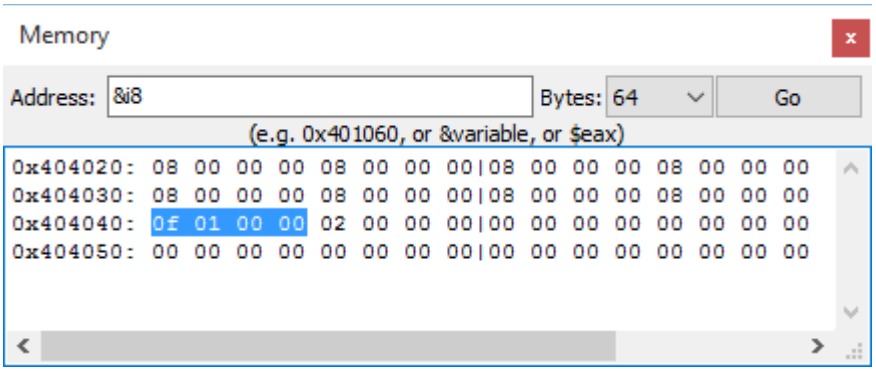


25. Select the red circle with the X to terminate the program.

26. Uncomment the line `i8(9) = 16*16 + 15`, rebuild the program, set a breakpoint on the line just uncommented, and run to the breakpoint. To view directly the contents of memory and show just how disastrous a subscript out of bounds can be, select *Debug* → *Debugging windows* → *Memory dump*. Enter *&i8* in the Address field, *64* Bytes, and *Go*. The eight values of `i8` are shown in the first two lines; the bytes are in reverse order, so, for example, `i8(1)` is hexadecimal 00000008 and the variable `i` has the value hexadecimal 00000100, which is 16x16 = 256 decimal (highlighted in the screen shot).



27. Step to the next line. Notice that the value of `i` has been changed to hexadecimal 10f, which is decimal 16x16+15 = 271. This also can be verified by placing the cursor over the variable `i` in the source code or looking in the watch window.



2.14 Terminating a Program

Usually, an executing program being debugged can be stopped by clicking on the red square containing the white "X" in the Debug icon bar. Or select the *Debug* tab, then *Stop debugger.*

2.15 Other Sources of Information

1. The `Help` tab does not yield much useful information.
2. A Code:Blocks manual is `code_blocks_user_manual.pdf` in the `doc` folder. There are links to other information at the web site `http://wiki.codeblocks.org`.

3 Command Line Compilation

The Fortran compiler may be used with the graphical development tool Code::Blocks or may simply be invoked from the command line.

3.1 Usage

Let us go through the steps to create, compile, and run a simple Fortran program. Suppose we want to find the value of *sin*(0.5).

The first step is to open a Command Prompt window. Select the *Start* icon at the lower left of your computer screen. Then select *All Programs*, then *Accessories*, then *Command Prompt*. In Windows 10, enter *Command Prompt* in the search box at the lower left of your computer screen containing "*Type here to search*"; then select *Command Prompt* from the dialog that is displayed. In Windows 8, it should simply be an "app" under *Accessories*.

It is probably convenient to create a special folder to hold the Fortran programs.

```
mkdir C:\FTPrograms
```

This also can be done with `Explorer`, of course.

The next step is to use any simple text editor to create a file with the suffix `.f90` that contains the Fortran program to print this value; do not use `Word` as it will insert control characters that are not legal in a Fortran program. For example `Notepad` might be used as follows:

```
cd C:\FTPrograms
notepad print_sin.f90
```

Suppose the file contains the following Fortran program:

```
program print_sin
   print *, sin(0.5)
end program print_sin
```

A nice convention is to name the file the same as the name of the program, but with the `.f90` suffix.

The next step is to compile the program. The Fortran command has the following form:

`gfortran` [*option*] . . . [*file*] . . .

so the command for our example is:

```
gfortran print_sin.f90
```

The executable file is named `a.exe` and can be run by entering the command `a` or `a.exe`.

Compiling and running a program that uses coarrays is described in Section 14.

Sometimes, particularly with large, complicated programs, it is desirable to separate the compilation and linking steps into multiple commands. `gfortran` is used for both steps.

```
gfortran -c main.f90
gfortran -c sub1.f90 sub2.f90 func.f90
gfortran -o prog.exe sub1.o sub2.o func.o main.o
```

If it is not convenient to type a whole command on one line, the caret (^), which is Shift+6, can be used to continue a command to another line.

```
gfortran -o prog.exe ^
More? sub1.o sub2.o func.o main.o
```

"More?" is a prompt typed by the system after you type the caret (^) and Enter.

3.2 Using Gfortran with the Fortran Tools

When using the special software that comes with the Fortran Tools, it is often necessary to add information to the command line to tell the compiler where this software is located. For example, to compile the program print_pi.f90 in the examples folder, which uses the math module, the compiler command might look something like

```
gfortran -IC:\Fortran_Tools\include print_pi ^
More?   C:\Fortran_Tools\lib\libfortrantools.a
```

To emphasize what is needed during compilation and what is needed during linking, here is how to do this in two steps:

```
gfortran -c -IC:\Fortran_Tools\include print_pi.f90
```

```
gfortran print_pi.o ^
More? C:\Fortran_Tools\lib\libfortrantools.a
```

These commands asume that the Fortran Tools are installed in the folder C:\Fortran_Tools. In general, it should be the value of the environment variable %FORTRAN_TOOLS%, which is set to the installation folder during installation. Even better, the environment variables %FT_INC% and %FT_LIB% can be used for the separate compile and link steps

```
gfortran -c %FT_INC% print_pi.f90
```

```
gfortran print_pi.o %FT_LIB%
```

or in just one step

```
gfortran print_pi.f90 %FT%
```

3.3 Description

Gfortran is a Fortran 03 compiler with many Fortran 08 features. It translates programs written in Fortran into executable programs, relocatable binary modules, or assembler source files.

The suffix of a filename determines the action gfortran performs upon it. Files with names ending in .f90 or .f95 are taken to be Fortran source files. Files ending in .F90 or .F95 are taken to be Fortran source files requiring preprocessing (Section 4). It is probably a good practice to name source files used with Code::Blocks and Gfortran with the suffix .f90. Files with suffix .f or .for are treated as obsolete fixed source form.

Modules and include files are expected to exist in the current working directory or in a directory named by the `-I` option.

Options not recognized by `gfortran` are passed to the link phase.

3.4 Some Options

A description of some options may be obtained by typing "`gfortran --help`" in a Command Prompt window. All of the options are described in the `gfortran` manual in the `doc` folder.

`-c`

Compile only (produce a `.o` file for each source file); do not link the `.o` files to produce an executable file.

`-D`*name*

Defines *name* as a preprocessor variable. This is equivalent to passing the `-D` option directly to `the` preprocessor.

`-fbounds-check`

Checks array and character string bounds at runtime.

`-finit-real=nan`

Initialize all local real values to NaN (not a number) to help find uninitialized variables.

`-ffixed-form`

Assumes the source file is fixed source form.

`-ffree-form`

Assumes the source file is free source form.

`--help`

Display information about the compiler.

`-fdefault-integer-8`

Sets the default kind of integers to 8.

`-fdefault-real-8`

Sets the default kind of reals to 8.

`-I`*pathname*

Add *pathname* to the list of directories which are to be searched for module information (`.mod`) files and include files. The current working directory is always searched first, then any directories named in `-I` options.

`-o` *output*

Name the output file *output* instead of `a.exe`. This also may be used to specify the name

of the output file produced under the `-c` and `-S` options.

`-O`

Normal optimization.

`-std=f95`

Check that program conforms to Fortran 95 standard.

`-std=f2008`

Check that program conforms to the Fortran 2008 standard.

`-S`

Produce assembler output only. Do not assemble and link.

`-Wall`

Enable most warning messages.

4 Preprocessors

Three preprocessors are available with the Fortran Tools distribution: a C preprocessor which also works for Fortran, `fppr` a simpler preprocessor written by Michel Olagnon, and COCO (conditional compilation), which is an ancillary Fortran standard.

4.1 The C Preprocessor

The C preprocessor is invoked when a source file with the `.F90` or `.F95` extension is compiled. For a description of `cpp`, consult Linux, Unix, or GNU documentation.

4.2 fppr

`fppr` is a preprocessor and "pretty printer". Here is a simple example.

```
$define WINDOWS 1
$define FPPR_KWD_CASE FPPR_LOWER
$define FPPR_USR_CASE FPPR_LEAVE
$define FPPR_MAX_LINE 132
program test_fppr

$if WINDOWS
character(len=*), parameter :: slash = "\"
$else
character(len=*), parameter :: slash = "/"
$endif

character(len=*), parameter :: file_name = &
   "." // slash // "fppr_out.f90"
integer :: ios
character(len=99) :: line

open (file=file_name, unit=35, &
      iostat=ios, status="old", &
      action="read", position="rewind")
if (ios == 0) then
   print *, "Successfully opened ", file_name
   read (unit=35, fmt="(a)") line
   print *, "First line: ", trim(line)
else
   print *, "Couldn't open ", file_name
   print *, "IOSTAT = ", ios
end if

end program test_fppr

! fppr < fppr_in.f90 > fppr_out.f90
! This is f90ppr: @(#) fppridnt.f90  V-1.3 00/05/09 Michel Olagnon
! ( usage: f90ppr < file.F90  > file.f90 )
! gfortran fppr_out.f90
! a
!  Successfully opened .\fppr_out.f90
!  First line: program test_fppr
```

Running `fppr` with input from `fppr_in.f90` (shown above) produces an output file `fppr_out.f90`. `fppr` must be executed explicitly; it is not invoked by the Fortran compiler based on the suffix `.F90`, the way the C preprocessor is. Because the `fppr` variable `WINDOWS` is defined to be true, the generated code will include the `parameter` statement that sets the variable slash to the backslash; if `WINDOWS` were false (`0`), it would be the forward slash. Here is the output file `fppr_out.f90`.

```
program test_fppr
!
      character (len=*), parameter :: slash = "\"
!
      character (len=*), parameter :: file_name = &
          "." // slash // "fppr_out.f90"
      integer :: ios
      character (len=99) :: line
!
      open (file=file_name, unit=35, iostat=ios, &
         status="old", action="read", &
         position="rewind")
      if (ios == 0) then
         print *, "Successfully opened ", file_name
         read (unit=35, fmt="(a)") line
         print *, "First line: ", trim (line)
      else
         print *, "Couldn't open ", file_name
         print *, "IOSTAT = ", ios
      end if
!
end program test_fppr
```

`fppr` does not make use of any command line argument, and the input and output files need thus to be specified with redirection, (they default to the standard input and the standard output).

4.2.1 Options

All options have to be specified through the use of directives.

4.2.2 Directives

All `fppr` directives start with a dollar symbol (`$`) as the first nonblank character in an instruction. The dollar sign was chosen because it is an element of the Fortran character set, but has no special meaning or use. The question mark, which is also an element of the Fortran character set with no special meaning, is used as a "vanishing" separator (see `$define` below)

`$define` *name token-string*

Replace subsequent instances of *name* with *token-string*. *name* must be identified as a token. In order to enable replacement of substrings embedded within tokens, `?` is a special "vanishing" separator that is removed by the preprocessor.

`$define` *name* `$`"*token-string*"

Replace subsequent instances of *name* with *token-string* where *token-string* must not be analyzed since it may consist of multiple instructions, for instance.

`$eval` *name expression*

Replace subsequent instances of *name* with *value* where value is the result, presently of default real or integer kind, of the evaluation of *expression.*

`$undef` *name*

Remove any definition for the symbol name.

`$include` "*filename*"

Read in the contents of *filename* at this location. This data is processed by `fppr` as if it were part of the current file.

`$if` *constant-expression*

Subsequent lines up to the matching `$else`, `$elif`, or `$endif` directive, appear in the output only if *constant-expression* yields a nonzero value. All non-assignment Fortran operators, including logical ones, are legal in *constant-expression*. The logical constants are taken as 0 when false, and as 1 when true. Many intrinsic functions are also legal in *constant-expression*. The precedence of the operators is the same as that for Fortran. Logical, integer, real constants and `$defined` identifiers for such constants are allowed in *constant-expression*.

`$ifdef` *name*

Subsequent lines up to the matching `$else`, `$elif`, or `$endif` appear in the output only if name has been defined.

`$ifndef` *name*

Subsequent lines up to the matching `$else`, `$elif`, or `$endif` appear in the output only if name has not been defined, or if its definition has been removed with an `$undef` directive.

`$elif` *constant-expression*

Any number of `$elif` directives may appear between an `$if`, `$ifdef`, or `$ifndef` directive and a matching `$else` or `$endif` directive. The lines following the `$elif` directive appear in the output only if all of the following conditions hold:

- The *constant-expression* in the preceding `$if` directive evaluated to zero, the name in the preceding `$ifdef` is not defined, or the name in the preceding `$ifndef` directive was defined.
- The constant-expression in all intervening `$elif` directives evaluated to zero.

- The current constant-expression evaluates to non-zero.

If the constant-expression evaluates to non-zero, subsequent `$elif` and `$else` directives are ignored up to the matching `$endif`. Any constant-expression allowed in an `$if` directive is allowed in an `$elif` directive.

`$else`

This inverts the sense of the conditional directive otherwise in effect. If the preceding conditional would indicate that lines are to be included, then lines between the `$else` and the matching `$endif` are ignored. If the preceding conditional indicates that lines would be ignored, subsequent lines are included in the output. Conditional directives and corresponding `$else` directives can be nested.

`$endif`

End a section of lines begun by one of the conditional directives `$if`, `$ifdef`, or `$ifndef`. Each such directive must have a matching `$endif`.

`$macro` *name* `(` *argument* [`,` *argument*] . . . `)` *token-string*

Replace subsequent instances of *name*, followed by a parenthesized list of arguments, with *token-string*, where each occurrence of an argument in *token-string* is replaced by the corresponding token in the comma-separated list. When a macro with arguments is expanded, the arguments are placed into the expanded *token-string* unchanged. After the entire *token-string* has been expanded, `fppr` does not re-start its scan for names to expand at the beginning of the newly created *token-string*, the opposite of the C preprocessor.

4.2.3 Macros and Defines

Macro names are not recognized within character strings during the regular scan. Thus:

```
$define abc xyz
print *, "abc"
```

does not expand `abc` in the second line, since it is inside a quoted string.

Macros are not expanded while processing a `$define` or `$undef`. Thus:

```
$define abc zingo
$define xyz abc
$undef abc
xyz
```

produces `abc`. The token appearing immediately after an `$ifdef` or `$ifndef` is not expanded.

Macros are not expanded during the scan which determines the actual parameters to another macro call. Thus:

```
$macro reverse(first,second) second first
$define greeting hello
reverse(greeting,        &
```

```
$define greeting goodbye &
)
```

produces

```
$define greeting goodbye greeting.
```

4.2.4 Options

A few pre-defined keywords are provided to control some features of the output code:

`FPPR_FALSE_CMT !`*string*

Lines beginning with `!`*string* should not be considered as comments, but processed. For instance, one may define:

```
$define FPPR_FALSE_CMT !$OMP
```

in order to use OpenMP (Section 15) directives in one's code.

`FPPR_MAX_LINE` *expression*

The current desirable maximum line length for deciding about splitting to a continuation line is set to the value resulting of evaluation of *expression*. If the value is out of the range 2-132, the directive has no effect.

`FPPR_STP_INDENT` *expression*

The current indentation step is set to the value resulting of evaluation of *expression*. If the value is out of a reasonable range (0-60), the directive has no effect. Note that it is recommended to use this directive when current indentation is zero, otherwise unsymmetrical back-indents would occur.

`FPPR_NMBR_LINES` *expression*

If *expression* evaluates to true, or non-zero, or is omitted, line numbering information is output in the same form as with `cpp`. If *expression* evaluates to 0, line numbering information is no longer output.

`FPPR_FXD_IN` *expression*

If *expression* evaluates to true, or non-zero, or is omitted, the input treated as fixed-form. If *expression* evaluates to 0, the input reverts to free-form.

`FPPR_USE_SHARP` *expression*

If *expression* evaluates to true, or non-zero, or is omitted, the sharp sign (`#`) may be used as well as the dollar sign as the first character of preprocessing commands. If *expression* evaluates to 0, only commands starting with dollar are processed.

`FPPR_FXD_OUT` *expression*

If *expression* evaluates to true, or non-zero, or is omitted, the output code is intended to be fixed-form compatible. If *expression* evaluates to 0, the output code reverts to free-form.

`FPPR_KWD_CASE` *expression*

If *expression* evaluates to 1, or is the keyword `FPPR_UPPER`, Fortran keywords are output in upper case. If *expression* evaluates to 0, or is the keyword `FPPR_LEAVE`, Fortran keywords are output in mixed case. If *expression* evaluates to –1, or is the keyword `FPPR_LOWER`, Fortran keywords are output in lower case.

`FPPR_USR_CASE` *expression*

If *expression* evaluates to 1, or is the keyword `FPPR_UPPER`, user-defined Fortran identifiers are output in upper case. If *expression* evaluates to 0, or is the keyword `FPPR_LEAVE`, user-defined Fortran identifiers are output in the same case as they were input. If *expression* evaluates to –1, or is the keyword `FPPR_LOWER`, user-defined Fortran identifiers are output in lower case.

4.2.5 Output

Output consists of a copy of the input file, with modifications, formatted with indentation, and possibly changes in the case of the identifiers according to the current active options.

4.2.6 Diagnostics

The error messages produced by `fppr` are intended to be self-explanatory. The line number and file name where the error occurred are printed along with the diagnostic.

4.2.7 Source Code

The source code is available in the `src` directory of the distribution. It is provided by Michel Olagnon and more information about this program and others provided by Michel may be found at

```
http://www.ifremer.fr/ifremer/ditigo/molagnon/
```

4.3 COCO

The program Coco provides preprocessing as per Part 3 of the Fortran Standard (Coco stands for "conditional compilation"). It implements the auxiliary third part of ISO/IEC 1539-1:1997 (better known as Fortran 95). (Part 2 is the ISO_VARYING_STRINGS standard, which is sometimes implemented as a module.) A restore program, similar to that described in the standard, is also available for download.

The Fortran source code for Coco may be found at http://daniellnagle.com/coco.html.

Generally, Coco programs are interpreted line by line. A line is either a Coco directive or a source line. The Coco directives start with the characters `??` in columns 1 and 2. Lines

are continued by placing & as the last character of the line to be continued. Except for the ?? characters in columns 1 and 2, Coco lines follow the same rules as free format lines in Fortran source code. A Coco comment is any text following a ! following the ?? characters. A Coco comment may not follow the &.

A description of Coco may be found in the file coco.html in the doc directory. Here is a simple example.

Statement of the problem to be solved: A Fortran program needs to use full path names for file names. The separator in the file names should be / if the system is not Windows and \ if it is Windows. A file slash.inc contains the following, which indicates whether the system is Windows or not.

```
?? logical, parameter :: windows = .true.
```

Then the following program will produce the correct character.

```
module slash

?? include "slash.inc"

   character, parameter, public :: &
?? if (windows) then
      slash = "\"
?? else
      slash = "/"
?? end if

end module slash

program p

   use slash
   print *, "Path is folder" // slash // "progs"

end program p
```

The Coco preprocessor is run with

```
coco < slash.f90 > new_slash.f90
```

which produces the file new_slash.f90:

```
module slash

!?>?? include "slash.inc"
!?>??! INCLUDE slash.inc
!?>?? logical, parameter :: WINDOWS = .true.
!?>??! END INCLUDE slash.inc

   character, parameter, public :: &
!?>?? if (windows) then
      slash = "\"
!?>?? else
!?>      slash = "/"
!?>?? end if
```

```
end module slash

program p

   use slash
   print *, “Path is folder” // slash // “progs”

end program p
```

Compiling and running the program produces the output:

```
 Path is folder\progs
```

5 Calling C Programs

Fortran programs may call C programs compiled with `gcc`, which is the C compiler in the Fortran Tools.

5.1 Calling a C Function

There are several problems that make it a little difficult to correctly call a C procedure. The most obvious is that the actual Fortran and dummy C arguments must match. Another problem is simply getting the name of the C procedure right, as the system may use something different than the name given by the C programmer.

5.2 Interfaces to C Procedures

To have a Fortran program make a correct call to a C procedure, an interface describing the C procedure is written in the calling Fortran program. The main new feature that is needed is to put `bind(c)` at the end of the subroutine or function statement in the interface describing a C procedure. This tells the Fortran compiler that it is a C procedure being called, even though the interface information is all written in Fortran.

C has only functions, no subroutines. However, a function can be declared as returning `void`, a special indication that the function does not return a regular value; such a function should be called as a subroutine from Fortran, whereas all other C functions should be called as Fortran functions.

5.3 The `iso_c_binding` Intrinsic Module

In Fortran, there is a built-in (intrinsic) module named `iso_c_binding`. All you have to do is use it to have access to several named constants and procedures that will help create a correct call to a C procedure, on whatever system the program is run.

One of the main features of this intrinsic module is a collection of named constants (parameters) that are Fortran kind numbers that indicate kinds that correspond to C data types. C does not use kinds, but uses a different data type for each kind, so that, for example, the C data types `float` and `double` correspond to two different kinds of `real` in Fortran. The following table lists a few of these parameters for the most common C data types.

C Type	Fortran Type and Type Parameter
`int`	`integer(kind=c_int)`
`short int`	`integer(kind=c_short)`
`long int`	`integer(kind=c_long)`
`float`	`real(kind=c_float)`
`double`	`real(kind=c_double)`
`char`	`character(len=1, kind=c_char)`

C does not have the character string as a data type. Strings are represented by arrays of single characters. However, a Fortran program may call a C function with a dummy argument that is an array of characters and pass a character string actual argument. This is one of the things that is taken care of by putting `bind(c)` on the interface of the procedure being called. Note that in the interface, the dummy character array is declared to have `dimension(*)` (do not ask why—just do it).

Another parameter in the `iso_c_binding` intrinsic is `c_null_char`. It is a single character that is used to terminate all C strings and hence needs to occur at the end of a character string passed to a C function.

The `iso_c_binding` intrinsic module contains other things useful for interoperating with C.

5.4 An Example of Interoperation with C

The following example illustrates the use of some of these features. The example shows how to pass a character string, an array, and a structure, as well as simple variables.

```
module type_def

   use, intrinsic :: iso_c_binding
   implicit none
   private
   type, public, bind(c) :: t_type
      integer(kind=c_int) :: count
      real(kind=c_float) :: data
   end type t_type

end module type_def

program fortran_calls_c

   use type_def
   use, intrinsic :: iso_c_binding
   implicit none

   type(t_type) :: t
   real(kind=c_float) :: x, y
   integer(kind=c_int), dimension(0:1, 0:2) :: a

   interface
      subroutine c(tp, arr, a, b, m) bind(c)
         import :: c_float, c_int, c_char, t_type
         type(t_type), value :: tp
         integer(kind=c_int), dimension(0:1, 0:2) :: arr
         real(kind=c_float), value :: a
         real(kind=c_float) :: b
         character(kind=c_char), dimension(*) :: m
      end subroutine c
   end interface

   t = t_type(count=99, data=9.9)
   x = 1.1
   a = reshape([1, 2, 3, 4, 5, 6], shape(a))
```

```
   call c(t, a, x, y, "doubling x" // c_null_char)
   print *, x, y
   print *, t
   print *, a

end program fortran_calls_c
```

Following is the C program that implements the function `c`.

```
#include <stdio.h>

typedef struct {int amount; float value;} newtype;

void c(newtype nt, int arr[3][2], float a, float *b, char msg[])
{
   printf (" %d %f\n", nt.amount, nt.value);
   printf (" %d %d %d\n", arr[0][1], arr[1][0], arr[1][1]);
   printf (" %s\n", msg);
   *b = 2*a;
}
```

Following is the output:

```
 99 9.90000
 2 3 4
 doubling x
 1.1 2.2
 99 9.9
 1 2 3 4 5 6
```

The module defines the derived type `t_type`. A structure `tp` of this type is passed as an actual argument, so the components must be declared to agree with the structure `nt` of type `newtype` in the C function. This uses the parameters `c_int` and `c_float` from the intrinsic module. The type itself is defined with the `bind(c)` attribute so that the components will be laid out in a manner similar to the components of a C structure of type `newtype`.

The program `fortran_calls_c` uses the module `type_def` to access the derived type `t_type`. The structure `t` is declared to be `t_type`. `x` and `y` are real variables with the kind to match a C `float` data type.

All C arrays have lower bound 0 and are stored by rows rather than by columns, as Fortran arrays are stored. Hence, it makes it a little easier to give the array lower bounds 0. Also note that the subscripts are reversed in the declaration of the Fortran actual array `a` and the C dummy array `arr`.

An interface for the C function `c` includes `bind(c)` and it uses the `import` statement to make some of the parameters in the intrinsic modules accessible within the interface, which has its own scope.

`t` is given a value with a structure constructor, `x` is given the value 1.1, and `a` is given the value

$$\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

with an array constructor and the `reshape` function.

The C procedure is called as a subroutine and values are printed in both the C procedure and the main program.

The program can be built from the command line with the commands:

```
gfortran -c fortran_calls_c.f90
gcc c.c
gfortran fortran_calls_c.o c.o
```

These source files are in the `examples` folder.

There are many more features involved with the interoperability with C; for example, pointers can be passed and a C function can call a Fortran procedure under the right circumstances. If you need to interact with C in a more complicated manner, please consult a manual or reference work, such as *The Fortran 2003 Handbook*.

6 The Input/Output Module

The input/output module currently contains no items.

7 The Math Module

When using these features and compiling from the command line the compiler must use the option

```
-I%FORTRAN_TOOLS%\include or %FT_INC%
```

and the linker must include the file

```
%FORTRAN_TOOLS%\lib\libfortrantools.a or %FT_LIB%
```

or compile and link in one step using `%FT%`.

```
gfortran print_pi.f90 %FT%
```

These are included for you when using Code::Blocks.

7.1 Math Constants

The module `math_module` contains definitions of parameters for the constants π, e, ϕ, and γ. The names of the constants are `pi`, `e`, `phi`, `gamma`, `pi_double`, `e_double`, `phi_double`, and `gamma_double`. An example of its use is

```
program print_pi

   use math_module

   print *, pi_double

end program print_pi
```

7.2 Randomizing the Random Number Generator

Even if the intrinsic subroutine `random_seed` is called with no arguments, the `random_number` intrinsic subroutine produces the same sequence of numbers each time a program is run. Different numbers can be generated by calling the Fortran Tools subroutine `randomize`. It sets the random number generator see to a value which depends on the system clock. This is not a very sophisticated scheme and may need to be modified for some applications. For example, it is possible that more than one coarray image could generate the same sequence of numbers; this might be fixed by multiplying the seed by the image number, for example.

```
program test_randomize
   use math_module
   implicit none
   real, dimension(4) :: x
   call randomize()
   call random_number(x)
   print*,x
end program test_randomize
```

Running the program twice produces the following output.

```
 0.733499706       0.131829679       0.460422099       0.470994234

 0.985081255       0.657111406       0.515435159       0.217222929
```

7.3 The gcd Function

Also in the math module is the elemental function `gcd` that computes the greatest common divisor or two integers or two integer arrays.

```
program test_gcd
   use math_module
   print *, gcd([432,16],[796,48])
end program test_gcd
which prints

 4 16
```

8 The Slatec Library

The Slatec library is a collection of mathematical routines developed jointly by Sandia National Laboratories, Los Alamos National Laboratory, and the Air Force Phillips Laboratory, all in New Mexico.

The Fortran Tools include a few "wrapper" functions that make it easier to call some of the Slatec procedures; they may be used in a Fortran program as a "built-in" module. Invoke any of the procedures described below from any Fortran program containing the following statement:

```
use slatec_module
```

When using these features and compiling from the command line the compiler must use the option `%FT_INC%` and the linker must include `%FT_LIB%` or compile and link in one step using `%FT%`.

```
gfortran slatec_ode.f90 %FT%
```

These are included for you when using Code::Blocks.

