Pro Fortran

Windows™

User Guide

For 32-bit and 64-bit Windows

absoft
development tools and languages

Pro Fortran

Windows™

User Guide

For 32-bit and 64-bit Windows

absoft
development tools and languages

2075 West Big Beaver Road, Suite 250
Troy, MI 48084
U.S.A.
Tel (248) 220-1190
Fax (248) 220-1194
support@absoft.com

Contents

CHAPTER 1

Introduction

INTRODUCTION TO ABSOFT PRO FORTRAN

Absoft specializes in the development of Fortran compilers and related tools. Full implementations of Fortran 77 and Fortran 90/95 are available for Windows, Macintosh and Linux platforms. Absoft will continue to focus on Fortran in the future, but the popularity of C/C++ in the Unix environment has required many of today's Fortran programmers, who are moving code to their desktop, to link Fortran code with C libraries. To facilitate this process, certain Absoft Fortran implementations are object code compatible with C/C++ objects, allowing users to create mixed Fortran/C applications from within a single development environment.

Absoft Fortran is a native 32-bit application designed for Windows™ XP/Vista/7.

Absoft Fortran implementations include all of the tools necessary for you to create stand-alone, double-clickable Windows applications. The purpose of this User Guide is to offer step-by-step instructions on the operation of each compiler, writing, compiling, debugging, linking and running your program. AWE, Absoft's application framework, can automatically build a standard Windows interface for each compiled application.

Absoft Fortran 90/95

A complete ANSI Fortran 90/95 implementation plus extensions, Absoft Fortran 90/95 is the result of a five year joint development effort with Cray Research. It utilizes a version of the CF90 front-end and is source compatible with several Cray F90 releases. It provides full support for the Win32 API directly from Fortran and is capable of building DLLs. Several popular VAX and workstation extensions have also been added.

Absoft FORTRAN 77

Refined over 15 years, with emphasis on porting legacy code from workstations, Absoft Fortran 77 is full ANSI 77 with MIL-STD-1753, Cray-style POINTERs, plus most extensions from VAX FORTRAN as well as many from IBM, Sun, HP, and Cray. Absoft Fortran 77 supports legacy extensions which are not part of the Fortran 90/95 standard. See the chapter on **Porting Code** in this manual for further information. Fortran 77 is fully link compatible with Fortran 90/95 so existing, extended FORTRAN 77 routines can be easily compiled and linked with new Fortran 90/95 code. The entire Win32 API is supported and DLLs can be created directly from Fortran.

CONVENTIONS USED IN THIS MANUAL

There are a few typographic and syntactic conventions used throughout this manual for clarity.

- `[]` square brackets indicate that a syntactic item is optional.
- ... indicates a repetition of a syntactic element.
- Term definitions are underlined.
- **-option** font indicates a compiler option.
- *Italics* is used for emphasis and book titles.
- On-screen text such as menu titles and button names look the same as in pictures and on the screen (e.g. the **File** menu).
- The modifier keys on PC keyboards are Shift, Alt, and Control. They are always used with other keys and are referenced as in the following:

Shift-G	press the Shift and 'G' keys together
Alt-F4	press the Alt and F4 function keys together
Control-C	press the Control and 'C' keys together

- Unless otherwise indicated, all numbers are in decimal form.
- FORTRAN examples appear in the following form:

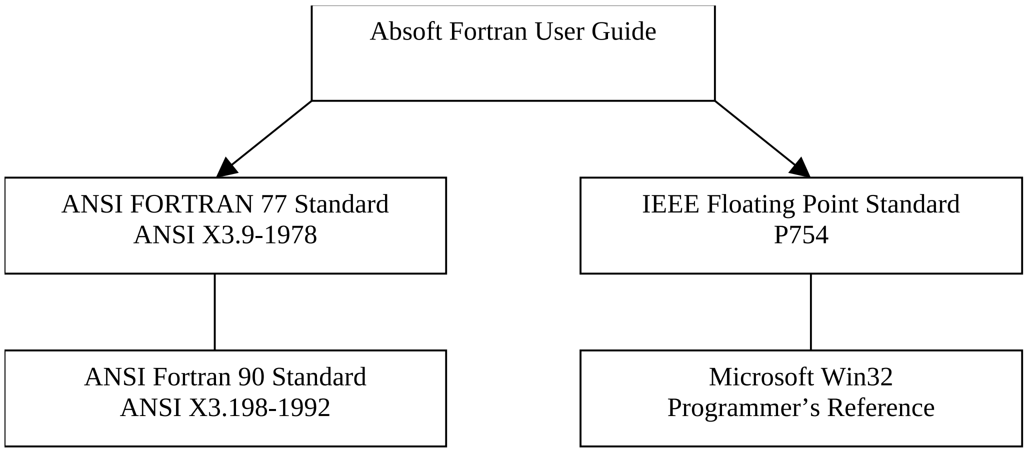
```
PROGRAM SAMPLE
WRITE(9,*) "Hello World!"
END
```

ROAD MAPS

Although this manual contains all the information needed to write programs with Absoft Fortran for Windows, there are a number of other manuals that describe FORTRAN extensions and the Windows programming environment in further detail. The two *road maps* in this chapter will guide you to these manuals for introductory or advanced reference. The bibliography in the Appendices lists further information about each manual.

FORTRAN Road Map

The Absoft implementations of Fortran 90/95 and FORTRAN 77 are detailed in the online manual, *Absoft Fortran Language Reference Manual*, also in the **Documentation** folder of the Pro Fortran CDROM. A discussion of floating point precision can be found in the chapter, **Porting Code** of this User Guide. Figure 1-1 shows additional manuals that can be used for referencing the FORTRAN language and internal math operations.



FORTRAN language road map

Windows Programming Road Map

Absoft Fortran 90/95 and FORTRAN 77 each provide complete access directly to the Windows System Services routines. The **Windows Programming** chapter of this manual describes the interface to these routines, but does not describe each of the hundreds of routines available. The **Win32 SDK Help** command in the Absoft Pro Fortran menu describes most of the Win32 API functions. Programmers wishing to make use of these routines to add graphics to their programs or to extend the user interface provided by MRWE may wish to obtain additional documentation on the Win32 programming model.

YEAR 2000 PROBLEM

All versions of Absoft Pro Fortran products for Macintosh, Power Macintosh, Windows 95/98, Windows NT/2000/XP/Vista/7, Linux, and UNIX operate correctly across the date transition to the year 2000. Neither the compilers nor the runtime libraries have ever used 2-digit years in their internal operation.

The only caveat may be for those porting code from VAX/VMS systems. Since the early 1980s, Absoft Pro Fortran products have included software libraries designed to facilitate porting code from the VAX/VMS environment. Included in these VAX compatibility libraries are two subroutines that emulate the VAX/VMS DATE and IDATE subroutines. These subroutines return the year using a two-digit format. If you use DATE or IDATE in a program that stores or compares dates, you may need to recode portions of your application. Below are listed some of the alternatives supplied with Pro Fortran:

Fortran 90 `DATE_AND_TIME` Subroutine

This subroutine is part of the Fortran 90/95 language and returns integer data from the date and real time clock. Refer to the *Absoft Fortran Reference Manual* for further information.

Unix Compatibility Library

There are a number of subroutines in the Unix Compatibility Library that return the date and time in both `INTEGER` and `CHARACTER` format. Refer to the *Support Library Guide* for information on their format and use.

CHAPTER 2

Getting Started

The tutorial in this chapter introduces the two main functions of the Absoft Pro Fortran Software Development package for Windows: compiling source code and running compiled applications. If you are familiar with the basics of compiling and running programs, please see the table below as a guide to topics you may find useful.

TO DO THIS…	TURN TO THIS SECTION…
Use the editor	**Using the Absoft Editor**, Chapter 3
Use the tools interface	**Developer Tools Interface**, Chapter 4
Use the compiler and options	**Using the Compilers**, Chapter 5
Port from other platforms	**Porting Code**, Chapter 6
Create applications	**Building Programs**, Chapter 7
Program Windows	**Windows Programming**, Chapter 8
Debug programs	**FX Debugger Manual**

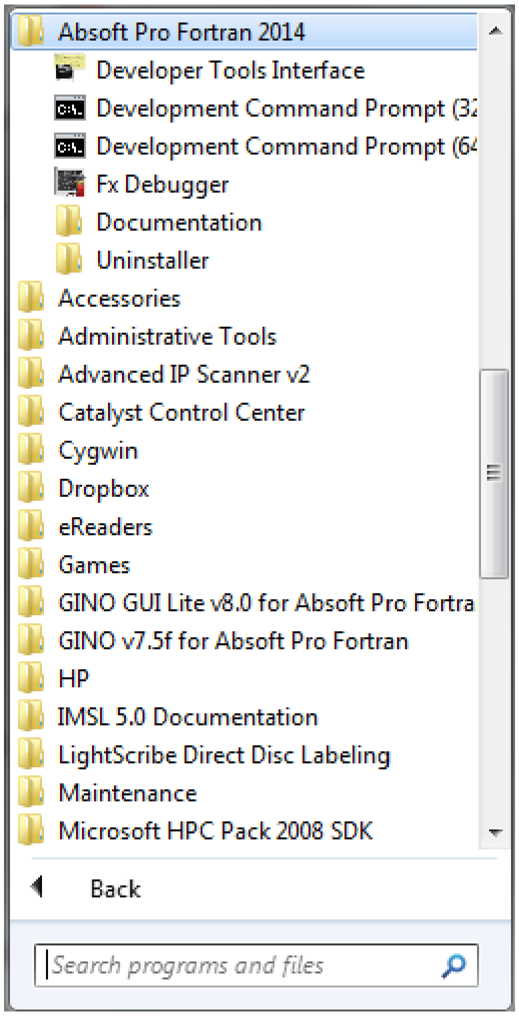
Road map for experienced users

COMPILING BASICS

The Absoft compilers can be run either from a command line or from the Absoft Developer Tools Interface. This chapter describes how to use the Developer Tools Interface —the command line interface is described in the Chapter 5, **Using the Compilers**.

Note: Throughout this chapter and the rest of the manual, it is assumed that the compiler has been installed on the `C:` drive. If this is not the case, substitute the correct drive letter in the examples as appropriate.

Selecting **Programs** from the **Start** menu and then **Developer Tools Interface** from the **Absoft Pro Fortran** submenu opens the Absoft Developer Tools Interface. It can also be started by selecting **Run…** from the **Start** menu and typing `c:\Absoft15.0\bin\AbsoftTools` or by typing `AbsoftTools` at a console or command line window. The Windows **Command Prompt** or should be opened by selecting **Programs** from the **Start** menu and then **Development Command Prompt** from the **Absoft Pro Fortran** submenu. Opening the console window in this way automatically sets the environment variables by executing the `c:\Absoft15.0\bin\absvars32.bat` batch file.


Absoft Pro Fortran 2014
Developer Tools Interface
Development Command Prompt (32
Development Command Prompt (64
Fx Debugger
Documentation
Uninstaller
Accessories
Administrative Tools
Advanced IP Scanner v2
Catalyst Control Center
Cygwin
Dropbox
eReaders
Games
GINO GUI Lite v8.0 for Absoft Pro Fortra
GINO v7.5f for Absoft Pro Fortran
HP
IMSL 5.0 Documentation
LightScribe Direct Disc Labeling
Maintenance
Microsoft HPC Pack 2008 SDK
Back
Search programs and files

During the installation process, several example programs were placed in the `c:\Absoft15.0\examples` directory. The example program used in this tutorial is `Fibonacci.f95`. Follow the tutorial on the following pages to learn how to use the graphical interface to quickly compile small to medium size programs.

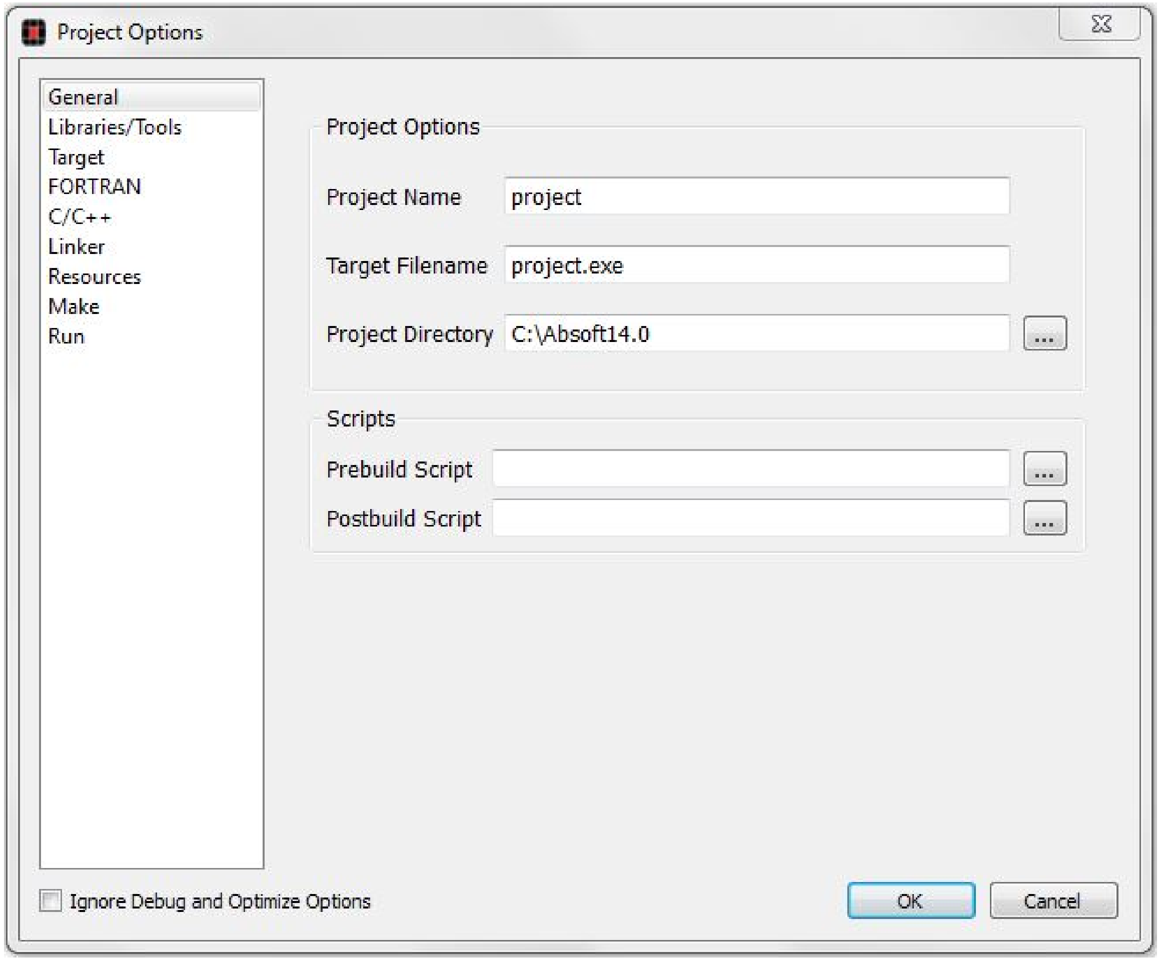
First, start up the interface to the compiler:

What to do	**How to do it**
Invoke the Absoft Developer Tools Interface.	Select **Programs** from the **Start** menu, then select **Developer Tools Interface** from the **Absoft Pro Fortran** submenu.

Note: The first time you run the interface, it may ask you if you want certain standard file extensions to be associated with the Absoft Editor. This will allow you to automatically open the editor by double-clicking on files with these extensions. You can choose to have this association established at this time, or defer the decision to later.

The Absoft Developer Tools Interface is project oriented, so the first thing you must do is to establish a name and location for your project.

What to do	How to do it
Set the project name and location.	Select **New Project** from the **Project** menu or type Ctrl+Shift_N.

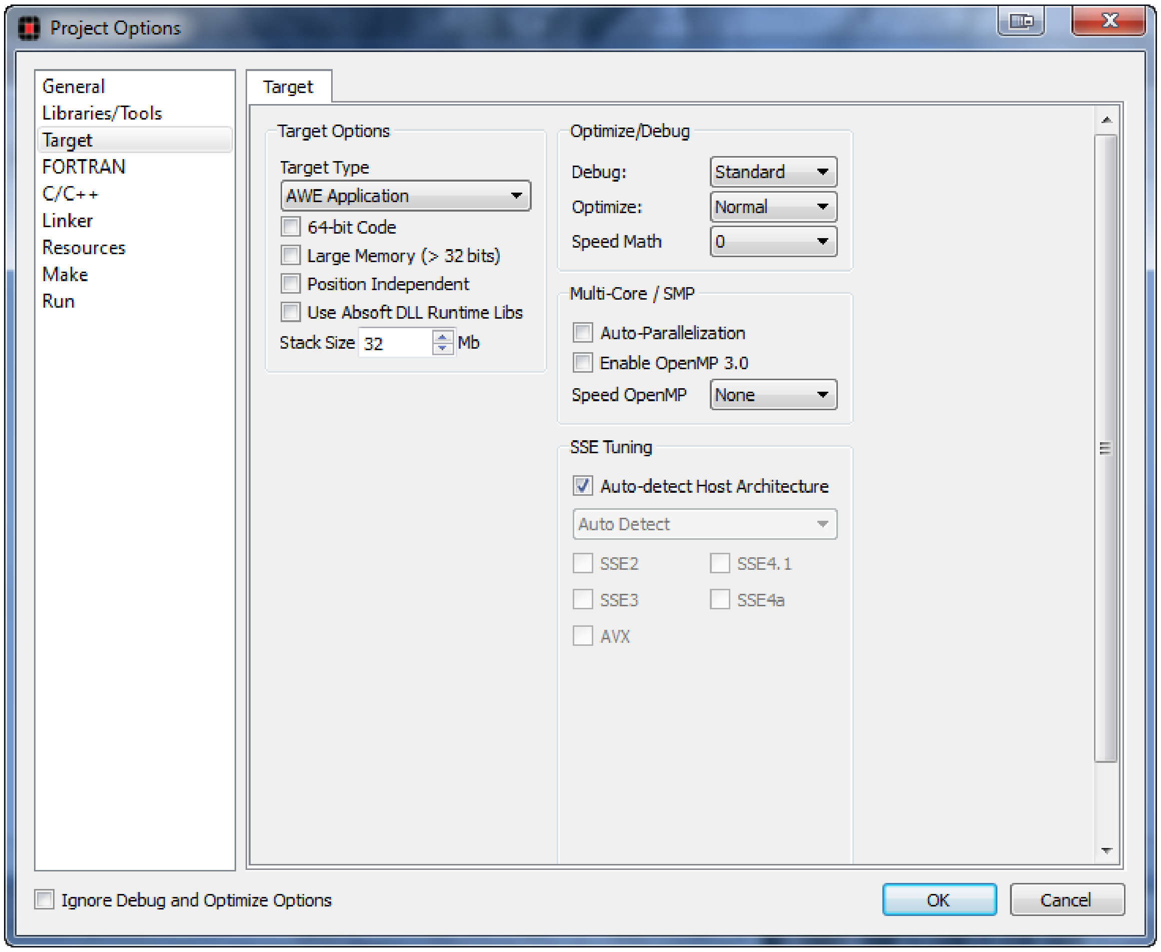


Project Name: project
Target Filename: project.exe
Project Directory: C:\Absoft14.0
Prebuild Script:
Postbuild Script:
☐ Ignore Debug and Optimize Options

On the **General** options page, change the **Project Name** to “Fibonacci” and the **Target Filename** to “Fibonacci.exe” as shown above. You may also want to change the **Project Directory** from the default to a Fortran specific directory.

You will now want to set the target type to **AWE Application**. AWE is the Absoft Window Environment. It provides an automatic Windows interface for your program with menus, a scrollable text window for program output, and the ability to print.

What to do	How to do it
Set the project **Target Type** to **AWE Application** (a Windows program).	Click on **Target** in the left panel to select the target options and then choose **AWE Application** from the **Target Type** drop menu in the upper left corner.