8.1 Finding Roots in an Interval

```
find_root_in_interval(f, a, b, root, indicator)
```

is a subroutine that searches for a zero of a function *f(x)* between the given values *a* and *b*.

`f` is a function of one variable. `a` and `b` specify the interval in which to find a root of `f`. `root` is the computed root of `f` in the interval `a` to `b`. These are all type default real.

`indicator` is an optional default integer argument—if it is zero, the answer should be reliable; if it is positive, it is not.

Here is an example using the subroutine `find_root_in_interval`.

```
module function_module

   public :: f

contains

function f(x) result(r)

   real, intent(in) :: x
   real :: r

   r = x**2 - 2.0

end function f

end module function_module

program find_root

   use function_module
```

```
   use slatec_module
   real :: root
   integer :: indicator

   call find_root_in_interval&
        (f, 0.0, 2.0, root, indicator)

   select case (indicator)
   case (0)
      print *, "A root is", root
   case (1)
      print *, "A possible root is", root
   case default
      print *, "Root not found"
   end select

end program find_root
```

Running this program produces

```
A root is   1.41421568
```

8.2 Finding Roots of a Polynomial

The subroutine

```
find_roots_of_polynomial(coefficients, roots, indicator)
```

accepts the coefficients of a polynomial and finds its roots (values where the polynomial is zero).

`coefficients` is a default real array; the element with the smallest subscript is the constant term, followed by the first degree term, etc. Thus, a reasonable choice is to make the lower bound of the array `coefficients` 0 so that the subscript matches the power of the coefficient.

`roots` is a complex array with at least as many elements as the degree of the polynomial. The roots of the polynomial will be found in this array after calling `find_roots_of_polynomial`.

`indicator` is a default integer optional argument; if it is positive, the solution is not reliable. In particular, if `indicator` is 1, a solution was not found in 30 iterations, if it is 2, the high-order coefficient is 0, if it is 3 or 4, the argument array sizes are not appropriate; if it is 5, allocation of a work array was not successful.

Here is a simple example that computes the roots of $x^2 - 3x + 2 = 0$.

```
program poly_roots

   use slatec_module

   complex, dimension(2) :: roots
   integer :: ind

   call find_roots_of_polynomial &
         ( [ 2.0, -3.0, 1.0 ], roots, ind)
   print *, "Indicator", ind
```

```
   print *, "Roots", roots

end program poly_roots
```

Running the program finds the roots 1 and 2.

```
 Indicator 0
 Roots (2.00000,0.00000E+00) (1.00000,0.00000E+00)
```

8.3 Computing a Definite Integral

```
integrate(f, a, b, value, tolerance, indicator, evaluations)
```

is a general purpose subroutine for evaluation of one-dimensional integrals of user-defined functions. `integrate` will pick its own points for evaluation of the integrand and these will vary from problem to problem. Thus, it is not designed to integrate over data sets.

`f` must be a function with a single argument. `a` and `b` are the limits of integration. `tolerance` is an optional requested error tolerance; if it is not present, 10^{-3} is used. `value` is the calculated integral. These are all type default real.

If the returned value of the optional default integer argument `indicator` is positive, the result is probably not correct. The value of `evaluations` indicates the number of integrand evaluations needed.

```
module sine_module

public :: sine

contains

function sine (x) result (sine_result)

   intrinsic :: sin
   real, intent (in) :: x
   real :: sine_result

   sine_result = sin (x)

end function sine

end module sine_module

program integration

use sine_module
use slatec_module
real :: answer
integer :: indicator

call integrate(sine, a=0.0, b=3.14159, &
   value=answer, tolerance=1.0e-5, &
   indicator=indicator, evaluations=evaluations)
print *, "Indicator is", indicator
```

```
print *, "Number of fn evaluations", evaluations
print *, "Value of integral is", answer

end program integration
```

Running this program produces

```
 Indicator is 0
 Number of fn evaluations is 25
 Value of integral is   2.0000000
```

8.4 Special Functions

`ln_gamma(x)` is a function that returns the natural logarithm of the gamma function for positive real values of `x`. `asinh(x)`, `acosh(x)`, and `atanh(x)` return the inverse hyperbolic function values. The program

```
program test_gamma
use slatec_module
print *, "4! = ", exp(ln_gamma(5.0))
end program test_gamma
```

produces

```
 4! =   24.0000000
```

All of these functions are intrinsic in Fortran 03 and later. To access the intrinsic functions, simply remove the `use` statement.

8.5 Differential Equations

```
solve_ode(f, x0, xf, y0, yf, tolerance, indicator)
```

is a subroutine that solves an ordinary differential equation

$$\frac{du}{dx} = f(x, u)$$

using a fifth-order Runge-Kutta method.

`f` must be a function of two variables. `x0` is the initial value of x. `y0` is the initial value of y. `xf` is the final value of x. `yf` is the final solution value of y. `tolerance` is an optional requested tolerance; if not present 10^{-3} is used. All of these are type default real.

`indicator` is an optional default integer value—if it is positive, the solution is not reliable; a value of 0 indicates success.

Here is a simple example with $f(x, u) = -0.01y$, $x_0 = 0$, $y_0 = 100$, and $x_f = 100$.

```
module f_module

   public :: f

contains

function f(x, y) result(r)
```

```
    real, intent(in) :: x, y
    real :: r

    r = -0.01 * y

end function f

end module f_module

program test_ode

use slatec_module
use f_module

real :: x0 = 0.0, xf = 100.0, &
        y0 = 100.0, yf

call solve_ode (f, x0, xf, y0, yf)

print *, "Answer is", yf

end program test_ode
```

Running the program produces

```
 Answer is  36.7878761
```

8.6 Linear Equations

The subroutine `solve_linear_eqns(A, x, b, indicator)` solves the set of linear equations given by

```
A x = b
```

`A` is a square array of the equation coefficients. `b` is the vector of constants that make up the right side of the equations. The solution is `x`.

`indicator` is an optional argument that returns the status of the result. If it is 0, the solution is reliable; if it is positive, it is not; if it is negative, the absolute value of the indicator is the approximate number of significant digits in the answer.

Here is an example:

```
program slatec_linear_eqns

use slatec_module

integer, parameter :: n = 3
integer :: indicator

real, dimension(n, n) :: A
real, dimension(n)    :: b, x

! Put some values in the matrix A
A = reshape( [ ((real(i+j),i=1,n),j=1,n) ], shape = [ n,n ] )
A(n, n) = -A(n, n)  ! Make sure A is not singular
```

```
! Put some values in the vector b
b = [20, 26, -4]

! Solve the linear equations
call solve_linear_eqns(A, x, b, indicator)

print *
print "(a, 3f8.4)", "The solution to Ax = b is", x
print *, "The indicator is", indicator

b = [1, 2, 3]

! Solve the linear equations
call solve_linear_eqns(A, x, b, indicator)

print *
print "(a, 3f8.4)", "The solution to Ax = b is", x
print *, "The indicator is", indicator

! Put some values in the matrix A
A(1, :) = [ 2, -1, 3 ]
A(2, :) = [ -1, 3, 2 ]
A(3, :) = [ 3, -2, -2 ]

b = [9, 11, -7]

! Solve the linear equations
call solve_linear_eqns(A, x, b, indicator)

print *
print "(a, 3f8.4)", "The solution to Ax = b is", x
print *, "The indicator is", indicator

end program slatec_linear_eqns
```

Running this program produces the following results. Each solution has approximately six significant digits. Matran (Section 10.1) also includes a linear equation solver.

```
The solution to Ax = b is  1.0000  2.0000  3.0000
 The indicator is          -6

The solution to Ax = b is  2.0000 -1.0000  0.0000
 The indicator is          -6

The solution to Ax = b is  1.0000  2.0000  3.0000
 The indicator is          -6
```

9 Defined Data Types

There are several modules available to the Fortran programmer that define new data types and a selection of operations on those types. Code for interval arithmetic, varying strings, big integers, rationals, quaternions, and Roman numerals are available; the source for each of these modules is available in the `src` directory to provide information about the modules and examples of how to build these abstract data types.

When using these features and compiling from the command line the compiler must use the option `%FT_INC%` and the linker must include `%FT_LIB%` or compile and link in one step using `%FT%`.

```
gfortran test_roman.f90 %FT%
```

These are included for you when using Code::Blocks.

9.1 Interval Arithmetic

Using interval arithmetic, each "value" is really an interval of values. For example, if it is known that the value of `a` is between 1.3 and 1.5, it is represented as the interval [1.3, 1.5] between 1.3 and 1.5. If `b` is the interval [2.6. 2.9] then the sum `a + b` is in the interval [3.9, 4.4] and this is the sum of `a` and `b` in interval arithmetic. This is illustrated by the following simple program that uses interval arithmetic.

```
program add_intervals

   use interval_arithmetic
   implicit none
   type(interval) :: a, b, s
   character(len=*), parameter :: &
      interval_fmt = "(' [', f0.2, ', ', f0.2, ']')"

   a = interval(1.3, 1.5)
   b = interval(2.6, 2.9)
   s = a + b

   write (unit=*, fmt=interval_fmt) s

end program add_intervals
```

Running this program produces the output:

```
 [3.90, 4.40]
```

The module `interval_arithmetic` contains extensions for many of the usual arithmetic operations and functions as well as some special ones for intervals. The document *Interval_Arithmetic* in the `doc` folder of the Fortran Tools describes the features of the interval arithmetic module in more detail.

9.2 Varying Length Strings

The ISO varying length string module provides the type `iso_varying_string` with the operations you would expect to have for character string manipulations (concatenation, input/output, character intrinsic functions). Unlike Fortran 95 character variables, a varying string variable has a length that changes as different values are assigned to the variable. Here is a simple program illustrating these features.

```
program string
   use iso_varying_string
   type(varying_string) :: s
   call get(string=s)
   s = s // s
   call put(string=s)
   print *, len(s)
end program string
```

The following lines show what happens when the program is compiled and run.

```
> gfortran string.f90 %FT%
> a.exe
A nice string.
A nice string.A nice string. 28
```

The current version of the source code is from Rich Townsend and is in the source code directory.

9.3 Big Integers

The `big_integer` data type can represent very large nonnegative integers. The representation of a big integer is a structure with one component that is an array of ordinary Fortran integers. In this version, the largest integer that can be represented is fixed, but the size is specified by a parameter that can be changed. The module may then be recompiled. The source for this module is in the `examples` directory of the distribution. All of the intrinsic operations and functions for intrinsic Fortran integers are available for big integers.

```
program factors
   use big_integer_module
   type(big_integer) :: b, n, s

   b = “9876543456789”

   n = 2
   call check_factor()
   s = sqrt(b)
   n = 3
   do
      if (n > s) exit
      call check_factor()
      n = n + 2
   end do
   if (b /= 1) then
      call print_big(b)
```

```
      print *
   end if

contains

   subroutine check_factor()
      do
         if (modulo(b, n) == 0) then
            call print_big(n)
            print *
            b = b / n
            s = sqrt(b)
         else
            exit
         end if
      end do
   end subroutine check_factor

end program factors
```

Running the program produces

```
3
3
3
3
17
97
1697
43573
```

9.4 High Precision Reals

9.4.1 The MP Module

This module provides the capability of computing with large precision real values. It was written by David Bailey of Lawrence Berkeley National Laboratory. A description of the module is in the files `mp.ps` and `mp.pdf` in the `doc` directory. Here is a simple example of its use.

```
program mp
   use mp_module
   type(mp_real) :: pi

   call mpinit()
   pi = 4.0 * atan(mpreal(1.0))
   call mpwrite(6, pi)
end program mp
```

The result printed consists of quite a few digits of π.

```
10 ^      0 x  3.14159265358979323846264338327950288419716939937510582097,
```

9.4.2 The XP Module

This module also provides the capability of computing with large precision real values. It was written by David Smith. A description of the module is in the file `xp.txt` in the `doc` directory. Here is a simple example of its use.

```
program test_xp
   use xp_real_module
   type (xp_real) :: x, y
   x = 1.0
   y = 4.0
   call xp_print(y*atan(x))
end program test_xp
```

```
 3.14159265358979323846264338327950288419 7E+0
```

9.5 Rationals

A module to compute with rational numbers is provided by Dan Nagle. Some details are provided in the file `rationals.txt` in the `doc` directory. Here is a simple example.

```
program test_rationals
   use rationals_module
   type(rational) :: r1, r2

   r1 = [3, 4]
   r2 = [5, 6]
   r1 = r1 + r2
   print *, real(r1)
end program test_rationals
```

```
   1.5833333333333333
```

9.6 Quaternions

The quaternions module was written by David Arnold of the College of the Redwoods. The only documentation is the source file `quaternions_module.f95` in the `src` directory. There is some information about quaternions in the file `quaternions.pdf` in the `doc` directory and the original article about quaternions presented by William Hamilton in 1843 can be found at `http://www.maths.tcd.ie/pub/HistMath/People/Hamilton/Quatern2/Quatern2.html`. Here is an example.

```
program Quaternions
  use Quaternions_module
  type(quaternion) :: u, v
  u=quaternion(1,2,3,4)
  v=quaternion(5,6,7,8)
  call quaternion_print(u+v)
  print *, 3+4
  print *
  call quaternion_print(u-v)
  print *, 3-4
  print *
  call quaternion_print(3.0*u)
```

```
  call quaternion_print(u*v)
  print *, 3*4
  print *
  call quaternion_print(conjg(u))
  print *, conjg((3,4))
  print *
  print *, (abs(u))
  print *, abs((3,4))
end program Quaternions
```

```
(    6.000000    8.000000   10.000000   12.000000)
 7

(   -4.000000   -4.000000   -4.000000   -4.000000)
 -1

(    3.000000    6.000000    9.000000   12.000000)
(  -60.000000   12.000000   38.000000   24.000000)
 12

(    1.000000   -2.000000   -3.000000   -4.000000)
 (3.0000000,-4.0000000)

   5.4772258
   5.0000000
```

9.7 Roman Numerals

This module to compute with Roman numerals was written by Jeanne Martin, former convenor of the international Fortran standards committee and an author of *The Fortran 2003 Handbook*. The only documentation available is in the source file in the `src` directory.

```
program test_roman
use roman_numerals_module
implicit none

type(roman) :: r
integer :: i

write (unit=*, fmt="(a)") "Integer  Roman Number"
do i = 1900, 2000
  r = i
  write (unit=*, fmt="(/, tr4, i4, tr2)", advance = "NO") i
  call print_roman (r)
end do
write (unit=*, fmt="(/)")

end program test_roman
```

Here is the result of running the program.

```
 Integer  Roman Number

    1900  MCM
    1901  MCMI
    1902  MCMII
```

```
1903  MCMIII
1904  MCMIV
1905  MCMV
1906  MCMVI
 . . .
1998  MCMXCVIII
1999  MCMXCIX
2000  MM
```

10 Matrix Operations

Fortran has extensive built-in operations on arrays, which may be used to do matrix manipulations when the matrices are represented as ordinary Fortran arrays. For example, two matrices may be added by writing

```
A + B
```

and their matrix product may be formed as

```
matmul(A, B)
```

because `matmul` is a standard Fortran intrinsic function.

However, more complicated operations require sophisticated programs to do the calculations effectively and efficiently. Fortunately, a lot of work has been done in this area and the results are included in the Fortran Tools. The BLAS and LAPACK libraries (10.2) have been used widely for years; in addition, MATRAN (may-tran) provides a higher level interface to these routines.

When using these features and compiling from the command line, the linker must include `%LAPACK%` or compile and link in one step

```
gfortran lapack_linear_equations.f90 %LAPACK%
```

If a Matran routine is used, the program must be compiled with `%FT_INC%` and linked with `%FT_LIB%` and `%LAPACK%` or compile in one step.

```
gfortran matran_linear_equations.f90 %FT% %LAPACK%
```

`%FT%` is included for you when using Code::Blocks; you must add `%LAPACK%` under *Other linker options* under the compiler settings.

10.1 MATRAN

MATRAN is a collections of modules containing procedures that may be used to perform a variety of matrix operations, such as solving linear equations and computing eigenvalues. These procedures call BLAS and LAPACK routines.

All computations are performed with double precision values.

MATRAN was developed by G. W. (Pete) Stewart, Department of Computer Science, Institute for Advanced Computer Studies, University of Maryland. The web site is:

```
http://www.cs.umd.edu/~stewart/
```

A few of the features of MATRAN are described here. More complete documentation may be found in *MatranWriteup* in PDF format in the `doc` directory of the Fortran Tools distribution.

10.1.1 The Rmat and Rdiag Derived Types

Most of the matrix computations in MATRAN are performed on objects of type `Rmat` (real matrix) and `Rdiag` (real diagonal matrix). These are derived types provided with MATRAN. For example, when solving a system of linear equations, objects of type `Rmat` are passed to the solver, not plain Fortran arrays. Here is a partial description of the `Rmat` type:

```
type :: Rmat
   real(wp), pointer :: a(:,:) => null()
   integer :: nrow = 0, ncol = 0
      . . .
end type Rmat
```

The first component `a` is a real array pointer. `wp` is the kind of the working precision, which is double precision for the matrix libraries provided with the Fortran Tools. The component `a` is default initialized to null, which means that it will be initialized to the null pointer for each `Rmat` object created. `nrow` and `ncol` are the number of rows and columns of the matrix, respectively.

For example if `X` is declared to be type `Rmat`

```
X % ncol
```

is the number of columns in matrix `X` and

```
X % a(nrow, :)
```

is the last row of the matrix. Thus, `Rmat` objects may be manipulated directly (their components are not private) as well as with the procedures provided by MATRAN.

The type `Rdiag` (real diagonal matrix) represents a diagonal matrix as a one-dimensional array, consisting of the diagonal elements, and other components, such as the size of the matrix.

10.1.2 Example: Linear Equations

Let us look at some of the MATRAN operations used to construct a program to solve a set of linear equations.

Here is the program; it solves $Ax = b$.

```
program matran_linear_equations

use matran_module

integer, parameter :: n = 3
integer :: i, j

type(Rmat) :: A, b, x

real(wp), dimension(n,n) :: AA
real(wp), dimension(n,1) :: bb

! Put some values in the matrix A
AA = reshape( [ ((real(i+j),i=1,n),j=1,n) ], shape = [ n,n ] )
```

```
A = AA
A%a(n,n) = -A%a(n,n) ! Make sure A is not singular

! Put some values in the vector b
bb = reshape( [(real(i), i = 1, n)], [n,1])
b = bb
b = A * b

! Solve the linear equations
x = A .xiy. b

call Print(A, 15, n, "Array A", e=1)
call Print(b, 15, n, "Vector B", e=1)
print *
print *
print "(a, 3f7.4)", "The solution to Ax = B is", x%a(1:n, 1)
print *

call Clean(A)
call Clean(b)
call Clean(x)

end program matran_linear_equations
```

The program could be run using Code::Blocks or from the command line.

`A`, `b`, and `x` are type `Rmat` objects. All three are, in effect, representations of matrices, even though in this program we think of `b` and `x` as vectors to hold the constants of the equations and the solution to the equations, respectively. Remember that Fortran is case insensitive, but case is used in the program in the traditional way: uppercase for matrices and lowercase for vectors.

The `use` statement in the subroutine accesses a module containing many of the MATRAN features.

`A` and `b` are given values using an extended assignment statement. In the statement

```
b = A * b
```

the operation is matrix multiplication, provided by MATRAN. This assigns values to `b` that will produce a solution we will recognize.

The statement

```
x = A .xiy. b
```

assigns the solution to `x` by computing $A^{-1}b$ using the operator `.xiy.`, which is intended to suggest "*x* inverse times *y*". (Of course, the inverse of `A` is not actually calculated in order to solve the equations.)

The MATRAN subroutine `Print` is used to verify the values of `A` and `b` used for the equations.

The subroutine `Clean` releases allocatable storage. Note that there are other MATRAN subroutines that deal with the deallocation of dummy arguments of type `Rmat`.

Here is the output from the program.

```
  Array A
```

```
 3 3 3 3 GE T 0
                  1              2              3
 1        2.000E+0       3.000E+0       4.000E+0
                  1              2              3
 2        3.000E+0       4.000E+0       5.000E+0
                  1              2              3
 3        4.000E+0       5.000E+0      -6.000E+0

 Vector B
 3 1 3 1 GE T 0
                  1
 1        2.000E+1
                  1
 2        2.600E+1
                  1
 3       -4.000E+0


 The solution to Ax = B is 1.0000 2.0000 3.0000
```

Slatec (Section 8.6) also includes a linear equation solver.

10.1.3 Example: Eigenvalues

For this example, we assume that the main program uses ordinary Fortran arrays to store matrices. To compute the eigenvalues of a matrix, we want to call the subroutine `eigen-values`, passing such an array. This subroutine will use MATRAN objects and the routine `Eig` to compute the eigenvalues.

The program could be run using Code::Blocks or from the command line.

Here is the program.

```
program matran_eigenvalues

integer, parameter :: n = 5
real, dimension(n,n) :: A
complex, dimension(n) :: e
integer :: i, j

! Put some values in the matrix A
call random_seed()
call random_number(A)
A = 10.0 * A - 5.0

call eigen_values(A, e)

print *
print *, "The eigenvalues of A are:"
print *
do i = 1, n
   print "(i3, 2(f7.2, a))", i, &
         real(e(i)), " +", aimag(e(i)), "i"
end do
print *
print *
```

```
contains

subroutine eigen_values (D, e)

   use matran_module

   real, dimension(:, :), intent(in) :: D
   complex, dimension(:), intent(out) :: e
   type(Rmat) :: RD
   type(RmatEig) :: eigD

   RD = D
   call Print(RD, 9, 2, "Array D", e = 1)

   call Eig(eigD, RD)

   ! D is the eigenvalue component of eigD
   e = eigD % D

   call Clean(RD)
   call Clean(eigD)

end subroutine eigen_values

end program matran_eigenvalues
```

The Fortran intrinsic subroutines `random_seed` and `random_number` are used to put some values in the array `A`. The statement

```
A = 10.0 * A - 5.0
```

causes the numbers to be between –5 and +5 instead of between 0 and 1. The array `A` is passed to the subroutine `Eig` with a vector `e` to hold the eigenvalues. The dummy argument `D` is assigned to the variable `RD` of type `Rmat`. The array is printed.

The call to subroutine `Eig` computes the eigenvalues of `RD` and stores the results in `eigD`, a MATRAN object of type `RmatEig`, defined in the module `matran_module`. The component `D` (not the same as the dummy array `D`) contains the eigenvalues and this is assigned to the dummy array `e`, which is returned as the value of the actual argument `e`. The eigenvalues stored in `e` are then printed.

Here is the result of one execution of the program.

```
 Array D
 5 5 5 5 GE T 0
           1        2        3        4        5
 1  -3.54E+0 -1.88E+0  3.01E+0  2.51E+0  3.33E+0
           1        2        3        4        5
 2   4.19E+0  2.04E+0 -4.80E+0 -3.19E+0  1.02E+0
           1        2        3        4        5
 3  -3.99E+0  7.37E-1 -3.98E+0 -1.34E+0 -7.92E-1
           1        2        3        4        5
 4  -3.44E+0  3.74E+0  1.49E-1 -3.59E+0 -1.79E+0
           1        2        3        4        5
 5  -4.16E+0 -1.70E-1  1.48E+0  4.89E+0  4.46E+0

 The eigenvalues of A are:
```

```
1  -1.31 +   7.13i
2  -1.31 +  -7.13i
3  -3.02 +   1.91i
4  -3.02 +  -1.91i
5   4.05 +   0.00i
```

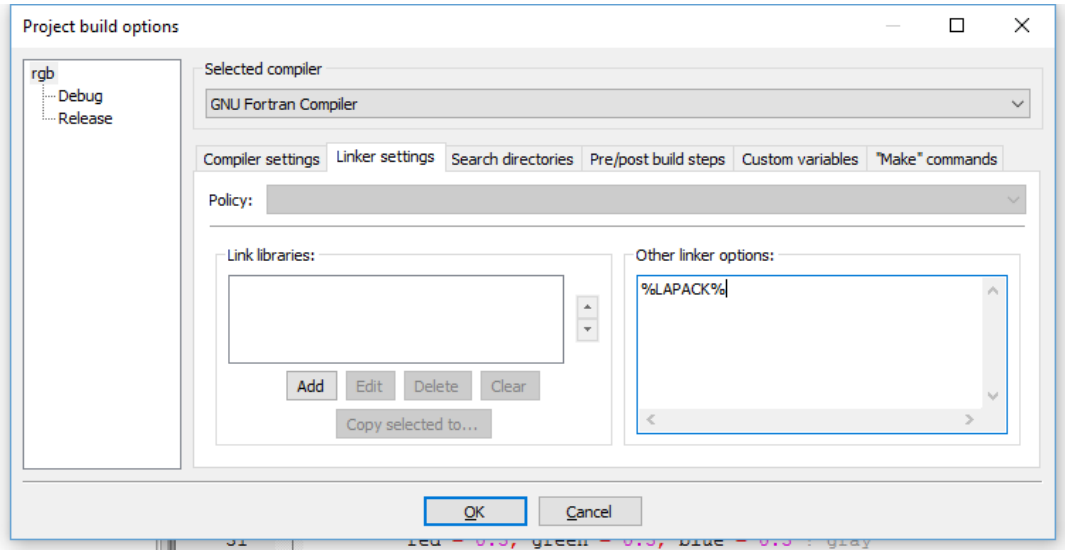
10.2 BLAS and LAPACK Libraries

It is possible to invoke any of the BLAS or LAPACK procedures directly in a Code::Blocks project or from the command line. Four versions of the routines are available: single, double, single complex, and double complex. All of the routines are described in the LAPACK manual in the `doc` folder of the Fortran Tools distribution.

To build (actually link) a program that uses LAPACK, add the environment variable `%LAPACK%`, whose value is

```
-L%FORTRAN_TOOLS%\lib -llapack -lblas
```

With Code::Blocks, add `%LAPACK%` to the Project build options Linker settings, Other linker options.



Here is an example that solves three simultaneous linear equations by calling the LAPACK routine `sgesv`. The program `lapack_linear_equations` is in the `examples` folder.

```
C:\Fortran_Tools\examples>type lapack_linear_equations.f90
program lapack_linear_equations

! Solving the matrix equation A*x=b using LAPACK

   implicit none
   real, dimension(3, 3) :: A
   real, dimension(3) :: b
   integer :: i, j, ok
```

```
    integer, dimension(3) :: pivot

    A = reshape ( [ &

         3.1, 1.3, -5.7, &
         1.0, -6.9, 5.8, &
         3.4, 7.2, -8.8  &

               ], shape(A), &
               order = [2, 1])

    b = [ -1.3, -0.1, 1.8 ]


    call SGESV(3, 1, A, 3, pivot, b, 3, ok)

    ! Parameters in the order as they appear in the subroutine call
    !    order of matrix A, number of right hand sides (b), matrix A,
    !    leading dimension of A, array that records pivoting,
    !    result vector b on entry, x on exit, leading dimension of b
    !    return value


    ! Print the result, stored in b
    print "(a, 3f9.3)", "Solution: ", b
    print *, "The process indicator is", ok

end program lapack_linear_equations

C:\Fortran_Tools\examples>gfortran lapack_linear_equations.f90 %LAPACK%

C:\Fortran_Tools\examples>a
Solution:     1.000    1.000    1.000
 The process indicator is           0
```

11 Plotting

The Fortran Tools contain two plotting packages: Gnuplot and Plplot. To generate a plot with Gnuplot, commands can be entered from a command line or a file can be created by a Fortran program, which may then display the plot. A Plplot may be displayed directly on the screen by a Fortran program, but also may be saved for later use.