Target Type: AWE Application
- [ ] 64-bit Code
- [ ] Large Memory (> 32 bits)
- [ ] Position Independent
- [ ] Use Absoft DLL Runtime Libs

Stack Size 32 Mb

Debug: Standard
Optimize: Normal
Speed Math: 0

- [ ] Auto-Parallelization
- [ ] Enable OpenMP 3.0

Speed OpenMP: None

- [✓] Auto-detect Host Architecture

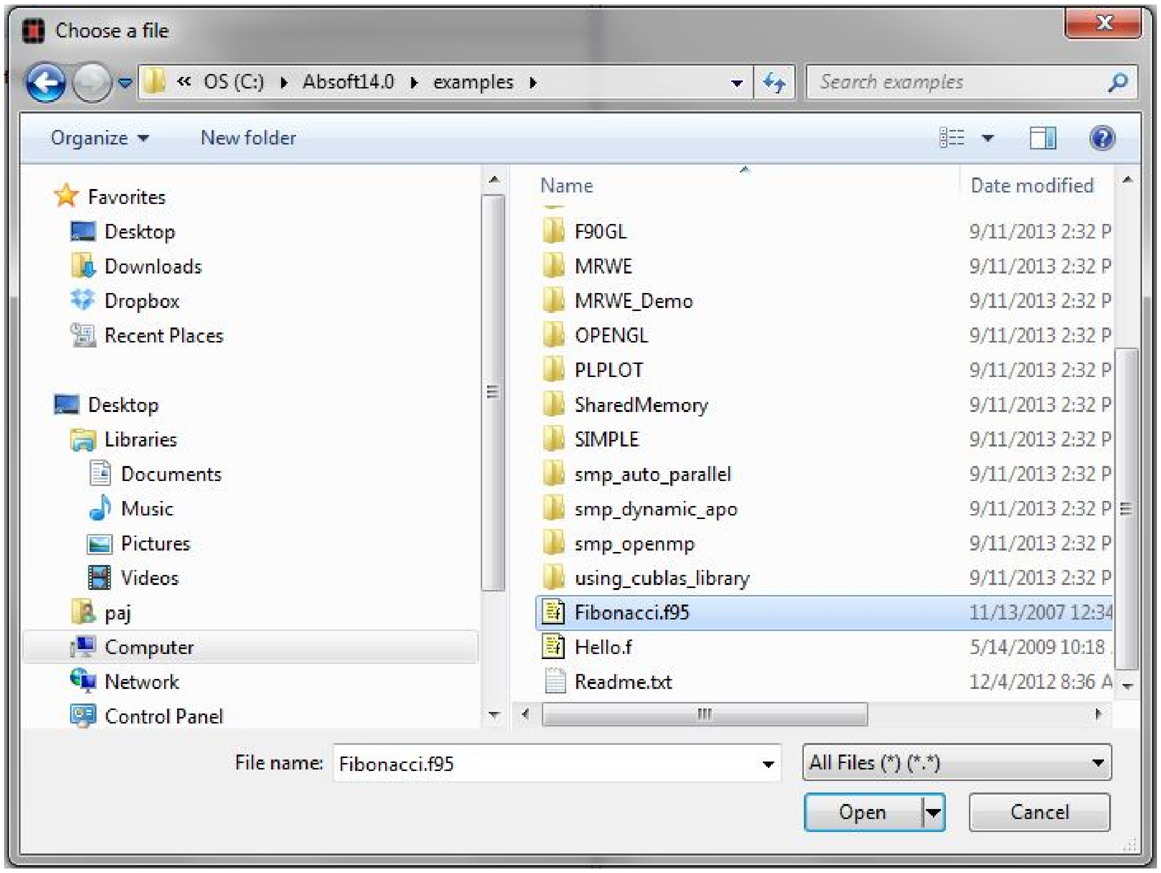
Auto Detect
- [ ] SSE2
- [ ] SSE4.1
- [ ] SSE3
- [ ] SSE4a
- [ ] AVX

- [ ] Ignore Debug and Optimize Options

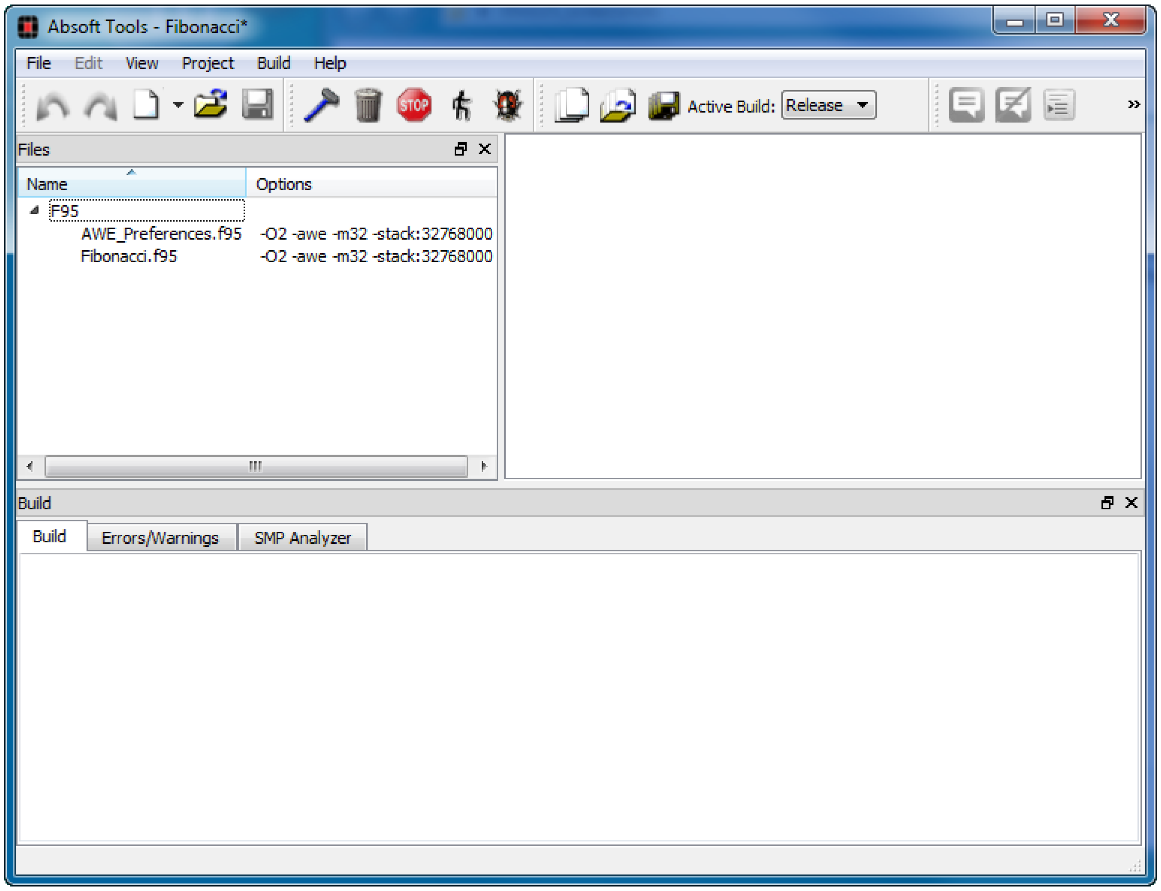
These are the only options you will want to set for this application, so click on the OK button to dismiss the **Default Tools Options** dialog.

The next step is to specify the file (or files) that the project consists of:

What to do	How to do it
Add the file **Fibonacci.f95** to the project.	Choose **Add File(s)…** from the **Project** menu. The file section dialog will open automatically

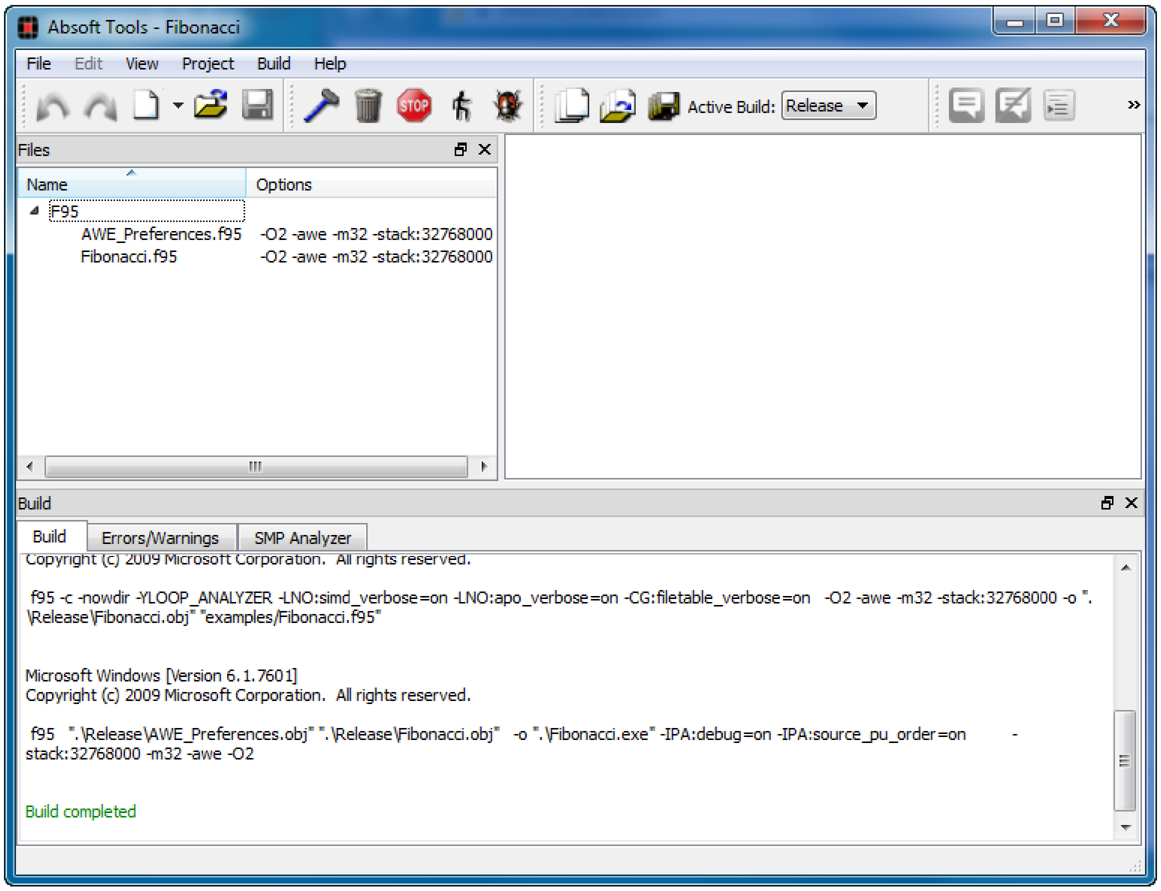


If you are not already in the **C:\Absoft 15.0\examples** folder, browse to that directory. Click on the file named Fibonacci.f95 and click **OK** to add it to the project. The project **Files** pane will now contain your source file and the options that will be used to compile it. This pane maintains all of the files in your project. Each file type will be kept in a separate folder. If you wish, you can also manage the files in your project directly from this window; you delete selected files and drag new files into this window.



The last step is to build (compile) your application:

What to do	How to do it
Compile the source file **Fibonacci.f95** into the application file **Fibonacci.exe**.	Choose the **Build** command from the **Build** menu.



Name	Options
F95	
AWE_Preferences.f95	-O2 -awe -m32 -stack:32768000
Fibonacci.f95	-O2 -awe -m32 -stack:32768000

```
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

 f95 -c -nowdir -YLOOP_ANALYZER -LNO:simd_verbose=on -LNO:apo_verbose=on -CG:filetable_verbose=on   -O2 -awe -m32 -stack:32768000 -o ".
\Release\Fibonacci.obj" "examples/Fibonacci.f95"


Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

 f95  ".\Release\AWE_Preferences.obj" ".\Release\Fibonacci.obj"   -o ".\Fibonacci.exe" -IPA:debug=on -IPA:source_pu_order=on        -
stack:32768000 -m32 -awe -O2


Build completed
```

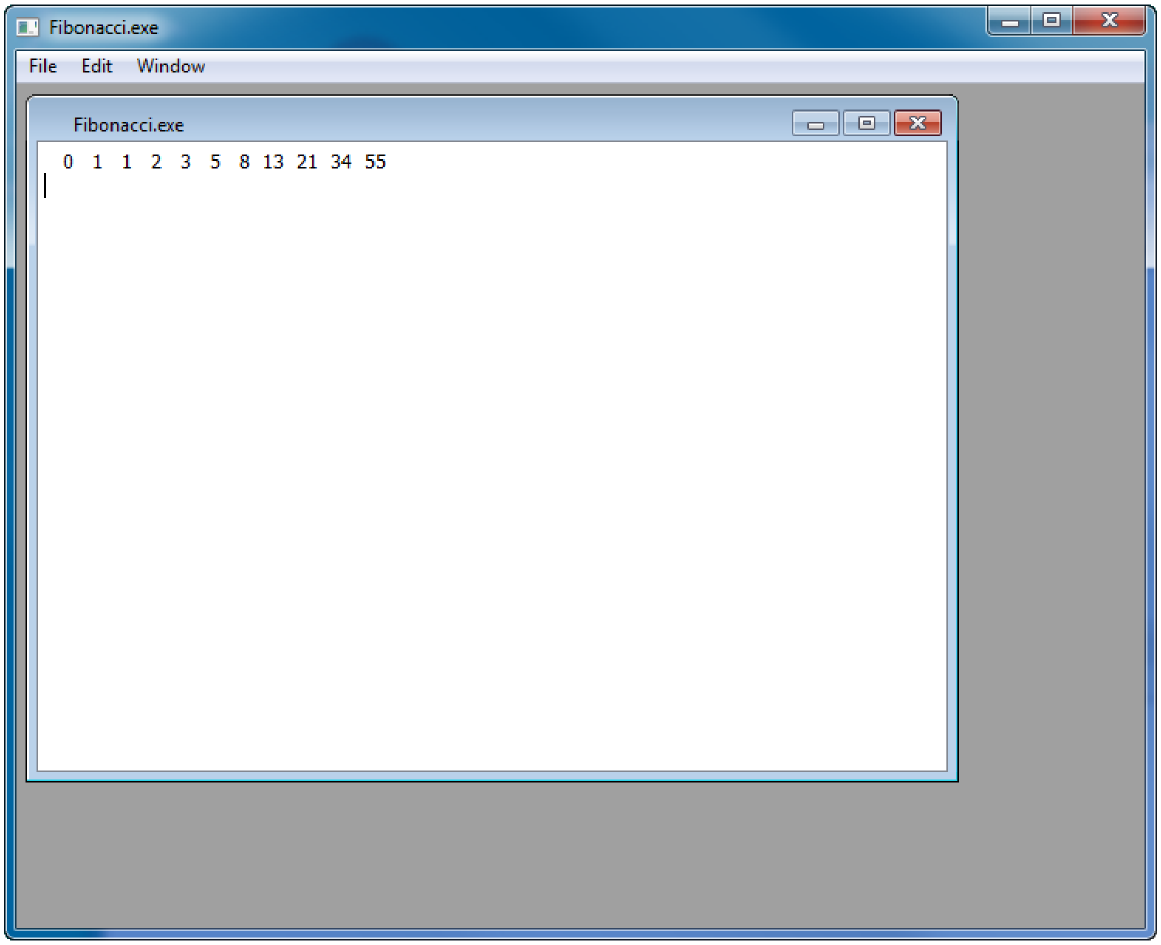
The compiler will then create `Fibonacci.exe` from `Fibonacci.f95`. More detailed information concerning the creation of an application can be found in the chapters **Developer Tools Interface** and **Using the Compilers**.

APPLICATION BASICS

The application is now ready to execute.

What to do	How to do it
Execute the compiled application.	Choose the **Execute** command from the **Build** menu.

You can also select **Run...** from the **Start** menu, browse to the `c:\Absoft15.0\examples` directory, select `Fibonacci.exe`, and run it or double-click on the application icon in an Explorer window.



```
  0  1  1  2  3  5  8 13 21 34 55
```

Additional examples that may be helpful in writing Fortran 90/95 or FORTRAN 77 programs can be found in the `c:\Absoft15.0\examples` directory. Each example source file starts out with a large comment, referred to as the header. Before compiling an example, look at the header in the source code. It will list all of the compiler options necessary to insure that the example will compile and run correctly. In addition, the header describes the purpose of the example and other useful information.

CHAPTER 3

Using The Editor

This chapter describes how to use the editor in Absoft Tools to create and edit source files written in FORTRAN. Since word processors embed formatting characters in a document, using a word processor to create source files is not recommended. You can create source files in a word processor or another editor and export them in text format, but the features of the Absoft Editor make this unnecessary. The Absoft Editor incorporates powerful features for editing FORTRAN 77, FORTRAN 90/95, C, and C++ source files. However, this chapter will concentrate specifically on editing FORTRAN programs.

The Absoft Editor is a powerful tool for creating and maintaining program source files. It is source language sensitive and will display keywords and comments in different text colors, making them easier to distinguish in your source code.

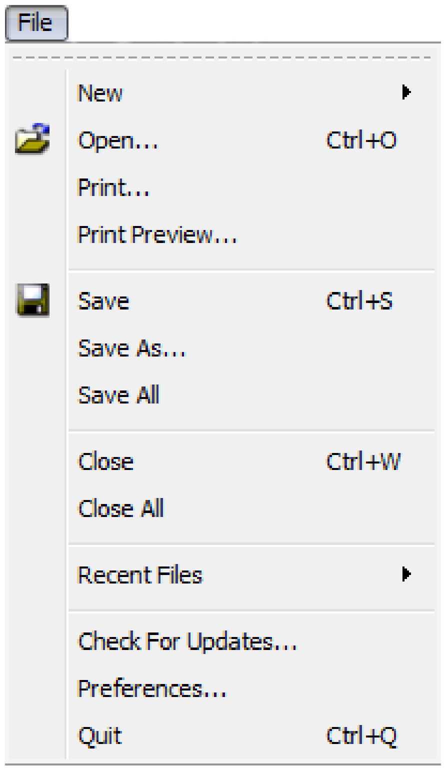
Basic editing functions are available as menu commands and there is usually more than one way to initiate any command:

- Select the command from the menu or tool bar.
- Type in the key equivalent (such as typing the Control and the letter O for the Open command).
- Right click on the text edit window to display a context menu

TEXT SELECTION

Text may be selected by dragging the cursor over the text while holding the mouse button down. Choosing Select All from the Edit menu or Ctrl+A will select the whole document.

FILE MENU



The File menu contains commands for creating, opening, saving, and closing files. There are also commands for printing and for establishing your preferences for the way that Absoft Tools operates.

New…(Ctrl+N)

This menu contains commands for creating new tabs for entering and editing text. The tab will be untitled (it will have the name “Untitled”) with the extension of the type of file you choose until the first time you save it.

Open…(Ctrl+O)

Use this command to open an existing file. This command displays a standard file selection dialog box to select the file to be opened. If you select a file that is already open, the tab that contains that file will be brought to the front of the editor.

Save (Ctrl+S)

Choose this command to save the text in the active tab. If the file does not exist, you will be asked to provide a name and a path for the file.

Save As…

Use the Save As command to save the text in the active tab to a different file. A standard file save dialog will appear, allowing you to specify the name of file. The active tab becomes the newly named file.

Save All

Use this command to save the text in all open tabs.

Close (Ctrl+W)

This command closes the file displayed in the active tab. If any unsaved changes had been made to the text, you will be asked to save it. This action is also available by right-clicking on the tab name.

Close All

This command closes all files. If any unsaved changes had been made to any files, you will be asked to save them.

Close Others

This command is only available by right-clicking on the tab name. The command closes all files except for the active tab. If any unsaved changes had been made to any files, you will be asked to save them.

Recent Files

Up to 8 files will appear in this list. Each menu item represents the file that has been most recently opened or saved. They are listed as a convenience for quickly opening files for editing. The **Clear Recent Files** selection in this menu will remove all 8 files from the list without any warning.

Check For Updates

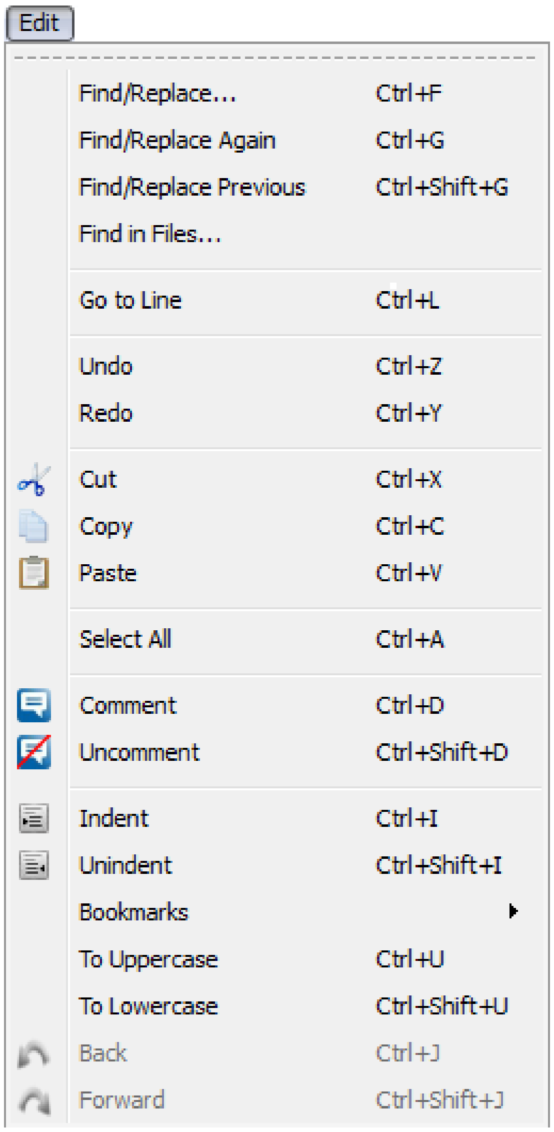
This menu selection opens a dialog to check for updates to your Absoft product.

Preferences

Opens a dialog to edit the preferences for Absoft Tools.

EDIT MENU AND POP-UP MENUS

Right-clicking the mouse button in a text edit window will display a pop-up menu of context sensitive commands. These commands are also available under the Edit menu. The Edit menu is not available if a file is not open for editing.



Find/Replace (⌘F)

Use this command to open the Find dialog for locating or replacing specified text within the front-most window. The controls in the Find dialog are used as follows:

Text in File

Enter the text string you wish to locate here.

Replace With

Enter the text string that will replace found text. This text is used with the **Replace All** and **Replace** buttons.

Replace

Replaces selected text with **Replace With**

Find and Replace

Executes a **Find** and then a **Replace**.

Replace and Find

Executes a **Replace** on selected text and executes a **Find**

Replace All

Replaces all text in the file.

Match Case

Check this box to find text occurrences in your source file that match your specified text exactly. Uncheck to search regardless of case.

Find Previous

Check this box to find text searching backwards from the cursor.

Whole Words

Match only whole word matches of the find text. For example, if the find text is soft, Absoft will not match when this is checked.

Find/Replace Again (Ctrl+G)

This command repeats the last Find/Replace command in the active tab.

Go to Line (Ctrl+L)

This command opens the Goto dialog. Enter the line number of the line you wish to go to and click on the Ok button.

Undo (Ctrl+Z)

The undo command undoes the last edit in the active tab. You can undo all actions since the document was opened.

Redo (Ctrl+Y)

The redo command redoes the last edit in the active tab. You can redo all actions since the document was opened.

Cut (Ctrl+X)

The cut command removes the selected text from the active tab and places it on the clipboard. Text on the clipboard may be pasted into other windows.

Copy (Ctrl+C)

The Copy command copies the selected text from the active tab and places it on the clipboard. Text on the clipboard may be pasted into other windows.

Paste (Ctrl+V)

The paste command replaces the selected text in the active tab with the text on the clipboard. If no text is selected in the active tab, the clipboard text is inserted at the insertion point.

Select All (Ctrl+A)

The Select All command selects all text in the document.

Comment (Ctrl+D)

This command inserts a comment character in column one of the current line if there is no selected text section. Otherwise, it will comment the entire selected text. . For C/C++ files this is a double forward slash ("//") and for all other files it is an exclamation mark ('!').

Uncomment (Ctrl+Shift+D)

This command deletes a comment character in column one of the current line or the selected lines. For C/C++ files this is a double slash ("//") and for all other files it is an exclamation mark ('!').

Indent (Tab)

Use this command to shift either the selected text or current line to the right by one tab stop.

Unindent (Ctrl+Shift+Tab)

Use this command to shift either the selected text or current line to the left by one tab stop.

To Uppercase (Ctrl+U)

This command converts the selected text to upper case.

To Lowercase (Ctrl+Shift+U)

This command converts the selected text to lower case.

Back (Ctrl +J)

Use this command to navigate back to the last cursor position in the file or project.

Forward (Ctrl +Shift+J)

Use this command to navigate forward to the last cursor position in the file or project.

Bookmarks

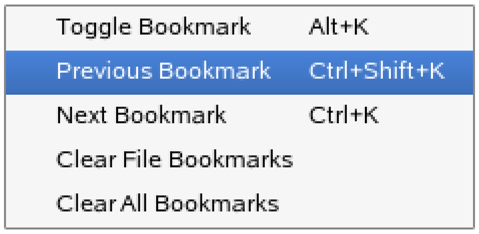
Bookmarks provide an easy way to "save your place" in a file so that you can later return there quickly. Positioning the insertion caret on the line where you want the bookmark set and then typing Alt+K sets (or unsets) a bookmark. In other words, Alt+K toggles a bookmark.