11.1 Plplot

Plplot consists of procedures callable from Fortran that can produce plots, graphs, charts, etc.

A plot may be viewed on the computer screen and may be incorporated into a Word document as a screen capture. Also several other formats, such as PDF, PNG, and SVG may be generated.

11.1.1 Compiling a Plplot Program

The following line illustrates how to compile and run a Plplot program from the command line.

```
gfortran -c %FT_INC% ft_x00f.f90
gfortran -o ft_x00f.exe ft_x00f.o %PLOT%
ft_x00f.exe
```

To run a Plplot program using Code::Blocks, add `%PLOT%` into the window viewed by *Project → Build options → Linker settings → Other linker options*.

11.1.2 A Simple Example

The basic procedures used to build a plot are used in the example in this section; it is in the file `ft_x00f.f90` in the `examples\plplot` folder. Here is the program.

```
program ft_x00f

! This is a modified version of x00f.f90
! which was written by Alan Irwin

   use plplot
   implicit none

   integer, parameter :: NSIZE = 100

   real(kind=plflt), dimension(0:NSIZE) :: x, y

   real(kind=plflt) :: xmin =   0.0_plflt, &
                       xmax =   1.0_plflt, &
                       ymin =   0.0_plflt, &
                       ymax = 100.0_plflt
   integer :: i, parse_result

   ! Prepare data to be plotted.
```

```
    ! x = .00, .01, .02, ..., .99, 1.00
    x = [(i, i=0,NSIZE)] / real(NSIZE)
    y = ymax * x**2

    ! Parse and process command line arguments
    parse_result = plparseopts( PL_PARSE_FULL )
    if (parse_result /= 0) stop “Error parsing options”

    ! Initialize plplot
    call plinit( )

    ! Create a labelled box to hold the plot.
    call plenv( xmin, xmax, ymin, ymax, just=0, axis=0 )
    call pllab( "x", "y=100 x#u2#d", &
                "Simple PLplot demo of a 2D line plot" )

    ! Plot the data that was prepared above.
    call plline( x, y )

    ! Close PLplot library
    call plend( )

end program ft_x00f
```

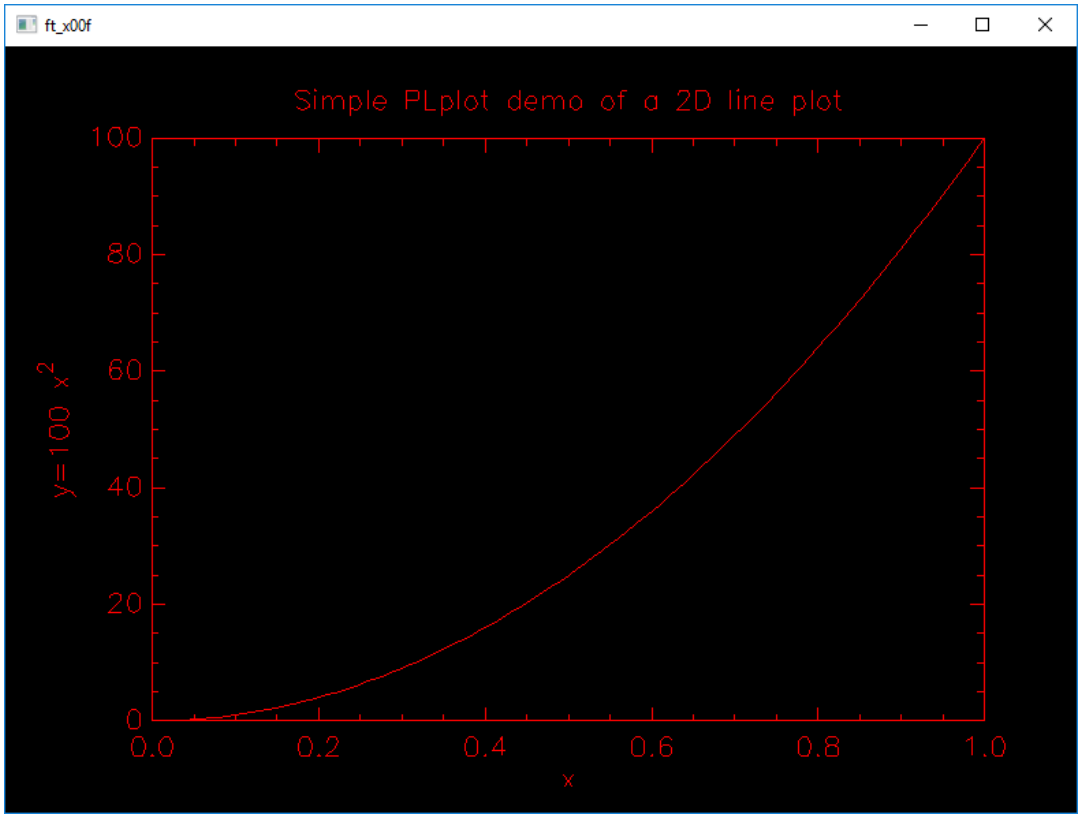
This program can be compiled with the following command.

```
gfortran ft_x00f.f90 %PLOT% -o ft_x00f
```

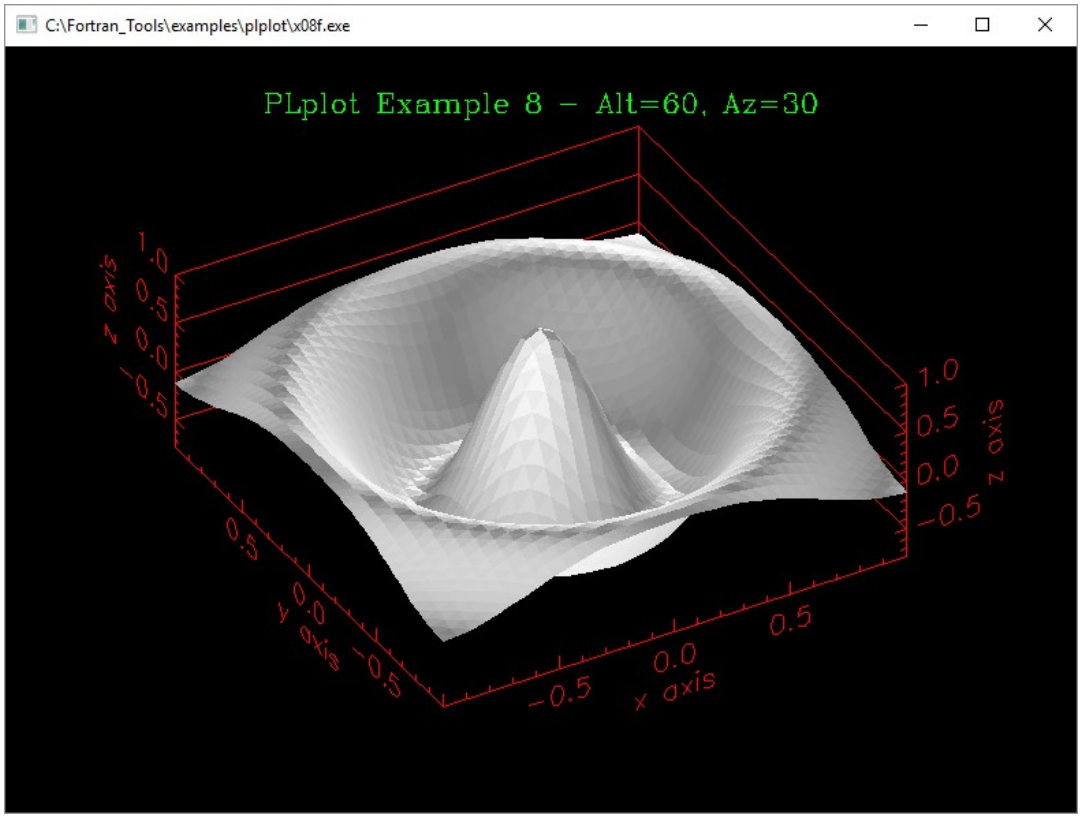
Executing the program produces the following in a console output window.

```
Plotting Options:
 < 1> wingcc     Win32 (GCC)
 < 2> ps         PostScript File (monochrome)
 < 3> psc        PostScript File (color)
 < 4> xfig       Fig file
 < 5> null       Null device
 < 6> ntk        New tk driver
 < 7> mem        User-supplied memory device
 < 8> svg        Scalable Vector Graphics (SVG 1.1)
 < 9> pdfcairo   Cairo PDF Driver
 <10> pscairo    Cairo PS Driver
 <11> epscairo   Cairo EPS Driver
 <12> svgcairo   Cairo SVG Driver
 <13> pngcairo   Cairo PNG Driver
 <14> memcairo   Cairo Memory Driver
 <15> extcairo   Cairo External Context Driver
 <16> wincairo   Cairo Microscoft Windows Driver

Enter device number or keyword: 1
```

Entering 1 for the device number causes the following plot to be displayed.



Here is another example produced by executing the program `x08f.f90` in the plplot examples folder.



11.1.3 Real Kinds for Plotting Values

The Plplot Fortran procedures that are called are basically "wrappers" that call Plplot routines written in C. Thus, it is important that values that are passed match the appropriate C values for the C routines called. This is achieved by using the kind parameter `PLFLT` to declare real variables that are used as actual arguments when calling the plotting routines. This value is defined in the module `plplot`.

11.1.4 Plotting Procedures

Some of the basic plotting procedures used in this program are described here. All of the plotting routines are described in `plplot.pdf` in the `doc` folder of the Fortran Tools distribution. Most of those descriptions are in terms of the C programs, but the Fortran versions can be understood fairly easily from them.

Before any plotting routines are called, the x and y values of 101 points are stored to display the graph of $y = 100x^2$ for values of x between 0 and 1.

```
x = [(i, i=0,NSIZE)] / real(NSIZE)
y = ymax * x**2
```

Next, the subroutine `plparseopts` is called to establish the values set by any execution options (Section Error: Reference source not found).

The subroutine `plinit` always must be called, but after `plparseopts`.

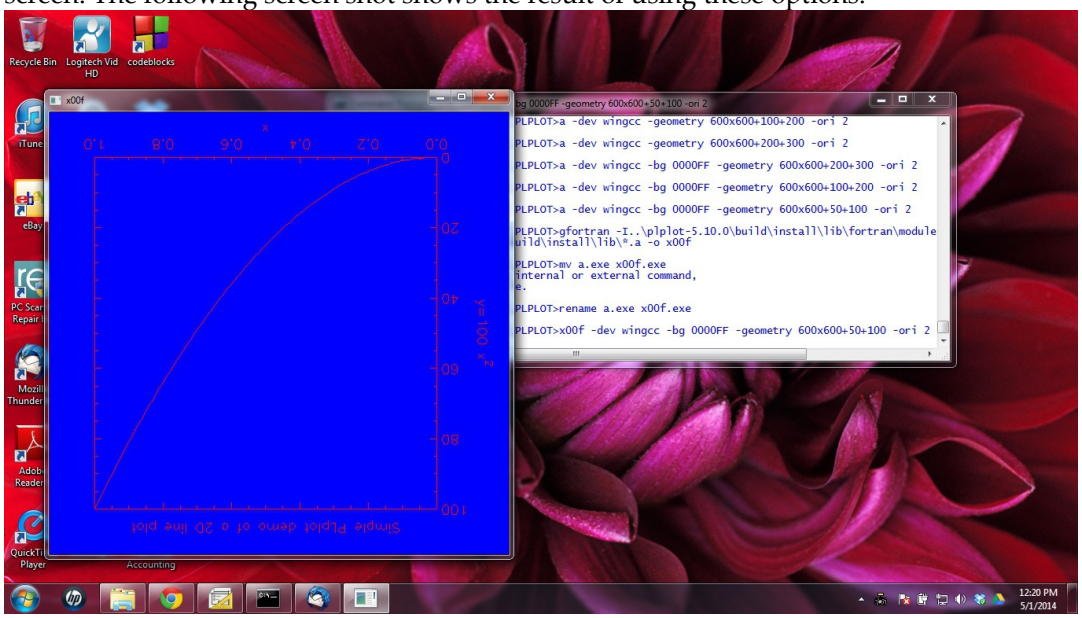
The subroutine `plenv` establishes the basic parameters of the plot. The first four arguments set the lower and upper limits of the horizontal and vertical axes. The `just` argument indicates something about the scales of the axes, and the `axis` argument provides several options related to drawing and labeling the axes. `axis=0` specifies drawing a box, labeled with coordinate values around edge. These are described in the Plplot manual.

11.1.5 Execution Options

Several things about the plot can be changed by supplying options when the program is executed. One option is `-h`, which displays all the options. A few other options are

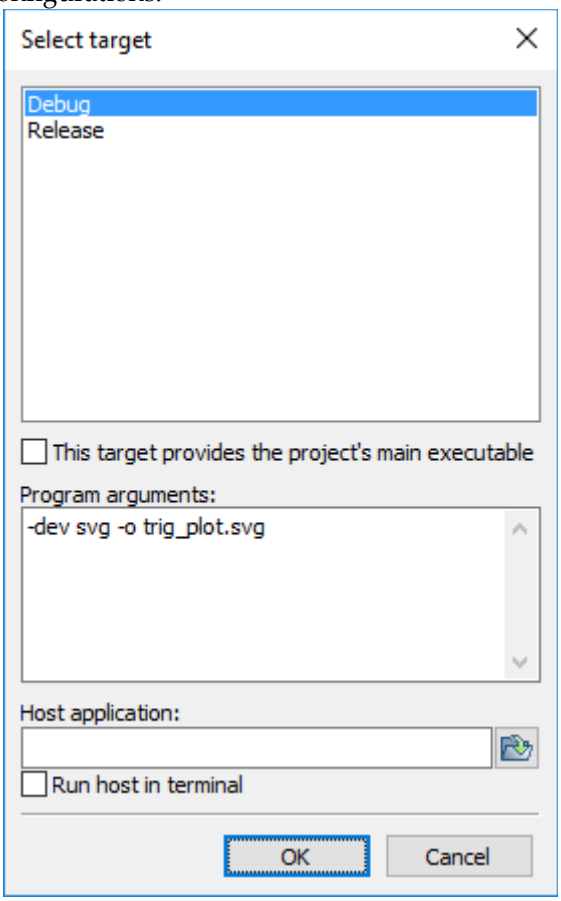
`-dev`	ouput device name
`-o`	output file name
`-bg`	background color RRGGBB, red, green, blue
`-ori`	orientation given as rotation
`-geometry`	size and location of plot

For example, `-dev svg` produces an SVG file, `-o ft_x00f.svg` stores the plot in the file `x00f.svg`, `-bg 0000FF` makes the background blue (black is the default), `-ori 2` rotates the plot 180 degrees (1 = 90 and 3=270), and `-geometry 600x600+200+300` plots using 600x600 pixels, shifted 200 pixels to the right and 300 pixels down from the upper left corner of the screen. The following screen shot shows the result of using these options.



11.1.6 Execution Options in Code::Blocks

To set execution options when using Code::Blocks, select the *Project* tab, then select *Set programs' arguments . . .* In the *Program arguments* pane, type the options to be used as shown in the following screen capture. This will cause the display to appear on the screen without having to select that option when the program is executed. Set the options for either or both of the configurations.



11.1.7 Plotting Trigonometric Functions

The program in this section produces a plot similar to the one produced by Gnuplot that graphs three trigonometric functions. It introduces some subroutines for making more complicated labels. The program is in the file `trig_plot.f90` in the `examples\plplot` folder and is `trig_plot.cbp` in the `projects` folder.

The first few executable statements set up the plot.

```
call plparseopts(PL_PARSE_FULL)
call plsdev("wingcc")
call plspage(0.0_plflt, 0.0_plflt, &
```

```
                1000, 600, 100, 100)
   call plscolbg(255, 255, 255)  ! White background
   call plinit()
```

The `plparseopts` routine establishes any options provided when the program is executed. The `plsdev` routine sets the display device to `wingcc`, so that the option to do this is not needed. The `plspage` routine sets the size and position of the page, just the same as the `-geometry` option. The `plscolbg` routine sets the background color to white. All of these should be called before `plinit`.

The next statements set up the plot.

```
call plenv(-PI, 3*PI, -0.6_plflt, 1.0_plflt, &
           just=0, axis=1)
call date_and_time(date=date, time=time)
call pllab("Graph of trig functions", "", "Plotted "// &
   date(1:4)//"-"//date(5:6)//"-"//date(7:8)// &
   " at "//time(1:2)//":"//time(3:4))
```

`plenv` establishes a plotting area with x values ranging from $-\pi$ to $+3\pi$ (`PL_PI` is a parameter defined in the `plplot` module renamed `PI` in this program) and y values ranging from –0.6 to 1. `axis=1` causes coordinate axes to be drawn and labeled. Calling `pllab` puts the first label at the bottom of the graph and the third label, which contains the date and time, at the top of the graph.

The statement

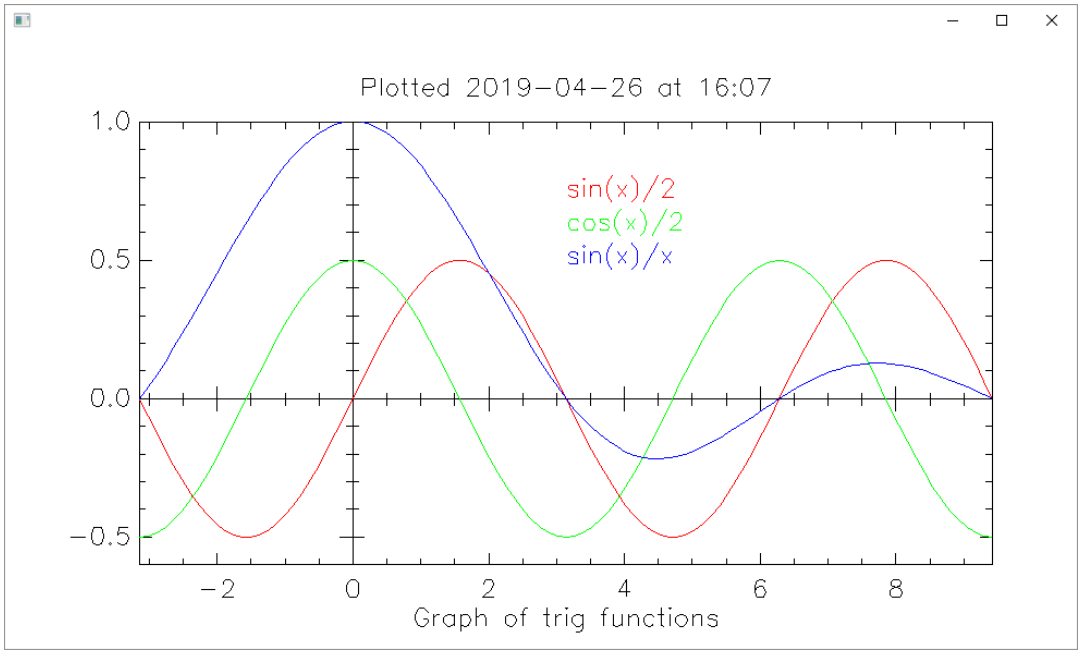
```
x = -PI + 4 * PI * [(i, i=0,nr_of_points)] / nr_of_points
```

sets an array of 100 x values to equally spaced values between $-\pi$ to $+3\pi$ and the statements

```
y = sin(x) / 2
call plcol0(RED)s
call plline(x, y)
call plmtex(top, -4.0_plflt, date_left, 0.0_plflt, &
      text="sin(x)/2")
```

set each y value to $sin(x)/2$ for each x value, set the current color to red (`RED` is a parameter with the appropriate value for color map 0), draw a line through all of the points (x, y), and put the text “sin(x)/2” four character heights down from the top of the plotting area. There are similar statements for the other two functions.

Here is the output.



11.1.8 Plot of Heat Transfer

The program discussed in this section produces a plot showing the same results as obtained by the heat transfer program plotted previously (Section 11.2.3) by Gnuplot. The program is in the file `examples\plplot\heat_plot.f90`. The computational portion of the program is the same as the program that produced the Gnuplot, so we will concentrate on some of the plotting procedures that are used.

This plot is a variation of a contour plot; the space between the contours is filled with a color reflecting the temperature in that region of the plate.

```
call plspal1("cmap1_blue_red.pal", 1)

contours = [(i, i=0, nr_contours)] / real(nr_contours)
```

The call to `plspal1` in the subroutine `initialize_plot` sets color map 1 to a spectrum of colors from blue to green to yellow to red. The next statement assigns values from 0 to 1 to the array `contours`; `nr_contours` is a parameter set in the program (it is 100 in the example). This means that if a value of one of the elements of the plate is, for example, 0.26, that region of the plot is filled with the color that is in the color palette between contour values 0.2 and 0.3 (a light shade of blue, in this case—see the color bar in the plot below). It is special in this case that the values of the plate and the color bar are the same.

Here are the first few statements of the subroutine `draw_plot`.

```
call initialize_plot()
```

```
call plsdev("wingcc")
call plspage(0.0_plflt, 0.0_plflt, &
             600, 600, 100, 100)
call plscolbg(255, 255, 255)  ! White background
call plinit()
call pladv(0)
call plvpor(0.05_plflt, 0.85_plflt, 0.1_plflt, 0.9_plflt)
call plwind(0.0_plflt, 1.0_plflt, 0.0_plflt, 1.0_plflt)

call plshades(transpose(real(plate,kind=plflt)), defined, &
              0.0_plflt, 1.0_plflt, 1.0_plflt, 0.0_plflt, &
              contours, &
              fill_width, cont_color, cont_width)
```

`plsdev`, `plspage`, and `plscolbg` set the device, position, and background color as in the trigonometric plot (11.2.2).

After `plinit` is called, `plenv` is not called as in previous plots, but `pladv`, `plvpor` and `plwind` are called instead because, although `plenv` provides the same basic functionality as these three procedures, it does not have the flexibility required for the heat plot. The routine `plvpor` creates a *viewing port* within the plot; the four arguments indicate the margin within the plot around the viewing port. The routine `plwind` creates a window within the viewing port; the four arguments provide coordinates to be referenced when drawing a plot within the window.

The subroutine `plshades` fills in the appropriate colors as indicated by the values of the `plate` and `contours` arrays. The transpose of the plate array is used because C stores arrays in row order and Fortran stores arrays in column order. The y minimum and maximum values are reversed from what might be expected because otherwise the value in the first row of the plate are plotted at the bottom, rather than the top.

Next, a color bar is drawn to the right of the plate plot. Contours for the color bar are set up as values of the array `bar_values` when `initialize_plot` is called. There are many additional argument not shown that indicate how the color bar is displayed.

```
call plschr(0.0_plflt, 0.65_plflt)
call plcolorbar(bar_width, bar_height, &
   . . .
   num_bar_values, bar_values)
```

Finally, a label is placed near the bottom of the plot.

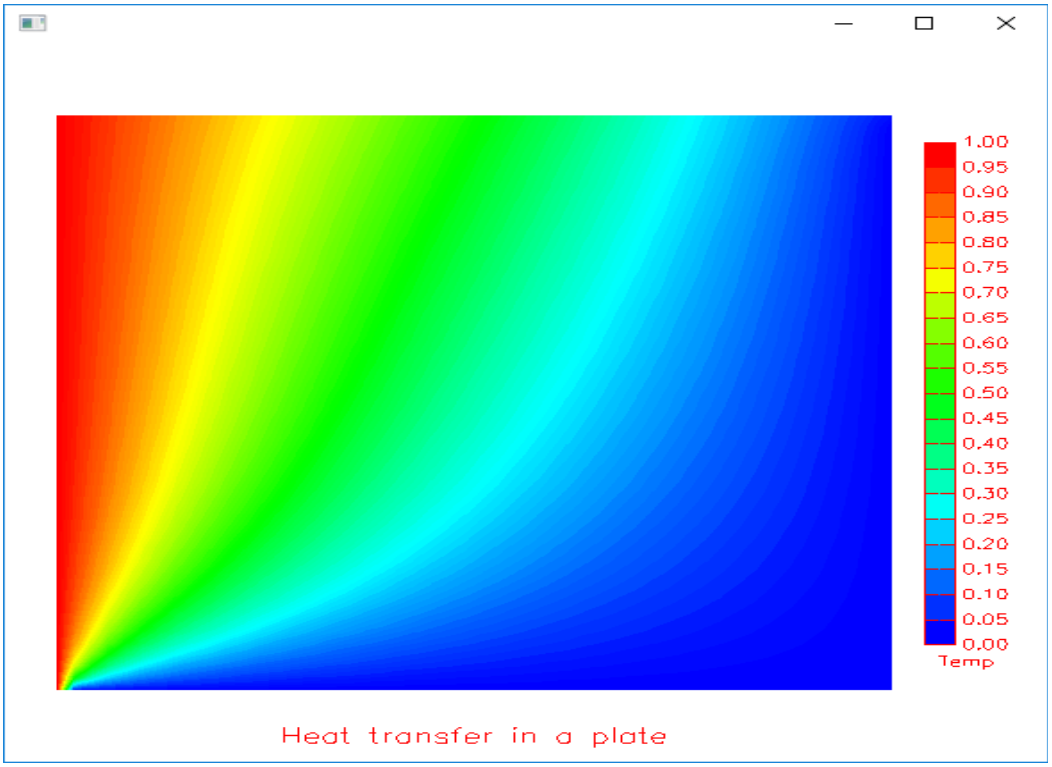
```
call pllab("Heat transfer in a plate", "", "")
```

One of the trickier aspects of this program is displaying several plots in succession as the heat spreads across the plate. Normally when a `plend` and `plinit` are executed after drawing a plot, the user must type something (`Enter`, for example) for the next plot to be displayed. This is avoided by including the calls to `plpause`. The call to `sleep` (nonstandard Fortran) allows the consecutive plots to be displayed slowly enough to be viewed.

```
! More plots
! Remove plspause to wait for CR
call plspause(.false.)
call plend()
```

```
call plspause(.true.)
call sleep(pause_time)
```

Here is the last plot produced after the computation has converged.



11.2 Gnuplot

One way to use Gnuplot is to type the commands into the `gnuplot` program. Select the file `wgnuplot.exe` for execution (`wgnuplot` is the Windows version of `gnuplot`).

In the window that appears, enter Gnuplot commands, such as

```
plot sin(x)
exit
```

11.2.1 Running Gnuplot with a Fortran Program

As is the case for all Fortran programs, you can either use the command line or Code::Blocks. To run Gnuplot with a Fortran program, create a new file called, for example, `trig_plot.gp` (which is in the `examples` folder of the Fortran Tools distribution) containing the following Gnuplot commands.

```
# Plot some trig functions with Gnuplot
```

```
# Draw the axes
set xzeroaxis lt 22
set yzeroaxis lt 22
set xlabel "X"
set ylabel "Y"

# Establish the time stamp
set time "Plotted 20%y-%m-%d at %H:%M" \
   offset 40,21 font "Helvetica,10"

# Provide identifying label
set label "Graph of trig functions" at 4,0.6 \
   font "Helvetica,12"

# Set plot type and output file
set terminal pdf
set output "trig_plot.pdf"

# Create the plot
plot [-pi:3*pi] cos(x)/2 lt 2 lw 2, \
                sin(x)/2 lt 1 lw 2, \
                sin(x)/x lt 3 lw 2
```

Create a Fortran program that runs `wgnuplot` as follows:

```
program trig_plot

   call execute_command_line ("wgnuplot trig_plot.gp")

end program trig_plot
```

The plot is placed in the output file `trig_plot.pdf`, which may be viewed with a PDF reader in the project workspace.