A bookmark appears as a small flag at the beginning of the line. Pressing the Ctrl+K key alone moves the insertion caret to the next bookmarked line in the file. Holding the Shift key down and pressing the Ctrl+K key moves the insertion caret to the previous bookmarked line in the file. The Clear File Bookmarks action in the Edit menu or context menu clears all bookmarks in the file. The Clear All Bookmarks action clears all the bookmarks for all files.

The View->Bookmarks action will open a display showing all available bookmarks in all files. Clicking on a bookmark opens the file (if not already opened) and sets the cursor to the line of the bookmark clicked. Double clicking on a bookmark name will allow you to edit the name.

Note: Bookmarks are either associated with a specific project (see Developer Tools Interface in the next chapter) or with no project (editor bookmarks). Editor bookmarks are only accessible with no project open, and project bookmarks are only accessible with the associated project open. Bookmarks are saved in a project save file.

Bookmarks Menu



The Bookmarks sub-menu provides commands for setting, clearing, and moving between bookmarks.

Toggle Bookmark (Alt+K)

Use this command to set or unset a bookmark on the line where the insertion caret is positioned.

Previous Bookmark (Ctrl+Shift+K)

Use this command to move to a previous bookmark location in the file.

Next Bookmark (Ctrl+K)

Use this command to move to the next bookmark location in the file.

Clear File Bookmarks

Use this command to remove all bookmarks in the file.

Clear All Bookmarks

Use this command to remove all bookmarks in all files.

Code Completion (Ctrl+E)

Code completion is a pop-up box that suggests possible ways of completing the words or strings based on previously used words in that file. It is automatically turned on once the length of the word typed is more than 3 characters. Typing Ctrl+E can also manually bring up the pop-up box. To navigate the pop-up box, use up and down arrow keys. The selection can be made by either pressing the enter key or left clicking the mouse. When the pop-up box is open, you can dismiss it by pressing the Esc key.

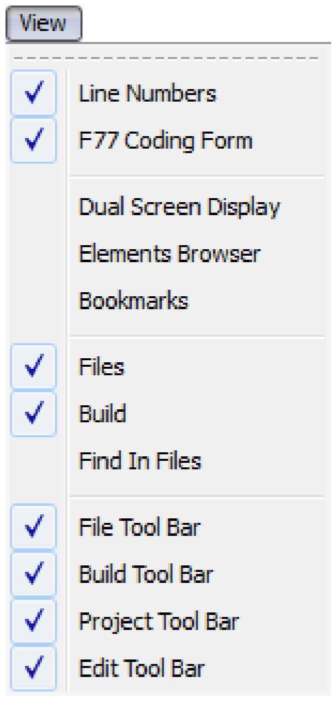
SYNTAX HIGHLIGHT (CONTEXT MENU ONLY)

The Syntax Highlight command will specify which programming language to highlight the document with. This is automatic for common FORTRAN and C file extensions. The current highlighting language may be changed through this menu. Choices are F95, F77, C/C++, and None. It is recommended to use standard file extensions so you do not have to change this setting.

Standard file extensions are:

F95:	*.f95, *.f90, .F95, .F90
F77:	*.f, *.for, .F, .FOR
FORTRAN headers:	*.inc
C:	*.c, *.C
C++:	*.cpp
C/C++ headers:	*.h

VIEW MENU AND POP-UP MENUS



The view menu allows you to change what is displayed in Absoft Tools.

Line Numbers

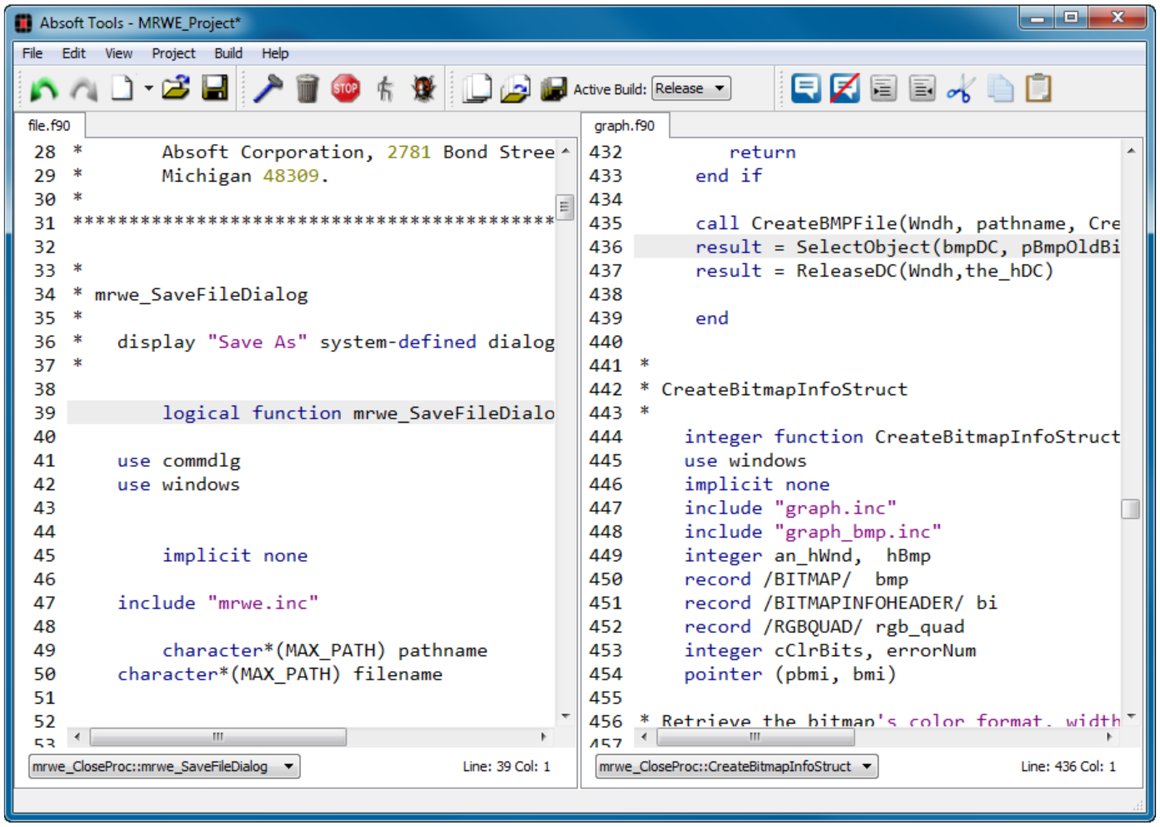
Toggles line number display in the margin for the active tab.

F77 Coding Form

Toggle coding form background for the active tab. Coding form highlights significant F77 columns in gray. The highlighted columns are columns 6, and 72-80. You must have a fixed form font for this the columns to be highlighted correctly. You can change fonts in the File->Preferences menu. F77 Coding Form cannot be toggled on in other file formats.

Dual Screen Display

Dual Screen display provides a convenient way to open files side-by-side. Toggling dual screen display on will create another text editor window pane to the right of the existing one. By default, toggling it on will give the focus to this second text editor pane. For example, if you open a file, the file will be opened in this new pane. This can be used for comparing two different files. Note that the same file cannot be opened twice. All the edit actions such as cut, copy, and paste apply on both panes.

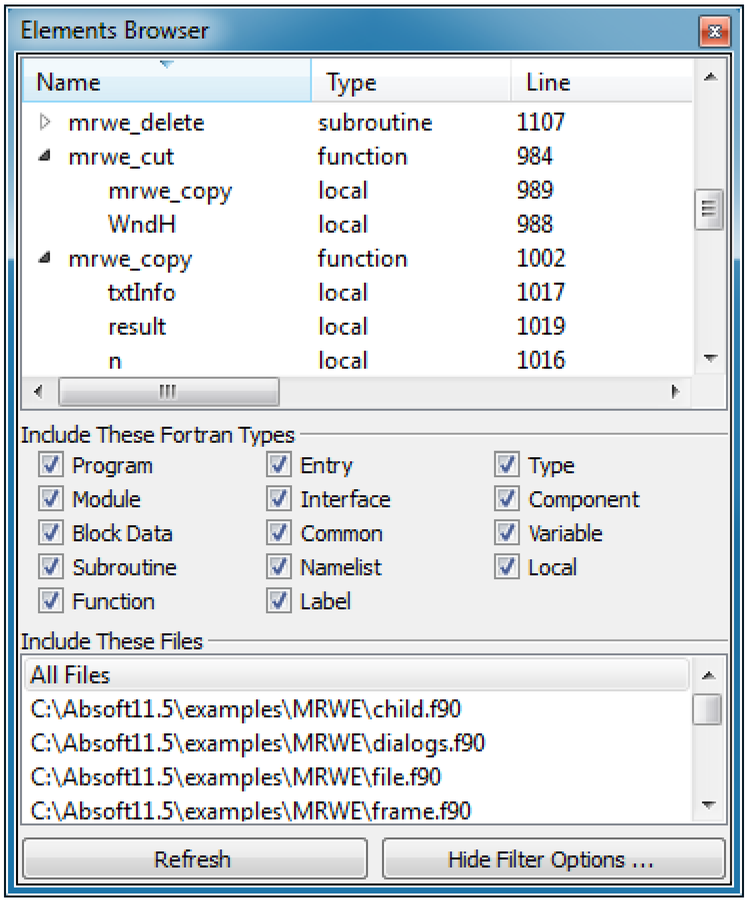

Absoft Tools - MRWE_Project*
File
Edit
View
Project
Build
Help
Active Build:
Release
file.f90
graph.f90
mrwe_CloseProc::mrwe_SaveFileDialog
Line: 39 Col: 1
mrwe_CloseProc::CreateBitmapInfoStruct
Line: 436 Col: 1

```
*       Absoft Corporation, 2781 Bond Stree
*       Michigan 48309.
*
*******************************************

*
* mrwe_SaveFileDialog
*
*   display "Save As" system-defined dialog
*

        logical function mrwe_SaveFileDialo

    use commdlg
    use windows


        implicit none

    include "mrwe.inc"

        character*(MAX_PATH) pathname
    character*(MAX_PATH) filename

```

```
          return
       end if

       call CreateBMPFile(Wndh, pathname, Cre
       result = SelectObject(bmpDC, pBmpOldBi
       result = ReleaseDC(Wndh,the_hDC)

       end

*
* CreateBitmapInfoStruct
*
      integer function CreateBitmapInfoStruct
      use windows
      implicit none
      include "graph.inc"
      include "graph_bmp.inc"
      integer an_hWnd,  hBmp
      record /BITMAP/  bmp
      record /BITMAPINFOHEADER/ bi
      record /RGBQUAD/ rgb_quad
      integer cClrBits, errorNum
      pointer (pbmi, bmi)

* Retrieve the bitmap's color format, width
```

Elements Browser

Toggles the elements display. The elements display contains a hierarchal list of all elements in the program. Clicking on a '+' will expand the element to show all its children. Clicking on a '–' will collapse an element. Clicking on an element will open the file if it is not already open and move the text cursor to the element declaration or implementation line. Clicking the refresh button will cause the project or editor file to be re-parsed. Saving the file will also cause it to be reparsed. Clicking the Show Filter Options will show a selection of item types to filter. To exclude variables from the elements list, uncheck the Variable checkbox. To enable, check the Variable checkbox.



Name	Type	Line
▷ mrwe_delete	subroutine	1107
◢ mrwe_cut	function	984
mrwe_copy	local	989
WndH	local	988
◢ mrwe_copy	function	1002
txtInfo	local	1017
result	local	1019
n	local	1016

Include These Fortran Types

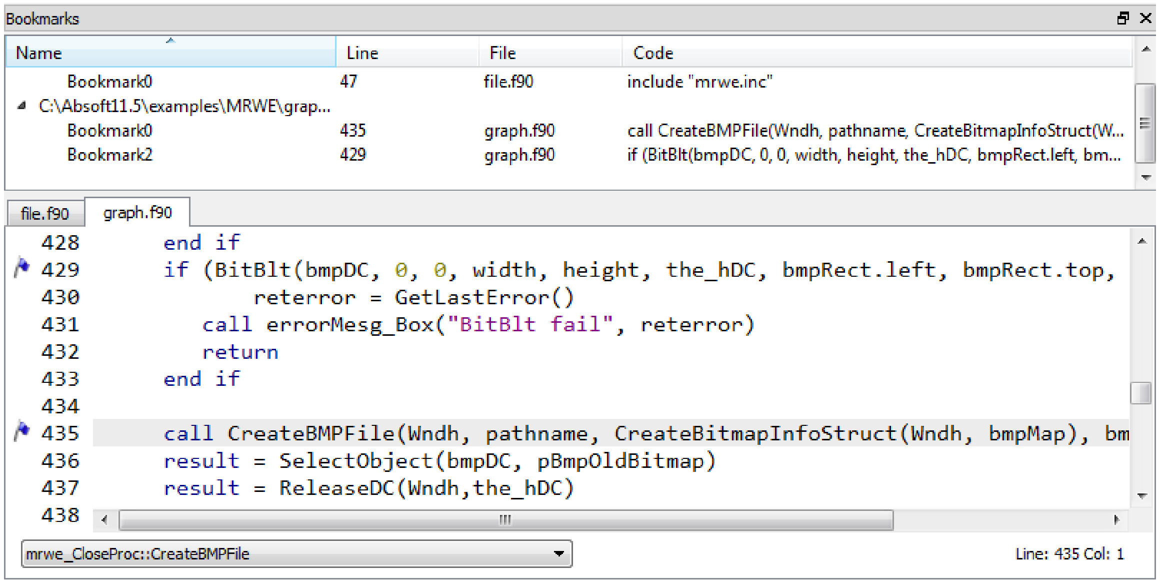
✓ Program	✓ Entry	✓ Type
✓ Module	✓ Interface	✓ Component
✓ Block Data	✓ Common	✓ Variable
✓ Subroutine	✓ Namelist	✓ Local
✓ Function	✓ Label	

Include These Files

```
All Files
C:\Absoft11.5\examples\MRWE\child.f90
C:\Absoft11.5\examples\MRWE\dialogs.f90
C:\Absoft11.5\examples\MRWE\file.f90
C:\Absoft11.5\examples\MRWE\frame.f90
```

Bookmarks

Toggles the bookmark display. The bookmark display contains all the bookmarks that are currently available sorted by file. (See Bookmarks description in the Edit menu for additional information). Whenever a bookmark is added or removed, this list will be updated. Clicking on a bookmark in this window will open the file if it is not already open and move the text cursor to the bookmarked line. The name of the bookmark may be edited by double clicking on the name and entering the text for the name.



Name	Line	File	Code
Bookmark0	47	file.f90	include "mrwe.inc"
C:\Absoft11.5\examples\MRWE\grap...			
Bookmark0	435	graph.f90	call CreateBMPFile(Wndh, pathname, CreateBitmapInfoStruct(W...
Bookmark2	429	graph.f90	if (BitBlt(bmpDC, 0, 0, width, height, the_hDC, bmpRect.left, bm...

```
428       end if
429       if (BitBlt(bmpDC, 0, 0, width, height, the_hDC, bmpRect.left, bmpRect.top,
430              reterror = GetLastError()
431          call errorMesg_Box("BitBlt fail", reterror)
432          return
433       end if
434
435       call CreateBMPFile(Wndh, pathname, CreateBitmapInfoStruct(Wndh, bmpMap), bm
436       result = SelectObject(bmpDC, pBmpOldBitmap)
437       result = ReleaseDC(Wndh,the_hDC)
438
```

mrwe_CloseProc::CreateBMPFile

Line: 435 Col: 1

Files

Toggles the files display for an open project. See project documentation for more details. Note: this will not be selectable if you do not have a project open.

Build

Toggles the build display for an open project. See project documentation for more details Note: this will not be selectable if you do not have a project open.

Find in Files

Toggles the Find in Files display for an open project. See project documentation for more details. Note: this will not be selectable if you do not have a project open.

File Tool Bar

Toggles the visibility of the File tool bar.

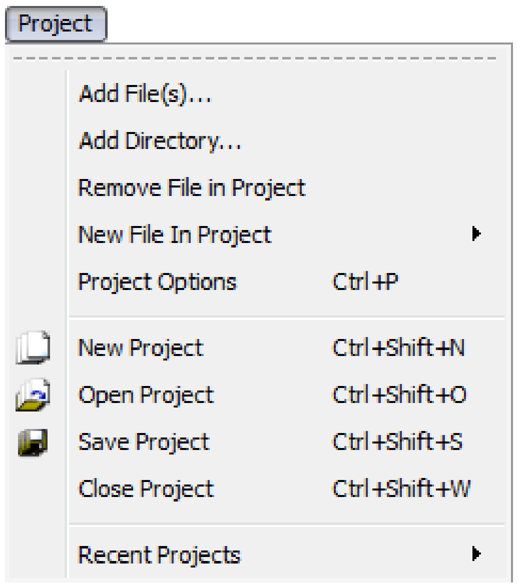
Build Tool Bar

Toggles the visibility of the Build tool bar. Note: this will not be selectable if you do not have a project open.

Project Tool Bar

Toggles the visibility of the Project tool bar.

PROJECT MENU



New Project

This command opens a new project dialog to create a new project. Your open files will not be closed or added to the new project.

Open Project

This command opens a file browser to select a saved project file dialog to open. Your open files will not be closed or added to the project.

Recent Projects

Up to 8 files will appear in this list. Each menu item represents the project file that has been most recently opened or saved. They are listed as a convenience for quickly opening projects.

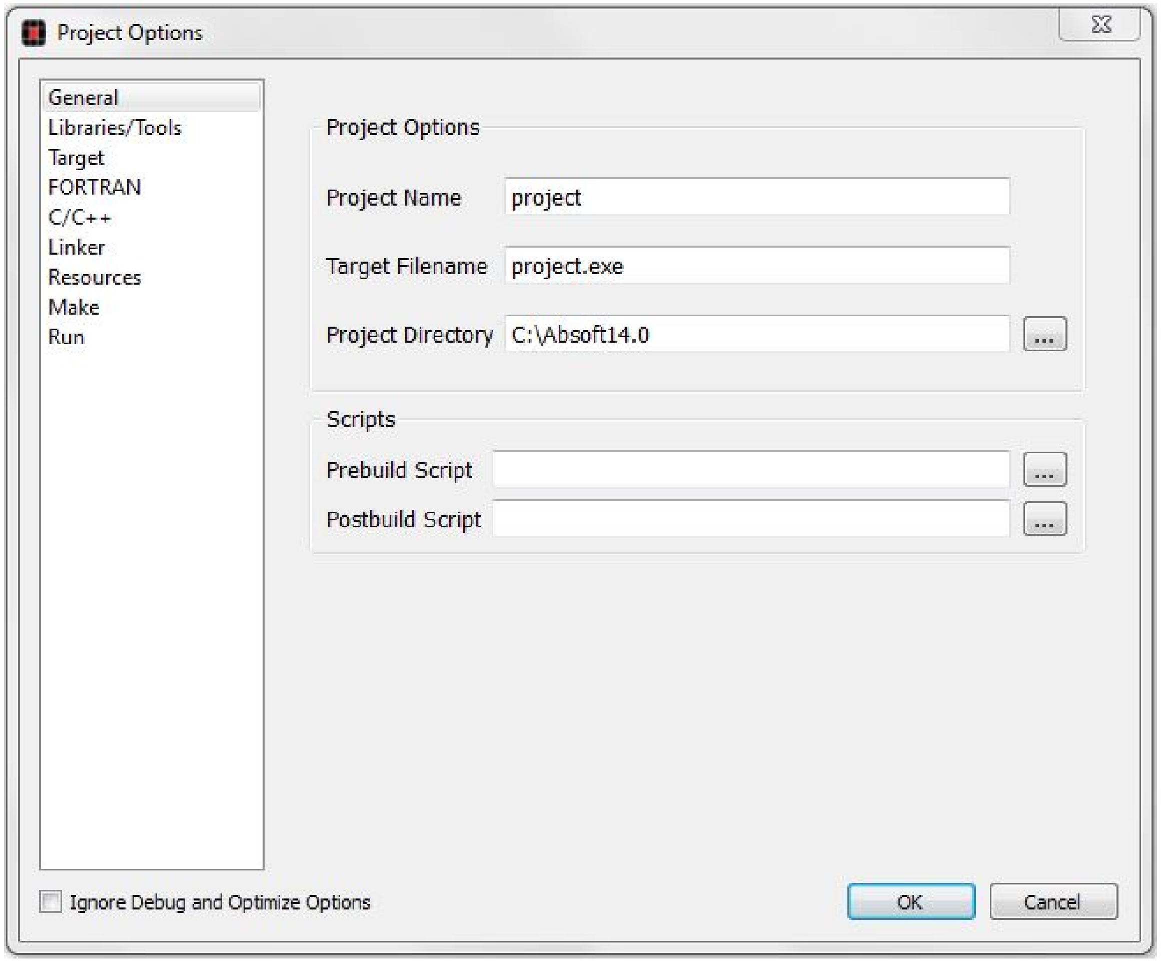
CHAPTER 4

Developer Tools Interface

WORKING WITH PROJECTS

A project allows you to organize the entire source, object, include, library, and resource files that constitute an application. It keeps track of which files are associated with the application, which ones are dependent on other files, which ones have been recently modified and need to be rebuilt. Also, it allows you to set specific options to be used with the compilation tool associated with the various files in the project.

The first step in working with a project is to create a new one. Use the **Project** menu **New Project…** command to create a new project. The **New Project** dialog will appear as shown below:



Project Name: project

Target Filename: project.exe

Project Directory: C:\Absoft14.0

Prebuild Script:

Postbuild Script:

Ignore Debug and Optimize Options

Project Name is the name that will be applied to the project. It will be shown whenever the project is identified in Absoft Tools. **Target Filename** is the name of the executable program or library to be created. **Project Directory** is the base directory of the project. Clicking the "..." button will allow you to choose a directory from a standard file dialog. This can only be set when creating a new project. Options Packages are libraries that are included or purchased as add-ons to the Absoft product. Checking the boxes will add the add-on to the project.

The left column contains groups of options for the general project or specific tools. **General**, **Target**, **Run**, **Make** and **Run** apply to the project globally. **FORTRAN**, **C/C++**, **Resources**, and **Linker** apply to the specific tools used for compiling their respective files. Changing options in the New Project dialog will create the initial options for the project. All may be changed any time a project is open. Clicking **OK** will create a project with the options specified.

DOCKED DISPLAYS

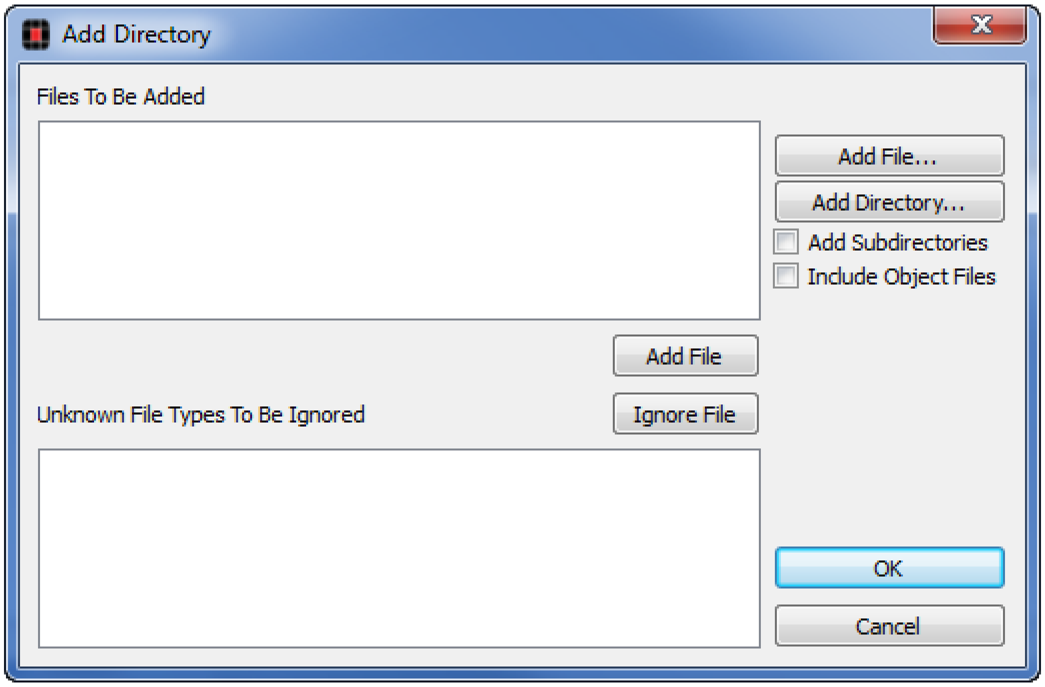
A dock is a movable, resizable, and detachable display window. Several project specific docks will appear after you have created a new project. The default docks are the Files dock, Find in Files dock and the Build Dock.

Docks can be moved to customize the appearance of Absoft Tools. Docks can be moved to the top, bottom, left, or right edged of the screen by clicking and dragging the name of the dock to a new location. Dropping a dock on top of an existing dock will create a tabbed set of docks. Selecting the name of the dock under the View menu will toggle the visibility of the dock.

When a project is created or opened, the default docks **Files**, **Find in Files**, and **Build** will be shown on the screen.

ADDING FILES TO THE PROJECT

Files can be added in two ways. The first is right clicking on the files window and choosing **Add File(s)** from the context menu or select **Add File(s)** from the **Project** menu. This will display a standard file dialog where you can select a single file by clicking or multiple files by holding down the Control key as you select the files to add. The second way is to select **Add Directory** from the **Project** menu or right click on the files display and choose **Add Directory** from the menu. This will bring up the following dialog:

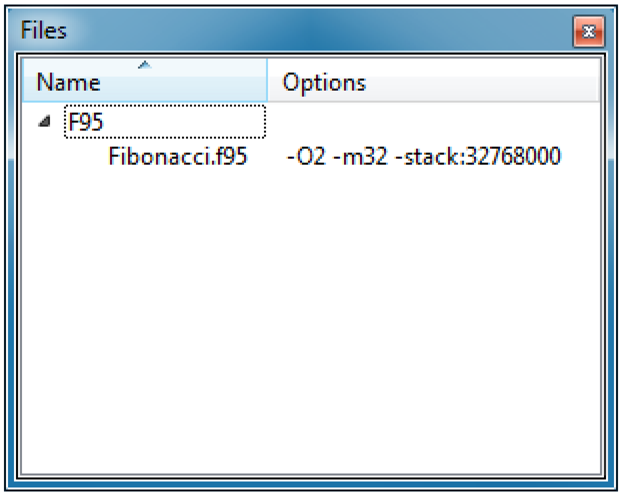


Click the **Add File** button to add files through a standard file dialog. Click the **Add Directory** button to add all files in a directory. Check **Add Directories** to add all files in the subdirectories under the chosen directory. Check **Include Object Files** to add object files to the project.

Once you finish with either the **Add File** or **Add Directories** selection and click **OK**, the chosen files will be added to the **Files to be Added** list box if Absoft Tools has a tool to compile the files. If a file is unknown, it will be added to the **Unknown File Types To Be Ignored** list box. Selecting a file in the **Files to be Added** list box and clicking the Ignore File button will move the file to the **Unknown File Types To Be Ignored** list box. Selecting a file in the **Unknown File Types To Be Ignored** list box and clicking the Add File button will move the file to the **Files to be Added** list box. All files in the **Files to be Added** list box will be added to the project once Ok is clicked.

FILES DOCK

Files that are in the project are located in the **Files** dock window. Files are organized by file type. Each type can be expanded or collapsed by clicking the + or – next to the type name.



Fibonacci.f95 -O2 -m32 -stack:32768000

Multiple files may be selected. Right-clicking on the file list will bring up a context menu. The menu commands are listed below:

New File in Project

This menu allows you to choose the type of new file to add to the project. Selecting the type of file will cause a standard file dialog to appear. Choose the name and directory to save the file as, and the new file will be saved and opened in the active window for editing.

Add File(s)

Opens a standard file dialog for adding files (see Adding Files To Project).

Add Directory

Opens a directory dialog for adding files (see Adding Files To Project).

Check Syntax

Compiles selected files using the options displayed in the files view. The results will appear in the build view. This is used to check the syntax of a file without recompiling the whole project.

Set Options for

Opens an option dialog to set options for the selected files only (see Setting Compiler Options).

Use Default Options

Removes any file specific options set by “Set Options for.” The project options set in **Project Options** from the **Project** menu will be used.

Remove

Removes files from project.

Show Full Paths

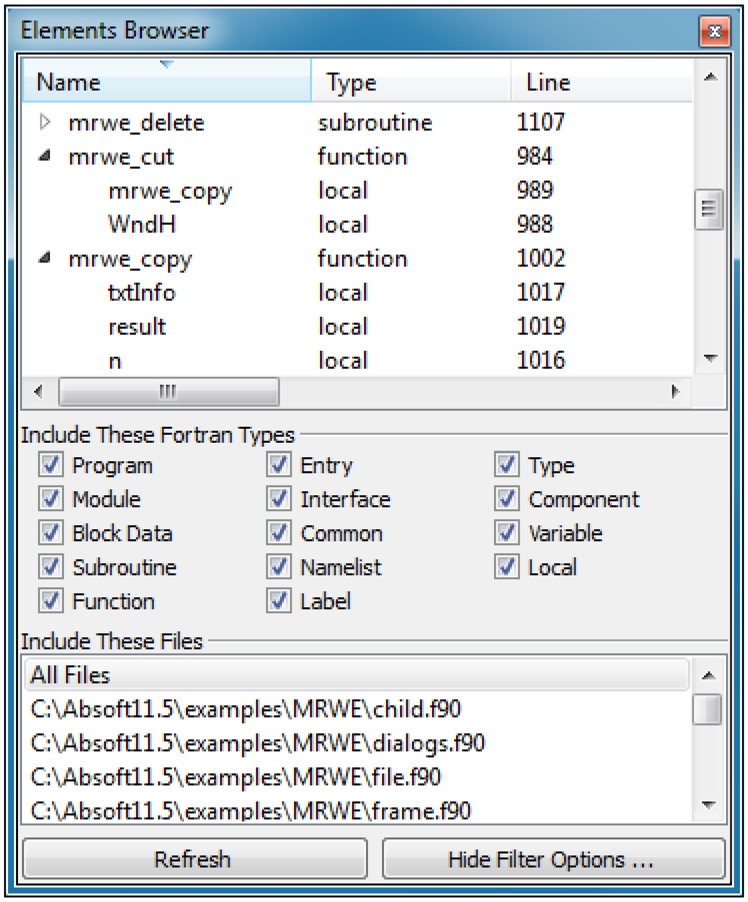
Adds another column to the file list that contains the full paths of each file.

Show Relative Paths

Adds another column to the file list that contains the relative paths of each file.

Elements Browser

Toggles the elements display. The elements display contains a hierarchal list of all elements in the program. Clicking on a ‘+’ will expand the element to show all its children. Clicking on a ‘–‘ will collapse an element. Clicking on an element will open the file if it is not already open and move the text cursor to the element declaration or implementation line. Clicking the refresh button will cause the project or editor file to be re-parsed. Saving the file will also cause it to be reparsed. Clicking the Show Filter Options will show a selection of item types to filter. To exclude variables from the elements list, uncheck the Variable checkbox. To enable, check the Variable checkbox.


Elements Browser

Name	Type	Line
mrwe_delete	subroutine	1107
mrwe_cut	function	984
mrwe_copy	local	989
WndH	local	988
mrwe_copy	function	1002
txtInfo	local	1017
result	local	1019
n	local	1016

Include These Fortran Types

✓ Program	✓ Entry	✓ Type
✓ Module	✓ Interface	✓ Component
✓ Block Data	✓ Common	✓ Variable
✓ Subroutine	✓ Namelist	✓ Local
✓ Function	✓ Label	

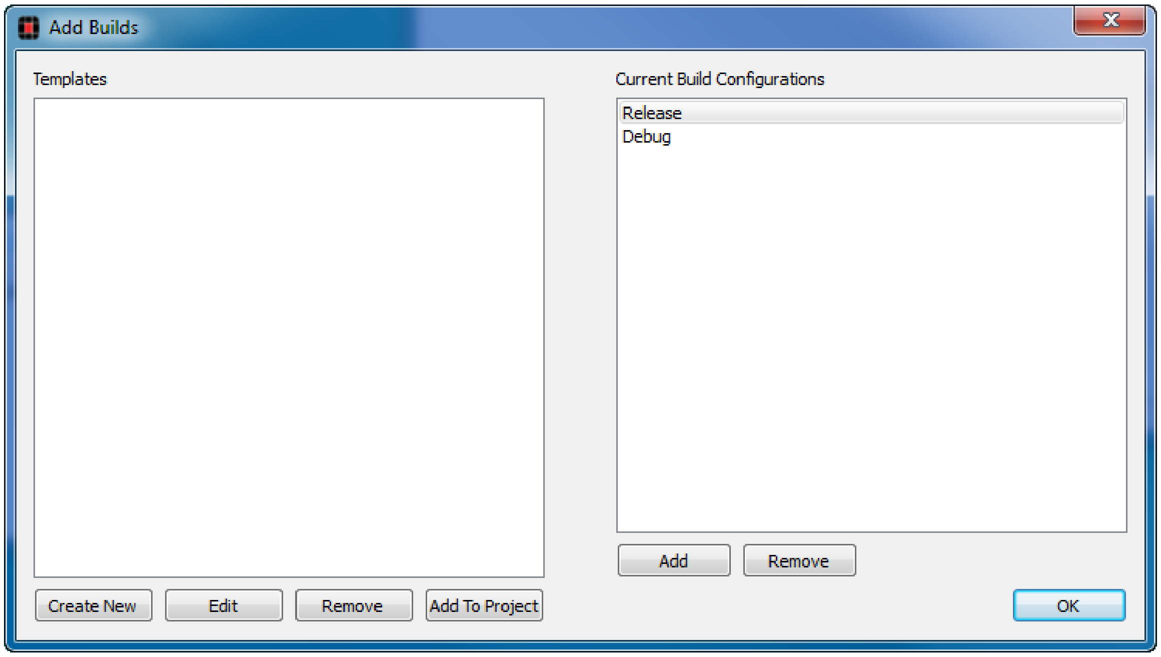
Include These Files

All Files
C:\Absoft11.5\examples\MRWE\child.f90
C:\Absoft11.5\examples\MRWE\dialogs.f90
C:\Absoft11.5\examples\MRWE\file.f90
C:\Absoft11.5\examples\MRWE\frame.f90

BUILD CONFIGURATIONS

Absoft Tools has two built in build configurations, Release and Debug. The Release build configuration is an optimized build. It uses the optimization options that are set in the options dialog and builds object and mod files in <project directory>/Release. The Debug build configuration is a debug build. It uses the debug options that are set in the options dialog and builds object and mod files in <project directory>/Debug. You can switch between build configurations by selecting **Set Build** from the **Build** menu or selecting the build name from the Active Build combo box on the project tool bar.

Absoft Tools allows you to create your own build configurations. To create a new configuration, select **Create New Build** from the **Build** menu. This will open a dialog that will allow you to add custom builds.



Adding a New Build Configuration

To add a new build configuration, click the **Add** button below the **Current Build Configurations** pane. A text cursor will appear in the list of current build configurations, allowing you to enter the name of the new configuration. If the name you enter ends with "debug", the new configuration will be considered a Debug build when using the **Project Options** dialog to modify build options. You can also add a build configuration to a project using a previously defined configuration template by selecting the name of the template in the **Templates** pane and clicking **Add To Project**.

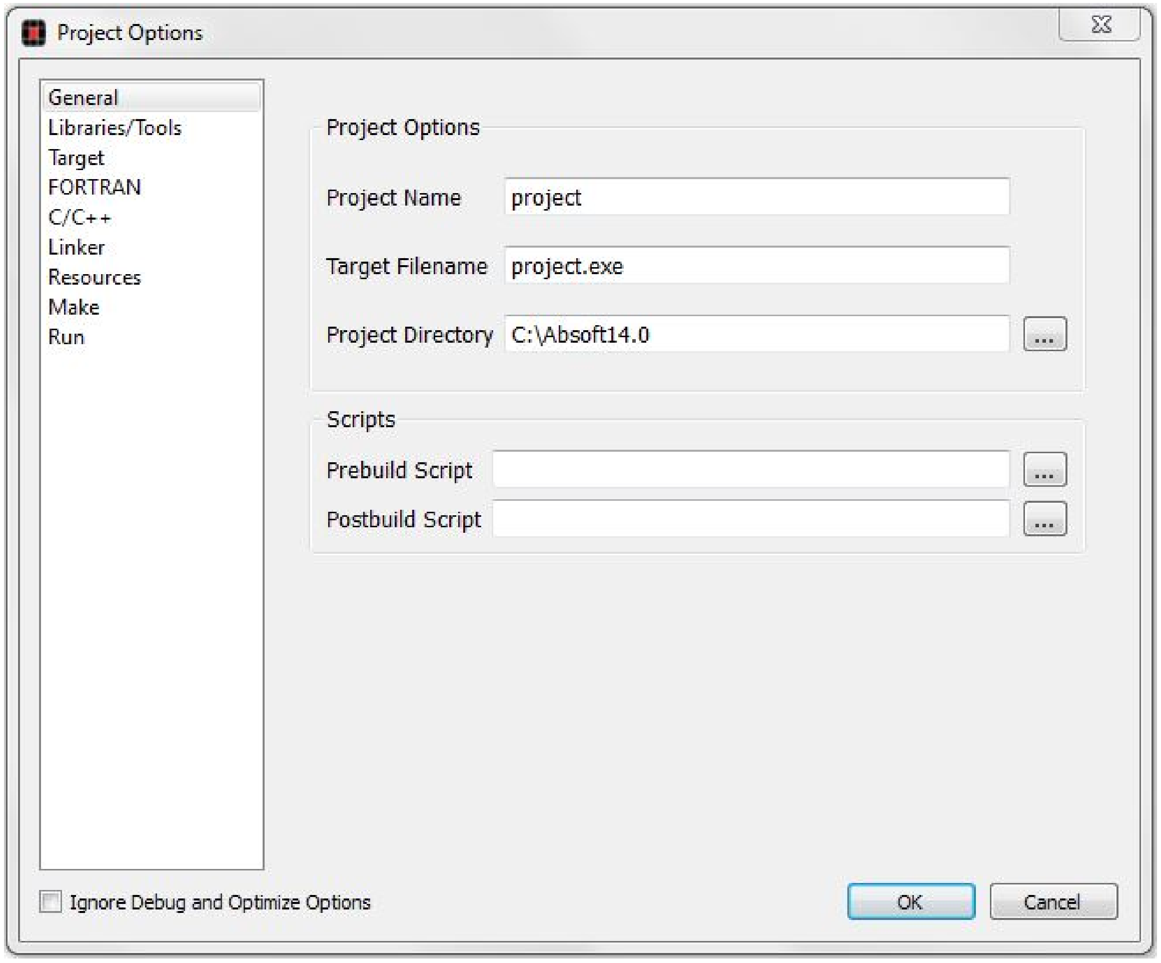
After a new build configuration has been added to a project, you can make it the active build using **Set Build** command the **Build** menu or selecting the build name from the **Active Build** combo box on the project tool bar.

Creating A New Build Configuration Template

To create a new build configuration template, click **Create New** below the **Templates** pane. Type in the name of the new build (case is significant) and press enter. If a template name ends with the characters "debug", it will be considered a Debug build when added to a project. After template name has been entered, the **Project Options** dialog will open and allow you set the various options for your new template. When you are finished, click **OK** and the new template will be added to the list of available templates. You can add the new template to the current project by clicking on **Add To Project**.

SETTING COMPILER OPTIONS

You can set options for tools or specific files. Options are described in the chapter **Using the Compilers**. Options for tools will apply to all files that the tool can compile. For example, setting the FORTRAN options will apply to all FORTRAN files (.f, .f90, f95, etc.). To set tool options, select **Project Options** from the **Project** menu. This will bring up an option dialog:



Project Name: project
Target Filename: project.exe
Project Directory: C:\Absoft14.0
Prebuild Script:
Postbuild Script:

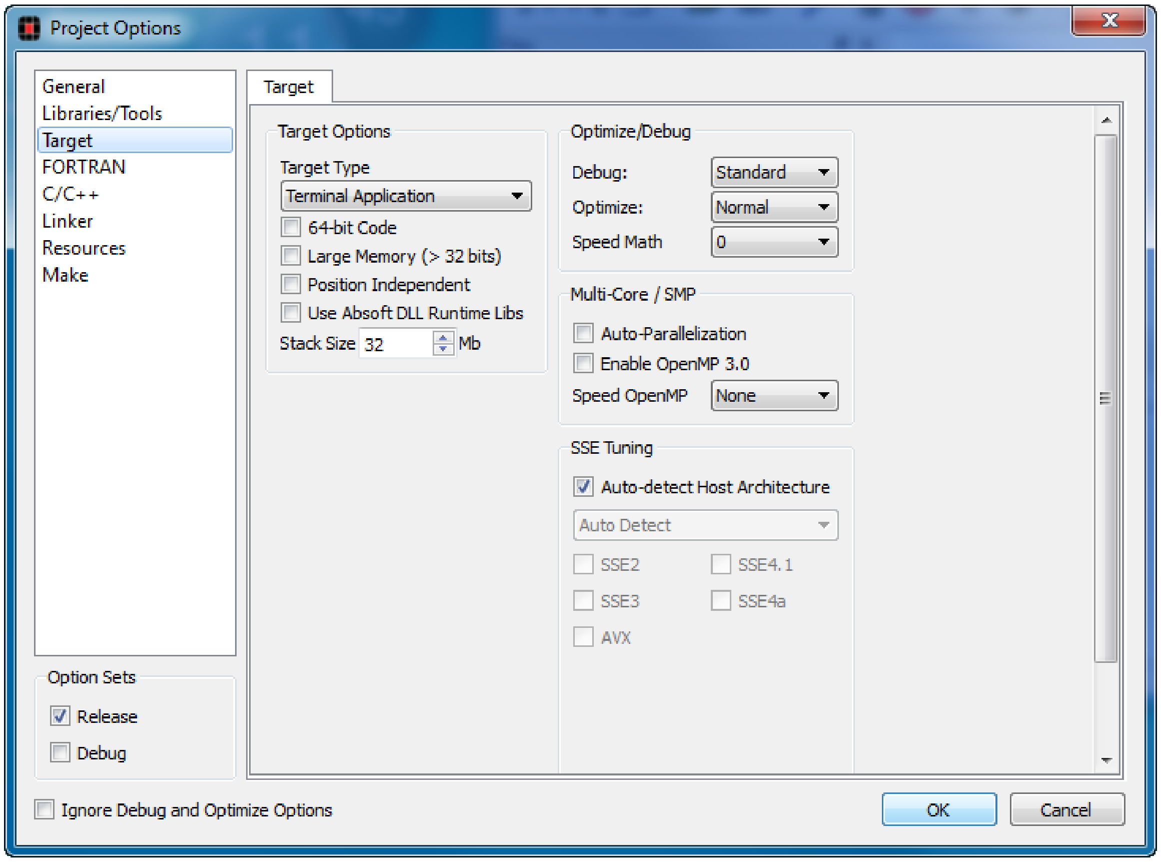
Selecting a tool name in the left list will show the corresponding options tabs in the right display. For example, selecting **FORTRAN** will show the **General, Warnings/errors, Format, Compatibility, Name Mangling and Optimize/Debug** tabs.

By default, any changes apply only to the option set for the active build configuration (shown in the Active Build combo box on the project tool bar). To apply the changes to the option sets for multiple build configurations, select the desired build configurations under **Option Sets**. If **Ignore Debug and Optimize Options** is checked, all options except for debug and optimization options will be applied to the selected builds. If **Ignore Debug and Optimize Options** is not checked, debug options will be applied to the selected debug builds (builds with names that end case insensitively in “debug”), and optimization options will be applied to the selected release builds.

When two or more builds are selected, the **Ignore Debug and Optimize Options** is automatically checked and you must explicitly uncheck it to change the behavior.

Target Options

Target Type


Project Options

General
Libraries/Tools
Target
FORTRAN
C/C++
Linker
Resources
Make

Option Sets
[✓] Release
[ ] Debug

Target

Target Options
Target Type: Terminal Application
[ ] 64-bit Code
[ ] Large Memory (> 32 bits)
[ ] Position Independent
[ ] Use Absoft DLL Runtime Libs
Stack Size 32 Mb

Optimize/Debug
Debug: Standard
Optimize: Normal
Speed Math 0

Multi-Core / SMP
[ ] Auto-Parallelization
[ ] Enable OpenMP 3.0
Speed OpenMP None

SSE Tuning
[✓] Auto-detect Host Architecture
Auto Detect
[ ] SSE2
[ ] SSE4.1
[ ] SSE3
[ ] SSE4a
[ ] AVX

[ ] Ignore Debug and Optimize Options

Terminal Application	Application that will be run from the terminal. This is the default.
MPI Application	Creates an application that will be built and executed with MPI.
Static Library	Creates a static library
DLL	Creates a DLL for Windows
AWE Application	Uses the Absoft AWE application framework. This framework can be used to create a native graphical application.

MULTIPLE BUILD AND OPTIONS EXAMPLE

To make multiple builds and setting options easier to understand, let's go through an example for editing the options for a single build:

Start a new project and add a file to it:

1. Open the AbsoftTools application
2. Select **New Project** from the **Project** menu
3. Enter the directory (type the directory name or click the "..." button and select the directory from the dialog)
4. Click Ok. You will now have a new project open
5. Right click in the Files window and select **F95 file** from the **New File In Project** menu.
6. Type a name in the dialog to save the file as. The name should appear in the Files list under F95. Options will be listed as -O2 and -m32 (or -m64 if you have a 64 bit processor)
7. Select "Debug" from the Active build combo box in the project tool bar. The Files list options should change to -g and -m32 (or -m64 if you have a 64 bit processor).