11.2.2 Some Gnuplot Commands

We can use the file named `trig_plot.gp` to briefly examine some of the common Gnuplot commands. Complete descriptions of all the commands may be found in `Gnuplot.pdf` in the `doc` directory of the Fortran Tools distribution.

```
set xzeroaxis lt 22
```

indicates that the x axis is to be drawn on the plot with line type (`lt`) 22. To see the line types and point types, execute `wgnuplot` and type *test*. The next three commands then do what is expected.

```
set time "Plotted 20%y-%m-%d at %H:%M" offset 40,21 font "Helvetica,10"
```

puts a time stamp on the graph at a position indicated by character offsets 40 (x) and 21 (y). The format is given by the first character string and the font and size by the second.

```
set label "Graph of trig functions" at 4,0.6 font "Helvetica,12"
```

places a label at coordinate position (4, 0.6) in Helvetica 12-point font.

```
set terminal pdf
set output "trig_plot.pdf"
```

indicate the format of the output file is PDF and the plot will be put in file `trig_plot.pdf`.

```
plot [-pi:3*pi] cos(x)/2 lt 2 lw 2, \
                sin(x)/2 lt 1 lw 2, \
                sin(x)/x lt 3 lw 2
```

This command generates the plot, with three curves, the label, the axes, the axis labels, and the time stamp. The notation in brackets indicates the range of x values to include in the plot. `pi` is a built-in variable. `lt` indicates the line type and `lw` means the line width is to be two times the normal size. Line types 1, 2, and 3 produce lines with color red, green, and blue, respectively.

11.2.3 Generating Data with a Fortran Program

The program `heat.f90` illustrates how a Fortran program can generate the data for a plot. This program generates data for many plots, in fact, and displays them sequentially.

```
!  A simple solution to the heat equation using
!  pointers. Results are plotted with Gnuplot.

program heat

implicit none

integer, parameter    :: nn = 20, &  ! Size of grid
                         plot_frequency = 10, &
                         pause_time = 1
real, dimension(nn, nn), target :: plate
real, dimension(nn-2, nn-2)     :: temp
real, pointer, dimension(:, :)  :: n,e,s,w, inside
real, parameter    :: tolerance = 1.0e-3
character(len=20)  :: filename = "heat_data."

real    :: diff
integer :: i,j, niter, ios

open(unit=11, file="heat.gp", status="replace",&
     action="write", iostat=ios)
if (ios > 0) then
   print *, "Couldn't open heat.gp"
   stop
end if

call execute_command_line ("del head_data.*")

write(unit=11, fmt="(a)") &
      "set terminal wxt", &
      "set pm3d", &
```

```
      "set palette", &
      "set ticslevel 0", &
      "set view 0,0"
write(unit=11, fmt="(a, i0)") "pause_seconds=", pause_time

! Set up initial conditions
plate = 0
plate(1:nn,nn) = 1.0  ! boundary values
plate(1,1:nn) = [ ( 1.0/nn*j, j = 1, nn ) ]

!  Point to parts of the plate
inside => plate(2:nn-1, 2:nn-1)
n => plate(1:nn-2, 2:nn-1)
s => plate(3:nn, 2:nn-1)
e => plate(2:nn-1, 3:nn)
w => plate(2:nn-1, 1:nn-2)

! Iterate
niter = 0
do
   niter = niter + 1
   temp = (n + e + s + w)/4.0
   diff = maxval(abs(temp-inside))
   inside = temp
   if (modulo(niter, plot_frequency) == 0) then
      call print_data()
   end if
   if (diff < tolerance) exit
end do

if (modulo(niter, plot_frequency) /= 0) then
   call print_data()
end if

close (unit=11, status="keep")

call execute_command_line("wgnuplot -persist heat.gp")

contains

subroutine print_data()
   write(unit=filename(11:), fmt="(i0)") niter
   open(file=trim(filename), &
        unit=22, status="replace", &
        action="write", iostat=ios)
   if (ios>0) then
      print *, "File open failed:", niter
      stop
   end if
   do i = 1,nn
      write (unit=22, fmt="(999f7.3)") plate(1:nn,i)
   enddo
   close(unit=22, status="keep")
   write (unit=11, fmt="(a)") &
      "splot """ // trim(filename) // &
      """ matrix pt 0", "pause pause_seconds"
end subroutine print_data
```

```
end program heat
```

This program is in the `examples` folder of the Fortran Tools distribution.

A constant heat source is placed along one edge of an `nn` × `nn` grid and a linearly diminishing heat source is placed along an adjacent edge. The other two edges are held at constant 0. The steady state condition of each point in the grid is found by iteratively replacing each value in the interior of the grid (the pointer variable `inside` is an alias of this portion of the plate) by the average of the points (`n`, `e`, `s`, `w`) around it. If the iteration is a multiple of `plot_frequency`, the values in the grid are placed in a file for plotting. `pause_seconds` indicates the number of seconds delay between plot displays. If the largest difference between an old value and a new value in the grid is less than the parameter `tolerance`, no more iterations are performed and the data for the last plot is generated, if it has not been generated already.

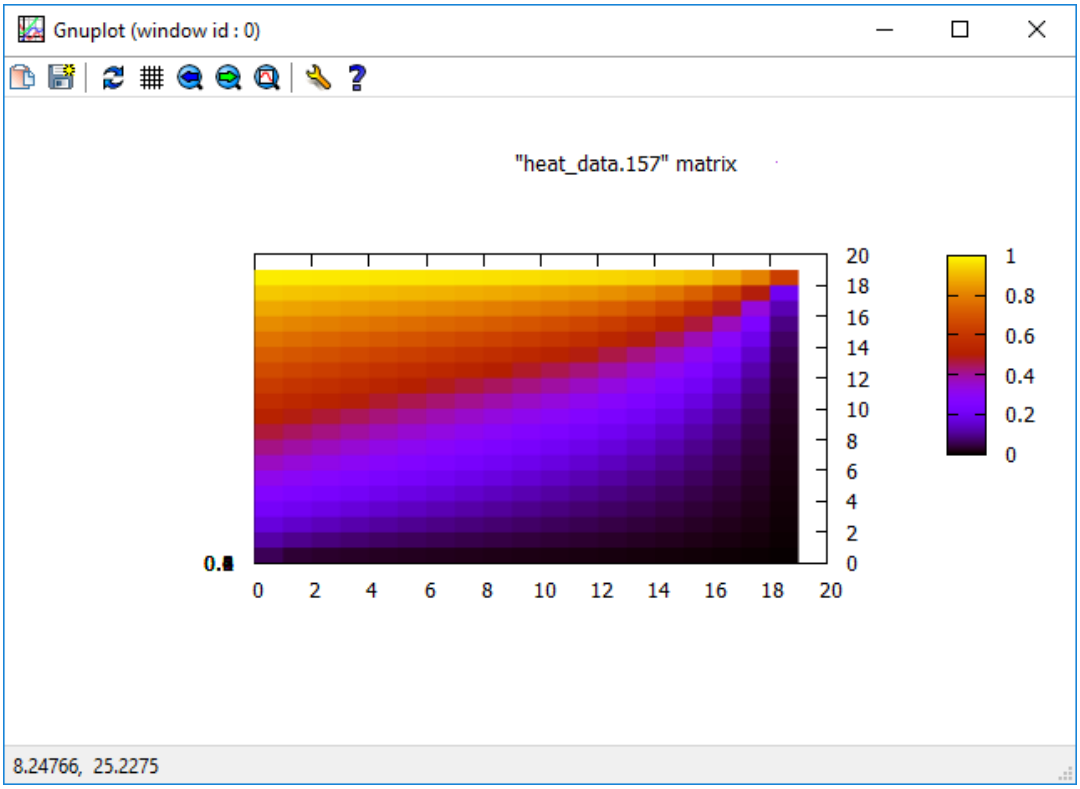
The intrinsic subroutine `execute_command_line` is called to execute the Gnuplot command file `heat.gp` generated by the program. Here is the content of the file generated in this case.

```
set terminal wxt
set pm3d
set palette
set ticslevel 0
set view 0,0
pause_seconds=1
splot "heat_data.10" matrix pt 0
pause pause_seconds
splot "heat_data.20" matrix pt 0
pause pause_seconds
   . . .
splot "heat_data.150" matrix pt 0
pause pause_seconds
splot "heat_data.157" matrix pt 0
pause pause_seconds
```

This sets the file format to `wxt` and the style to a special three-dimensional format (`pm3d`). The view is from straight above the plot (`0,0`). It draws three-dimensional (projected on the two-dimensional screen, of course) plots (`splot`) for each of the data files generated by the Fortran program. `matrix` indicates that the data in the file is in matrix (two-dimensional array) format. `pt 0` indicates that the point style is 0. Because the command that executes this file is

```
wgnuplot -persist heat.gp
```

the last plot remains displayed until closed. Here is the final plot.


Gnuplot (window id : 0)
"heat_data.157" matrix
20
18
16
14
12
10
8
6
4
2
0
0
2
4
6
8
10
12
14
16
18
20
1
0.8
0.6
0.4
0.2
0
8.24766, 25.2275

12 Libraries

Libraries are collections of Fortran procedures that may be accessed by other programs.

There are two kinds of Fortran libraries: static libraries and dynamically linked libraries. Each has its advantages.

Static libraries are linked to the Fortran program that uses them. That is, the procedures in the library are included in the executable file for the program. This has the advantage that the program may be executed on another system. The disadvantage is that all of the procedures in the library, even those that are not called by the program, are linked into the executable, so it may become unnecessarily large. The static library file suffix is `.a`. As an example, the main library provided by the Fortran Tools is `libfortrantools.a`.

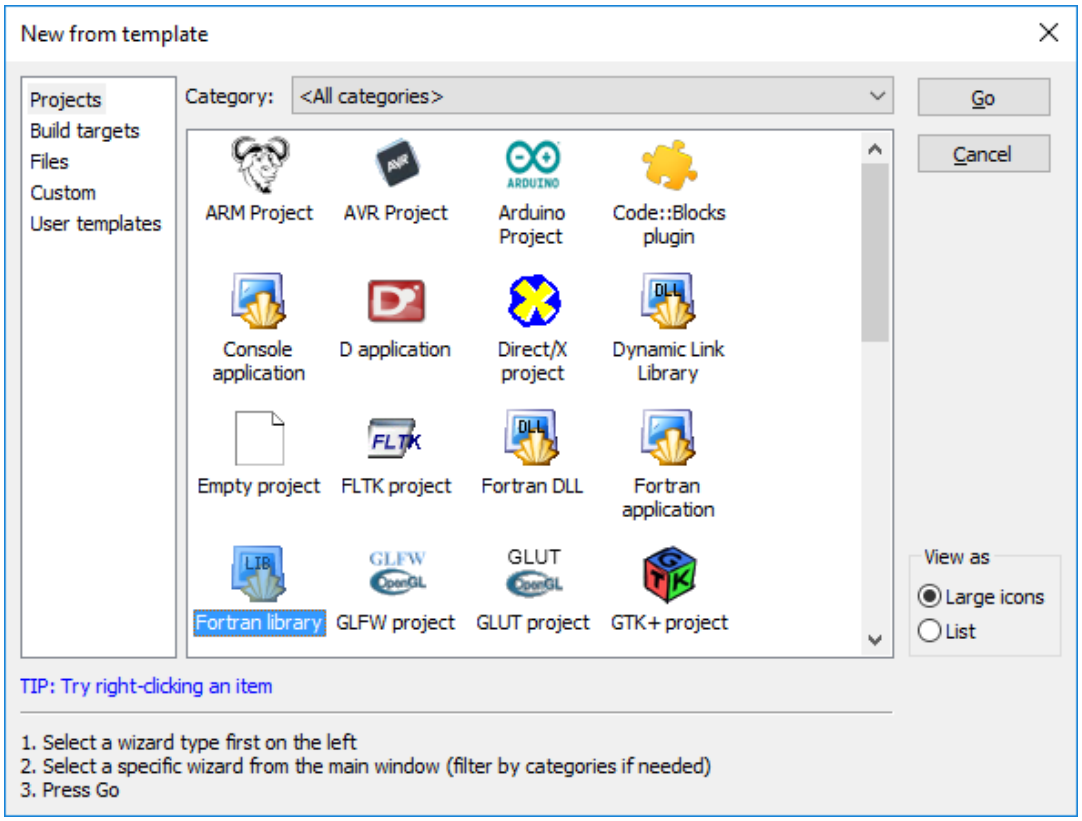
Dynamically linked libraries, or DLLs, are accessed by the program only while it is running. The disadvantage is that any program that uses the library must have access to it during execution. However, it allows the executable to remain small and provides a nice solution to providing libraries to many users. The file suffix for a DLL is `.dll`.

The same example is used to show how to build and use a library of each kind using Code::Blocks. It is possible to build either type of library using command lines. Probably the easiest was to do that is to observe the commands that are executed when building using Code::Blocks; these are displayed in the Build log window after a Build is executed. Building DLLs from the command line also is discussed in Section 12.4.

12.1 Static Libraries

12.1.1 Building a Static Library

A static library can be built using a Code::Blocks Fortran library project. Run Code::Blocks and select *File* → *New* → *Project*. Then select a Fortran library project by double clicking on the icon.



If the Welcome screen appears, select *Next* to display the Fortran library window. Enter a name for the project, such as *static_lib_mod* (this project will contain modules that contain the procedures to go in the library). Select a workspace if an appropriate one is not already displayed. Select *Next* and ensure that the GNU Fortran Compiler is selected in the next window; everything else should be OK. Select *Finish* and the `static_lib_mod` project should appear in the Code::Blocks Management/Projects window.

Make sure that `static_lib_mod` is the active project (if there is more than one in the workspace) and delete the file `main.f90` (highlight the file name, right click on it and select *Remove file from project*). Add a new empty file (*File* → *New* → *Empty file*). Name the file `static_lib_mod1.f90` and save it as a Fortran file. Type in the following module (or copy and paste it from this document):

```
module static_lib_mod1

   implicit none
   private
   integer, parameter :: answer = 42
   public :: static_sub1

contains

subroutine static_sub1(n)
```

```
   integer, intent(in) :: n
   integer :: i

   do i = 1, n
      print *, "The answer is", answer
   end do
end subroutine static_sub1

end module static_lib_mod1
```

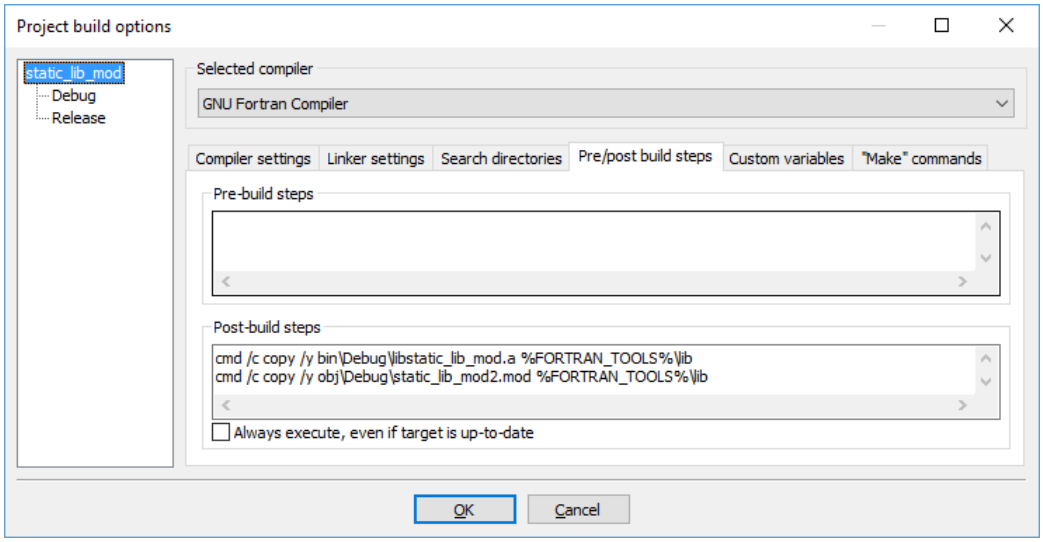
Similarly add another source file named `static_lib_mod2.f90` and enter the following module:

```
module static_lib_mod2

   use static_lib_mod1
   implicit none
   private
   public :: static_sub2

contains

subroutine static_sub2(k)
   integer, intent(in) :: k
   call static_sub1(k)
end subroutine static_sub2

end module static_lib_mod2
```

Building the project will create the file `libstatic_lib_mod.a`, which is the one of most interest. But any program that needs to access the library must know where it is during the linking phase. The folder `%FORTRAN_TOOLS%\lib` already contains the Fortran Tools library if that is where the Fortran Tools were installed, so it is convenient to put this library file there also. Also, when any program using the module is compiled, the module `.mod` files must be accessible, so we also put them in `%FORTRAN_TOOLS%\lib`. It is possible to copy these files by specifying a post-processing step when the project is built.

Make sure that the `static_lib_mod` project is active. Select the *Project* tab, then *Build options*, then the *Pre/post build steps* tab. Select *static_lib_mod* in the upper left-hand corner to apply the following to both releases (or select one of the configurations). In the *Post-build steps* window, type (or copy):

```
cmd /c copy /y bin\Debug\libstatic_lib_mod.a %FORTRAN_TOOLS%\lib
cmd /c copy /y obj\Debug\static_lib_mod2.mod %FORTRAN_TOOLS%\lib
```



The library can be built and the library file and the module files will be copied to a place where they can be accessed (it could be left where it was created, but usually it will be put in a place where multiple programs can access it). It makes no sense to run the project because it contains no main program.

12.1.2 Using Static Libraries

Another project can be created that uses a static library. As an example, create a new Fortran empty project named `call_static_lib`. Add to the project an empty Fortran source file named `call_static_lib.f90`. Enter the following program:

```
program call_static_lib

   use static_lib_mod2
   implicit none

   call static_sub2(k = 3)

end program call_static_lib
```

This program needs to link the static library as part of its executable file. To do this select *Project* → *Build options* → *Linker settings*. Make sure the project `call_static_lib` is selected. Below the *Linker libraries* window, select *Add*. Enter the following and then select *OK* twice.

```
%FORTRAN_TOOLS%\lib\libstatic_lib_mod.a
```

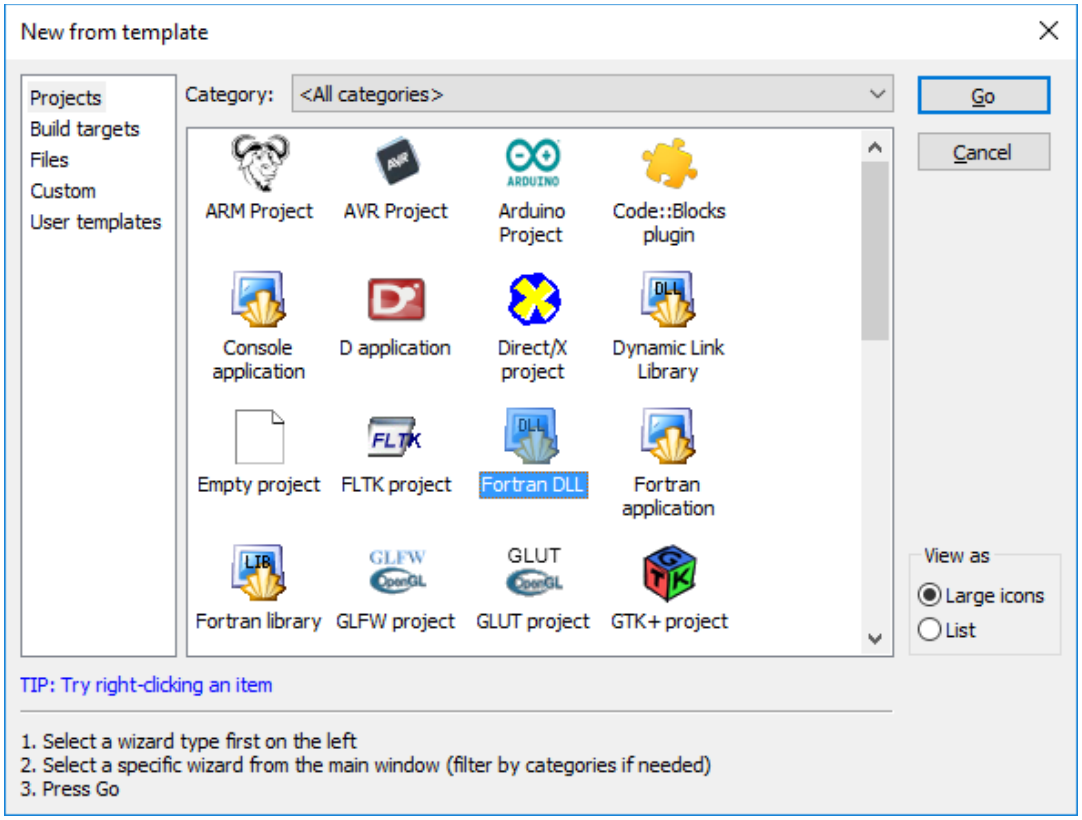
Now the project `call_static_lib` can be built and run.

```
C:\FTWorkspace\static\bin\Debug\static.exe
The answer is          42
The answer is          42
The answer is          42

Process returned 0 (0x0)   execution time : 0.011 s
Press any key to continue.
```

12.2 Dynamically Linked Libraries

12.2.1 Building a DLL

A dynamically linked library (DLL) can be built using a Code::Blocks Fortran DLL project. Run Code::Blocks and select *File* → *New* → *Project*. Then select a Fortran DLL project by double clicking on the icon.



If the Welcome screen appears, select *Next* to display the Fortran DLL window. Enter a name for the project, such as *dll_mod* (this project will contain modules that contain the procedures to go in the DLL). Select a workspace if an appropriate one is not already displayed. Select *Next* and ensure that the GNU Fortran Compiler is selected in the next window; everything else should be OK. Select *Finish* and the `dll_mod` project should appear in the Code::Blocks Management/Projects window.

Make sure that `dll_mod` is the active project (if there is more than one in the workspace) and delete the file `main.f90` (highlight the file name, right click on it and select *Remove file from project*). Add a new empty file (*File* → *New* → *Empty file*). Name the file `dll_mod1.f90` and save it as a Fortran file. Type in the following module (or copy and paste it from this document).

```
module dll_mod1

   implicit none
   private
   integer, parameter :: answer = 42
   public :: dll_sub1

contains

subroutine dll_sub1(n)
   integer, intent(in) :: n
   integer :: i

   do i = 1, n
      print *, "The answer is", answer
   end do
end subroutine dll_sub1

end module dll_mod1
```

Similarly add another source file named `dll_mod2.f90` and enter the following module:

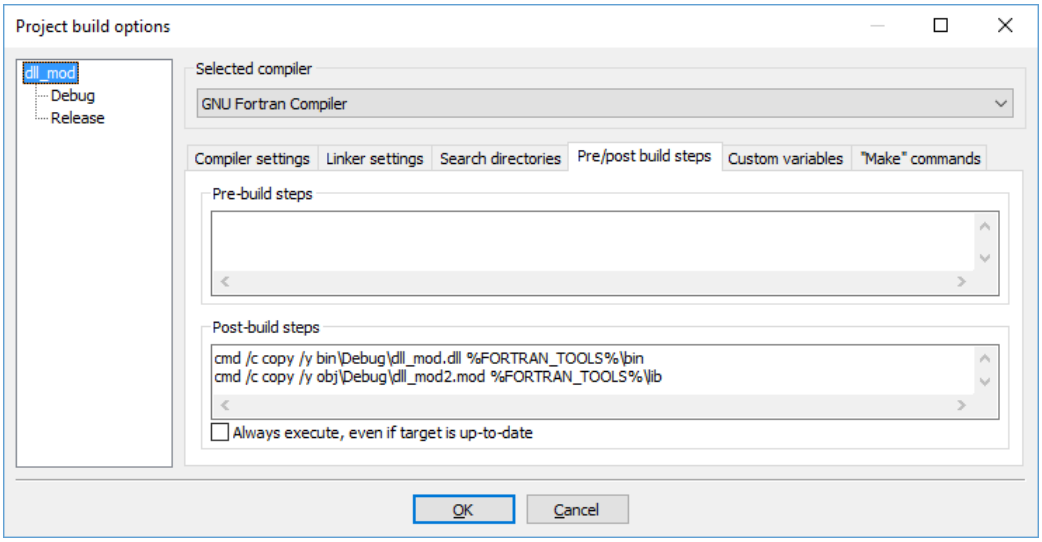
```
module dll_mod2

   use dll_mod1
   implicit none
   private
   public :: dll_sub2

contains

subroutine dll_sub2(k)
   integer, intent(in) :: k
   call dll_sub1(k)
end subroutine dll_sub2

end module dll_mod2
```

Building the project will create the file `dll_mod.dll`, which is the one of most interest. But any program that needs to access the library must know where it is both during

compilation and during execution. During execution, it must be in a folder that is in your execution path. The folder `C:\Fortran_Tools\bin` is in your path if it was set during installation as suggested (1.6); we will use this one (the `C:\Fortran_Tools` folder may be different if the Fortran Tools were installed in a different place). The folder `C:\Fortran_Tools\gfortran\bin` also is in the execution path if that is where Gfortran was installed. It is possible to copy the `.dll` file to the appropriate place by specifying a post-processing step. Also, when any program using the module is compiled, the module `.mod` file must be accessible, so we put it in `C:\Fortran_Tools\lib`, which already contains the modules for the Fortran Tools library.

Make sure that the `dll_mod` project is active. Select the *Project* tab, then *Build options*, then the *Pre/post build steps* tab. Make sure that Debug is selected in the upper left-hand corder. In the *Post-build steps* window, type (or copy):

```
cmd /c copy /y bin\Debug\dll_mod.dll %FORTRAN_TOOLS%\bin
cmd /c copy /y obj\Debug\dll_mod2.mod %FORTRAN_TOOLS%\lib
```



The library can be built and the library file and the module files will be copied to a place where they can be accessed (it could be left where it was created, but usually it will be put in a place where multiple programs can access it). It makes no sense to run the project because it contains no main program.

12.2.2 Using DLLs

Another project can be created that uses a DLL. As an example, create an empty Fortran application project named `call_dll`. Add to the project an empty Fortran source file named `call_dll.f90`. Enter the following program:

```
program call_dll

   use dll_mod2
   implicit none

   call dll_sub2(k = 3)

end program call_dll
```

The calling program needs to have access to the DLL during both compilation and execution. To provide access during compilation select *Project* → *Build options* → *Linker settings.* Make sure the project `dll_lib` is selected. Below the *Linker libraries* window, select *Add*. Enter the following and then select *OK* twice.

```
%FORTRAN_TOOLS%\bin\dll_mod.dll
```

During execution, the DLL is accessible because it has been copied into a `bin` directory in the execution path.

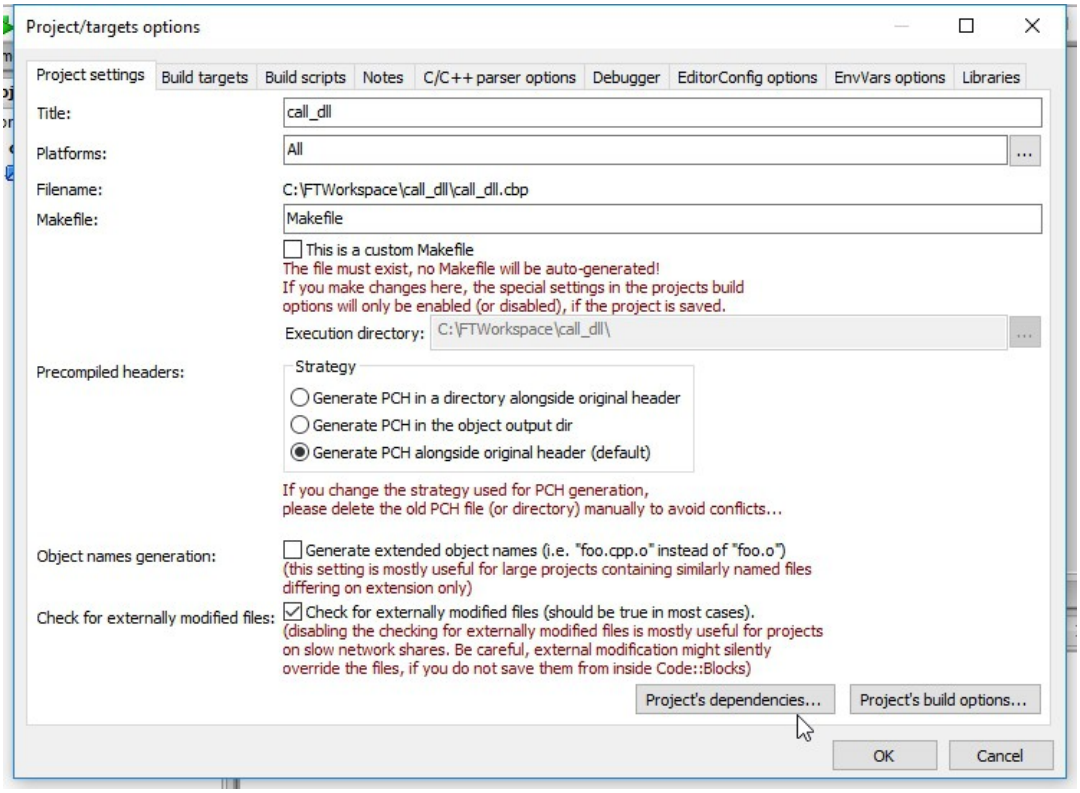
Now the project `call_dll` can be built and run.