Create multiple builds:

8. From the **Build** menu, select **Create New Build...** An **Add Builds** dialog will appear.
9. Click **Add** under the Current Build Configurations list.
10. Type "Fast" and press enter.
11. Click **Add** again **Add** under the Current Build Configurations list.
12. Type "FullDebug" and press enter.
13. Click **Ok**.

Examine builds:

The Active Builds combo box will now contain "Release", "Debug", "Fast", and "FullDebug". When you change active builds by selecting the build from the Active Build combo box, the Files list should contain -O2 for Release and Fast, and -g for Debug and FullDebug.

Set Options for each set:

15) Select “FullDebug” from the Active Build combo box.
16) Select **Project Options** from the **Project** menu. The option dialog should appear. We are editing the FullDebug build, since FullDebug is selected as the Active build.
17) Select **FORTRAN** and click on the **General** tab.
18) Select **Check Pointers**
19) Click **Ok**. The options in the File list should read -O2, -m32 (or -m64) and -Rp.
20) Select “Fast” from the Active Build combo box.
21) From the **Project** menu, select **Project Options**. The option dialog should appear. We are editing the Fast build, since Fast is selected as the Active build.
22) Select **FORTRAN** and click on the **Optimize/Debug** tab.
23) Select **Level 4** from the optimize combo box.
24) Click **Ok**. The options in the File list should read -O4, -m32 (or -m64).

We now have 4 builds to choose from.

Fast	sets -O4 for fast optimizations
Release	sets -O2 for normal optimizations
Debug	sets -g for debug
FullDebug	sets -g standard debug and -Rp for pointer checking

Add -s to all builds:

25) Select **Project Options** from the **Project** menu. The option dialog should appear.
26) Select **FORTRAN** and click on the **Compatibility** tab and check **Static Storage**.
27) Select all four builds (Release, Debug, Fast, and FullDebug) under **Option Sets**.
28) Note that **Ignore Debug and Optimize Options** is checked when you have more than one build checked.
29) Click **Ok**.

All the builds now have -s as an option. You can verify this by selecting the different releases in the Active Builds combo box.

BUILDING

To build a project, click the build icon on the tool bar or select **Build** from the **Build** menu. The build dock will display the output from the build. If an error occurs in a FORTRAN compilation, the build tab will switch to the Errors/Warnings tab and a summary of the FORTRAN error will be displayed. Clicking on the error in the Errors/Warnings tab will open the file with the error and go to the line and column of the error. Right-clicking on an error and clicking explain will cause a dialog to appear with a detailed explanation of the error.

Selecting **Clean** from the **Build** menu will remove all files created during the build process.

Selecting **Rebuild** from the **Build** menu will clean a project and then build from scratch.

Clicking the stop icon in the tool bar or selecting **Stop** from the **Build** menu may be used to stop a build.

EXECUTE/DEBUG

Once an executable program is built, you may execute the program by selecting **Execute** from the **Build** menu or clicking the execute icon in the tool bar. Clicking **Debug** from the **Build** menu or clicking the debug icon will start Fx3 with the executable program.

Environment variables and program arguments are set in the Project Options dialog under the Run item. If OpenMP is checked in the **Target** options page, the environment variables will be populated with the OpenMP runtime variables.

FIND IN FILES

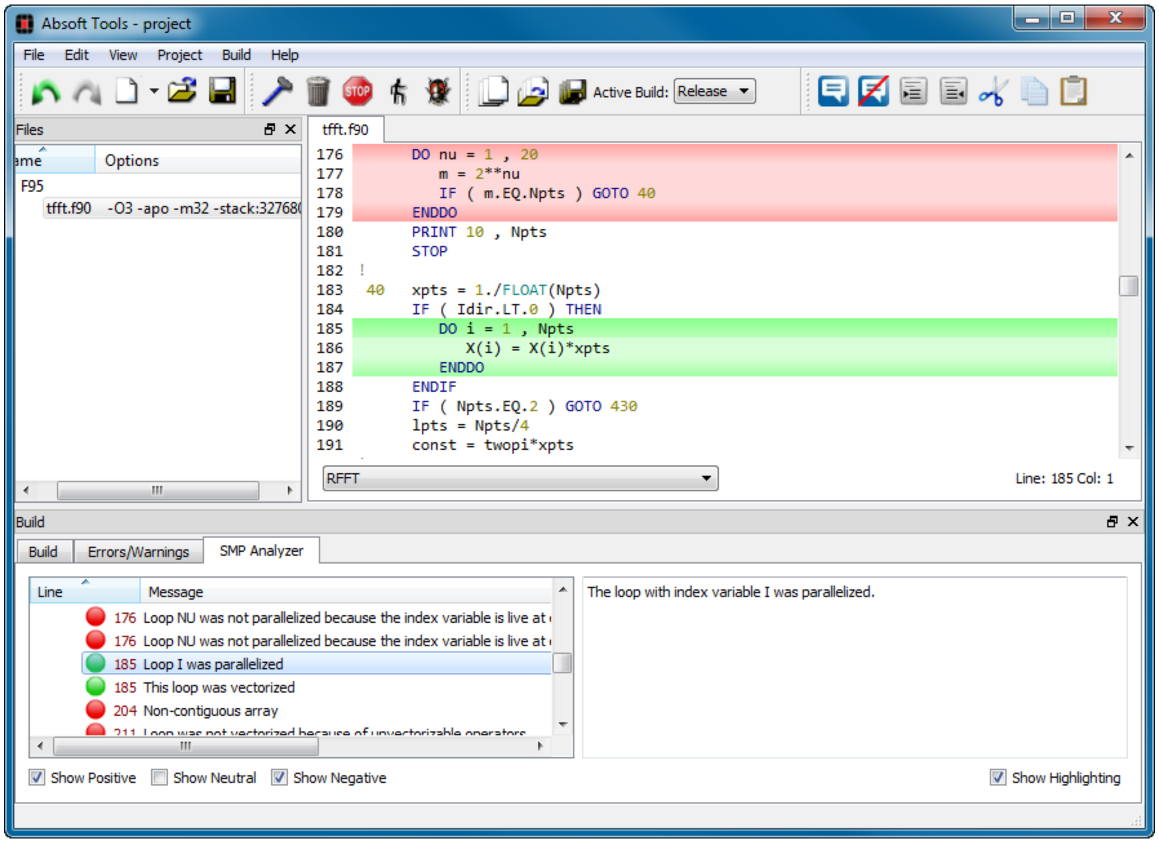
The Find in Files Dock can be hidden and shown by selecting **Find in Files** from the **View** menu. **Find in Files** will search each file in the project for the text specified. To search for text, make sure the **Find in Files** dock is visible, then enter the text to search for in the Find in File text box and press enter. Checking **Case Sensitive** will make case significant. Checking **Whole Words** will search for whole word references. A list of files will appear above the text. Clicking on a reference will open the file and go to the line the reference is in.

To replace, type the text to be replaced in the Find in File text box and the replacement text in the Replace With text box then press enter. This will only replace one occurrence in the current file; hitting the Replace button achieves the same result. If you hit the enter key again, it will replace the next occurrence. Both the Case Sensitive and Whole Words check boxes also apply when replacing. If you click Replace All button, it will replace every occurrence in all the files in the project, including the ones that are not opened. All the files that have been affected by this action are listed in the above text box. At the end, a summary of how many replacements have been made in how many files is displayed. Clicking on a reference will also open that file.

Note: if you want to replace every occurrence in one file only, use Find/Replace action under the Edit Menu.

SMP ANALYZER

When the **Auto-Parallelization** option (**-apo**) is checked, **Optimize Advanced** (**-O3**, auto-vectorization) is selected, or **Optimize Level 5** (**-O5**, auto-parallization and auto-vectorization) is selected, the Absoft SMP Analyzer is enabled. This tool provides detailed feedback and analysis of where advanced optimizations were performed and where they could not be performed. The analysis includes the line number of the code considered for optimization, a brief report, and an expanded explanation. The analyzer is selected by clicking on the **SMP Analyzer** tab in the **Build** window:



```
176       DO nu = 1 , 20
177          m = 2**nu
178          IF ( m.EQ.Npts ) GOTO 40
179       ENDDO
180       PRINT 10 , Npts
181       STOP
182 !
183  40   xpts = 1./FLOAT(Npts)
184       IF ( Idir.LT.0 ) THEN
185          DO i = 1 , Npts
186             X(i) = X(i)*xpts
187          ENDDO
188       ENDIF
189       IF ( Npts.EQ.2 ) GOTO 430
190       lpts = Npts/4
191       const = twopi*xpts
```

Line	Message
🔴 176	Loop NU was not parallelized because the index variable is live at
🔴 176	Loop NU was not parallelized because the index variable is live at
🟢 185	Loop I was parallelized
🟢 185	This loop was vectorized
🔴 204	Non-contiguous array
🔴 211	Loop was not vectorized because of unvectorizable operators

The loop with index variable I was parallelized.

When the SMP Analyzer is enabled, the source file is highlighted, indicating loops that were considered for optimization. Three types of highlighting are displayed:

1. Positive indicates the loop was optimized
2. Neutral indicates the loop could be optimized, but was not
3. Negative indicates the loop could not be optimized

Unchecking its box can selectively disable each type of highlighting. Typically, loops that could be optimized, but were not, have an iteration count too small to benefit from parallelization. Loops that cannot be optimized at all typically contain constructs that cannot be parallelized such as I/O statements and external function references with unknown side effects.

CHAPTER 5

Using the Compilers

This chapter describes how to use the Absoft Fortran 90/95 and FORTRAN 77 compilers to create executable files on the Windows operating system for the Intel and AMD families of processors. Beginning with an overview of the compilers, this chapter explains how to compile a small number of Fortran source files into an executable application. File name conventions and process control options are described first. The final sections of this chapter describe the compiler options in detail.

COMPILING PROGRAMS

Three methods of compiling programs are available: a traditional command line, the Absoft Developer Tools Interface, and makefiles. The Absoft Developer Tools Interface was discussed in the previous chapter. Makefiles and the Absoft make utility, `amake`, are described in the chapter **Building Programs**. All three methods allow you to compile source files quickly and easily

Source file names and compiler options are selected with the mouse pointer in the Absoft Developer Tools Interface. Arguments to the command line version are typed in on the command line.

USING THE COMMAND LINE

To use a command line version of any of the Absoft compilers, you must first open a command line window and set a number of environment variables that assist and control the use of the compilers. A shortcut to open a command line window has been provided in the Absoft menu: **Development Command Prompt (32-bit)** and **Development Command Prompt (64-bit)** for 32-bit applications and 64-bit applications respectively. These commands are shortcuts to the normal command line prompt for your system. On startup, they execute a batch file: **absvars32.bat** and **absvars64.bat** for 32-bit and 64-bit applications respectively. These batch files are located in the **Absoft15.0\Bin** directory and set a number of environment variables. Examine the batch file for further details.

A command line version of an individual compiler can be started with one of the commands: f95, f77, or cl.

```
f95 [options] file[s]
f77 [options] file[s]
cl  [options] file[s]
```

The various options are described in the specific compiler options sections next in this chapter.

FILE NAME CONVENTIONS

Compilation is controlled by the two compiler drivers: `f77` and `f95`. These drivers take a collection of files and, by default, produce an executable output file. Acceptable inputs to `f95` are:

File Type	Default form
Free format Fortran 90/95 source files	*file*`.f90` or *file*`.f95`
Free format Fortran 90/95 preprocessor files	*file*`.F90` or *file*`.F95`
Fixed format Fortran 90/95 source files	*file*`.f`
Fixed format Fortran 90/95 preprocessor files	*file*`.F`
Assembly language source files	*file*`.s`
Relocatable object files	*file*`.o`

Acceptable inputs to `f77` are:

File Type	Default form
FORTRAN 77 source files	*file*`.f` or *file*`.for`
FORTRAN 77 preprocessor files	*file*`.F` or *file*`.FOR`
Assembly language source files	*file*`.s`
Relocatable object files	*file*`.o`

File names that do not have one of these default forms are passed to the linker.

Output file names take the form:

File Type	Default form
Assembly language source files	*file*`.s`
Relocatable object files	*file*`.obj`
Precompiled module file	*file*`.mod`
Executable object files	*file*`.exe`

COMPILER PROCESS CONTROL

By default the `f77` and `f95` compiler drivers construct and execute the necessary commands to produce an executable application. This process requires compilation, assembly and linkage. As each of these processes finishes, all files that were created by the preceding stage are deleted. In some cases it may be desirable to save these intermediate files. Options controlling this are described here. These switches, in conjunction with the input file names, can also be used to stop the compilation process at any stage.

Generate Assembly Language (-S)

Specifying the **`–S`** option will cause the compilers to generate assembly language output in a form suitable for the system assembler. The file created will have the suffix "`.s`". For example, compiling `test.f` with the **`–S`** option will create `test.s`. If any C source files are given as arguments to `f77` or `f95`, this option will be passed to the C compiler. If no other compiler process control options are specified and there are no relocatable object files specified on the command line, the compilation process will halt after all Fortran 90/95, FORTRAN 77, and any C source code files have been compiled to assembly language source.

Generate Relocatable Object (-c)

Specifying the **`–c`** option will cause the compilers to generate relocatable object files. In the Windows environment, this option indicates that all source files (Fortran 90/95, FORTRAN 77, C, and assembly) should be processed to relocatable object files. If no linker options are present (see below), then the compilation process stops after all object files have been created. If any C source files are given as arguments to `f77` or `f95`, this option will be passed to the C compiler.

Passing Options To The Linker

For ease of use within the Windows environment, many of the options that are available to the system linker are also available to the `f77` and `f95` compiler drivers. Specifying any of these options indicates that all files specified on the command line should be processed through the linkage phase. Unless the **`–S`** or **`–c`** options are specified, all intermediate files (relocatable objects and/or assembly source) will be deleted. See the section *Linking Programs* in the chapter **Building Programs** for documentation on *link*. In brief, the options are as follows:

Executable File Name (-o *name*)

Use of the **`–o`** ***name*** option will cause the linker to produce an executable file called ***name***. The default is to produce an executable file called *`file`*`.exe` where *`file`* is the root name of the first source file provided on the command line.

Linker Options (-X)

Use the **`–X`*****option*** switch to pass an option directly to the linker. The FORTRAN 77 or Fortran 90/95 driver will pass ***option*** to the linker. If you want to pass an option that takes an argument, use the **`–X`** option twice.

Library Specification

On Windows, no special option is necessary to specify a library. Simply give the entire library name to pass a library to the linker.

Linker/Library Manager Preference (-use_vctools)

This option instructs the compiler to invoke the linker and library manager from the active version of Microsoft Visual C/C++ instead of using the versions bundled with Absoft Pro Fortran. The active version is determined from the contents of the environment variable named `VCINSTALLDIR`..

Preprocessor Options (-cpp and –no-cpp)

If a source file name has an upper case extension (F, FOR, F90, F95), the compiler first passes it to the C preprocessor to handle C-style includes, macros, and conditional directives. Use the **–cpp** option to force the compiler to invoke the C preprocessor regardless of the source file extension. Use the **–no-cpp** option to force the compiler to *not* invoke the C preprocessor regardless of the source file extension.

Code Generation Model (-mcmodel={small | medium})

This option specifies the code generation model for 64-bit processors. The **small** code model limits the combined code and data size to 2 gigabytes. The **medium** code model allows data to be larger than 2 gigabytes. The default is the **small** model.

Stack Size (-stack:reserved)

Use this option to establish the amount of memory in bytes reserved for stack use only. The default **reserved** amount is 0x800000 (8 megabytes). The **reserved** argument can be given in decimal or hexadecimal.

Application Type

By default, a stand-alone console or command line application. Use **-cons** to select this option on a command line. An application linked with AWE creates an application with a Windows style interface (see the chapter, **Windows Programming** for more information). Use **-awe** to select this option on a command line. Use the **-plainappl** option when you are creating an application with an interface that you supply. This type of application will have neither an AWE interface nor a console interface – you are responsible for the interface presented to the user. This option includes all of the standard Windows API import libraries and leaves the linker `-entry` and `-subsystem` arguments to their respective defaults.

Generate 32-bit code (-m32)

Use the **–m32** option to generate code that can be run on any X86 class processor.

Generate 64-bit code (-m64)

Use the **–m64** option to generate code that can only be run on AMD or Intel 64-bit processors.

Generate Debugging Information (-g)

Specifying the **–g** option will cause the compilers to include Dwarf2 symbol and line information appropriate for debugging a compiled program with Fx3, the Absoft debugger, or other source level debuggers which can read Dwarf2 symbol information.

OPTIMIZATIONS

These options control compile time optimizations to generate an application with code that executes more quickly. Absoft Fortran 90/95 is a globally optimizing compiler, so various optimizers can be turned on which affect single statements, groups of statements or entire programs. There are pros and cons when choosing optimizations; the application will execute much faster after compilation but the compilation speed itself will be slow. Some of the optimizations described below will benefit almost any Fortran code, while others should only be applied to specific situations.

Basic Optimizations (-O1)

The **–O1** option will cause most code to run faster and enables optimizations that do not rearrange your program. The optimizations include common subexpression elimination, constant propagation, and branch straightening. This option is generally usable with debugging options. **–cpu:host** is implied with this option.

Normal Optimizations (-O2)

The **–O2** option enables normal optimizers that can substantially rearrange the code generated for a program. The optimizations include strength reduction, loop invariant removal, code hoisting, and loop closure. This option is not usable with debugging options. **–cpu:host** is implied with this option.

Advanced Optimizations (-O3)

The **–O3** option enables advanced optimizers that can significantly rearrange and modify the code generated for a program. The optimizations include loop permutation (loop reordering), loop tiling (improved cache performance), loop skewing, loop reversal, unimodular transformations, forward substitution, and expression simplification. This option is not usable with debugging options. **–cpu:host** is implied with this option.

Advanced Optimizations (-O4)

The **–O4** option enables advanced optimizers that can significantly rearrange and modify the code generated for a program. The optimizations include all optimizations that are included with **–O3** as well as turning on inter-procedural analysis.

Dynamic AP (-O5)

The **–O5** option enables auto parallelization and dynamic load scheduling. When your program begins execution, the CPU load is measured and your program will automatically only use those processors that are actually available (idle). The optimizations include all optimizations that are included with **–Ofast**.

Automatic Parallelization (-apo)

The **–apo** option enables automatic parallelization of your source program.

CPU Specific Optimizations (–cpu:type)

Use the **–march=** option to target object code to a specific type of processor. Valid values for ***type*** are:

anyx86	any processor using the x86 instruction set
pentium4	Intel Pentium 4
em64t	Intel Pentium 4 with 64-bit extensions
core	Intel Core and Core 2
opteron	AMD Athlon 64/FX/X2 and AMD opteron
barcelona	AMD Opteron and Phenom with K10 Barcelona architecture
wolfdale	Intel Core I7 technology
host	automatically establishes ***type*** based on the processor in the machine that the program is compiled with. If the CPU type cannot be determined, *anyx86* is used.

Loop unrolling (**-U** and **-h*nn*** and **-H*nn***)

The Absoft Fortran 95 compiler has the ability to automatically unroll some of the loops in your source code. Loops may be unrolled by any power of two. Generally it is beneficial to unroll loops that execute a large number of iterations, while the benefit is small for loops that iterate only a few times. Due to this, only innermost loops are considered for unrolling. The **-h*nn*** option will cause the compiler to unroll your innermost loops *nn* times, where *nn* is any power of two. The **-H*nn*** option will cause the compiler to consider loops containing *nn* or fewer operations for unrolling. When the **–O3** option is used, the default is to only consider loops of a forty operations and unroll them four times. Using the **-U** option is equivalent to using **-h4 -H40**, causing innermost loops of forty or fewer operations to be unrolled four times. Loop unrolling will provide a speed increase in most cases, but will make your application larger and it will require more memory to compile. Consider the following example:

Original code:

```
SUBROUTINE SUB(A,N,X)
INTEGER A(100)

DO i=1,N
   A(i) = X*A(i)
END DO
RETURN
END
```

Becomes:

```
SUBROUTINE SUB(A,N,X)
INTEGER A(100)

DO i=1,MOD(N,4)
   A(i) = X*A(i)
END DO
DO i=MOD(N,4)+1,N,4
   A(i)   = X*A(i)
   A(i+1) = X*A(i+1)
   A(i+2) = X*A(i+2)
   A(i+3) = X*A(i+3)
END DO
RETURN
END
```

This is similar to the effect of loop unrolling. At least three comparisons and three branch instructions are saved each time the second loop is executed. Note that if your code contains extended range `DO` loops, unrolling loops will invalidate your program.

SSE2 instructions (**-msse2 and –mno-sse2**)

The **–msse2** and **–mno-sse2** options enable and disable respectively the use of SSE2 instructions for floating-point operations. This **–msse2** option is automatically enabled on processors which support SSE2. It may be disabled with the **–mno-sse2** option.

SSE3 instructions (**-msse3**)

The **-msse3** option enables the use of SSE3 instructions for floating-point operations. This option is automatically turned on when the **-march=host** option is specified and the host supports SSE3 instructions.

SSE4a instructions (-msse4a)

The **-msse4a** option enables the use of SSE4a instructions. This option is automatically turned on when the **-march=host** option is specified and the host supports SSE4a instructions.

SSSE4.1 instructions (-msse41)

The **-msse4a** option enables the use of SSE4.1 instructions. This option is automatically turned on when the **-march=host** option is specified and the host supports SSSE4.1 instructions.

Math Optimization Level (-speed_math=*n*)

The **-speed_math=*n*** option enables aggressive math optimizations that may improve performance at the expense of accuracy. Valid arguments for ***n*** are 0-11. See ***speed_math option*** Appendix for more information.

Enable OpenMP Directives (-openmp)

The **-openmp** option enables the recognition of OpenMP directives. OpenMP directives begin in column one in the form of:

`C$OMP` for fixed source format
`!$OMP` for free source format

OpenMP optimization Level (-speed_openmp=*n*)

The **–speed_openmp=** enables progressively more aggressive OpenMP optimizations on the value of ***n*** as follows:

n	effect
0	allow code optimization and movement through OpenMP Barrier
1	enable loose memory equivalence algorithm during optimization
2	Enable MU generation in SSA generation for OpenMP pragma
3	Enable CHI generation in SSA generation for OpenMP pragma
4	Allow loop unrolling for loops with OpenMP chunksize directive
5	Use a risky but faster algorithm to handle thread private common blocks

Each level includes all previous optimizations (e.g. **3** includes 0,1, and 2).