12.3 Project Dependencies

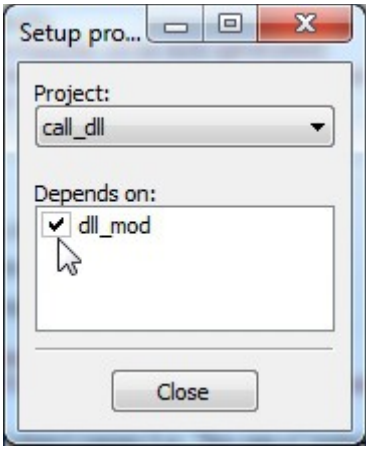
In more realistic cases, a library or set of libraries probably would be kept in its own workspace and programs that use the library kept in another. But in simple cases, a library project and the projects that use the library might be kept in the same workspace.

The programs that use a library depend on the library, just as a program that uses a module depends on the module. It is possible to set Code::Blocks so that if a library and the programs that use it are in the same workspace, the programs depend on the library. When a calling program is rebuilt, the library will be rebuilt, if necessary, before any programs that depend on it.

To set such a dependency, set the calling project to be active. Then select *Project* → *Properties*.



In the *Project/Target* options window, select the *Project dependencies...* button near the lower right-hand corner. Check the box for the DLL project on which the calling project depends. Then *Close* → *OK*.



12.4 DLLs Using the Command Line

Sometimes it is convenient to use the command line to build and use a DLL. This section shows how to do this in way that reflects a more realistic process for building large programs.

In this section, a DLL will be built from procedures in Fortran modules using the previous example. Using external procedures is similar except that there is no need to worry about `.mod` files.

In a more complex program, the parts that make up the program may not be kept all in one folder. To illustrate this, create a folder for the whole program called `Dll_program` and then make three subfolders `Dll`, `Lib`, and `Main`.

```
mkdir Dll_program
cd Dll_program
mkdir Dll Lib Main
```

By copying files or typing, put in the folder `Dll` the file `dll_mod1.f90` containing the program `dll_mod1` and the file `dll_mod2.f90` containing the program `dll_mod2`.

The file `dll_mod1.f90`:

```
module dll_mod1

   implicit none
   private
   integer, parameter :: answer = 42
   public :: dll_sub1

contains

subroutine dll_sub1(n)
   integer, intent(in) :: n
   integer :: i

   do i = 1, n
      print *, "The answer is", answer
   end do
end subroutine dll_sub1

end module dll_mod1
```

The file `dll_mod2.f90`:

```
module dll_mod2

   use dll_mod1
   implicit none
   private
   public :: dll_sub2

contains

subroutine dll_sub2(k)
   integer, intent(in) :: k
   call dll_sub1(k)
end subroutine dll_sub2
```

```
end module dll_mod2
```

The modules can be compiled with:

```
gfortran -c -J..\Lib dll_mod2.f90 dll_mod2.f90
```

The `-J` option puts the resulting `.mod` files in the folder `..\Lib`.
The DLL can be built with the command:

```
gfortran -shared -o dll_mod.dll dll_mod1.o dll_mod2.o
```

This produces the DLL file `dll_mod.dll`. It can be left in the current folder or placed in any other place where programs that want to use it can find it both during compilation and during execution. A possibility is to put it in `Lib`.

```
copy dll_mod.dll ..\Lib
```

Put the program `call_dll` in the file `call_dll.f90` in the folder `Main`.
The file `call_dll.f90`:

```
program call_dll

   use dll_mod2
   implicit none

   call dll_sub2(k = 3)

end program call_dll
```

Compile this program with:

```
gfortran -c -J ..\Lib call_dll.f90
```

Link it with:

```
gfortran -o call_dll.exe ..\Lib\dll_mod.dll call_dll.o
```

In order to execute the program, `dll_mod.dll` must be in the execution path (1.6). For example, it could be copied into the current folder (`Main`) or it could be copied into the folder containing the Gfortran executable (such as `%FORTRAN_TOOLS%\gfortran\bin`). It is probably not a good idea in the long run to put it in the Fortran Tools installation folder, so that the tools can be uninstalled or reinstalled without affecting the program.

Running `call_dll` produces

```
 The answer is          42
 The answer is          42
 The answer is          42
```

13 Timing and Profiling

In many, of not most, cases, the time it takes to execute a program is not important. But there are exceptions, such as compute-intensive applications like weather prediction and simulation. Also, in some applications, response to a human must be timely, so even a few extra seconds can be significant.

Thus, in most cases, the time spent writing a program and especially debugging and modifying it are far more important than the time it takes to run it. This implies that programs should be written in a way that is clear, conforms well to the algorithms being implemented, and is easy to maintain.

This section discusses some of the options to deal with a program whose execution time is important. In order to experiment with something, we consider the following program, some of which was extracted from a real program, but does nothing useful by itself.

```
module data

   implicit none
   integer, parameter :: N_ROWS = 150
   integer, parameter :: N_COLS = 250
   integer, parameter :: ABCD_SIZE = 1000
   integer, dimension(:,:), allocatable :: MAT
   real, dimension(:,:), allocatable :: A, B, C, D
   integer, dimension(:), allocatable :: temp

end module data

module setup

   use random
   use data
   implicit none
   private
   public :: setup_data

contains

   subroutine setup_data()
      integer :: alloc_stat
      allocate(MAT(N_ROWS, N_COLS), stat=alloc_stat)
      if (alloc_stat > 0) then
         print *, "Allocation of MAT failed"
         stop
      end if
      allocate(A(ABCD_SIZE, ABCD_SIZE), &
               B(ABCD_SIZE, ABCD_SIZE), &
               C(ABCD_SIZE, ABCD_SIZE), &
               D(ABCD_SIZE, ABCD_SIZE), &
         stat=alloc_stat)
      if (alloc_stat > 0) then
         print *, &
            "Allocation of A, B, C, or D failed"
         stop
      end if
```

```
      call random_int(MAT, 0, ABCD_SIZE)
      call random_number(B)
      call random_number(D)
      call mix_mat()
   end subroutine setup_data

   ! Rearrange MAT just for something to do
   subroutine mix_mat()
      integer, parameter :: mixes = 1000000
      integer :: i, ir, jr
      do i = 1, mixes
         call random_int(ir, 1, N_ROWS)
         call random_int(jr, 1, N_ROWS)
         temp = mat(ir, :)
         mat(ir, :) = mat(jr, :)
         mat(jr, :) = temp
      end do
   end subroutine mix_mat

end module setup

module compute

   use data
   implicit none
   private
   public :: do_loops

contains

   subroutine do_loops()

      integer :: I, J, K, L, II, JJ

      A = 0
      C = 0

      DO 379 I=1,N_ROWS
      DO 379 J=1,N_ROWS
      DO 378 K=1,N_COLS
      DO 378 L=1,N_COLS
      II=MAT(I,K)
      JJ=MAT(J,L)
      IF(II .EQ. 0 .OR. JJ .EQ. 0)GO TO 378
      A(II,JJ)=A(II,JJ)+B(I,J)
      C(II,JJ)=C(II,JJ)+D(I,J)
378   CONTINUE
379   CONTINUE

   end subroutine do_loops

end module compute

program loops

   use setup
   use compute
```

```
    use data, only: A, C
    implicit none

    real :: start_time, stop_time

    call cpu_time(start_time)
    call setup_data()
    call cpu_time(stop_time)
    print *, "Setup time:", &
       stop_time - start_time, "seconds"

    call cpu_time(start_time)
    call do_loops()
    call cpu_time(stop_time)
    print *, "Original loop time:", &
       stop_time - start_time, "seconds"

    print *, sum(A), sum(C)

end program loops
```

This program is in the file `loops.f90` in the `examples` folder of the Fortran Tools distribution. It uses a generic subroutine named `random_int` in the module `random`, which is also contained in the file `loops.f90`. The subroutine uses the intrinsic subroutine `random_number` to fill the first argument with pseudo-random integers between the second and third arguments.

13.1 Timing a Program

In order to time portions of the program, the intrinsic subroutine `cpu_time` may be used. It returns the time in seconds since the beginning of execution of the program. Thus, to time a portion of a program, record the cpu (central processing unit) time just before and just after the portion of the program to be timed. Then the execution time of the portion of the program is the difference between the two times.

Note: If a program is run on multiple processors using, for example, coarrays, OpenMP, or MPI, `cpu_time` returns the total time for all of the processors. For this situation, the intrinsic subroutine `system_clock` is better, as it returns elapsed time, which is probably a better indicator of the performance of the program. `system_clock` is discussed in the section on OpenMP (Section 15).

This scheme is used to time the portion of the program used to set up some values in the arrays and also to time the execution of the quadruple loops. The program is compiled with the command

```
gfortran loops.f90
```

The result of running this program is as follows.

```
 Setup time:   1.35899997     seconds
 Original loop time:   29.2810001     seconds
   699112512.       702719552.
```

The program prints `sum(A)` and `sum(C)` because some compilers could optimize away the entire quadruple loops if the results computed in the loops were never used.

The timing results shown in this section were obtained running an Intel i5-4440 four-processor system.

13.2 Compiler Optimization

Suppose the 29-second execution time is not acceptable. Probably trying some different compiler options is the simplest way to improve the running time. The main Gfortran compiler option is `-O`, although `-O1`, `-O2`, and `-O3` also are available.

To compile the program with optimization, use the command

```
gfortran -O3 loops.f90
```

Running the program produced by this compilation produces

```
 Setup time:   1.31099999     seconds
 Original loop time:   11.0939999     seconds
   699112512.       702719552.
```

13.3 Profiling a Program

Compiling the program with the optimization option set is definitely an improvement. But suppose this is still not good enough and we think we can improve the running time by modifying the program. In this simple case, we can tell where all the time is spent, but imagine a much more complicated program. It is a waste of effort to try to improve parts of the program that take little time to execute, so we need to determine the parts of the program that run for a significant amount of time.

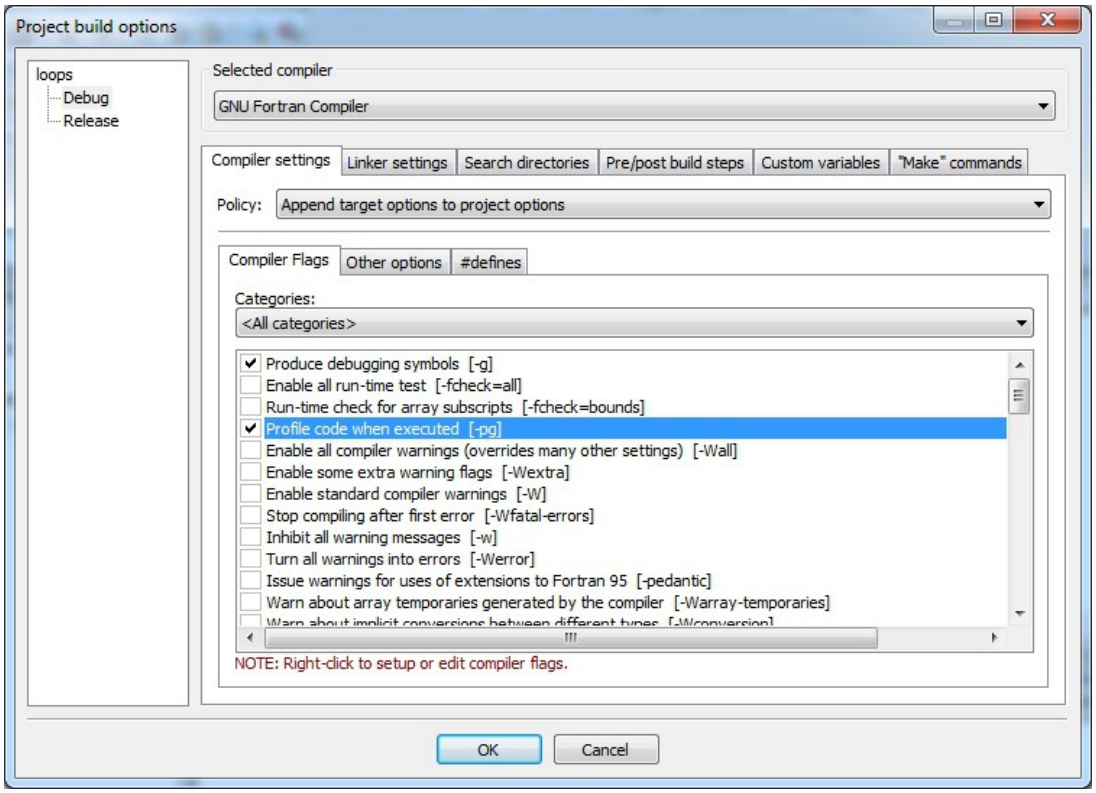
13.3.1 gprof

One of the Fortran Tools is a profiler called `gprof`. This tool produces a report indicating how much time is spent executing each part of the program, much like what would be accomplished by using `cpu_time` in all the parts of a program.

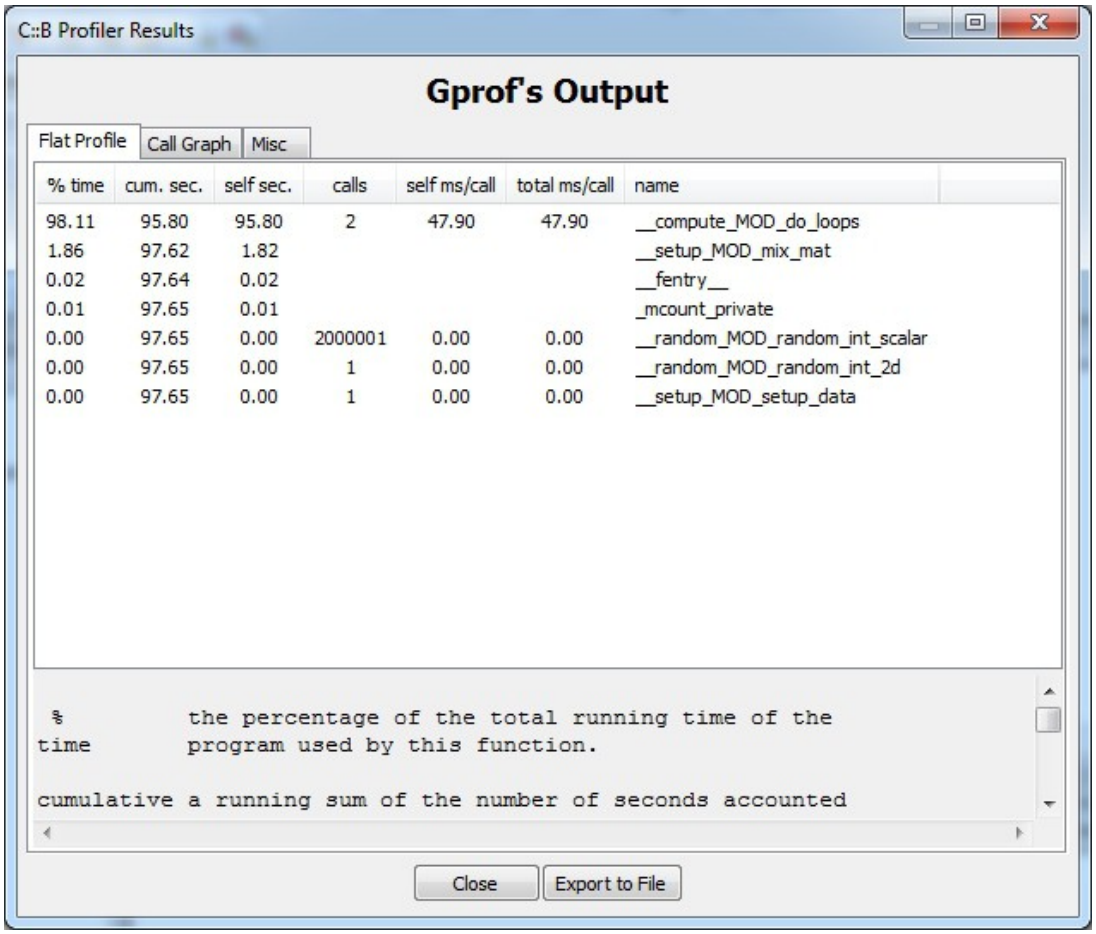
The program `gprof` may be used easily with a Code::Blocks project or the command line may be used.

13.3.2 Using gprof with Code::Blocks

To use `gprof` with Code::Blocks, set the compile option `-pg`. With a project selected that contains the source `loops.f90`, select *Project* → *Build options* → *Compiler settings* → *Compiler Flags* → <*All categories*> and check the box *Profile code when executed*, which sets the `-pg` option.



Build and run the program. After executing the program, select the *Plugins* tab, then select *Code profiler*, then from the next dialog, *Debug* (if that is the version you just ran). The `gprof` window is displayed.



The display indicates that 98.11% of the execution time was spent in the subroutine `do_loops`, 1.86% of the time in `mix_mat`, etc. More information about the contents of the window occurs below what is shown, seen by using the slider bar at the bottom right of the window. The file may be saved for later use with the *Export to File* button at the bottom of the window.

13.3.3 Using gprof from the Command Line

Profiling using the command line involves several steps. First, the program must be compiled with the `-pg` option. In this example, the executable is named `loops` using the `-o` option instead of the default `a`.

```
gfortran -pg -oloops loops.f90
```

Compiling without optimization produces a more informative report.

Next, the program is executed by typing `loops` or `loops.exe`. This produces a file named `gmon.out`.

Then the program gprof is executed.

```
gprof -b loops.exe > loops.prof
```

The output from gprof goes to the standard output and so it can be redirected to the file loops.prof. The .exe extension on loops.exe is necessary in this case. The -b option produces a brief report, part of which is shown below. The file loops.prof is in the examples folder of the Fortran Tools distribution.

```
  %  cumulative ...
 time seconds ... name
98.21   90.04 ... __compute_MOD_do_loops
 1.75   91.64 ... __setup_MOD_mix_mat
 0.03   91.67 ... _mcount_private
 0.01   91.68 ... __setup_MOD_setup_data
 0.00   91.68 ... __random_MOD_random_int_scalar
 0.00   91.68 ... __random_MOD_random_int_2d
```

This time 98.21% of the execution time was spent in the main program loops, 1.75% of the time was spent in the subroutine mix_mat, and so forth.

13.3.4 gcov

The tool gprof shows execution times for each procedure. The tool gcov shows how many times each statement is executed. Using gcov also requires several steps.

First compile the code with options -fprofile-arcs and -ftest-coverage.

```
gfortran -fprofile-arcs -ftest-coverage -o loops loops.f90
```

This produces a file named loops.gcno.

Next, execute the program by typing loops or loops.exe. This produces a file named loops.gcda.

Then, type

```
gcov loops.f90
```

This produces a file named loops.f90.gcov, parts of which are shown below. The entire file is in the examples folder of the Fortran Tools distribution.

```
        -:    0:Source:loops.f90
        -:    0:Graph:loops.gcno
        -:    0:Data:loops.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:module random
                  . . .
        -:   10:contains
        -:   11:
  2000000:   12:subroutine random_int_scalar . . .
                  . . .
        -:   22:
        1:   23:subroutine random_int_2d . . .
                  . . .
        -:   48:end module data
```

```
        -:   49:
        -:   50:module setup
                    . . .
        -:   59:
        1:   60:   subroutine setup_data()
                    . . .
        1:   83:      call mix_mat()
        1:   84:   end subroutine setup_data
                    . . .
        1:   87:   subroutine mix_mat()
                    . . .
  1000001:   90:      do i = 1, mixes
  1000000:   91:         call random_int( . . .
  1000000:   92:         call random_int( . . .
  1000000:   93:         temp = mat(ir, :)
  1000000:   94:         mat(ir, :) = mat(jr, :)
  1000000:   95:         mat(jr, :) = temp
        -:   96:      end do
        1:   97:   end subroutine mix_mat
        -:   98:
        -:   99:end module setup
        -:  100:
        -:  101:module compute
                    . . .
        -:  109:
        1:  110:   subroutine do_loops()
                    . . .
      151:  114:      DO 379 I=1,N_ROWS
    22650:  115:      DO 379 J=1,N_ROWS
  5647500:  116:      DO 378 K=1,N_COLS
1411875000:  117:       DO 378 L=1,N_COLS
1406250000:  118:       II=MAT(I,K)
1406250000:  119:       JJ=MAT(J,L)
1406250000:  120:       IF(II .EQ. 0 .OR. . . .
1403476369:  121:       A(II,JJ)=A(II,JJ)+B(I,J)
1403476369:  122:       C(II,JJ)=C(II,JJ)+D(I,J)
        -:  123:378   CONTINUE
        -:  124:379   CONTINUE
        -:  125:
        1:  126:   end subroutine do_loops
        -:  127:
        -:  128:end module compute
        -:  129:
        1:  130:program loops
                    . . .
        1:  140:   call setup_data()
                    . . .
        1:  146:   call do_loops()
                    . . .
        1:  153:end program loops
```

This confirms that most of the execution involves the quadruple loops.

The other important use of the gcov tool is for testing a program. By trying different test data and running gcov, it is possible to tell which parts of the program have been tested.

13.4 Modifying the Program

If it is still desirable to try to improve the performance of the program, the profiling has shown that it is effective to focus attention only on the quadruple loops. In this particular case, it turns out that modifying the code can reduce the running time.

The main modification involves the order of the `DO` loops. Because Fortran arrays are stored in the order specified by varying the first subscript first, and because of the characteristics of memory usage in modern computers, it is desirable to process array elements in that same order. This means that if there is a double loop that varies over the two subscripts of a two-dimensional array, the outer loop should involve the second subscript and the inner loop the first.

Often an optimizing compiler will make such a switch, but, of course, it must be determined that the results are the same with the order of the loops exchanged. In the case of the program `loops`, it is hard for the compiler to figure that out, but it is possible to see that the order does not matter.

Another common optimization is to take code out of a loop if it does not depend on being in the loop. In the loops program, the test for `JJ` can be removed from the inner loop after the loop ordering is changed.

Also, the code is modified to reflect a more modern style. This makes it easier to understand and maintain, but should have no effect on performance.

When code is changed in this manner, it is a good idea to comment out the original parts that have been replaced and indicate what changes have been made and why they have been made.

The modified portion of the code and the results produced when compiled with `-O3` are shown below. The complete modified program is in the file `loops_modified.f90` in the `examples` folder of the Fortran Tools distribution.

```
! Loops reordered
! Original order of loops:; I,J,K,L
! I, J should be inside K, L (MAT)
! I should be inside J (B, D)
! Loops changed to DO/END DO
! GO TO changed to CYCLE
! Test for JJ taken out of inner loop

   do L = 1, N_COLS
   do J = 1, N_ROWS
      JJ = MAT(J, L)
      if (JJ == 0) cycle
      do K = 1, N_COLS
      do I = 1, N_ROWS
         II = MAT(I, K)
         if (II == 0) cycle
         A(II, JJ) = A(II, JJ) + B(I, J)
         C(II, JJ) = C(II, JJ) + D(I, J)
      end do
      end do
   end do
   end do
```

```
Setup time:   1.29600000     seconds
Original loop time:   11.1410007     seconds
  699112512.       702719552.
Modified loop time:   2.17199993     seconds
  699112640.       702719424.
```

Notice that the printed values of the sums are slightly different. Changing the order of the loops causes the sums to be accumulated in a different order, changing the roundoff error slightly.

14 Coarrays

Coarrays provide a natural standard Fortran feature for spreading a computation across multiple processors. Unfortunately, the coarray features are not built into gfortran and hence are not built into the Fortran Tools. To run a Fortran program with coarrays, additional software must be installed. The instructions for doing this are described in the next section.

This software may be installed only on Windows 10 (not Windows 10 S).

14.1 Installing Coarray Fortran

To run a coarray Fortran program on Windows, the Windows System for Linux (Ubuntu), a coarray Fortran library, and an MPI library must be installed.

To install WSL, use your web browser to go to `https://microsoft.com`. Search (the little magnifying glass icon) for `ubuntu`. Click on `Ubuntu 18.04`. On the new page that comes up, select `Get the app`. On the next screen that comes up, select `Get`. You do not need to sign up with Microsoft. On the next screen, you may (optional) select ". . ." and `Pin to Taskbar`. You may also select `Pin to Start`. Then either select `Launch` or click on the icon just added to the task bar or Start menu. An Ubuntu screen will appear and make the announcement that the software is being installed (it does take several minutes). When asked, pick a user name and a password (the password must be entered twice).

If a prompt does not appear in a few minutes, hit the carriage return (it doesn't hurt to do that right away—the system will just wait).

When a prompt appears, enter the following commands:

```
sudo apt update

sudo apt install mpich
```

Ubuntu files are installed at `/mnt/c/Users/Walt/AppData/Local/Packages/Canonical*`, with your user name in place of `Walt`.

If you want, you can run

```
sudo apt upgrade
```

which also takes a while. It probably won't affect running CAF.

To install the coarrays library, enter the command:

```
git clone https://github.com/sourceryinstitute/opencoarrays.git
```

When the prompt reappears, type `ls` (list), the equivalent of Windows `dir`, to see that there is a directory (folder) `opencorrays`. Go to that directory by typing

```
cd opencoarrays
```

then to install the open coarrays library, type:

```
./install.sh -j 4
```

`-j 4` says to run on four processors (change it if you have a different number or omit it if you want). This step also takes several minutes.

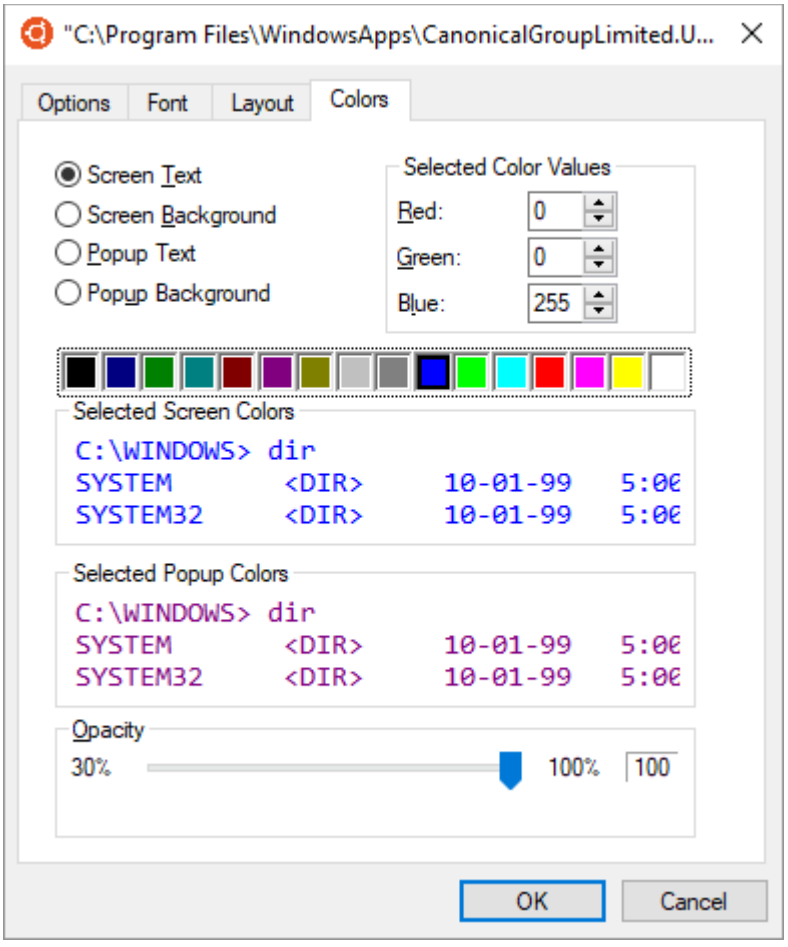
At the prompt, edit the file `~/.bashrc` to add the following line at the end. It will tell Ubuntu where to find the executable programs `caf` and `cafrun`.

```
export PATH=~/opencoarrays/prerequisites/builds/opencoarrays/2.1.0/bin:$PATH
```

14.2 Running Ubuntu

To run an Ubuntu command prompt window, either select it from the Start menu, click on the Ubuntu icon, or type `ubuntu1804` in a Fortran Tools command prompt window.

If you don't like the appearance of the screen, its properties can be changed by right clicking in the top margin and selecting `Properties`.



WSL/Ubuntu is a Linux-like system, so all of the usual Linux commands (such as `ls` and `grep`), as well as programs such as `perl`, are available when running Ubuntu.

14.3 Editing a CAF Program

Here is a very simple CAF program.

```
program caf_hello

   implicit none

   print *, "Hello from", this_image(), &
            "out of", num_images(), "images."

end program caf_hello
```

The editor `vi` is available; `emacs` can be installed with the command `sudo apt-get install emacs`. If you don't know either, a simple editor named `edit` is available. If you type `edit caf_hello.f90` and get an error message, try `echo xxx > caf_hello.f90` first, then `edit caf_hello.f90`. Then edit the file by removing the line with `xxx` and copying in the example above.

14.4 Building a CAF Program

CAF programs are compiled with the command `caf`.

```
caf caf_hello.f90 -O -o caf_hello.exe
```

14.5 Running a CAF Program

To run a CAF program, use the command `cafrun`. The number of images must be specified with the `-np` option followed by a space and the number of processors. The executable file name must include `./` so that the Linux system can find it.

```
cafrun -np 4 ./caf_hello.exe

 Hello from           3 out of           4 images.
 Hello from           1 out of           4 images.
 Hello from           2 out of           4 images.
 Hello from           4 out of           4 images.
```

14.6 Copying Files to/from the Windows File System to WSL

In WSL, the Windows folder `C:` is known as `/mnt/c`. So, for example, to copy the two CAF examples in the Fortran Tools examples folder to the current directory, use the command

```
 cp /mnt/c/Fortran_Tools/examples/caf* .
```

assuming the Fortran Tools are installed in `C:\Fortran_Tools`.

Remember that WSL is a Linux-like system so the path separator is `/` and the copy command is `cp`, rather than `copy`.

14.7 Introduction to Coarray Fortran

For Coarray Fortran (CAF), computations take place on one or *images*. The number of images is set when the program is run as described above and cannot be changed dynamically.

More information can be found in the Fortran 2008 standard in the `doc` folder, in the book *Guide to Fortran 2008 Programming* by Brainerd (Springer, 2015), or other books.

14.7.1 Images

The images are numbered 1, 2, ..., *n*.

An important thing to keep in mind is that each image executes the same program. Because the program code can depend on which image is executing it, the images may be executing different parts of the program at the same time or executing the same part of a program at different times.

Each image has its own copy of the program and the data. The program is the same on each image. The *execution* of the program may be different on each image.

The intrinsic function `num_images` returns the number of images.

The intrinsic function `this_image` returns the number of the image on which the code is executing.

14.7.2 Varying the Execution on Images

Because the intrinsic function `this_image` indicates which image the code is executing on, the `select case` or `if` constructs may be used to vary the execution on different images.

```
program trig

   implicit none
   real :: x = 0.5
   character(len=*), parameter :: &
         fmt = "(a, f0.1, a, f0.5)"

   x = 0.1 + this_image()/10.0

   select case (this_image())
   case (1)
      print fmt, "sine(", x, ") = ", sin(x)
   case (2)
      print fmt, "cosine(", x, ") = ", cos(x)
   case (3)
      print fmt, "tangent(", x, ") = ", tan(x)
   end select

end program trig
```

The output is

```
tangent(.4) = .42279
sine(.2) = .19867
cosine(.3) = .95534
```

The output from the three images occurs in an arbitrary order.

14.7.3 Declaring Coarrays

Declaring a coarray by giving it the codimension attribute allows its values to be accessed on any of the images. Some examples follow

```
real, dimension(100), codimension[*] :: ca, cb, cd
real :: dimension(0:9, 4:12), codimension[0:*] :: c2
integer, codimension[*] :: n
```

The last upper cobound must be `*`. `n` is a coarray scalar.

14.7.4 Referencing a Value on Another Image

The bracket notation used like an array subscript refers to a value on another image. Without the bracket, a value refers to the value on the local image.

The presence of the brackets indicates data movement from one image to another and may indicate a performance penalty. For example, `c2(:,:)[3]` refers to the entire array `c2` on image 3 and the values of `c2` will be copied to the current image.

In the following program, the value of `n` on image 1 is accessed by image 2 and stored as the value of `n` on image 2.

```
program ref_image_value

   implicit none
   integer, codimension[*] :: n = -99

   n = this_image()
   if (this_image() == 2) then
      print *, "Before assignment, n =", n
      n = n[1]
      print *, "After assignment, n =", n
   end if

end program ref_image_value
```

The output is

```
Before assignment, n = 2
After assignment, n = 1
```

14.7.5 The sync all Statement

The `sync all` statement causes execution on each image to wait at that point in the computation until all of the images reach that point.

Look again at the previous example, remembering that the program is being executed by all images in whatever order they get to the instructions. It is possible, but in this case not likely, that image 2 will reach the assignment statement

```
n = n[1]
```

before n as been given the value of its image on image 1. This is fixed by putting the statement

```
sync all
```

just before the if statement. Then image 2 will not execute the `if` construct and access `n[1]`until it has been set on image 1.

Other examples of the `sync all` statement occur later in this section.

14.7.6 Input and Output

Each image has its own units and file connections. The default output unit (*) is defined for all images. The default input unit (*) is defined only for image 1. The output from images is merged in a processor-defined manner; this means that even if output from the images is controlled to execute in order, the output itself may not appear in a file in any particular order.

14.7.7 A Sorting Example

To sort an array using two images, the first half of the array is sorted on image 1, the second half is sorted on image 2, and the two halves of the array are merged. A very inefficient interchange sort (not shown) is used. At each point in the merging process, the first elements of each sorted list are compared and the smaller one is selected for inclusion as the next element in the merged list. The process is made a little more complicated by handling the merge after one of the lists is exhausted.

```
module sort_mod

   implicit none

! subroutine interchange_sort(a) goes here

function merge2(a, b) result(m)

   real, dimension(:), intent(in) :: a, b
   real, dimension(size(a)+size(b)) :: m
   integer :: ka, kb, km

   ka = 1; kb = 1; km = 1

   do
      if (ka > size(a)) then
         m(km:) = b(kb:)
         return
      else if (kb > size(b)) then
         m(km:) = a(ka:)
         return
      else if (a(ka) < b(kb)) then
         m(km) = a(ka)
         km = km + 1; ka = ka + 1
      else
         m(km) = b(kb)
         km = km + 1; kb = kb + 1
```

```
      end if
   end do

end function merge2

end module sort_mod
```

In the program `sort2`, random numbers are generated, stored in the array `a`, and copied into the array `b`. To do the sort that uses two images, the latter half of `a` is copied to image 2; then the first half is sorted on image 1 and the second half is sorted at the same time on image 2; then the sorted latter half is copied back from image 2 to image 1; then the two halves of the array are merged on image 1.

To compare doing the sort on one image, the same steps are followed, but use just one image. The two halves are sorted and then merged into one array. This process must be used because simply using the interchange sort on image 1 to sort the whole array takes longer and would not be a fair comparison of the methods.

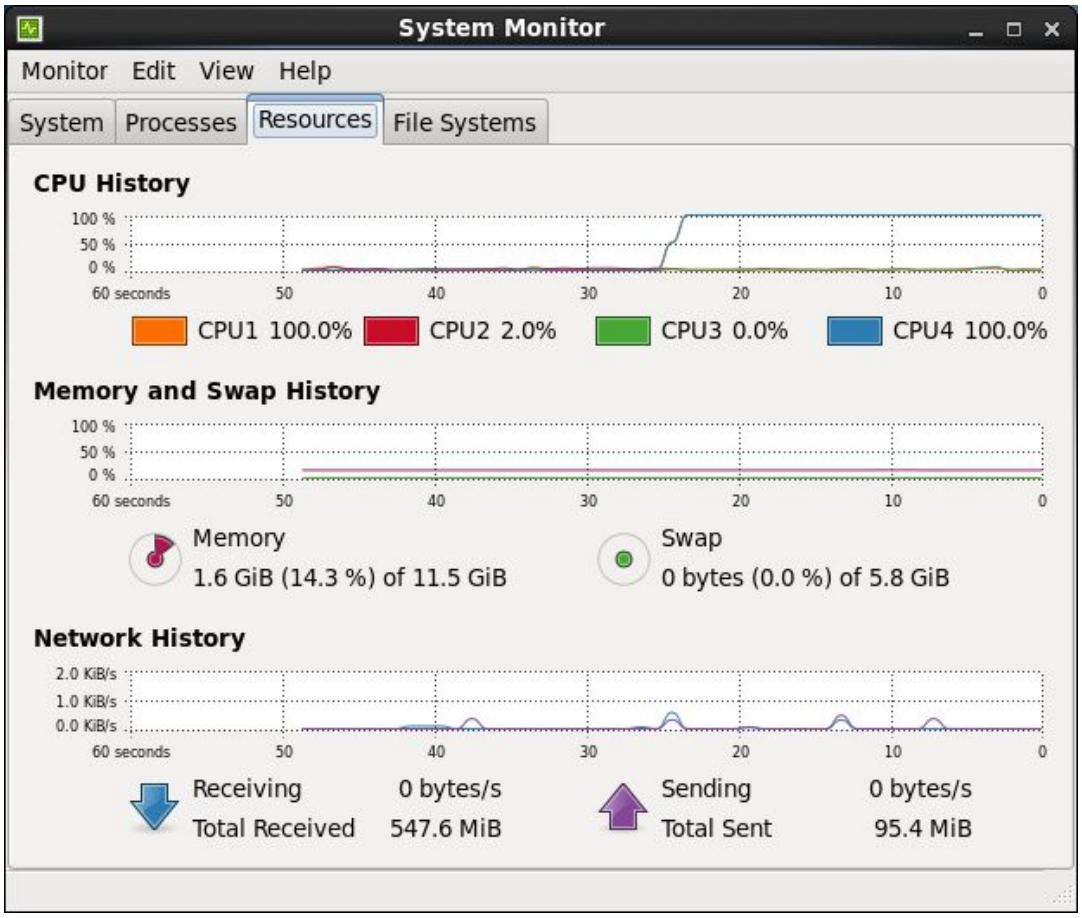
```
program sort2

   use sort_mod
   implicit none
   integer, parameter :: N = 200000
   real, dimension(N), codimension[*] :: a
   real, dimension(N) :: b
   integer :: start, stop, counts_per_second

   if (this_image() == 1) then
      call random_seed()
      call random_number(a)
      b = a
      call system_clock(start, counts_per_second)
   end if
   sync all

   if (this_image() == 2) then
      a(N/2+1:) = a(N/2+1:)[1]
   end if
   sync all

   select case (this_image())
   case (1)
      call interchange_sort(a(:N/2))
   case (2)
      call interchange_sort(a(N/2+1:))
   end select
   sync all

   if (this_image() == 1) then
      a(N/2+1:) = a(N/2+1:)[2]
      a = merge2(a(:N/2), a(N/2+1:))
      call system_clock(stop)
      print *, "For 2-image sort, time = ", &
            (stop - start) / counts_per_second, " seconds"
      print *, a(1), a(N), all(a(:N-1)<=a(2:))
   end if
```

```
    if (this_image() == 1) then
       call system_clock(start)
       call interchange_sort(b(:N/2))
       call interchange_sort(b(N/2+1:))
       b = merge2(b(:N/2), b(N/2+1:))
       call system_clock(stop)
       print *
       print *, "For 1-image sort, time = ", &
             (stop - start) / counts_per_second, " seconds"
       print *, b(1), b(N), all(b(:N-1)<=b(2:))
    end if

end program sort2
```

Here is the result of running this program. Note that the sorting time using two images is one-half of the time using one image.

```
For 2-image sort, time = 14 seconds
 5.8775768E-06 0.9999985 T

For 1-image sort, time = 28 seconds
 5.8775768E-06 0.9999985 T
```

A performance monitor provides more proof that execution is taking place on two of the computer's processors. One is shown while executing the program `sort2`, which uses two images. Note that two of the CPUs four processors are running at 100%, while the other two are almost idle.



14.8 Heat Transfer using Coarrays

Another example using coarrays is found in the program `caf_heat4` in the examples folder. It is more complicated, primarily the parts that involve updating the data along the borders of the quadrant of the plate. Build a project that uses `caf_heat4.f90` and study the code to learn more about coarrays. Running the program produced the following output.

```
For 4-image solution,  time =           45  seconds

Number of iterations (4 images):       32955

For 1-image solution,  time =          111  seconds

Number of iterations (1 image):       32808

Max difference between methods:   1.45575404E-03
```

The speedup is approximately 2.5x using four images.

15 OpenMP

Many modern computers have more than one processing unit. Even an inexpensive laptop usually has several "cores". Large computer clusters may have many thousands of them. The total elapsed running time of a program often can be shortened if different parts of the program can run simultaneously on different processing units.

OpenMP consists of a set of directives that can be inserted in a program to cause the computation to take place on multiple processors. (In fact, "MP" stands for multiple processors, and the "Open" part refers to the fact that the specification of OpenMP is available to anyone.) OpenMP directives may be used with Fortran, as well as C and other languages.

The purpose of this section is to show how to use OpenMP with the Fortran Tools and introduce some of the most basic OpenMP directives, just to get an idea how it works. There is an excellent tutorial covering the basics of OpenMP in the doc folder of the Fortran Tools distribution: "Parallel Programming in Fortran 95 using OpenMP" written by Miguel Hermanns.

15.1 A Simple Fortran Program Using OpenMP

Here is a simple example that doesn't do anything useful, except show how OpenMP works. Its features are explained in following subsections.

```
program omp_hello
   use omp_lib
   implicit none
   integer :: id, nthreads

   !$omp parallel private(id)
   id = omp_get_thread_num()
   print *, "Hello World from thread", id
   !$omp barrier

   if (id == 0) then
      nthreads = omp_get_num_threads()
      print *, 'There are', nthreads, 'threads'
   end if
   !$omp end parallel
end program omp_hello
```

15.2 Compiling from the Command Line

The program `omp_hello` may be compiled with the following command.

```
gfortran -fopenmp omp_hello.f90
```

If the program is compiled and linked with separate steps, the `-fopenmp` option must be given on both the compile command and the link command.

```
gfortran -fopenmp -c omp_hello.f90
gfortran -fopenmp omp_hello.o
```

This program also may be compiled and run using Code::Blocks. The `-fopenmp` option is set for compilation and linking by selecting the *Project* tab, then *Build options.* Select *Compiler settings* (if not already selected) and highlight the name of the project in the upper-lefthand corner so that the `-fopenmp` option is applied to all versions. Then check the box for *Enable the OpenMP extensions*. Next select *Linker settings* and add `-fopenmp` in the box to the right labeled *Other linker options*. Then select *OK*.

Running the program produces the following output.

```
Hello World from thread           1
Hello World from thread           3
Hello World from thread           2
Hello World from thread           0
There are           4 threads
```

15.3 Threads

The fundamental concept of OpenMP is to designate portions of a programs as *threads* and execute these threads on the physical processors available in the computer. In some cases more than one processor may be executing the same thread (possibly with different data) and sometimes a thread is executed by only one processor. The number of threads may be larger or smaller than the number of physical processors.

15.4 Some Basic OpenMP Features

The program `omp_hello` uses some simple OpenMP features. These features and a few others are described in this section.

15.4.1 OpenMP Functions

A subroutine that is available in OpenMP is `omp_set_num_threads()`; two functions that are available are `omp_get_num_threads()`, and `omp_get_thread_num()`. Calling `omp_set_num_threads` establishes the number of threads that will be used by some of the directives. There is a default number of threads used if this function is not used in a program. `omp_get_num_threads` indicates the number of threads in use and `omp_get_thread_num` indicates which thread is currently executing. (Be careful not to confuse these last two.)

The program `omp_hello` uses two of these functions to indicate the number of threads running and the number of each thread that is running. Note that the four threads are numbered 0, 1, 2, and 3.

15.4.2 The OpenMP Module

Each part of a program (main program, module, external procedure, etc.) that uses an OpenMP feature must have the statement

```
use omp_lib
```

15.4.3 OpenMP Directives

The special OpenMP directives are on a line that begins with `!$omp` (possibly preceded by spaces).

One of the basic directives is

```
!$omp parallel
```

which may be followed by some other information. This indicates that the portion of the program between this directive and

```
!$omp end parallel
```

is to be executed by all of the threads, in parallel when possible, and in any order.

Where an `!$omp barrier` occurs, each of the threads waits at that point until all of the threads have reached that point.

15.4.4 OpenMP Parallel DO

One of the most common and effective directives is `!$omp parallel do`. This indicates that the `do` loop which follows is to be split among threads. For example, if there are four threads, consider the loop

```
do n = 1, 113
```

Thread 1 might execute the loop for `n` = 1 to 28, thread 2 for `n` = 29 to 56, thread 3 for `n` = 57 to 84, and thread 4 for `n` = 85 to 113. Here is a simple program that illustrates the parallel loop.

```
module m
   use omp_lib
   implicit none

   integer :: k

contains
subroutine s()
!$OMP PARALLEL DO
   do k = 1, 10
      print "(2(a, i0))", "k: ", k, &
      "   thread: ", omp_get_thread_num()
   end do
!$OMP END PARALLEL DO
end subroutine s
end module m

program omp_test
use m
call s()
end program omp_test
```

Running this program produces

```
k: 4   thread: 1
```

```
k: 9   thread: 3
k: 7   thread: 2
k: 5   thread: 1
k: 1   thread: 0
k: 10   thread: 3
k: 8   thread: 2
k: 6   thread: 1
k: 2   thread: 0
k: 3   thread: 0
```

15.4.5 OpenMP Parallel Section

The directives `!$omp parallel sections`, `!$omp parallel section`, and `!$omp end parallel sections` indicate that different sections of the program are to be executed on different threads. Recall in the first example in this section, that with no directives, the entire program is executed by all of the threads. Here is a simple example.

```
program omp_sections

   use omp_lib
   implicit none
   integer :: n

   call omp_set_num_threads(2)
   !$omp parallel sections
   !$omp section
   do n = 11, 15
      print *, n, omp_get_thread_num()
   end do
   !$omp section
   do n = 21, 25
      print *, n, omp_get_thread_num()
   end do
   !$omp end parallel sections

end program omp_sections
```

Running this program produces the following output. Notice how the output from the two threads is mixed up.

```
          11           0
          21           1
          12           0
          22           1
          13           0
          23           1
          14           0
          24           1
          15           0
          25           1
```

15.4.6 The OpenMP Reduction Clause

Suppose the following loop is to be run in parallel.

```
do k = 1, size
   total = total + n(k)
end do
```

If a parallel DO directive is used, there is a potential problem that different threads may try to update the variable total at the same time. If it is made private to each thread, then the problem is adding the total from each thread to form the real desired total. These problems are solved by using the reduction clause on the DO directive, as seen in the following program.

```
program omp_reduction

   use omp_lib
   implicit none

   integer :: k, total = 0
   integer, parameter :: size = 100
   integer, dimension(size) :: n = [ (k, k = 1, size) ]

   !$omp parallel
   !$omp do reduction (+:total)
   do k = 1, size
      total = total + n(k)
      print *, k, omp_get_thread_num(), total
   end do
   !$omp end do
   !$omp end parallel

   print *, "Total =", total

end program omp_reduction
```

Running this program produces the following output. Note that each thread has its own running total, but all are added up after the loop is finished. For example, the next-to-last line of the output shows that the loop for k = 100 was executed by thread 3, and its total is only 2200, although the total after executing the loops is 5050.

```
           1           0           1
          51           2          51
          26           1          26
          76           3          76
           2           0           3
          52           2         103
          27           1          53
          77           3         153
           3           0           6
             . . .
          49           1         900
          99           3        2100
          25           0         325
          75           2        1575
          50           1         950
         100           3        2200

 Total =        5050
```

15.4.7 The OpenMP Private Clause

In the example in the previous subsection, the variable `total` cannot be a global variable in all the threads, because the different threads would be accessing it at different times in different orders or even accessing it at the same time.

Because it is listed in the reduction clause, a private copy is maintained for each thread (and the values from all threads are summed after the parallel section is finished). However, if this is needed for other variables, they can be listed in a `private` clause. This is illustrated in the program `omp_loops` discussed later in this section.

15.5 Timing an OpenMP Program

The intrinsic subroutine `system_clock` should be used to time an OpenMP program or other parallel program, such as one that uses MPI or coarrays. This is because `system_clock` records elapsed time on the system clock, where `cpu_time` adds the processing time of all the processors, thereby giving a time that is probably larger than the time to run the program sequentially (due to the overhead for splitting the program into threads executed on different processors). The subroutine `system_clock` returns "ticks" of the system clock and optionally the number of ticks per second. Thus, the elapsed time in seconds is the ending time minus the starting time divided by the number of ticks per second. The use of this subroutine to time some simple OpenMP programs occurs in the following subsections.

15.6 More Example OpenMP Programs

This section contains some OpenMP programs, in which the execution time for parallel execution is compared with that of serial execution. The programs were run on an Intel i5-4440 with four processors.

15.6.1 A Sort Using Sections

The program `omp_sort2` divides an array into two halves, sorts each half with the subroutine `interchange_sort` and then merges the two sorted halves with the function `merge2`. The running time is compared with doing the same thing using one processor.

The interchange sort is a very inefficient sort, but used to show the speedup for a process that takes some significant time. The interchange sort and merge routines are in the module `sort_mod`, which is part of the file `omp_sort2.f90` in the `examples` folder of the Fortran Tools distribution. The value of the function `all(a(:N-1)<=a(2:))` verifies that the array `a` is sorted into ascending order.

```
program omp_sort2

   use sort_mod
   use omp_lib
   implicit none

   integer, parameter :: N = 200000
```

```
    real, dimension(:), allocatable :: a, b
    integer :: start, stop, counts_per_second

    allocate(a(N), b(N))
    call random_seed()
    call random_number(a)
    b = a
    call omp_set_num_threads(2)
    call system_clock(start, counts_per_second)

    !$OMP PARALLEL SECTIONS
    !$OMP SECTION
    call interchange_sort(a(:N/2))
    !$OMP SECTION
    call interchange_sort(a(N/2+1:))
    !$OMP END PARALLEL SECTIONS

    a = merge2(a(:N/2), a(N/2+1:))
    call system_clock(stop)
    print *, "For 2-thread sort,  time = ", &
          (stop - start) / counts_per_second, " seconds"
    print *, a(1), a(N), all(a(:N-1)<=a(2:))

    call system_clock(start)
    call interchange_sort(b(:N/2))
    call interchange_sort(b(N/2+1:))
    b = merge2(b(:N/2), b(N/2+1:))
    call system_clock(stop)
    print *
    print *, "For 1-thread sort,  time = ", &
          (stop - start) / counts_per_second, " seconds"
    print *, b(1), b(N), all(b(:N-1)<=b(2:))

end program omp_sort2
```

Compiling this program with `-O3` and `-fopenmp` and running produces:

```
 For 2-thread sort,  time =           15  seconds
   2.50339508E-06  0.999993384     T

 For 1-thread sort,  time =           29  seconds
   2.50339508E-06  0.999993384     T
```

15.6.2 Matrix Multiply Using Parallel DO

The program in this section performs a matrix multiplication testing the `!$omp parallel do` directive.

```
module matrix_multiply

  use omp_lib
  implicit none
  private
  integer, parameter, public :: num = 2048
  integer, parameter, public :: dp = kind(0.0d0)
  public :: multiply_d, init_arr
```

```
contains

  subroutine multiply_d(a, b, c)
    real(kind=dp), dimension(num, num) :: a, b ,c
    integer :: i, j, k
    intent(in) :: a, b
    intent(out) :: c

    !$OMP PARALLEL DO
    do i = 1, num
      do j = 1, num
        c(i, j) = 0.
        do k = 1, num
          c(i, j) = c(i, j) + a(i, k)*b(k, j)
        enddo
      enddo
    enddo
    !$OMP END PARALLEL DO

  end subroutine multiply_d

  ! routine to initialize an array with data
  subroutine init_arr(row, col, off, a)
    real, intent(in) :: row, col, off
    real(kind=dp), dimension(num, num), intent(out) :: a
    integer :: i, j

    do i = 1, num
      do j = 1, num
        a(i, j) = row*i+col*j+off
      enddo
    enddo
  end subroutine init_arr

end module matrix_multiply

program omp_matrix

   use matrix_multiply
   implicit none
   real(kind=dp), dimension(:, :), allocatable :: a, b, c
   integer :: start, finish, counts

   allocate(a(num,num),b(num,num),c(num,num))
   ! initialize the arrays with data
   call init_arr(3.0, -2.0, 1.0, a)
   call init_arr(-2.0, 1.0, 3.0, b)

   ! start timing the matrix multiply code
   call system_clock (start, counts)
   call multiply_d(a, b, c)
!   c = matmul(a, b)
   call system_clock (finish)

   print *, sum(c)  ! To ensure computation
   print *, "Elapsed time =" , (finish - start) / counts, "seconds"
```

```
end program omp_matrix
```

Compiling this program with `-O3 -fopenmp` and running it produces the following output.

```
   3011225193545728.0
 Elapsed time =         35 seconds
```

Without either option, the elapsed time was 173 seconds. With just `-O3`, it was 79 seconds. With `-O3 -fopenmp` and the `i` and `j` loops interchanged inside the parallel do section, the time was 24 seconds.

This program is special because it is timing a matrix multiplication. The way to get really good performance is to use the `matmul` intrinsic function. Replacing the call to `matrix_multiply` with a call to `matmul` reduced the running time to six seconds!

15.6.3 A Final Example

For the last example, we examine the program `loops_modified` from the section on Timing and Profiling (13.4). This uses a parallel do for the outermost loop of the quadruple loops. Since the purpose of the loop is to sum some values in the arrays `A` and `C`, a reduction clause is needed. Also the variables `II` and `JJ` must be private to each thread. Only the quadruple loops in the main program are shown; the program is in the file `omp_loops.f90` in the `examples` folder of the Fortran Tools.

```
!$omp parallel private (II, JJ)
!$omp do reduction (+:A,C)
   do L = 1, N_COLS
   do J = 1, N_ROWS
      JJ = MAT(J, L)
      IF (JJ == 0) cycle
      do K = 1, N_COLS
      do I = 1, N_ROWS
         II = MAT(I, K)
         IF (II == 0) cycle
         A(II, JJ) = A(II, JJ) + B(I, J)
         C(II, JJ) = C(II, JJ) + D(I, J)
      end do
      end do
   end do
   end do
!$omp end do
!$omp end parallel
```

Running the program produced the following results.

```
 Setup time:   1.30700004     seconds
 Modified loop time:   0.607999980    seconds
   699112576.       702719360.
```

The running time for the quadruple loops was about 0.6 seconds, compared with about 2.1 seconds without the OpenMP directives.