Safe Floating-Point (-safefp)

The **–safefp** option is used to disable optimizations that may produce inaccurate or invalid floating point results in numerically sensitive codes. The effect of this option is to preserve the FPU control word, enable NAN checks, disable `CABS` inlining, and disable floating-point register variables.

Report Parallelization Results (-LNO:verbose=on)

The **–LNO:verbose=on** option is used to display the results of the **–apo** option. It will report which loops were parallelized and which were not and why not.

Report Vectorization Results (--LNO:simd_verbose=on)

The **–LNO:simd_verbose=on** option is used to display the results of vectorization of loops which occurs at optimization levels greater than **–O3**. It will report which loops were vectorized and which were not and why not.

DEBUGGING

Generate Debugging Information (-g)

The **–g** option produces an object file containing debugging information with entry points, line numbers, and program symbols. This is the standard debugging option.

This **-gmin** switch produces an object file containing debugging information with entry points and line numbers only. No information for program symbols is produced. Use this option when you are only interested in stepping through the program.

FPU CONTROL OPTIONS

These options provide control over several aspects of the operation of the *Floating-Point Unit* of the processor including rounding mode, exception handling, control word state, and FPU stack integrity.

FPU Rounding Mode (-OPT:roundoff=*n*)

Set the level of acceptable rounding (# can be 0,1,2, or 3)

0 - Turn off optimizations that may be harmful to floating point calculations.
1 - Allow simple optimization that may affect floating point accuracy.
2 - Allow more extensive optimization that may affect floating point accuracy.
3 - Allow all optimizations affecting floating point accuracy.

FPU Exception Handling

When a floating-point exception is produced, the default action of an application is to supply an IEEE P754 defined value and continue. For undefined or illegal operations (such as divide by zero or square root of a negative number) this value will usually be either Infinity (INF) or Not A Number (NaN) depending on the floating-point operation.

Checking any of the exception boxes will cause the program to stop and produce a core dump, rather than continue, if the exception is encountered. If the program is being debugged, it will stop in the debugger at the statement line that caused the exception. The syntax for using this option on the command line is:

-TENV:*exception*=off

where ***exception*** is one of:

simd_imask – Invalid operation exception.
simd_dmask – Denormalized operand exception.
simd_zmask – Divide by zero exception.
simd_omask – Overflow exception.
simd_umask – Underflow exception.
simd_pmask - Precision exception.

ABSOFT FORTRAN 95 OPTIONS

The compiler options detailed in this section give you a great deal of control over the compilation and execution of Fortran 90/95 programs. Select the **Set Project Options** command in the **Configure** menu to access the Options Property Sheet. The Fortran 90/95 options fall into four categories: General, Compatibility, Miscellaneous, and Format.

Each option is listed with the corresponding option letter(s) and a description. Options that take arguments may optionally have a space to separate the option from its argument. The only exceptions are the `B` and `N` options; they cannot have a space between the option and its argument (e.g. `-N33`).

Compiler control

These options control various aspects of the compilation process such as warning level, verbosity, code generation, where module files can be found, and the definition of compiler directive variables. The generation of debugging information, for the symbolic source-level debugger, Fx, is also controlled by compiler control options.

Warn of Non-Standard usage (-en)

Use of the **-en** option will cause the compiler to issue a warning whenever the source code contains an extension to the Fortran 90/95 standard. This option is useful for developing code which must be portable to other environments.

Suppress warnings (-w)

Suppresses the listing of warning messages. For example, unreachable code will generate a warning message.

Suppress Warning number(s) (-Z*nn*)

Use the **-Z*nn*** option to suppress messages by message number, where ***nn*** is a message number. This option is useful if the source code generates a large number of messages with the same message number, but you still want to see other messages. See also the **-z*nn*** option.

Quiet (-q)

The Absoft Fortran 90/95 compiler normally displays information to standard output (the command line window) as it compiles an application. Enabling the **-q** option will suppress any messages printed to standard output. Errors will still be printed, however.

Verbose (-v)

Enabling the **-v** option will cause the **f95** command, described later in the **Building Programs** chapter, to display the commands it is sending to the compiler and linker.

Warning level (-z*nn*)

Use the **-z*nn*** option to suppress messages by message level, where ***nn*** is a message level. Diagnostics issued at the various levels are:

0	errors, warnings, cautions, notes, comments
1	errors, warnings, cautions, notes
2	errors, warnings, cautions
3	errors, warnings
4	errors

The default level is **-z3**; the compiler will issue error and warning diagnostics, but not cautions, notes, and comments. See also the **-Z*nn*** option.

Error Handling (-dq and -ea)

Normally, the Absoft Fortran 90/95 compiler will stop if more than 100 errors are encountered. This many errors usually indicate a problem with the source file itself or the inability to locate an `INCLUDE` file. If you want the compiler to continue in this circumstance, select the **Allow > 100** or **-dq** option. The **Stop on Error** or **-ea** option will cause the `f95` compiler to abort the compilation process on the first error that it encounters.

Output Version number (-V)

The **-V** option will cause the `f95` compiler to display its version number. This option may be used with or without other arguments.

Default Recursion (-eR)

If you select the **-eR** option, all `FUNCTION`s and `SUBROUTINE`s are given the `RECURSIVE` attribute. Normally, if the compiler detects a recursive invocation of a procedure not explicitly given the `RECURSIVE` attribute, a diagnostic message will be issued. The **-eR** option disables this.

Max Internal Handle (-T *nn*)

This option is used to change the number of handles used internally by the compiler. Under most conditions, the default value of 100000 handles is sufficient to compile even extremely large programs. However, under certain circumstances, this value may be exceeded and the compiler will issue a diagnostic indicating that the value should be increased.

The default value can be increased by powers of ten by specifying the **-T** *nn*, where *nn* is a positive integer constant. When this option is specified, the number of handles will be 100000×10^{nn} bytes.

Temporary string size (-t *nn*)

In certain cases the compiler is unable to determine the amount of temporary string space that string operations will require. The compiler will assume that the operation in question will require 1024 bytes of temporary string space. This default value can be increased by powers of ten by specifying the **-t** *nn*, where *nn* is a positive integer constant. When this option is specified, the default temporary string size will be 1024×10^{nn} bytes.

Set Module Paths (-p path)

The Absoft Fortran 90/95 compiler will automatically search the local directory for precompiled module files. If module files are maintained in other directories, use the **-p** option to specify a path or complete file specification. See **Fortran 90/95 Module Files** in the chapter, **Building Programs** for more information.

Module File Output Path (-YMOD_OUT_DIR=*path*)

The Absoft Fortran 90/95 compiler will automatically create module files in the current directory. If module files are to be maintained in another directory, the **-YMOD_OUT_DIR=*path*** option can be used to specify target directory.

Compatibility - F95 Options

These options allow Absoft Fortran 90/95 to accept older or variant extensions of Fortran source code from other computers such as mainframes. Many of these can be used for increased compatibility with Fortran compilers on various mainframe computers.

Disable compiler directive (-x*directive*)

The **-x** option is used to disable compiler directives in the source file. ***directive*** may be any of the following:

```
NAME
FIXED
FREE
STACK
INTEGER
```

See the section **Absoft Fortran 90/95 Compiler Directives** for more information on using compiler directives in your source code.

Integer Sizes (-i2 and -i8)

Without an explicit length declaration, `INTEGER` data types default to thirty-two bits or four bytes (`KIND=4`). The **–i2** option can be used to change this default length to sixteen bits or two bytes (`KIND=2`). The **–i8** option can be used to change the default `INTEGER` size to 64 bits or 8 bytes (`KIND=8`). However, an explicit length specification in a type declaration statement always overrides the default data length.

Demote Double Precision to Real (-dp)

The **-dp** option will cause variables declared in a `DOUBLE PRECISION` statement and constants specified with the `D` exponent to be converted to the default real kind. Similarly, variables declared in a `DOUBLE COMPLEX` statement and complex constants specified with `D` exponents will be converted to the complex kind in which each part has the default real kind.

Promote REAL to REAL(KIND=8) (-N113)

Without an explicit length declaration, single precision `REAL` and `COMPLEX` data types default to thirty-two bits or four bytes (`KIND=4`) and sixty-four bits or eight bytes (`KIND=4`), respectively. The **-N113** option is used to promote these to their double precision equivalents (`KIND=8`). This option does not affect variables which appear in type statements with explicit sizes (such as `REAL (KIND=4)` or `COMPLEX (KIND=4)`).

One trip DO loops (-ej)

Fortran 90/95 requires that a `DO` loop not be executed if the iteration count, as established from the `DO` parameter list, is zero. The **-ej** option will cause all `DO` loops to be executed at least once, regardless of the initial value of the iteration count.

Static storage (-s)

The **-s** option is used to allocate local variables statically, even if SAVE was not specified as an attribute. In this way, they will retain their definition status on repeated references to the procedure that declared them. Two types of variables are not allocated to static storage: variables allocated in an ALLOCATE statement and local variables in recursive procedures.

Check Array Boundaries (-Rb)

When the **–Rb** compiler option is turned on, code will be generated to check that array indexes are within the bounds of an array. Assumed size arrays whose last dimension is * cannot be checked. In addition, file names and source code line numbers will be displayed with all run time error messages.

Check Array Conformance (-Rc)

The **–Rc** compiler option is used to check array conformance. When array shapes are not known at compile time and where they must conform, runtime checks are created to insure that two arrays have the same shape.

Check Substrings (-Rs)

When the **–Rs** compiler option is turned on, code will be generated to check that character substring expressions do not specify a character index outside of the scope of the character variable or character array element.

Check Pointers (-Rp)

Use **–Rp** compiler option is used to generate additional program code to insure that Fortran 90 style POINTER references are not null.

Character Argument Parameters (-YCFRL={0|1})

Use the **–YCFRL=1** option to force the compiler to pass CHARACTER arguments in a manner that is compatible with g77 and f2c protocols. Use the **–YCFRL=0** option (the default) to pass CHARACTER arguments in a manner that is compatible with Absoft Compilers on other platforms. **Note:** this option should be used consistently on all files that will be linked together into the final application.

Pointers Equivalent To Integers (YPEI={0|1})

This option controls whether or not the compiler will allow a CRI style pointer to be equivalent to an integer argument. By default the Absoft Fortran 90/95 compiler allows this. Even with this relaxed error checking the compiler will correctly choose the right interface for the following example:

```
interface generic
  subroutine specific1(i)
      integer i
  end subroutine specific1
  subroutine specific2(p)
      integer i
      pointer (p,i)
  end subroutine specific2
end interface
call generic(i)
call generic(loc(i))
end
```

Regardless of the switch setting, this example will compile and the executable generated will be equivalent to:

```
call specific1(i)
call specific2(loc(i))
```

DVF/CVF Character Arguments (-YVF_CHAR)

The **–YVF_CHAR** option causes the compiler to pass and expect `CHARACTER` arguments in a manner compatible with Digital/Compaq Visual Fortran. The length of the argument (as a value) immediately follows the argument itself as opposed to the more common method of passing the length(s) at the end of the argument list.

Format - F95 Options

For compatibility with other Fortran environments and to provide more flexibility, the compiler can be directed to accept source code that has been written in a number of different formats. The two basic formats are free-form and fixed-form.

Free-Form (-f free)

The **-f free** option instructs the compiler to accept source code written in the format for the Fortran 90/95 Free Source Form. This is the default for file names with an extension of “`.f95`”.

Fixed-Form (-f fixed)

The **-f fixed** option instructs the compiler to accept source code written in the format for the Fortran 90/95 Fixed Source Form which is the same as the standard FORTRAN 77 source form.

Alternate Fixed form (-f alt_fixed)

The **-f alt_fixed** option instructs the compiler to accept source code written in following form:

If a tab appears in columns 1 through 5, then the compiler examines the next character. If the next character is not a letter (a-z, or A-Z) then it is considered a continuation character and normal rules apply. If it is a zero, a blank, another tab, or a letter, the line is not a continuation line.

Fixed line length (-W *nn*)

Use the **-W** option to set the line length of source statements accepted by the compiler in Fixed-Form source format. The default value of *nn* is 72. The other legal values for *nn* are 80 and 132 — any other value produces an error diagnostic.

YEXT_NAMES={ASIS | UCS | LCS}

The **-YEXT_NAMES** option is used to specify how the external names of globally visible symbols, such as `FUNCTION` and `SUBROUTINE` names, are emitted. By default, names are emitted entirely in lower case. Set this option to **UCS** to emit names entirely in upper case. Set this option to **ASIS** to force external names to emitted exactly as they appear in the source program. This option controls how external names will appear to other object files.

Treat as Big-Endian (-N26)

Use this option to force the compiler to consider the byte ordering of all unformatted files to be big-endian by default. The `CONVERT` specifier in the `OPEN` statement may be used to override this setting for individual files. In the absence of specification, handling of byte ordering depends on the system.

Treat as Little-Endian (-N27)

Use this option to force the compiler to consider the byte ordering of all unformatted files to be little-endian by default. See discussion under N26

External Symbol Prefix (-YEXT_PFX=*string*)

The **-YEXT_PFX** option can be used to prepend a user specified ***string*** to the external representation of external procedure names.

External Symbol Suffix (-YEXT_SFX=*string*)

The **-YEXT_SFX** option can be used to append a user specified ***string*** to the external representation of external procedure names.

Escape Sequences in Strings (-YCSLASH=1)

If the **-YCSLASH=1** option is turned on, the compiler will transform the following escape sequences marked with a ‘\’ embedded in character constants:

```
\a                Audible Alarm (BEL, ASCII 07)
\b                Backspace (BS, ASCII 8)
\f                Form Feed (FF, ASCII 12)
\n                Newline (LF, ASCII 10)
\r                Carriage Return (CR, ASCII 13)
\t                Horizontal Tab (HT, ASCII 09)
\v                Vertical Tab (VT, ASCII 11)
\xh[h]            Hexidecimal, up to 2 digits
\o[o[o]]          Octal number, up to 3 digits
\\                Backslash
```

The default is **-YCSLASH=0**.

No Dot for Percent (-YNDFP=1)

This option instructs the compiler to disallow the use of a ‘.’ (period) as a structure field component dereference operator. The default is to allow both ‘%’ (percent), which is the Fortran 90/95 standard, and a period which is typically used with DEC style RECORD declarations. The use of a period may cause certain Fortran 90/95 conforming programs to be mis-interpreted (a period is used to delineate user defined operators and some intrinsic operators). The default is **-YNDFP=0**. This switch implements Fortran 90/95 standard parsing for structure component referencing.

MS Fortran 77 Directives (-YMS7D)

The **-YMS7D** option causes the compiler to recognize Microsoft Fortran 77 style directives in the form of `$directive` where the dollar-sign character is in column one of the source file. `directive` must be from the set of supported MS directives.

Miscellaneous - F95 Options

These options are used to control the global names of `COMMON` blocks. Their primary use is for managing the character case and decoration applied to `COMMON` block names when interfacing with external procedures written in FORTRAN 77 or the C Programming Language.

COMMON Block Name Prefix (-YCOM_PFX=*string*)

The **-YCOM_PFX** option can be used to prepend a user specified ***string*** to the external representation of `COMMON` block names.

COMMON Block Name Suffix (-YCOM_SFX=*string*)

The **-YCOM_SFX** option can be used to append a user-specified ***string*** to the external representation of `COMMON` block names.

COMMON Block Name Character Case (**-YCOM_NAMES={ ASIS | UCS | LCS }**)

The **-YCOM_NAMES** option is used to specify how the external names of `COMMON` blocks are emitted. The default (-**YCOM_NAMES=LCS**) is to emit `COMMON` block names entirely in lower case. Set this option to **UCS** to emit names entirely in upper case.

Loop unrolling (**-U** and **-h** ***nn*** and **-H** ***nn***)

The Absoft Fortran 95 compiler has the ability to automatically unroll some of the loops in your source code. Loops may be unrolled by any power of two. Generally it is beneficial to unroll loops that execute a large number of iterations, while the benefit is small for loops that iterate only a few times. Due to this, only innermost loops are considered for unrolling. The **-h** *nn* option will cause the compiler to unroll your innermost loops *nn* times, where *nn* is any power of two. The **-H** *nn* option will cause the compiler to consider loops containing *nn* or fewer operations for unrolling. When the **–O3** option is used, the default is to only consider loops of a forty operations and unroll them four times. Using the **-U** option is equivalent to using **-h 4 -H 40**, causing innermost loops of forty or fewer operations to be unrolled four times. Loop unrolling will provide a speed increase in most cases, but will make your application larger and it will require more memory to compile. Consider the following example:

Original code:

```
SUBROUTINE SUB(A,N,X)
INTEGER A(100)

DO i=1,N
   A(i) = X*A(i)
END DO
RETURN
END
```

Becomes:

```
SUBROUTINE SUB(A,N,X)
INTEGER A(100)

DO i=1,MOD(N,4)
   A(i) = X*A(i)
END DO
DO i=MOD(N,4)+1,N,4
   A(i)   = X*A(i)
   A(i+1) = X*A(i+1)
   A(i+2) = X*A(i+2)
   A(i+3) = X*A(i+3)
END DO
RETURN
END
```

This is similar to the effect of loop unrolling. At least three comparisons and three branch instructions are saved each time the second loop is executed. Note that if your code contains extended range `DO` loops, unrolling loops will invalidate your program.

Add Microsoft GLOBAL prefix (**-YMSFT_GLB_PFX**)

This option causes common block names declared with GLOBAL to be prefixed with "__imp_" for the Microsoft linker. It is useful when creating DLLs that will share data through Fortran common blocks.

Other F95 Options

The following options are not available with the graphical interface to the compiler but may used with the command line interface or the make facility (See the chapter, **Building Programs**).

Conditional Compilation (-YX)

Statements containing an X or a D in column one are treated as comments by the compiler unless the **-YX** compiler option is selected. This option allows a restricted form of conditional compilation designed primarily as a means for removing debugging code from the final program. When the **-YX** option is selected, a blank character replaces any occurrence of an X or a D in column one. The only source formats for which conditional compilation is valid are standard FORTRAN 77, VAX Tab-Format, and wide format.

Check Argument Interface (-Ra)

When the **–Ra** compiler option is specified, code will be generated to check 1) actual and dummy argument count mismatches, 2) non-writable arguments passed to dummy arguments declared as `INTENT OUT` or `INTENT INOUT`, and 3) type/kind mismatches. Note that this option requires that all components of an executable be compiled with the **–Ra** option, including module procedures.

Check Argument Count (-Rn)

The **–Rn** compiler option is used to check actual and dummy argument count mismatches. Note that this option requires that all components of an executable be compiled with the **-Rn** option, including module procedures.

Disable Default Module File Path (-nodefaultmod)

The Absoft Fortran 90/95 compiler will automatically search the directory `%ABSOFT%\F90INC` for precompiled module files. Use the **–nodefaultmod** to disable this.

Variable Names Case Sensitivity (-YVAR_NAMES={ASIS | UCS | LCS})

The **-YVAR_NAMES** option is used to specify how the case of variable names is treated. By default, variable names are processed entirely in lower case (**LCS**), regardless of the how they appear in the source code. Set this option to **UCS** to fold variable names to upper case. Set this option to **ASIS** to force variable names to be processed exactly as they appear in the source program.

Variable Names Case Sensitivity (-YALL_NAMES={ASIS | UCS | LCS})

The **-YALL_NAMES** option is used to specify how the case of all symbolic names is treated. By default, symbolic names are processed entirely in upper case (**UCS**), regardless of the how they appear in the source code. Set this option to **LCS** to fold all symbolic names to lower case. Set this option to **ASIS** to force symbolic names to be processed exactly as they appear in the source program. This option is the same as using the **-YVAR_NAMES**, **-YCOM_NAMES**, and **-YEXT_NAMES**, which may appear after the **-YALL_NAMES** option to control an individual symbolic name type.

Ignore CDEC$ directives (-YNO_CDEC)

The compiler recognizes CDEC$ directives that contain conditional compilation directives. Use this option disable them.

Absoft Fortran 90/95 Compiler Directives

Compiler directives are lines inserted into source code that specify actions to be performed by the compiler. They are not Fortran 90/95 statements. If you specify a compiler directive while running on a system that does not support that particular directive, the compiler ignores the directive and continues with compilation.

A *compiler directive line* begins with the characters `CDIR$` or `!DIR$`. How you specify compiler directives depends on the source form you are using.

If you are using fixed source form, indicate a compiler directive line by placing the characters `CDIR$` or `!DIR$` in columns 1 through 5. If the compiler encounters a nonblank character in column 6, the line is assumed to be a compiler directive continuation line. Columns 7 and beyond can contain one or more compiler directives separated by commas. If you are using the default 72 column width, characters beyond column 72 are ignored. If you have specified 80 column lines, characters beyond column 80 are ignored.

If you are using free source form, indicate a compiler directive line by placing the characters `!DIR$` followed by a space, and then one or more compiler directives separated by commas. If the position following the `!DIR$` contains a character other than a blank, tab, or newline character, the line is assumed to be a compiler directive continuation line.

NAME Directive

The `NAME` directive allows you to specify case-sensitivity for externally visible names. You can use this directive, for example, when writing calls to C routines or declaring functions to be called outside of Fortran 90/95. The case-sensitive external name is specified on the `NAME` directive, in the following format:

```
!DIR$ NAME (fortran=“external” [,fortran=“external”]...)
```

where: *fortran* is the name used for the object throughout the Fortran program whenever the external name is referenced.

external is the external name.

FREE Directive

The `FREE` directive specifies that the source code in the program unit is written in the free source form. The `FREE` directive may appear anywhere within your source code. The format of the `FREE` directive is:

```
!DIR$ FREE
```

You can change source form within an `INCLUDE` file. After the `INCLUDE` file has been processed, the source form reverts back to the source form that was being used prior to processing the `INCLUDE` file.

FIXED Directive

The `FIXED` directive specifies that the source code in the program unit is written in the fixed source form. The `FIXED` directive may appear anywhere within your source code. The format of the `FIXED` directive is:

```
!DIR$ FIXED
```

You can change source form within an `INCLUDE` file. After the `INCLUDE` file has been processed, the source form reverts back to the source form that was being used prior to processing the `INCLUDE` file.

STACK Directive

The `STACK` directive causes the default storage allocation to be the stack in the program unit that contains the directive. This directive overrides the **`-s`** command line option in specific program units of a compilation unit. The format for this compiler directive is:

```
!DIR$ STACK
```

ABSOFT FORTRAN 77 OPTIONS

The compiler options detailed in this section are provided for compatibility with the Absoft legacy FORTRAN 77 compiler. This compiler is no longer supplied as all of its capabilities have been incorporated into the Fortran 95 compiler. These options are deprecated and will eventually no longer be supported. It is suggested that the equivalent Fortran 95 options be used instead.

Each option is listed with the corresponding option letter(s) and a short description. Options that take arguments (e.g. `-h 4` or `-o file`) must have a space to separate the option from the argument. The only exceptions are the `B` and `N` options; they do not have a space between the option and the argument (e.g. `-N33`).

General - F77 Options

These options control the general characteristics of the FORTRAN 77 components of the program being built. They are primarily concerned with debugging information.

Suppress Warnings (-w)

Suppresses the listing of warning messages. For example, unreachable code or a missing label on a `FORMAT` statement generate warning messages. Compile time diagnostic messages are divided into two categories: errors and warnings. Error messages indicate that the compiler was unable to generate an output file. Warning messages indicate that some syntactic element was not appropriate, but the compiler was able to produce an output file.

Warn of non-ANSI Usage (-N32)

Use of the **-N32** option will cause the compiler to issue a warning whenever the source code contains an extension to the ANSI FORTRAN 77 standard (American National Standard Programming Language FORTRAN, X3.9-1978). This option is useful for developing code which must be portable to other environments.

Quiet (-q)

The Absoft Fortran 77 compiler normally displays information to standard output (the command line window) as it compiles an application. Enabling the **-q** option will suppress any messages printed to standard output. Errors will still be printed, however.

Show Progress (-v)

Enabling the **-v** option will display the individual commands that are sent to the command line window, such as the front and back ends of the compiler and the linker.

Check Array Boundaries (-C)

When the **-C** compiler option is turned on, code will be generated to check that array indexes are within the bounds of an array. Exceptions: arrays whose last dimension is * and dummy arguments whose last dimension is 1 cannot be checked. In addition, file names and source code line numbers will be displayed with all run time error messages.

Conditional compilation (-x)

Statements containing an X or a D in column one are treated as comments by the compiler unless the **-x** compiler option is selected. This option allows a restricted form of conditional compilation designed primarily as a means for removing debugging code from the final program. When the **-x** option is selected, any occurrence of an X or a D in column one is replaced by a blank character. The only source formats for which conditional compilation is valid are standard FORTRAN 77, VAX Tab-Format, and wide format. The compiler also incorporates a complete set of statements for conditional compilation which are described in the **Conditional Compilation Statements** section **The Fortran Program** chapter of the *Absoft Fortran Language Reference Manual*.

Max Internal Handle (-T nn)

This option is used to change the number of handles used internally by the compiler. Under most conditions, the default value of 20000 handles is sufficient to compile even extremely large programs. However, under certain circumstances, this value may be exceeded and the compiler will issue a diagnostic indicating that the value should be increased.

Temporary string size (-t *nn*)

In certain cases the compiler is unable to determine the amount of temporary string space that string operations will require. This undetermined length occurs when the `REPEAT` function is used or when a `CHARACTER*(*)` variable is declared in a subroutine or function. In these cases, the compiler will assume that the operation in question will require 1024 bytes of temporary string space. This default value can be changed by specifying the **-t** *nn*, where *nn* is a positive integer constant. When this option is specified, the default temporary string size will be *nn* bytes.

Compiler Directives (-D*name*[=*value*])

Use this text box to enter the names and optional values of conditional compilation variables. The **-D** option is used to define conditional compilation variables from the command line. *value* can only be an integer constant. If *value* is not present, the variable is given the value of 1. Conditional compilation is described in the **Conditional Compilation Statements** section of the chapter **The Fortran Program** of the *Absoft Fortran Language Reference Manual*.

Compatibility - F77 Options

These options allow Absoft Fortran 77 to accept older or variant extensions of FORTRAN source code from other computers such as mainframes. Many of these can be used for increased compatibility with FORTRAN compilers on various mainframe computers.

Folding to Lower Case (-f)

The **-f** option will force all symbolic names to be folded to lower case. By default, the compiler considers upper and lower case characters to be unique, an extension to FORTRAN 77. If you do not require case sensitivity for your compilations or specifically require that the compiler not distinguish between case, as in FORTRAN 77, use this option. This option should be used for compatibility with VAX and other FORTRAN environments.

Static Storage (-s)

In FORTRAN 66, all storage was static. If you called a subroutine, defined local variables, and returned, the variables would retain their values the next time you called the subroutine. FORTRAN 77 establishes both static and dynamic storage. Storage local to an external procedure is dynamic and will become undefined with the execution of a `RETURN` statement. The `SAVE` statement is normally used to prevent this, but the **-s** compiler option will force all program storage to be treated as static and initialized to zero.

Folding to Upper Case (-N109)

By default, the compiler considers upper and lower case characters to be unique, an extension to FORTRAN 77. If you do not require case sensitivity for your compilations or specifically require that the compiler not distinguish between case, as in FORTRAN 77, including the **-N109** option on the compiler invocation command line will force all symbolic names to be folded to upper case.

One-Trip DO Loops (F66) (-d)

FORTRAN 66 did not specify the execution path if the iteration count of a `DO` loop, as established from the `DO` parameter list, was zero. Many processors would execute this loop once, testing the iteration count at the bottom of the loop. FORTRAN 77 requires that such a `DO` loop not be executed. The **-d** option will cause all `DO` loops to be executed at least once, regardless of the initial value of the iteration count.

Promote REAL and COMPLEX (-N113)

Without an explicit length declaration, single precision `REAL` and `COMPLEX` data types default to thirty-two bits (four bytes) and sixty-four bits (eight bytes), respectively. The **-N113** option is used to promote these to their double precision equivalents: `DOUBLE PRECISION` and `DOUBLE COMPLEX`. This option does not affect variables which appear in type statements with explicit sizes (such as `REAL*4` or `COMPLEX*8`).

Integer Sizes (-i2 and -i8)

Without an explicit length declaration, `INTEGER` and `LOGICAL` data types default to thirty-two bits (four bytes). The **–i2** option can be used to change this default length to sixteen bits (two bytes) for both `INTEGER` and `LOGICAL`. The **–i8** option can be used to change the default `INTEGER` size to 64 bits (8 bytes). However, an explicit length specification in a type declaration statement always overrides the default data length.

Format - F77 Options

For compatibility with other FORTRAN environments and to provide more flexibility, the compiler can be directed to accept source code that has been written in a variety of different formats. The default setting is to accept only ANSI standard FORTRAN source code format. See the chapter **The Fortran Program** of the *Absoft Fortran Language Reference Manual* for more information on alternative source code formats.

ANSI Fortran 77 Fixed

The default source form is ANSI FORTRAN 77 as described in the chapter **The Fortran Program** of the *Absoft Fortran Language Reference Manual*. There is no option for this setting.

Fortran 90 Free-Form (-8)

Use of the **-8** option instructs the compiler to accept source code written in the format for the Fortran 90 Free Source Form.

VAX Tab-Format (-V)

Use of the **-V** option causes the compiler to accept source code in the form specified by VAX Tab Format.

Wide Format (-W)

Use of the **-W** option causes the compiler to accept statements which extend beyond column 72 up to column 132.

Treat as Big-Endian (-N26)

Use this option to force the compiler to consider the byte ordering of all unformatted files to be big-endian by default. The `CONVERT` specifier in the `OPEN` statement may be used to override this setting for individual files. In the absence of specification, handling of byte ordering depends on the system

Treat as Little-Endian (-N27)

Use this option to force the compiler to consider the byte ordering of all unformatted files to be little-endian by default. See discussion under N26.

Escape Sequences in Strings (-K)

If the **-K** option is turned on, the compiler will transform certain escape sequences marked with a '`\`' embedded in character constants. For example '`\n`' will be transformed into a newline character for your system. Refer to chapter **The Fortran Program** of the *Absoft Fortran Language Reference Manual* for more information on the escape sequences that are supported.

DLL Compatibility

The actual form of an external entry point name in a DLL (an exported name) is dependent on the system that created the DLL. Various forms of *name mangling* are employed by programming language suppliers. Name mangling involves decorating external names in such a way that they do not conflict with other global names or so that they can supply argument list and stack size information to the linker. In addition, there are two call/return sequences defined in the Win32 API (Application Programming Interface):

CDECL

This is the default call/return sequence generated by Absoft compilers. The caller pushes arguments from right to left onto the stack. The callee accesses the parameters in the stack and returns. The caller cleans up (removes the arguments from) the stack.

STDCALL

This is the call/return sequence used by most of the Windows 32-bit operating system functions. The caller pushes arguments from right to left onto the stack. The callee accesses the parameters in the stack, but is also responsible for removing them from the stack.

Obviously, the two mechanisms cannot be intermixed and passing too many or two few arguments with the STDCALL protocol is disastrous (the wrong number of arguments will be removed from the stack by the callee). As a protection against this, STDCALL function names in the object code are often *mangled* by appending a commercial at sign (‘@’) and the size of the stack (in bytes) to the function name. In this way, the caller and the callee must agree on the number of arguments, or the program will not link.

CHAPTER 6

Porting Code

This chapter describes issues involved in porting FORTRAN 77 code from other platforms. One of the major design goals for Absoft Fortran 77 is to permit easy porting of FORTRAN source code from mainframe computers such as VAX and IBM, and from workstations such as Sun. The result is the rich set of statements and intrinsic functions accepted by Absoft Fortran 77. The last section of this chapter describes Windows-specific issues about porting code.

The Absoft Fortran 77 compiler is recommended for porting most legacy codes because of the number extensions and features it supports. Consequently, FORTRAN 77 options and language features will be described in this chapter. However, in most cases, the Fortran 90/95 compiler has equivalent options and can also be used. Refer to the **Using the Compilers** chapter for information on Fortran 90/95 compile time options.

As a general rule when porting code, use the following compiler option:

-s Force all program storage to be treated as static and initialized to zero.

Ported programs that have incorrect runs or invalid results are usually caused by the differences between Windows and other environments such as floating point math precision or stack-size issues. See the section **Other Porting Issues** later in this chapter for special considerations when porting code to Windows. In addition, you may want to use this option:

-C Check array boundaries and generate better runtime errors. Using this option makes programs slightly larger and they will execute slower.

If you want to use the Absoft debugger, Fx3, add the **-g** option to generate debugging information.

PORTING CODE FROM VAX

Absoft FORTRAN 77 automatically supports most of the VAX FORTRAN language extensions. Below is a list of key VAX FORTRAN extensions that are supported and a list of those that are not supported. Using various options, the compiler can also accept VAX Tab-Format source lines and/or 132-column lines. Otherwise, only ANSI FORTRAN 77 fixed format lines are accepted.

Key Supported VAX FORTRAN Extensions

- `NAMELIST`—the `NAMELIST` terminator may be either “`$`” or “`&`”
- `STRUCTURE`, `RECORD`, `UNION`, `MAP`, `%FILL` statements

- `DO WHILE` loops
- `INCLUDE` statement
- `ENCODE`, `DECODE`, `ACCEPT`, `TYPE`, and most `OPEN` I/O specifiers
- Hollerith and hexadecimal constant formats
- “`!`” comments
- Variable Format Descriptors (I<w>.<d> where w and d are variables)

Key Unsupported VAX FORTRAN Extensions

- Absoft Pro Fortran uses IEEE floating point representation
- I/O statements `DELETE`, `DEFINE FILE`, and `REWRITE`
- Data dictionaries

Compile Time Options and Issues

Absoft Fortran 77 can be made even more compatible with VAX FORTRAN by using the compiler option:

-s Force all program storage to be treated as static and initialized to zero.

VAX-compatible time, date, and random number routines are available by linking with the library file `vms.lib` in the `lib` directory. The routine names are:

`DATE` subroutine	returns current date as `CHARACTER*9`
`IDATE` subroutine	returns current date as 3 `INTEGER*4`
`TIME` subroutine	returns current time as `CHARACTER*8`
`SECNDS` subroutine	returns seconds since midnight
`RAN` function	returns random number

The following list of VAX FORTRAN “qualifiers” shows the equivalent Absoft Fortran 77 options or procedures:

`/ANALYSIS_DATA`	no equivalent
`/CHECK BOUNDS`	**-C** to check array boundaries
`/CHECK NONE`	do not use the **-C** option
`/CHECK OVERFLOW`	no equivalent
`/CHECK UNDERFLOW`	no equivalent
`/CONTINUATIONS`	no equivalent
`/CROSS_REFERENCE`	no equivalent
`/DEBUG`	**-g** to generate debugging information
`/D_LINES`	**-x** to compile lines with a “D” or “X” in column 1
`/DIAGNOSTICS`	append `2>filename` to the f77 command line to create a file containing compiler warning and error messages
`/DML`	no equivalent
`/EXTEND_SOURCE`	**-W** to permit source lines up to column 132 instead of 72
`/F77`	do not use the **-d** option
`/NOF77`	**-d** for FORTRAN 66 compatible `DO` loops
`/G_FLOATING`	see the section on “Numeric Precision” later in this chapter
`/I4`	do not use the **-i** option
`/NOI4`	**-i** for interpreting `INTEGER` and `LOGICAL` as `INTEGER*2` and `LOGICAL*2`
`/LIBRARY`	no equivalent
`/LIST`	no equivalent
`/MACHINE_CODE`	no equivalent
`/OBJECT`	no equivalent—you can use the `COPY` command to copy an object file to another name
`/OPTIMIZE`	**-O** to use basic optimizations
`/PARALLEL`	no equivalent
`/SHOW`	no equivalent
`/STANDARD`	**-N32** to generate warnings for non-ANSI FORTRAN 77 usage
`/WARNINGS DECLARATIONS`	the `IMPLICIT NONE` statement may be used to generate warnings for untyped data items
`/WARNINGS NONE`	**-w** to suppress compiler warnings

PORTING CODE FROM IBM VS FORTRAN

Absoft Fortran 77 automatically supports most of the IBM VS FORTRAN language extensions. Below is a list of key VS FORTRAN extensions that are supported and not supported. Using a compiler option, Absoft Fortran 77 can also accept VS FORTRAN Free-Form source lines that use 80 columns, otherwise, only ANSI FORTRAN 77 fixed format lines are accepted.

Key Supported VS FORTRAN Extensions

- “*” comments in column 1
- Can mix `CHARACTER` and non-`CHARACTER` data types in `COMMON` blocks
- The `NAMELIST` terminator may be an ampersand “&”
- Hollerith constants

Key Unsupported VS FORTRAN Extensions

- Absoft Fortran 77 uses IEEE floating point representation (more accurate)
- Debug statements
- I/O statements `DELETE`, `REWRITE`, and `WAIT`
- `INCLUDE` statement syntax is different

Compile-time Options and Issues

Absoft Fortran 77 can be made even more compatible with VS FORTRAN by using this compiler option:

-s	Force all program storage to be treated as static and initialized to zero

PORTING CODE FROM MICROSOFT FORTRAN

Absoft Fortran 77 automatically supports many of the Microsoft FORTRAN language extensions. Below is a list of key Microsoft FORTRAN extensions that are supported and not supported. Absoft Fortran 77 does not have the code size restrictions found in the segmented Microsoft FORTRAN models.

Key Supported Microsoft FORTRAN Extensions

- The NAMELIST terminator may be an ampersand “&”
- The Free-Form Source Code is very similar to VS FORTRAN (**-V** option)
- `STRUCTURE`, `RECORD`, `UNION`, `MAP` statements
- `SELECT CASE` statements
- `DO WHILE` loops
- `INCLUDE` statement
- Conditional compilation statements

Key Unsupported Microsoft FORTRAN Extensions

- Metacommands
- MS-DOS specific intrinsic functions
- `INTERFACE TO` statement
- `OPEN` statement displays standard file dialog when using `FILE=""`

Compile-time Options and Issues

Absoft Fortran 77 can be made even more compatible with Microsoft FORTRAN by using this compiler option:

-s Force all program storage to be treated as static and initialized to zero

The following list of Microsoft FORTRAN metacommands shows the equivalent Absoft Fortran 77 options or procedures:

`$DEBUG`	**-C** to check array boundaries and other run-time checks
`$DECLARE`	the `IMPLICIT NONE` statement may be used to generate errors or warnings for untyped data items
`$DO66`	**-d** for FORTRAN 66 compatible `DO` loops
`$FLOATCALLS`	all floating point is calculated inline or with a threaded math library in Absoft Fortran 77
`$FREEFORM`	**-V** for IBM VS FORTRAN Free-Form source code
`$INCLUDE`	use the `INCLUDE` statement
`$LARGE`	not necessary — Absoft Fortran 77 does not have the data size restrictions found in the segmented Microsoft FORTRAN models
`$LINESIZE`	not applicable
`$LIST`	no equivalent
`$LOOPOPT`	**-O** for optimization
`$MESSAGE`	no equivalent
`$PAGE`	not applicable
`$PAGESIZE`	not applicable
`$STORAGE:2`	**-i** for interpreting `INTEGER` and `LOGICAL` as `INTEGER*2` and `LOGICAL*2`
`$STORAGE:4`	do not use the **-i** option
`$STRICT`	**-N32** to generate warnings for non-ANSI FORTRAN 77 usage
`$SUBTITLE`	not applicable
`$TITLE`	not applicable
`$TRUNCATE`	no equivalent

PORTING CODE FROM SUN WORKSTATIONS

Absoft Fortran 77 automatically supports most of the Sun FORTRAN language extensions. Below is a list of key Sun FORTRAN extensions that are supported and not supported. The Sun FORTRAN compiler appends an underscore to all external names to prevent collisions with the C library. Absoft Fortran 77, by default, does not append an underscore to maintain compatibility with Windows functions and other development languages.

Key Supported Sun FORTRAN Extensions

- the `NAMELIST` terminator may be either “`$`” or “`&`”
- `STRUCTURE`, `RECORD`, `POINTER`, `UNION`, `MAP`, `%FILL` statements
- `DO WHILE` loops
- `INCLUDE` statement
- `ENCODE`, `DECODE`, `ACCEPT`, `TYPE`, and most `OPEN` I/O specifiers
- Hollerith and hexadecimal constant formats
- “`!`” comments in column 1

PORTING CODE FROM THE NEXT WORKSTATION

Absoft FORTRAN 77, formerly available, but now discontinued on the NextStep operating system for either Motorola or Intel microprocessors had the same optimizations and language extensions as Absoft Fortran 77. The object-oriented extensions of the NeXT compiler are specific to the NextStep environment and are not supported with Absoft Fortran 77 for Windows with Intel or PowerPC processors. The compilers are 100% source-compatible.

PORTING CODE FROM THE IBM RS/6000 WORKSTATION

Absoft FORTRAN 77, formerly available, but now discontinued for the IBM RS/6000 computer and had the same optimizations and language extensions as Absoft Fortran 77 for Windows with Intel or PowerPC processors. The compilers are 100% source compatible.

Distribution Issues

If you plan to distribute executable programs generated with Absoft Fortran 77, you must obtain a copy of the Absoft “Redistribution License Agreement”, complete it, and return it to Absoft. There is no charge for this license or the redistribution of programs created with Absoft Fortran 77. To obtain the Absoft “Redistribution License Agreement”, write to:

Absoft Corporation
2075 West Big Beaver Road, Suite 250
Troy, MI 48084

OTHER PORTING ISSUES

Not all porting and compatibility issues can be solved automatically by Absoft Fortran 77 or by using various option combinations. There are six issues that must be addressed on a program-by-program basis for Windows based computers:

Memory Management	Tab Character Size
Stack Issues	Numeric Precision
File and Path Names	Floating Point Math Control

Memory Management

A Win32 API application's address space differs slightly between Windows NT/XP and Windows 95/98. A Windows NT/XP address spaces ranges from 0x00010000 to 0x7ffeffff, while Windows 95/98 makes the space between 0x0040000 and 0x7fffffff available to the application. Since Win32 is a virtual, memory-managed environment, there is no real need for the programmer to be concerned with address spaces or virtual to physical memory mapping. A Win32 application will generally run without any memory management intervention by the programmer. There are however, two tunable memory parameters available that may be used to improve performance with memory usage intensive programs.

The Windows memory manager will automatically allocate memory beyond the initial 0x100000 byte (1 megabyte) heap and stack allocations to your program as it requires it. If you know that your program will use significantly more memory than this, it can be more efficient to reserve it initially rather than allocate it incrementally. Initial stack and heap allocations are established with the **-stack** and **-heap** compiler options (refer to the chapter **Using the Compilers** for details on the stack size and heap size options). The optional **commit** argument is used to specify the size of actual memory pages.

File and Path Names

Almost every operating system has a unique set of rules for valid file and path names and Windows is no exception. File names may be up to 255 characters in length, but may not contain any of the following characters: `? “ \ / < > * | :`. Case is preserved in file names, but file names themselves are not case sensitive. To reference a file in the current directory of a running application, the file name can be used without having to specify a path as in this example with a file called `file.dat` (8 characters including the period).

```
OPEN(UNIT=5,FILE="file.dat")
```

Path names are the concatenation of drive names, directory names, and file names and are used to specify files in other directories. Each component of a path name is separated by a back slash (\). A full path name always begins with the name of a drive and includes each of the directory names in the path to the file. A full path name is a complete and unambiguous identification for a file. Another type of path name is a partial path name that describes the path to a file starting from the current directory. Parent directories can be specified by beginning the path name with two periods and a backslash.

Programmers should be aware that Windows XP "personalizes" directories so the complete path name for a file may appear different from different sign in accounts. The same file will be seen in "Jane's Documents" by Fred, but in "My Documents" by Jane.

Tab Character Size

Absoft Fortran 77 expands each tab character in a FORTRAN source file into the equivalent number of spaces during compilation. The size of a tab character is determined from the following list in order.

- From the environment variable `TABSIZE`, which can be established with the `SET` command as follows:

`SET TABSIZE=6`	set tabs to six characters
`SET TABSIZE`	"unset" the `TABSIZE` variable

- If the environment variable `TABSIZE` is not set, the value `8` is used.

Tabs are also expanded at runtime when reading formatted files. They are expanded modulo TABSIZE where TABSIZE is an environment variable. If TABSIZE is not set, tabs are expanded modulo 8. If TABSIZE is set to 0, the tab is passed unmolested to the application.

Runtime Environment

A number of the aspects of the runtime environment can be controlled with the `ABSOFT_RT_FLAGS` environment variable. This variable can be a combination of any of the following switches (the leading minus sign is required for each switch and multiple switches must be separated by one or more spaces):

-defaultcarriage

> Causes the units preconnected to standard output to interpret carriage control characters as if they had been connected with `ACTION='PRINT'`.

-fileprompt

> Causes the library to prompt the user for a filename when it implicitly opens a file as the result of I/O to an unconnected unit number. By default, the library creates a filename based on the unit number.

-vaxnames

Causes the library to use 'vax style' names (FORnnn.DAT) when creating a filename as the result of I/O to an unconnected unit number.

-unixnames

Causes the library to use 'unix style' names (fort.nnn) when creating a filename as the result of I/O to an unconnected unit number.

-bigendian

Causes the library to interpret all unformatted files using big endian byte ordering.

-littleendian

Causes the library to interpret all unformatted files using little endian byte ordering.

-noleadzero

Causes the library to surpress the printing of leading zeroes when processing an Fw.d edit descriptor. This only affects the limited number of cases where the ANSI standard makes printing of a leading zero implementation defined.

-reclen32

Causes the library to interpret the value specified for `RECL=` in an `OPEN` statement as 32-bit words instead of bytes.