16 GTK

GTK is software that allows a Fortran program to perform graphical input and output operations using windows containing such things as text boxes, slider bars, and radio buttons. None of the features described in this section are included in the 32-bit version of the Fortran Tools.

GTK stands for GIMP Tool Kit.

GTK-Fortran provides Fortran interfaces to the C GTK routines; they have been written by Jerry DeLisle and others.

The dialog that appears when the Fortran Tools are started was written using GTK-Fortran.

16.1 Setting Up GTK-Fortran

To use GTK-Fortran, the PATH environment variable (Section 1.6) must contain %FORTRAN_TOOLS%\bin; this is set during installation.

16.2 Using GTK-Fortran

Probably the best way to learn to use GTK-Fortran is to run some of the examples whose file names start with "hl" in the Fortran Tools examples\gtk folder and then study the code.

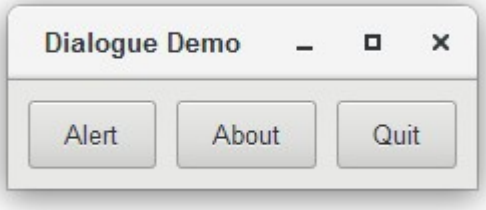
To run a program using GTK-Fortran with Code::Blocks, create a project as described in Section 2.3 and add one of the programs in the examples\gtk folder to the project. Make sure everything is set up as described in the previous section and build and run the program.

To run a GTK program using Code::Blocks, add %GTK% into the window viewed by *Project → Build options → Linker settings → Other linker options*. This is an environment variable set during installation.

Note: In this version of the Fortran Tools, a program that uses GTK cannot be run from the command line; use Code::Blocks as described above. To run a GTK-Fortran program from the command line, the include path must contain the GTK-Fortran module files and the library libgtkfortran.a must be linked. The following should work if the environment variable GTK is set as it is during installation.

```
gfortran hl_dialog.f90 %GTK%
```

Executing this program displays the following dialog box. The buttons are active, the window can be resized and moved, and it can be minimized, maximized, or closed in the usual way. Compile and run it so you can see for yourself.



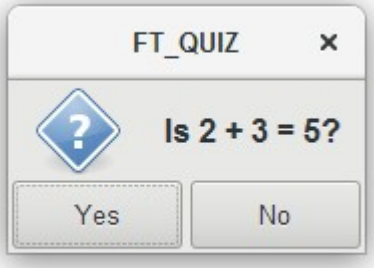
All of the high-level interfaces used in these examples are described in the GTK HTML document in the doc folder. Knowledge of Fortran interoperability with C (Section 5) is important and it probably helps to know some C, but it is not essential.

As yet, there is no comprehensive tutorial about using all of these interfaces, but a few of them are described in more detail later in this section.

16.3 A Simple Dialog

In order to see how to build programs that use GTK, we start with something even simpler than the dialog program in the previous section. This program produces a window with a couple of buttons that can be selected. Here is the program.

```
program ft_gtk_quiz

   use gtk_hl
   use gtk, only: gtk_init
   implicit none
   integer(kind=c_int) :: response
   type(c_ptr) :: quiz_window

   call gtk_init()

   quiz_window = hl_gtk_window_new()

   ! Display a window with a message
   ! and YES and NO buttons
   response = hl_gtk_message_dialog_show &
        (message = ["Is 2 + 3 = 5?"], &
         button_set = GTK_BUTTONS_YES_NO, &
         title = "FT_QUIZ"//c_null_char, &
         parent = quiz_window)

   if (response == GTK_RESPONSE_YES) then
      print *, "You are correct!"
   else
      print *, "Refresh your adding skills."
   end if

end program ft_gtk_quiz
```



Running this program produces the following output.

When one of the buttons is selected, a response is printed in the standard output.
A few comments on the code follow.

- The `use` statements access the high-level GTK routines and the interfaces to the base GTK routines. It is a good idea to have an `only` clause with `use gtk` because there are many routines in the `gtk` module that are not needed and using them could slow down the computation.
- Most of the uses of GTK routines eventually result in calls to C functions, so most values used should be made C interoperable. An example is `response`, which is declared to be a Fortran integer with kind interoperable with the C `int` data type. The `title` argument has a `c_null_char` concatenated at the end. The `message` argument is one case where a Fortran character string is what is required.
- The types of all the arguments for calls to the high-level procedures are given in the documentation for these routines.
- `gtk_init` always should be called before executing any other GTK procedures.
- Execution of the `response = ...` statement causes the window to be displayed and the response to the button assigned to the variable `response`.
- The file `gtk-enums-auto.f90` in the Fortran Tools `src\gtk` folder contains the definitions of the parameters `GTK_BUTTONS_YES_NO` and `GTK_RESPONSE_YES` (and others).

16.4 Windows and Widgets

The previous example is simple because it uses only one response window. More typical examples involve several components.

A GTK *window* usually has the capabilities present in windows: a border, a title, minimize, maximize, and delete buttons, sides and corners that can be selected to resize it, etc. A window may contain at most one *widget*, which can be a button, text box, slider bar, etc. It may be used to get input during execution of the program and display output. A program may generate more than one window.

A typical program creates a window, places an empty box in the window, and then places other widgets inside the empty box. This is illustrated in the program `ft_gtk_gas`, which allows the user to push a button and get the current price of gas. It has two active widgets, one containing the button and one displaying the price. The program is in `examples\gtk`.

The statement

```
price_window = hl_gtk_window_new &
     (title = "Gas price" // c_null_char, &
      wsize = [200_c_int, 60_c_int], &
      destroy = c_funloc(gas_destroy))
```

sets `price_window` as the name of a window; it does not display the window. It indicates that `gas_destroy` is to be executed if the window is deleted. `price_window` must be declared `type(c_ptr)`.

The statements

```
empty_box = hl_gtk_box_new(homogeneous=TRUE)
call gtk_container_add(price_window, empty_box)
```

create an empty box and insert it into the price window. The argument indicates that the widgets placed in the box will all be the same size (there are just two of them in this example). Widgets placed in the box will be arranged vertically; (`horizontal=TRUE`) would change that.

The statements

```
message_widget = hl_gtk_button_new &
     (label = "Click for current price"//c_null_char, &
      tooltip = &
         "The gas price is updated in real time"//c_null_char, &
      clicked = c_funloc(get_gas_price))
call hl_gtk_box_pack(empty_box, message_widget)
```

create a widget containing a button. When the button is clicked, the subroutine (C has only functions, hence the use of `c_funloc`) `get_gas_price` is executed. The tool tip is displayed when the user lets the cursor hover over the button. The widget is placed in the empty box.

The statements

```
price_widget = hl_gtk_text_view_new &
     (initial_text = [" "//c_null_char], &
      editable = FALSE)
call hl_gtk_box_pack(empty_box, price_widget)
```

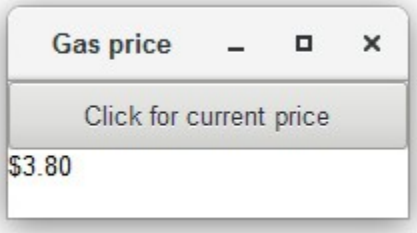
create a widget containing a message. The message is composed and displayed by the subroutine `get_gas_price`; the user may not modify the message directly. The widget is added to the empty box.

The statement

```
call gtk_widget_show_all(price_window)
```

displays the window, including the contained widgets.

The following shows the window after the message button has been clicked.



16.5 Event-Driven Programming

Traditional Fortran programs execute a statement and go on to the next statement to be executed. If there is input to be read, the program executes a `read` statement and processes the input for that statement. With event-driven programming, the program can be waiting for any number of inputs and when one occurs, the appropriate action is taken. This is how common windows-oriented programs work: the operating system is waiting for the cursor to be moved, the mouse to be clicked, or some text to be typed.

A program that responds to such a signal is called a *callback* or an *event handler*. In the program above, `get_gas_price` is an event handler that is called when the button in the message widget is clicked.

In `ft_gtk_gas`, the event handlers are placed in a module, which is a good place for all Fortran procedures. They are declared `public` so that the main program can reference them.

When using GTK-Fortran, event handlers are written as part of a Fortran program, but they are not called by the Fortran program; they are called by GTK responding to an event that occurs when the program is executed. This means that the interface (the arguments) of the event handler must match what is expected by GTK. For example, the procedures are `bind(c)` and the arguments are C interoperable and must have the characteristics indicated by GTK (just follow the examples).

One of the callback routines in `ft_gtk_gas is gas_destroy`.

```
subroutine gas_destroy(widget, gdata) bind(c)
   type(c_ptr), value :: widget, gdata
   call gtk_main_quit()
end subroutine gas_destroy
```

There must be two arguments and they must be declared as shown; note that they are never used. The subroutine simply executes `gtk_main_quit`, which cleans up everything.

The other event handler is `get_gas_price`.

```
subroutine get_gas_price(widget, gdata) bind(c)
   type(c_ptr), value :: widget, gdata
   character(len=20) :: price_text = "$"
   real, parameter :: base_price = 3.30
```

```
    real :: price
    call random_number(price)
    price = base_price + 0.5 * price
    write (unit = price_text(2:), fmt = "(f0.2)") price
    call hl_gtk_text_view_insert &
          (price_widget, [trim(price_text)], replace=TRUE)
end subroutine get_gas_price
```

The unused arguments must be declared as before.

It would be simpler to just put "$3.80" in the message widget, but to make things a little more interesting, a random price between $3.30 and $3.80 is generated instead. The random number generator returns a real value, but a character string must be displayed, so the internal `write` statement is used to convert the real value to a character value. The write starts in position 2 to preserve the "$" initialized in position 1.

Note that the text to be inserted must be an array. If there was anything there previously, it is replaced.

The program `ft_gtk_currency`, also in the `examples\gtk` folder, provides a little more complicated example that uses text boxes.

16.6 More Widgets

The example program `ft_gtk_rgb` uses radio buttons and sliders as input devices and a drawing as output. It also uses a table widget to arrange the slider buttons and label widgets to identify the sliders. Because it is a little larger than the example programs seen so far in this section, it is organized a bit differently. There is a module containing the handlers, another module containing the code to set up the window and widgets, and a third to specify things needed by both modules.

16.6.1 Radio Buttons

A radio button allows the user to select one of several options. The statement

```
type(c_ptr) :: radio_button_group = c_null_ptr
```

declares `radio_button_group` to be a widget that will hold one group of related buttons. The statement

```
square_button = hl_gtk_radio_button_new &
     (group = radio_button_group, &
      label = "Square"//c_null_char, &
      toggled = c_funloc(square_selected), &
      tooltip = "Select this button to make the " &
                //"shape a square"//c_null_char)
```

adds the square button to the empty group and the statement

```
call hl_gtk_box_pack(button_box, square_button)
```

places the button into the button box. Other statements create the circle button in the same group and place it in the same button box.

The names of the widgets are declared to be `type(c_ptr)`.

Here is what the radio buttons look like.

When one of the buttons is selected, the appropriate event handler is called. For the square button, the callback `square_selected` in the module `rgb_handlers` is called.

```
subroutine square_selected(widget, gdata) bind(c)
   type(c_ptr), value :: widget, gdata
   shape = square
   call redraw_figure()
end subroutine square_selected
```

All it does it set the variable `shape` to `square` and then call `redraw_figure`. `square` and `circle` are parameters used in the program to indicate which button is currently selected. The subroutine `circle_selected` is similar.

16.6.2 Tables

A table provides a rectangular grid into which other widgets can be placed. One of the benefits is that the cells in the table line up nicely. The following statement creates a table. The argument indicates that all the rows will be the same size. The columns will hold labels and sliders, which will have different sizes.

```
table = hl_gtk_table_new(row_homogeneous=TRUE)
```

The table will be added to the window after the sliders have been added to the table.

16.6.3 Labels

A label widget holds text that cannot be changed. The following statement creates a label widget named `label` that identifies the red slider

```
label = gtk_label_new("Red"//c_null_char)
```

and the statement

```
call hl_gtk_table_attach(table, label, &
         ix = 1_c_int, iy = 1_c_int)
```

inserts the label into the upper left-hand cell of the table (`ix` is the row; `iy` is the column).

16.6.4 Sliders

A slider allows the user to enter numerical input data by changing the position of the slider. A slider can input either integer or real values depending on the form of the statement that creates it. The statement

```
red_slider = hl_gtk_slider_new &
     (imin = 0, imax = slider_max, &
      initial_value = slider_max/2, &
```

```
        value_changed = c_funloc(red_slider_changed), &
        length = slider_length)
```

creates a slider that can input integer values between 0 and `slider_max`, has an initial value `slider_max/2` when first displayed, and has a length `slider_length`. When the slider is moved, the event handler `red_slider_changed` is called.

The statement

```
call hl_gtk_table_attach(table, red_slider, &
      ix = 2_c_int, iy = 1_c_int)
```

inserts the slider in row 1 and column 2 of the table.

There are similar statements to create the green and blue sliders.

Here is what the table containing the slider labels and the sliders looks like.

The statement

```
call hl_gtk_box_pack(empty_box, table)
```

inserts the entire table into the window.

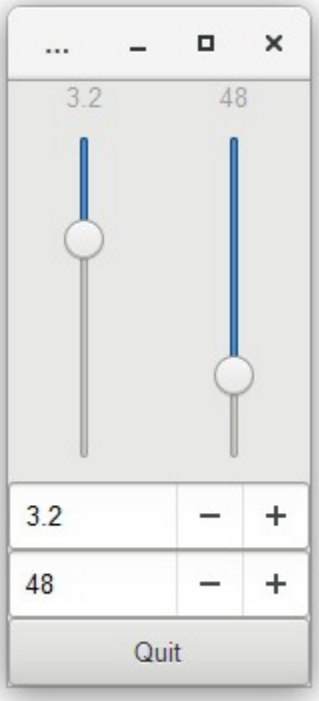
The event handler for each slider also is quite simple. Here is the one for the red slider.

```
subroutine red_slider_changed(widget, gdata) bind(c)
   type(c_ptr), value :: widget, gdata
   red = hl_gtk_slider_get_value(red_slider) / slider_max
   call redraw_figure()
end subroutine red_slider_changed
```

The function `hl_gtk_slider_get_value` returns the value of the slider, a real value between 0 and `slider_max`. It is divided by `slider_max` to get a real value between 0 and 1, which is what is required by the drawing program when setting the color. Then the figure is redrawn.

16.6.5 Spinners

A spinner is similar to a slider, but a value is set by selecting the "+" or "-" buttons. There are examples of spinners (and sliders) in `hl_sliders.f90` and `hl_sliders2.f90` in the `examples\gtk` folder of the Fortran Tools distribution. Here is the dialog produced by `hl_sliders.f90`; moving either a slider or a spinner moves the corresponding spinner or slider.



16.7 Drawing

The program `ft_gtk_rgb` draws either a square or a circle depending on which value of the radio button is selected. The color of the figure is given by the values of the red, green, and blue sliders, each between 0 and `slider_max`. But note that the values of the *variables* `red`, `green`, and `blue` used to set the slider are real values between 0 and 1.

The drawing in this program is done by routines from a collection of programs called "Cairo". Modules containing interfaces to these programs must be used by the drawing program. These `use` statements can be found in the program in the examples folder.

The drawing tools provided with GTK-Fortran are described on a separate page linked at the bottom of the main page of the GTK-Fortran high-level documentation in the `doc` folder of the Fortran Tools distribution. There is also documentation of the Cairo drawing tools used by GTK-Fortran at

```
www.cairographics.org/documentation
```

Parts of the subroutine `redraw_figure` are shown and explained.

The following statements create a drawing area named `figure` and paint the whole area white.

```
figure = hl_gtk_drawing_area_cairo_new(drawing_area)
call cairo_set_source_rgb(figure, &
```

```
      1.0_c_double, 1.0_c_double, 1.0_c_double)
! start with white screen
call cairo_paint(figure)
```

The next statement sets the color to be used later when drawing a figure.

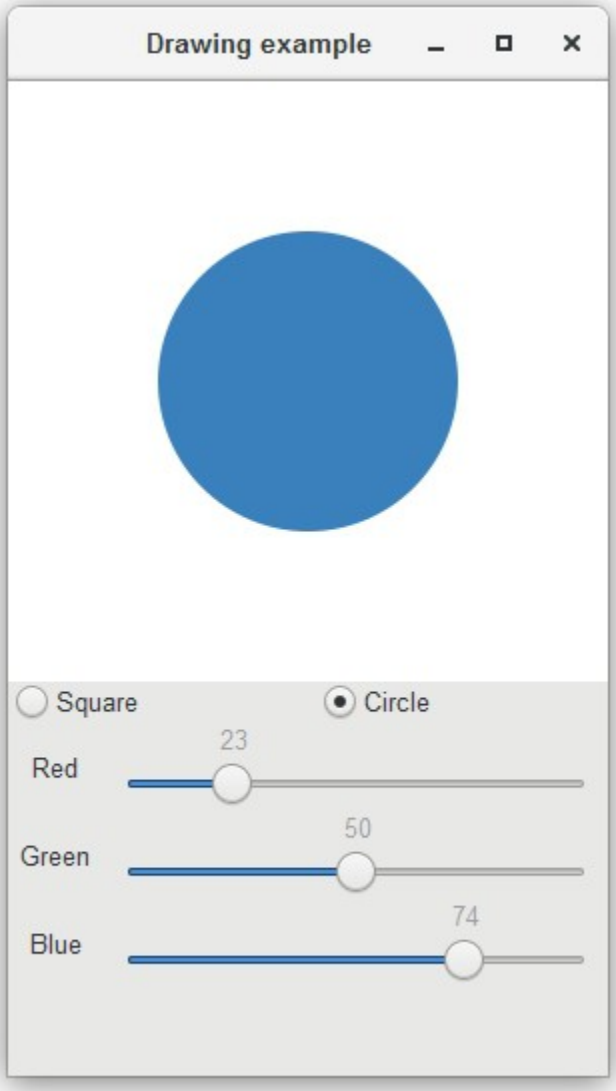
```
call cairo_set_source_rgb(figure, red, green, blue)
```

The next statements create either a square or circle, depending on which radio button has been selected. `radius` is a parameter declared in the subroutine; it is one-fourth of the parameter `width`, which determines the size of the window.

```
select case (shape)
case (square)
   call cairo_rectangle(figure, &
         radius, radius, &  ! upper left-hand corner
         2*radius, 2*radius) ! length of sides
case (circle)
   call cairo_arc(figure, &
         2*radius, 2*radius, & ! center of circle
         radius, &
         0.0_c_double, 2*pi_double) ! angle range
end select
```

The next statements fill the figure (square or circle) with the color selected, release the descriptor for the drawing area, and draw the figure on the computer screen.

```
call cairo_fill(figure)
call hl_gtk_drawing_area_cairo_destroy(figure)
call gtk_widget_queue_draw(drawing_area)
```



This is what the entire window looks like after it is drawn.
This project called `rgb` is in the `projects` folder.

16.8 Glade

Glade is a program that allows you to create a GTK-Fortran graphical interface using a graphical interface. Glade runs as a completely independent program using either Code::Blocks or a command line. The output from Glade is an XML file that can be read by

a program that uses GTK to establish the form of the windows that are used. The handlers still must be written as part of the Fortran program.

For simple cases, writing the code to build the GUI is not much more difficult than writing the code that incorporates the Glade output into a Fortran program, but when the graphical interface is complicated, Glade greatly helps getting all of the parts of a window set up correctly.

Thus, using Glade to create the interface and GTK-Fortran to interact with the interface makes an effective combination.

Unfortunately, there is not sufficient documentation for Glade. Some can be learned by looking at the examples in this section and by looking at both the examples and the source code that comes with GTK-Fortran.

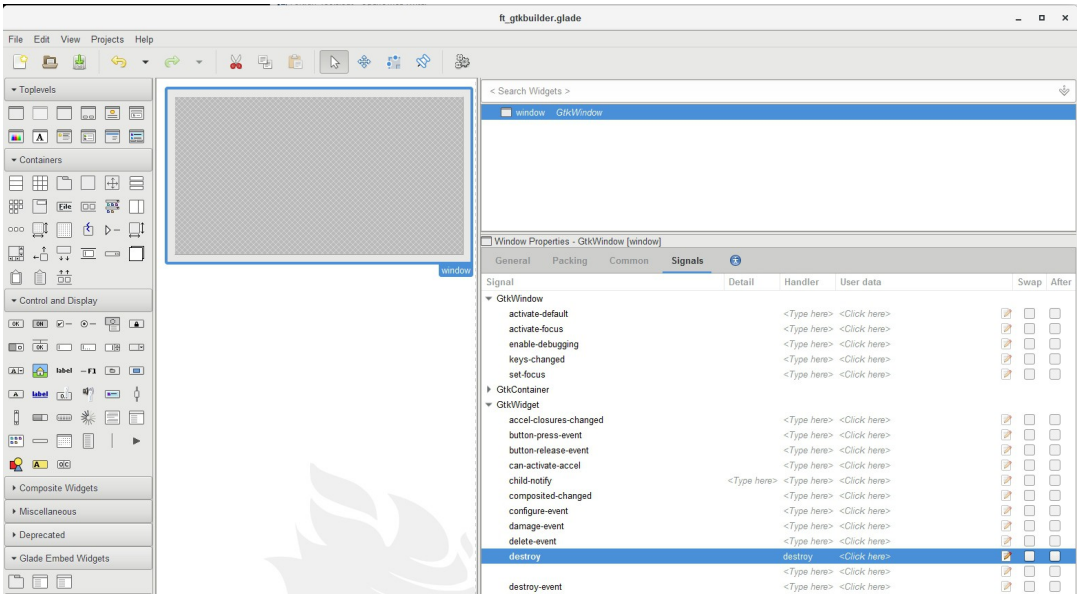
In the `examples\gtk` folder, there is a Glade file `gtkbuilder.glade` that is used by the program `gtkbuilder.f90`. That program can be compiled and run to see how it works. In this section, we will show how to run Glade and produce a Glade file that is similar to `gtkbuilder.glade`. The file produced is called `ft_gtkbuilder.glade` and is used by the program `ft_gtkbuilder.f90`, which is similar to `gtkbuilder2.f90`.

The Glade and Fortran programs described in this section also are contained in the `examples\gtk` folder of the Fortran Tools distribution.

The Glade program is in `..\Fortran_Tools\bin`. This can be executed by simply typing `glade` in a command window because that directory has been added to your path (Section 1.6) or you may create a shortcut (Section 1.7) that can be selected. This produces a graphical interface. The first thing to do is save it as `ft_gtkbuilder` (the file name extension `.glade` will be added) in a directory of your choice.

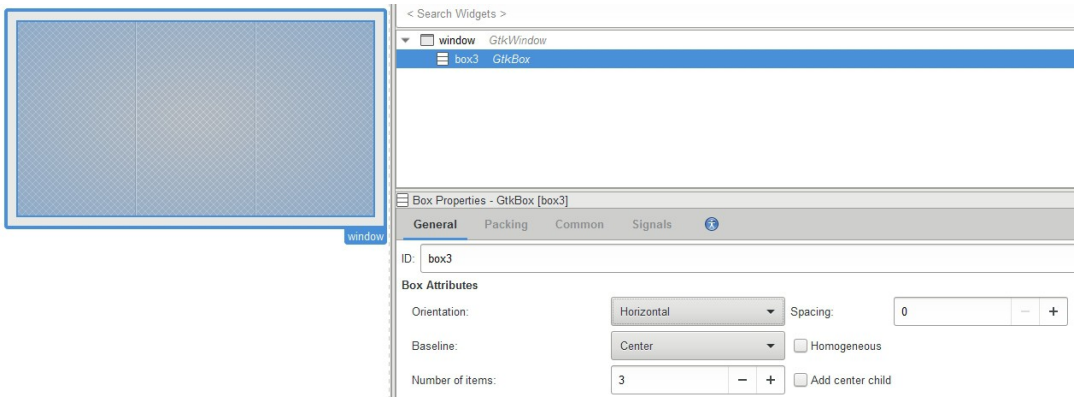
In the project, create a window by selecting the window icon (upper left icon under *Toplevels*). Select *General* and set the *ID* of the window as `window` and set the *Title* under *Appearance* to `My Title`. Set the border width by selecting *Common* and entering the value `10` for *Border width* at the bottom of the pane. When results of this are displayed, there will be a menu bar at the top; it does not show here.

To establish the handler that is called when the window is destroyed, select the *Signals* tab. Move down in the display until you see *GtkWidget*. Expand that list and look for *destroy*. In that row, to the right where is says *<Type here>*, enter `gtk_main_quit`. Using Glade, it is not necessary to define a handler in the Fortran program that does nothing but call `gtk_main_quit`.



We want to put three buttons into the window, but before that can be done, three containing windows must be established. From the icons in the *Containers* category, select the *Box* icon (at the upper left corner). When it is selected, a border is drawn around it. Drag it into the shaded window and click to place the window. In the little dialog that is displayed, enter `3` for the *Number of items*, and select *Create*. A new item, *GtkBox*, should appear in the upper right pane, which shows the widgets that will be placed on the screen.

In the *General* category, set the *ID* to `box3` and change the orientation to `Horizontal`.

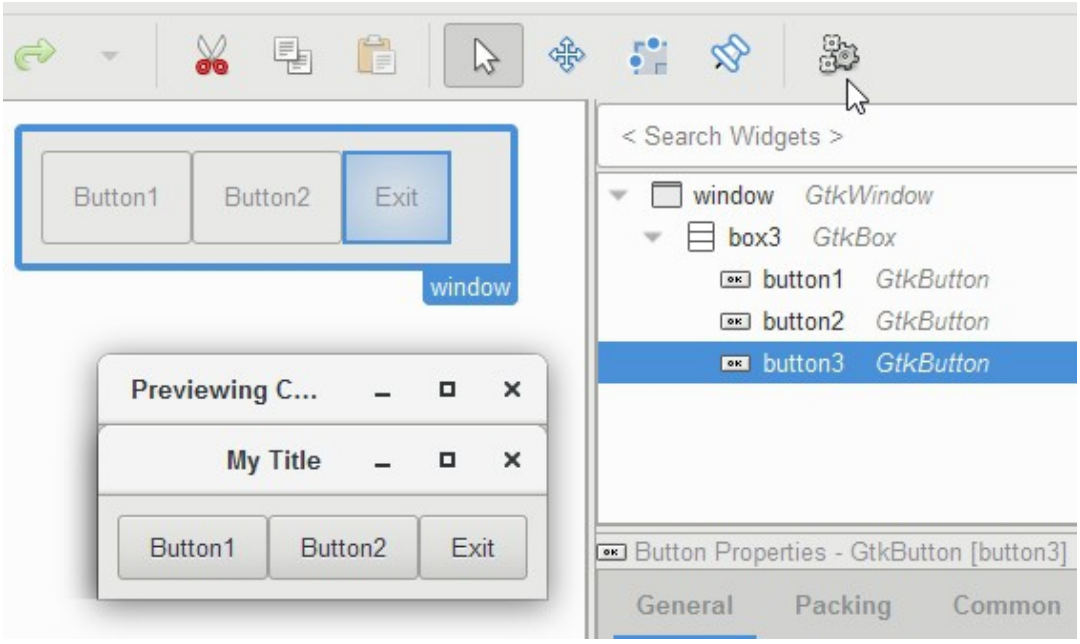


Now the three buttons can be placed in the box. Select *box3* in the upper pane on the right. In the panel of *Control and Display* widgets, select the *Button* icon (labeled *OK* in the upper left corner). Drag it into the leftmost display window and click. Under the *General* category of *Button Properties*, set the *ID* to `button1`, change the *Label* to `Button1`. Select the *Signals* tab and under the *GtkButton* category, set the handler for *clicked* to `hello`. Create a

second and third button similarly. Set the *ID* of the second button to `button2`, set the label to `Button2`, and set its handler to `button2clicked`. Set the ID of the third button to `button3`, set its label to `Exit`, and set its clicked handler to `gtk_main_quit`.

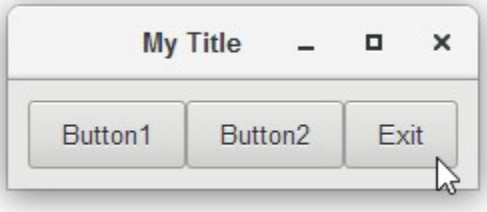
By grabbing and dragging the corner of the display, make it smaller. Save it.

By selecting the gear icon at the right end of the menu bar, a preview of the window will be displayed.



Now the program `ft_gtkbuilder.f90` can be compiled and run. Make sure the Glade file is available by putting it in the same folder with the executable or in the main project folder when using Code::Blocks.

```
gfortran ft_gtkbuilder.f90 %GTK% -o gtkbuilder.exe
```




```
Hello World!
Hello World!
Button 2 clicked!
```

The next sections will show examples of some of the other widgets that can be used by Glade.

16.8.1 Displaying the Current Time

For this example, a GUI is constructed that allows the user to display the current time in any of the zones of the continental USA with a choice of two formats.

Execute `glade` by typing `glade` or double clicking on `glade.exe` in the directory ..\`Fortran_Tools\bin`.

Open a *New* project and save it as `time.glade`.

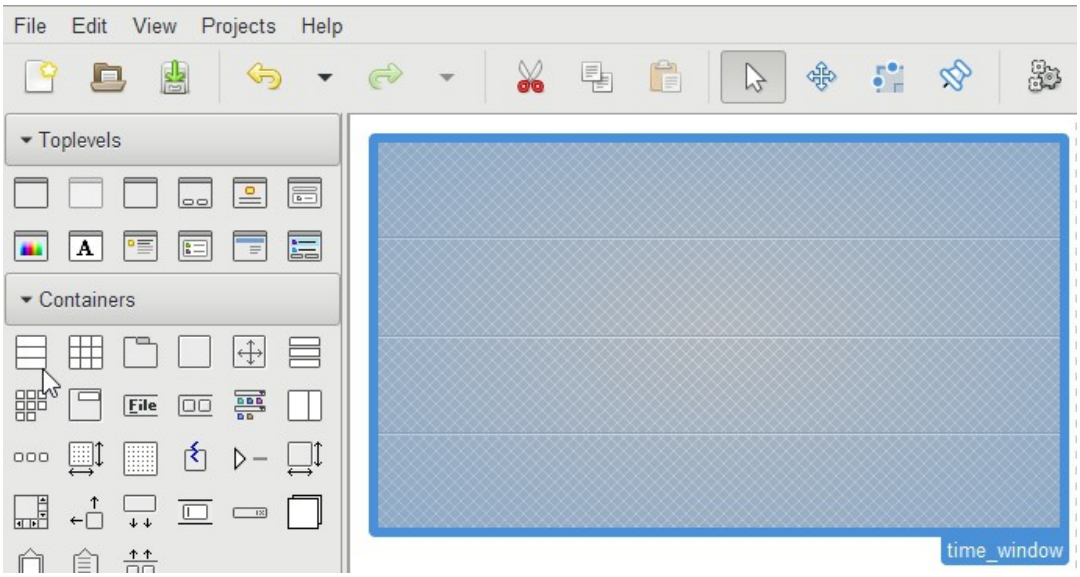
Create a window for the entire display: Click on *Toplevels* → *Window* (upper left).

Set the name of the window: *General* → *ID*: `time_window`. Set the title (under *Appearances*) to `TIME`.

Establish the subroutine to call when the window is deleted: *Signals* → *GtkWidget* → *destroy* → `gtk_main_quit`. If necessary, expand the list under *GtkWidget* by clicking on the little down arrow to the left.

Put a vertical row of boxes in the window: *Containers* → *Box*. When it is placed in the window, select 4 in the dialog box to create four boxes. *Create*.

Set the box *ID* to `vbox`. (*General* → *ID*)



16.8.2 The Label Widget

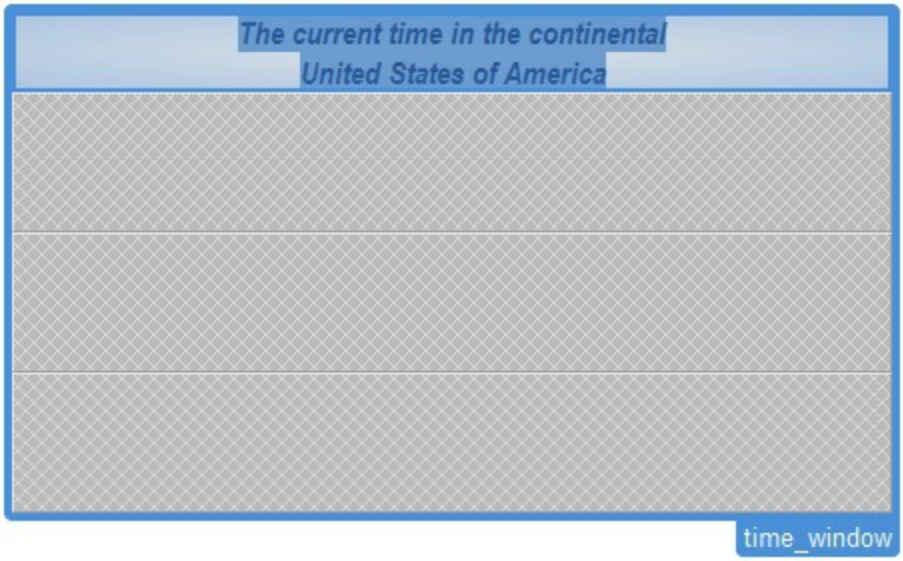
Put a text label in the top box: *Control and Display* → *label* (the one with no underline).

General → *ID*: `time_label`.

General → *Appearance* → *Label*:

```
The current time in the continental
United States of America
```

General → *Appearance* → *Attributes* → *Edit Attributes*
Font Description: `Arial Narrow Bold Oblique 12`
Foreground Color: Pick a dark blue
Background Color: Pick a light blue
Absolute Size: `100`
OK
General → *Formatting* → *Justification:* `Center`
General → *Alignment and Padding* → *Padding*: `10` (Select the box)



16.8.3 Text View

The time will be displayed in the second box. This is a little more complicated because the Fortran program does not get data from the GUI, but must place the time into the GUI. This is done with a Text View widget.

Control and Display → *Text View*. It is the icon that looks like a page with lines of text on it. Place it in the second box.

General:
ID: `time`
Monospace: `OFF`
Purpose: `Free Form`
Editable: `No` (just click on the box)
Justification: `Center`
Margins → *Top:* `15`
Margins → *Bottom*: `15`
Common → *Height request*: → `30` (and check the box)

16.8.4 Tables

Next, four items are to be arranged in a table, called a *grid* in Glade. The boxes for the table are placed in the third row.

Containers → Grid. The Grid icon looks like a 3x3 tic-tac-toe board. Place it in the third row of the window. In the dialog box that appears, select two rows and two columns. *Create*.

16.8.5 Radio Buttons

In the top row of the table, two radio buttons will be placed so that one of two time formats may be selected when the program is run. Radio buttons have the property that when any one button in a group is selected, the others are deselected, so that exactly one button is always selected. The icon for the radio button has a circle with a solid circle inside it.

Control and Display → Radio Button. Place it in the upper-left section of the table.

General

ID: `radio_button_24`

Button content → Label with optional image → Label: `24-hour format`

A tool tip appears when the user of the GUI lets the mouse hover over an item.

Common

Tooltip Text: `Select this button for 24-hour time format.`

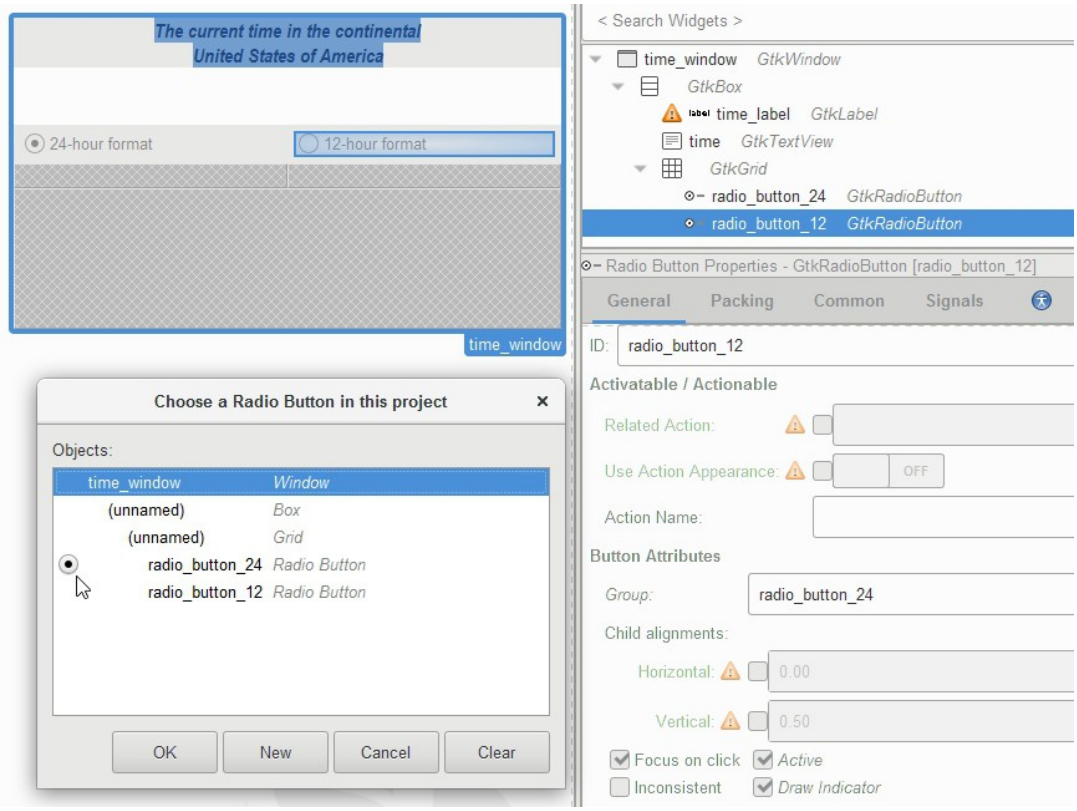
Border width: `5`

Signals → GtkButton → clicked: `radio_button_24_clicked`. This is the Fortran subroutine that will be executed when this button is selected.

Put a second radio button in the upper-right section of the table and change all "24" to "12".

It is necessary to indicate that the two radio buttons are in the same group, as follows:

General → Group. Click on the pencil icon at the right end of the box. In the dialog that opens, select the radio button to the left of *radio_button_24*. *OK*. The group will then be shown in the properties window for the 12-hour button, but not the 24-hour button.



16.8.6 Combo Box

A combo box provides a selection from a list of items. In our case, the time zone in the continental United States is to be selected.

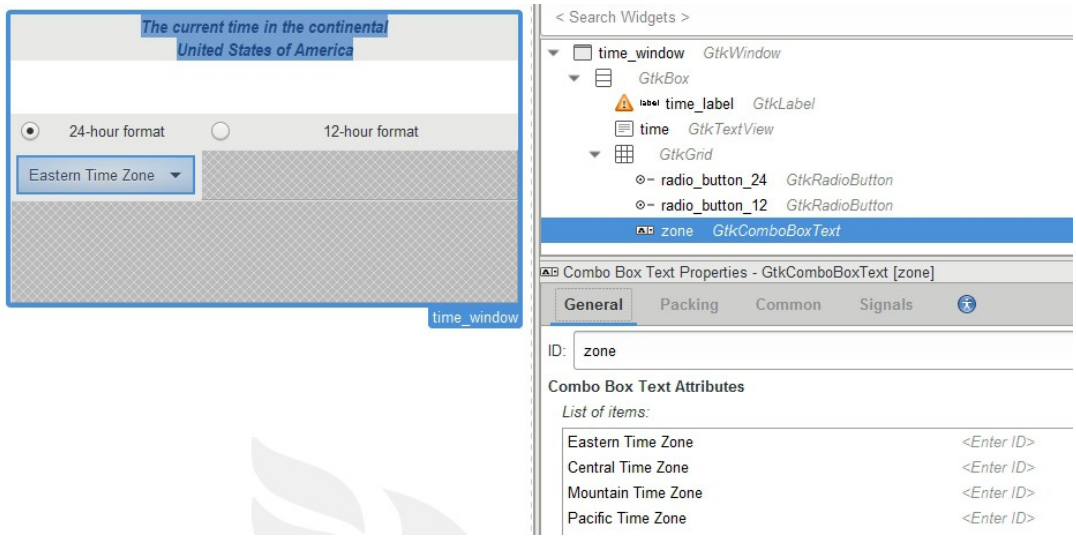
Control and Display → *Combo Box Text* (containing the letter "A" and the down arrow). Place it in the lower-left section of the table.

General → *ID*: `zone`.

General → *List of items*: Select the pencil icon to the right and type in `Eastern Time Zone`, `Central Time Zone`, `Mountain Time Zone`, and `Pacific Time Zone`.

Common → *Border width*: 5

Signals → *GtkComboBox* → *changed* → `zone_changed`.



16.8.7 Check Button

The check button in the example will indicate daylight saving time.

Control and Display → *Check Button* (the one with the check symbol). Place it in the lower-right section of the table.

General

ID: `dst_button`

Label: `Daylight Saving Time`

Common

Border width: `5`

Tooltip Text: `Check this button if daylight saving time is in effect.`

Signal → *GtkToggleButton* → *toggled* → `dst_button_toggled`

16.8.8 Button

When this last button is selected, the time is displayed.

Command and Display → *Button* (the one with "OK"). Place it in the bottom section of the window.

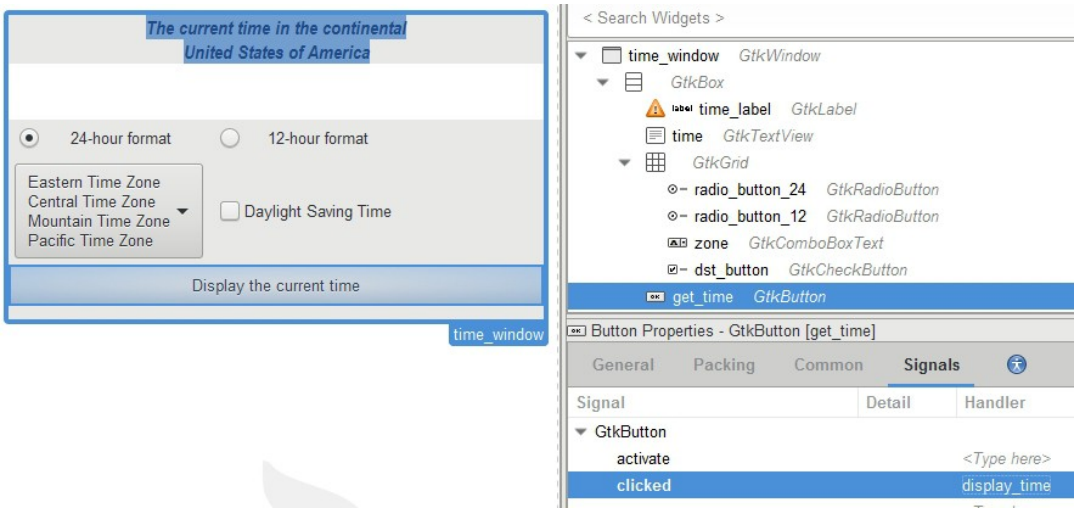
General

ID: `get_time`

Label: `Display the current time`

Signals → *GtkButton* → *clicked* → `display_time`

Here is what the Glade program displays at this point. Note the "tree" view of the parts of the window shown on the right.



16.8.9 The Fortran Program that Displays the Time

The Fortran program `time.f90` is In the `examples\glade` folder of the Fortran Tools. Some of the interesting parts are discussed here.

First, the program must read the data created by Glade in the file `time.glade`.

```
builder = gtk_builder_new()
b_int = gtk_builder_add_from_file(builder, &
   "time.glade"//c_null_char, error)
if (b_int == 0) then
   print *, "Could not open time.glade"
   stop
end if
```

Then, connections between the Glade objects and the Fortran program must be established.

```
time_window = gtk_builder_get_object &
   (builder, "time_window"//c_null_char)

time = gtk_builder_get_object &
   (builder, "time"//c_null_char)

zone = gtk_builder_get_object &
   (builder, "zone"//c_null_char)

call gtk_builder_connect_signals(builder, c_null_ptr)

call g_object_unref(builder)
```

Subroutines specified by the handlers in Glade are called when something changes. Only one of these handlers is shown. All it does is change the value of the variable `hour_format`.

```
subroutine radio_button_24_clicked(widget, gdata) bind(c)
   !GCC$ ATTRIBUTES DLLEXPORT :: radio_button_24_clicked
   type(c_ptr), value :: widget, gdata
   hour_format = 24
end subroutine radio_button_24_clicked
```

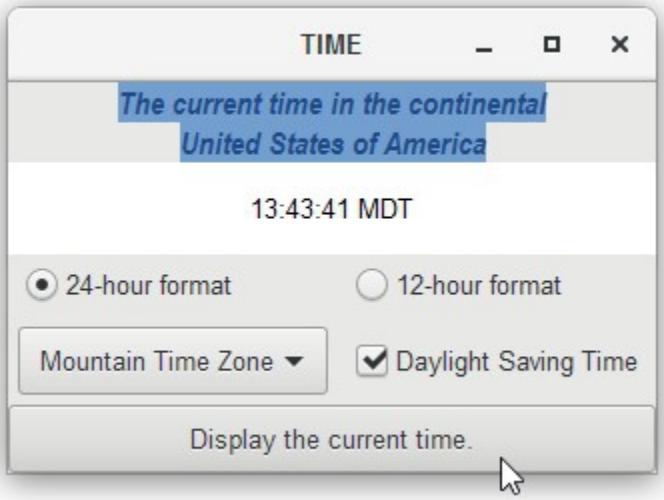
The subroutine to compute the time uses the Fortran intrinsic subroutine `date_and_time` to create the time as a string `time_text` and the following statement displays the time in the GUI. Note that the text is an array, but with only one element in this case.

```
call hl_gtk_text_view_insert &
     (time, [trim(time_text)], replace=TRUE)
```

All of these routines are put in a module which is used by the main program.

```
program time
   use gtk_hl
   use gtk, only: &
      gtk_init, &
      gtk_main
   use time_mod
   implicit none

   call gtk_init()
   call set_up_handlers()
   call gtk_widget_show_all(time_window)
   call gtk_main()
end program
```

Here is what is displayed by the running program.



16.8.10 Setting the Computation Speed

This next program itself is not very useful, but does illustrate the use of three more widgets, the file chooser, the progress bar, and the slider bar.

To follow along with this example, create a new Glade project saved as `file_chooser.glade` and insert a *Toplevel* window with ID `file_chooser_window`. Under *Appearance*, uncheck the box *Client Side Window Decorations* and make the *Title* `Select a file to sort`. Make the *Default Width* `600` (and check the box). Under the *Common* tag, make the *Border Width* `10`. For the *GtkWidget* signal *destroy*, enter the subroutine name `gtk_main_quit`.

In the container window, place a *Box* with three horizontal sections.

16.8.11 The File Chooser

In the top section of the window, place a *File Chooser Button*. It is the icon in the *Control and Display* section that is a rectangle with a file folder inside. Name it `file_chooser`. Make the *Title* `Select a file`. Make the *Width in Characters* `20`. Set the *FtkFileChooserButton file-set* signal to `file_selected`.

16.8.12 The Progress Bar

In the middle section, place a progress bar. It is the *Control and Display* icon that is about 2/3 shaded gray. Name it `progress_bar`. Check the box *Show Text* and use `Progress of sort` for the *Text*.

16.8.13 The Slider Bar

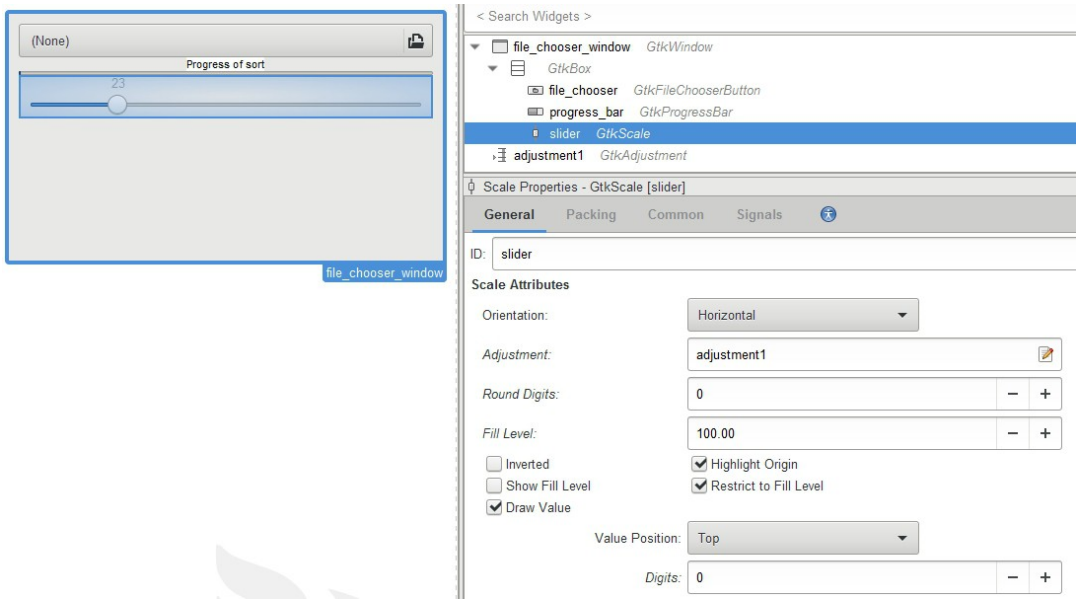
In Glade, what is often called a "slider bar" is called a "scale". Place a *Scale* in the lowest section of the window. Name it `slider`. Set the *Orientation* to `Horizontal`. Set *Round Digits* to `0`. Set the *Fill Level* to `100`. Check the *Draw Value* box, set the *Value Position* to `Top`, and set *Digits* to `0`.

And here is the tricky part: Under the *General* tab, select the icon at the right end of the *Adjustment* entry. When the dialog box appears, select *New*. An entry named `adjustment1` will appear in the list of windows entities and in the *Adjustment* entry for the slider. Select this to edit its properties and change *Minimum Value* to `1.00`. Move the slider to make sure that it will change. All four sliders and spinners need an *Adjustment*.

Add a *Tooltip* if you want one.

Under *Signals*, expand the list for *GtkRange* and set *value-changed* to `delay_changed`.

Save the Glade file.

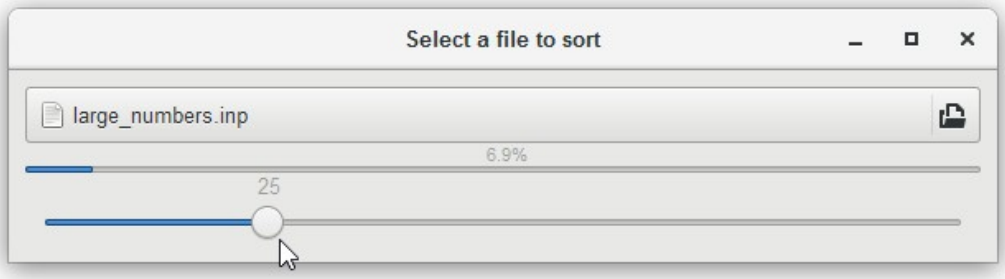


16.8.14 The Fortran Sorting Program

In the program `file_chooser.f90`, the connections between the Glade objects and the Fortran subroutines are established as before. In this case, we need handlers for the file chooser and the progress bar.

Two files are available in the `examples\gtk` folder to test the file sorting program: `small_numbers.inp` and `large_numbers.inp`. When the file chooser is selected, one of these files may be selected using the file chooser browser, which is similar to the usual Windows file explorer.

When the file is chosen, the numbers are sorted. The slider bar can be used to adjust the speed of the sort, either before selecting the file or during the sorting process.



After the numbers are sorted, the following is printed in a console window. It indicates that the numbers were successfully sorted.

```
The input file is
C:\Fortran_Tools\examples\gtk\large_numbers.inp
The         200 numbers are  sorted
The median number is   5.52902985
```

The file `large_numbers.inp` consists of random numbers from 0 to 10, so a median of about 5 is reasonable. The following statement tests to determine if the numbers are sorted.

```
sorted = merge("    ", "not ", all(a(:n-1)<=a(2:)))
```

The variable `k` is incremented each time the inner sorting loop is executed and is used to position the progress bar; the progress in the bar is displayed as the fraction $k/((n^2-n)/2)$ divided by 100 as $(n^2-n)/2$ is the number of steps needed to do the sort (it is a very inefficient sort) and the progress bar was set to have 100 steps.

The following code is executed each iteration through the outer sorting loop. The call to `pending_events` is necessary to make the whole thing work. Just copy the subroutine as it appears in the example.

The call to `g_usleep` suspends the computation for `delay` microseconds. Thus, the larger the value of `delay`, which is set by moving the slider bar, the slower the sorting speed.

Finally, the call to `hl_gtk_progress_bar_set`, displays the green progress bar.

```
call pending_events()
if (run_status == FALSE) exit
call g_usleep(delay)
call hl_gtk_progress_bar_set &
   (progress_bar, real(k,c_double)/((n**2-n)/2), string=TRUE)
```

A Software License Agreement

Read the terms and conditions of this license agreement carefully before installing the Software on your system.

By installing the Software you are accepting the terms of this Agreement between you and The Fortran Company. If you do not agree to these terms, promptly destroy all files and other materials related to the Software.

"Software" means the programs developed by The Fortran Company. The Fortran compiler, Code::Blocks, and other programs not developed by The Fortran Company are licensed under agreements with their developers.

The Fortran Company grants to you a nonexclusive license to use the Software with the following terms and conditions:

The Fortran Company retains title and ownership of the Software. This Agreement is a license only and is not a transfer of ownership of the Software.

The Software is copyrighted. You may copy the software provided that you include a copy of this license.

You may adapt and modify any source programs, but may not reverse engineer any object or executable files.

You may not sell, rent, or lease the software.

This license is effective until terminated by The Fortran Company. It will terminate automatically without notice if you fail to comply with any provision of this license. Upon termination, you must destroy all copies of the Software.

The failure of either party to enforce any rights granted under this Agreement or to take action against the other party in the event of any breach of this Agreement will not be deemed a waiver by that party as to subsequent enforcement of rights or subsequent action in the event of future breaches. If applicable statute or rule of law invalidates any provision of this Agreement, the remainder of the Agreement will remain in binding effect.

THE FORTRAN COMPANY MAKES TO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE OR PERFORMANCE OR ACCURACY. THE FORTRAN COMPANY SHALL IN NO EVENT BE LIABLE TO THE LICENSEE FOR ANY DAMAGES (EITHER INCIDENTAL OR CONSEQUENTIAL), EXPENSE, CLAIM, LIABILITY, OR LOSS, WHETHER DIRECT OR INDIRECT, ARISING FROM THE USE OF THE SOFTWARE.

Index

I

K

L

M

N

O

P

Q

R

S

T

U

V

W