-f90nlexts

Allows f90 `namelist` reads to accept non-standard syntax for array elements. Without this flag, the following input results in a runtime error:

```
$ONE
A(1)=1,2,3,4
$END
```

When -f90nlexts is set, the values are assigned to the first four elements of A.

-nounit9

Causes UNIT 9 not to be preconnected to standard input and output.

-maceol

Formatted sequential files are in Classic Macintosh format where each record ends with a carriage return,

-doseol

Formatted sequential files are in Windows format where each record ends with a carriage return followed by a line feed.

-unixeol

Formatted sequential files are in Unix format where each record ends with a line feed.

-hex_uppercase

Data written with the Z edit descriptor will use upper case characters for A-F.

Floating Point Math Control

This section describes the basic information needed to control the floating point unit (FPU) built into Intel processors. The FPU provides a hardware implementation of the IEEE Standard For Binary Floating Point Arithmetic (ANSI/IEEE Std 754-1985). As a result it allows a large degree of program control over operating modes. There are two aspects of FPU operation that can affect the performance of a FORTRAN program:

Rounding direction

Exception handling

A single subroutine is provided with the compiler that is used to retrieve the current state of the floating point unit or establish new control conditions:

```
CALL fpcontrol(cmd, arg)
```

where: *`cmd`* is an `INTEGER` expression that is set to 0 to retrieve the state of the floating point unit and 1 to set it to a new state.

`arg` is an `INTEGER` variable that receives the current state of the floating point unit if *`cmd`* is 0 or contains the new state if *`cmd`* is 1.

Rounding Direction

The first aspect of FPU operation that may affect a FORTRAN program is rounding direction. This refers to the way floating point values are rounded after completion of a floating point operation such as addition or multiplication. The four possibilities as defined in the fenv.inc include file are:

`FE_TONEAREST`	round to nearest
`FE_TOWARDZERO`	round toward zero
`FE_UPWARD`	round toward +infinity
`FE_DOWNWARD`	round toward -infinity

Exception Handling

The second aspect of FPU operation that affects FORTRAN programs is the action taken when the FPU detects an error condition. These error conditions are called exceptions, and when one occurs the default action of the FPU is to supply an error value (either Infinity or NaN) and continue program execution. Alternatively, the FPU can be instructed to generate a floating point exception and a run time error when an exception takes place. This is known as *enabling the exception*. The five exceptions that can occur in a FORTRAN program are:

`FE_INEXACT`	inexact operation
`FE_DIVBYZERO`	divide-by-zero
`FE_UNDERFLOW`	underflow
`FE_OVERFLOW`	overflow
`FE_INVALID`	invalid argument

For example, to retrieve the state of the FPU, and then enable divide-by-zero exceptions, the following sequence would be used:

```
INCLUDE "fenv.inc"
INTEGER state

CALL fpcontrol(0, state)
CALL fpcontrol(1, FE_DIVBYZERO)
```

CHAPTER 7

Building Programs

This chapter covers the specifics of building Fortran 90/95 and FORTRAN 77 programs, including a discussion of the linker, library manager, and make facility. This chapter details the Absoft tools available for advanced programming and linking using the command line. The `Fsplit` utility is also described. You use each tool on the command line – the syntax and a description of each command is given below.

This chapter also describes the features, capabilities, and extensions in the Absoft implementations of Fortran 90/95 discussed in language reference manuals. The Absoft implementation of Fortran 90/95 and FORTRAN 77 is described the *Absoft Fortran Language Reference Manual*

AN OVERVIEW OF PROGRAM BUILDING

There are several different ways of building an application with the Absoft software development tools. The general overview of building a completed application is as follows:

Create Fortran 90/95 or FORTRAN 77 and compile them into object files with the proper interface and include files. See the section on **Creating Object Files** later in this chapter.

Create non-code resources with `rc.exe`, the resource compiler. See the section on **Working with Resources** in this chapter.

Create the executable program by using the `link` tool to link object files with the necessary resources and library routines from the Windows system. For more information on the `link` tool, see the section on **Linking Programs**, also in this chapter.

The Components of an Application

Program code, system calls, library routines, and features of the Windows operating system and interface are all important components of an application. Output from tools such as `amake` and `link` are combined with your object code to create a Windows application.

Working with Resources

A resource is one of the most important concepts in Windows programming. A resource is a collection of information used by the Windows system, such as menus, dialog definitions, or icons. These and other types of special information are stored in the executable image of a program file. The application itself may use some of the resources and other applications may use the resources for getting information about the application.

Resources are added to your program by the linker and are created using special tools and programs. Various dialog editors provide an interactive method of modifying existing resources or copying resources between files. The Microsoft program, `rc.exe`, included with the Absoft software development tools, is a resource compiler that creates new resources based on a textual resource description file. `rc.exe` is documented in a help file supplied with the compiler.

CREATING OBJECT FILES

After you create and edit source files, or port files from other environments (see the chapter, **Porting Code**), these files are compiled using one or more of the Absoft compilers (described in the chapter, **Using the Compilers**).

The compiler is invoked by using one of the commands: `f95.exe`, `f77.exe`, or `f90.exe` – these command control both components of the compiler (front and back ends) and the `link` tool (see the section below on **Linking Programs**). The features of the `f95`, `f77`, and `f90` commands simplify the process of creating finished applications, especially if you are working with a limited number of source files.

To initiate one of the Absoft compilers from the command line, follow these command syntax guidelines:

```
f95 [option…] [file…]
f77 [option…] [file…]
cl  [option…] [file…]
```

where *`option…`* represents one or more of the compiler options described in the chapter, **Using the Compilers**. These options must begin with a dash (-); if more than one option is used, separate each option with a space. Also, some arguments appended to an individual option, such as a filename, may need to be separated from the option letter with a space — see the chapter, **Using the Compilers** for specific option rules.

When these commands are invoked on the command line, each *`file`* will be compiled to generate an executable application. By default, the resulting application will be given a name the same as the base file name of the first file on the command line with an extension of `.exe`. For example, if you enter `f77 hello.f` on the command line, the source code from the `hello.f` file is compiled and an application will be generated in the file `hello.exe`. To compile `hello.f` with the static local storage option, and generate an application named `welcome.exe`, enter:

```
f77 -s -o welcome.exe hello.f
```

The option, `-o name`, specifies the name of the executable file overriding the default name of `hello.exe`. The name of the file must appear after the `-o` option as shown above. This option is passed directly to the linker; therefore, it has no effect when used in conjunction with the `-c` option. In this case, a space is required between the `-o` and `name`.

Remember that the `f77` and `f95` commands are used to control the compilation process. The actual compilers consist of the front-end (parsers and syntax analyzers) and the back-end (code generator).

If you need to create object files that are to be combined in a library, use the compiler commands with the `-c` option. This will suppress any linking functions and an executable file will not be created, as in the following example:

```
f95 -c Hello.f95 Goodbye.f95
```

The files are compiled into the object files `Hello.obj` and `Goodbye.obj`. After a source file has been compiled into an object file, it contains object code as well as any symbolic external references not known at compile time.

Since the linker is directly accessed in the `f77` and `f95`, any set of options may be passed directly to the linker. To do this, append the following option to the compiler command:

```
-link opts
```

The argument *`opts`* is a string enclosed in quotes to be passed to the linker. For example, `-link -verbose` will pass the `-verbose` option (display additional information) to the linker.

Fsplit - Source Code Splitting Utility

When you need to manage large files, work on small portions of Fortran code, or port code from other environments, you may want to split large, cumbersome source files into one procedure per file. This can be done using the Fsplit tool. The command syntax for the tool is shown below.

```
fsplit [option…] [file…]
```

Fsplit splits FORTRAN source files into separate files with one procedure per file. The following command line will generate individual files for each procedure:

```
fsplit largefile.f
```

A procedure includes block data, function, main, program, and subroutine program declarations. The procedure, proc, is put into file *`proc`*`.f` with the following exceptions:

- An unnamed main program is placed in `MAIN.f`.
- An unnamed block data subprogram is placed in a file named `blockdataNNN.f`, where `NNN` is a unique integer value for that file. An existing block data file with the same name will not be overwritten.
- Newly created procedures (non-block data) will replace files of the same name.
- File names are truncated to 14 characters.

Output files are placed into the directory in which the `fsplit` command was executed. The tab size is pulled from the environment variable `TABSIZE` if it exists, otherwise, a tab size of 8 is used. Options for the command are:

-v	Verbose progress of `fsplit` is displayed on standard diagnostic.
-V	Source files are in VAX FORTRAN Tab-Format.
-I	Source files are in IBM VS FORTRAN Free-Form.
-8	Source files are in Fortran 90 Free Source Form.
-W	Source files are in wide format.

LINKING PROGRAMS

Linking programs is the process of combining a group of object files into an application. The result is a new executable file. The Absoft tool that provides this feature is the `link` tool, or otherwise called the Linker. This tool is also used to create a Dynamic-Link Library, or a DLL (see the section **Creating Libraries** below for details).

The `link` tool links these object files into an application or tool called the *output* file. The Linker creates (or replaces) program code and places the object files as a linked application in the output file. The default output name is the root name of the first object file with the extension `.exe` appended.

If you use the `f77` command with the `/c` option to create an ordinary object file only, it contains object code and symbolic references to global variables and identifiers unknown at compile time. If you execute the `f77` command without the `/c` option, the `link` command is automatically invoked, creating the executable file.
The format for the `link` command is:

```
link [option…] [file…]
```

The options for the `link` command are:

@

Specifies a response file

-ALIGN

Specifies the alignment of each section

-ALLOWBIND

Specifies that a DLL cannot be bound

-ALLOWISOLATION

Specifies behavior for manifest lookup.

-ASSEMBLYDEBUG

Adds the DebuggableAttribute to a managed image.

-ASSEMBLYLINKRESOURCE

Create a link to a managed resource.

-ASSEMBLYMODULE

Specifies that a Microsoft Intermediate Language (MSIL) module should be imported into the assembly

-ASSEMBLYRESOURCE

Embeds a managed resource file to an assembly

-BASE

Sets a base address for the program

-CLRIMAGETYPE

Sets the type (IJW, pure, or safe) of a CLR image.

-CLRSUPPORTLASTERROR

Preserves the last error code of functions called through the P/Invoke mechanism.

-CLRTHREADATTRIBUTE

Specify which threading attribute you want applied to the entry point of your CLR program.

-CLRUNMANAGEDCODECHECK

/CLRUNMANAGEDCODECHECK specifies whether the linker will apply the SuppressUnmanagedCodeSecurity attribute to linker-generated PInvoke stubs that call from managed code into native DLLs.

-DEBUG

Creates debugging information

-DEF

Passes a module-definition (.def) file to the linker

-DEFAULTLIB

Searches the specified library when resolving external references

-DELAY

Controls the delayed loading of DLLs

-DELAYLOAD

Causes the delayed loading of the specified DLL

-DELAYSIGN

Partially sign an assembly.

-DLL

Builds a DLL

-DRIVER

Creates a Windows NT kernel mode driver

-ENTRY

Sets the starting address

-errorReport

Report internal linker errors to Microsoft.

-EXPORT

Exports a function

-FIXED

Creates a program that can be loaded only at its preferred base address

-FORCE

Forces a link to complete in spite of unresolved or symbols defined more than once

-FUNCTIONPADMIN

Creates a hotpatchable image.

-HEAP

Sets the size of the heap in bytes

-IDLOUT

Specifies the name of the .idl file and other MIDL output files

-IGNOREIDL

Prevents processing attribute information into an .idl file

-IMPLIB

Overrides the default import library name

-INCLUDE

Forces symbol references

-INCREMENTAL

Controls incremental linking

-KEYCONTAINER

Specify a key container to sign an assembly.

-KEYFILE

Specify key or key pair to sign an assembly.

-LARGEADDRESSAWARE

Tells the compiler that the application supports addresses larger than two gigabytes

-LIBPATH

Allows the user to override the environmental library path

-LTCG

Specifies link-time code generation

-MACHINE

Specifies the target platform

-MANIFEST

Create a side-by-side manifest file.

-MANIFESTDEPENDENCY

Specify a <dependentAssembly> section in your manifest file.

-MANIFESTFILE (Name Manifest File)

Change the default name of the manifest file.

-MAP

Creates a mapfile

-MAPINFO

Includes the specified information in the mapfile

-MERGE

Combines sections

-MIDL

Specifies MIDL command line options

-NOASSEMBLY

Suppresses the creation of a .NET Framework assembly

-NODEFAULTLIB

Ignores all (or specified) default libraries when resolving external references

-NOENTRY

Creates a resource-only DLL

-NOLOGO

Suppresses startup banner

-NXCOMPAT

Marks an executable as having been tested to be compatible with Windows Data Execution Prevention feature.

-OPT

Controls LINK optimizations

-ORDER

Places COMDATs into the image in a predetermined order

-OUT

Specifies the output file name

-PDB

Creates a program database (PDB) file

-PDBSTRIPPED

Creates a program database (PDB) file with no private symbols

-PGD

Specify .pgd file for profile guided optionizations.

-PROFILE

Produces an output file that can be used with the Performance Tools profiler.

-RELEASE

Sets the Checksum in the .exe header

-SAFESEH

Specify that the image will contain a table of safe exception handlers.

-SECTION

Overrides the attributes of a section

-STACK

Sets the size of the stack in bytes

-STUB

Attaches an MS-DOS stub program to a Win32 program

-SUBSYSTEM

Tells the operating system how to run the .exe file

-SWAPRUN

Tells the operating system to copy the linker output to a swap file before running it

-TLBID

Allows you to specify the resource ID of the linker-generated type library

-TLBOUT

Specifies the name of the .tlb file and other MIDL output files

-TSAWARE

Creates an application that is specifically designed to run under Terminal Server

-VERBOSE

Prints linker progress messages

-VERSION

Assigns a version number

-WX

Treat linker warnings as errors.

CREATING LIBRARIES

Windows based computers support two types of libraries: *static* and *dynamic*. A static library is a collection of object files (modules), each containing one or more routines, which are maintained in a single file — a library. When a library file is presented to the linker, modules that are required to satisfy unresolved external references are selected for inclusion into the application file. The advantage of a library is that only those modules that are required to satisfy unresolved external references are linked into the application. The FORTRAN runtime library `af77math.lib` is an example of a static library. Not every FORTRAN program requires a hyperbolic tangent function, so it is only linked into those programs that require it.

A dynamic library is similar to a static library in that it contains a collection of routines in object modules. The difference is that the elements of the library are not linked into the final application file, but rather are available for linking when the application is executed. The advantage to this type of library is that the individual applications can be smaller and several applications can share the same library. The disadvantage is that the dynamic or shared library must be available on every computer where the application is to be run.

The `link` command, introduced earlier, is used to create dynamic libraries or DLL's by combining multiple object files into a single DLL and the loader glue code. The linker can also be used to create static library files, but the library manager, `lib`, supplied with the Absoft Fortran compiler, is the primary tool for manipulating static libraries. Use `lib` to create libraries from multiple object files and to add, delete, or replace object modules in existing libraries.

The syntax of the `lib` command is given below. Options can be preceded with either a dash (`-`) or a slash (`/`).

```
lib [option…] [files…]
```

A new library is created by using the `-out` option to specify the new library name and the `files` argument to indicate the object files. To add an object file to an existing library, specify both the existing library name and the object file name in the `files` argument and use the `-out` option to specify the resulting library name. The input and output library name may be the same.

`/extract:`*`objname`*

This option is used to copy the *`objname`* from the library to a file. The output filename is the same as *`objname`* unless the `/out` option is given. `/remove` and `/extract` cannot be used at the same time.

`/list`

Use this option to display to standard output a list of the object modules in the library.

`/out:`*`filename`*

This option sets the name of the output file.

/remove:objname

Use this option to delete the specified objname from the library. /remove and /extract cannot be used at the same time.

DLL Import Libraries

The Win32 API supports two types of libraries: static and dynamic, which were described earlier in this chapter. One requirement to link against a dynamic link library or DLL is an *import library*. An import library is a type of library that describes to the linker all of the available function entry points in the DLL. `link` automatically creates an import library when it creates a DLL. If you are given a DLL and no import library, you can still use the DLL by creating the import library yourself, provided you have a definition file (described below). The utility `imptool` creates import libraries from definition files.

The syntax of the `imptool` command is given below. Options can be preceded with either a dash (`-`) or a slash (`/`).

```
imptool [option…] [files…]
```

`/def:`*`filename`*

This option is required. It specifies the definition file (described below) which is used as input to `imptool`. The file must have an extension of `.def`.

`/out:`*`filename`*

This option specifies the name of the output file. If it is not used, the name of the output will be the root of the definition file name with an extension of `.lib` added.

`/nounderscores`

This option specifies that the symbol names not have an underscore prepended to the entry names.

`/w`

This option suppresses warnings. imptool will not output any warning messages.

Syntax of a Definition File

The input file for `imptool` is a definition file. This file describes the entry points into a dynamic-link library, the DLL. A definition file has a list of the entry point names, and some keywords to describe them. The syntax of a definition follows.

There are many statements available that can be used in a definition file. Most of them are not supported by `imptool`, but are used by other applications. `imptool` will produce a warning, and ignore them. The three statements that `imptool` supports are.

`NAME [`*`application`*`][BASE=address]`

> This statement specifies the application to which the definition file is associated with. If `NAME` is not specified, the default will be the output file name with a `.dll` extension. The `BASE` keyword has no effect as far as an import library is concerned; it is used by other applications, but is still legal syntax. This statement can only appear on the first line of the definition file; otherwise it will produce a warning. If you use a `NAME` statement, you cannot use a `LIBRARY` statement (described next).

`LIBRARY [`*`library`*`][BASE=address]`

> This statement specifies the DLL that the definition is associated with. If this is not specified, the default will be the output file name with a `.dll` extension. The `BASE` keyword has no effect as far as an import library is concerned; it is used by other applications, but is still legal syntax. This statement can only appear on the first line of the definition file; otherwise it will produce a warning. If you use a `LIBRARY` statement, you cannot use a `NAME` statement.

`EXPORTS definitions`

> This statement makes one or more definitions available as exports to other programs. The syntax of an export definition is:
>
> *`entryname`*`[=`*`internalname`*`] [@ordinal[NONAME]] [DATA] [PRIVATE]`

`entryname`	the name of the entry point into the DLL
`internalname`	is used by other applications and is ignored by `imptool`
`ordinal`	the ordinal associated with the entry point
`NONAME`	a keyword that means import by ordinal only
`DATA`	a keyword that means you are importing data
`PRIVATE`	a keyword that means to leave the entry name out of the import library

> Other statements ignored by `imptool` are:
>
> `DESCRIPTION`

```
STACKSIZE
IMPORTS
SECTIONS
VERSION
NAME (after the first line)
LIBRARY (after the first line)
```

Comments in a definition file are signified by a semi-colon (`;`) at the first position of a line.

Name Mangling

Entry point names are the most important parameter when creating a definition file. `imptool` will automatically prepend an underscore (_) to all entry point names unless the `/nounderscores` is specified on the command line. If `/nounderscores` is specified, `imptool` will place the entry point in the import library exactly as it appears in the definition file. The best way to insure that your import library will correctly describe the entry points of a DLL is to specify the entry point name exactly as it appears in the DLL and use the `/nounderscores` option. Name mangling, prepending an underscore to a symbol name is standard in the Win32 API, so it is best to add an underscore yourself and use the`/nounderscores` option.

When creating a definition file, keep in mind that using `STDCALL` will require you to append some characters to the entry point names that must be present in the definition file, since that is how they appear in the object code and the DLL. The extra characters are a commercial at sign (`@`) followed by the size, in bytes, of the parameter block for the function or subroutine. An example follows:

```
FUNCTION my_func(INTEGER parameter1, INTEGER parameter2)
STDCALL my_func
```

This should appear as:

```
_my_func@8
```

in your definition file.

Because `imptool` will not have any of this information available regarding the type or calling convention used with the function, it will not add any of these characters, so it is important that you append them yourself.

This form of name mangling, appending `@` and a number is different from using an ordinal value. If you are specifying an ordinal value, the value is separated by whitespace on the same line as the entry point name. If the value following the `@` is the size of the parameter block required when using `STDCALL`, it is appended to the entry point name, with no white space between. An example to illustrate this follows:

If using `STDCALL` as above:

```
_my_func@8
```

should be the export line name in your definition file.

If specifying an ordinal value:

```
_my_func @32
```

should be the export line in your definition file.

When you are creating the definition file it is important to keep in mind how the entry point names are being represented. Again, the most direct way is to add the characters that are needed and use the `/nounderscores` command line option. This will insure that names are handled exactly as they appear, and are not being mangled by `imptool` in any way.

BUILDING PROGRAMS

It is often necessary in software development to maintain large numbers of files, many of which are dependent on other files in some way. It can become very difficult and time-consuming to manage these complex file relationships manually and to ensure that the appropriate files are updated when modifications are made to other related files. For example, when a source file is altered, it is necessary to recompile it in order to build or rebuild an updated object file and to link the object file with the appropriate auxiliary files (such as libraries) to form a complete and up-to-date executable file. It may also be necessary to use multiple languages and other programming resources during this process.

The Absoft `amake` utility allows you to automate much of this process of file maintenance by keeping a record of file dependencies according to rules that are either *built-in* to `amake` or *specified by the user*. (The `amake` utility is also referred to as "`amake`", the "make program", or the "make command" throughout this section.) Following these rules, the program determines whether any files need to be updated, and if so, rebuilds them automatically. If a file needs to be updated and does not exist, `amake` will create it based on the dependency rules for that file.

While `amake` is used primarily in software development, it can also be employed in other types of routine project management activities that involve file dependency relationships such as deleting temporary files, updating documents, or performing backups. In this section, we will focus on the use of `amake` to maintain an up-to-date executable file during the course of a software project.

The major advantages of using `amake` in this type of environment are that it:

- saves considerable time and computing resources since only the files that need to be updated at a particular time are rebuilt;

- simplifies project management by performing many routine functions automatically and helping to coordinate the activities of projects involving multiple programmers; and

- frees programmers from the need to perform routine file maintenance activities manually.

This section discusses the operation of the Absoft `amake` program and explains how you can define your own rules to adapt the program to your particular environment. It also covers the topics of creating description files and macros, command usage and options, using environment variables, and handling errors. The level of presentation assumes a familiarity with programming and the process of developing software, but does not require any previous knowledge of the `amake` utility itself.

The Elements of amake

A key concept in understanding the operation of the `amake` program is that of *file dependency*. Files that are required to build (or rebuild) other files are referred to here as *prerequisite files* (or *prerequisites*). A file that is dependent on these prerequisites is called a *target file* (or *target*). For example, an object file (the target) is dependent on one or more source files (the prerequisites). The `amake` program searches through a file *dependency tree* to establish the relationships between targets and prerequisites. If a prerequisite file has been updated more recently than its target file (or at exactly the same time), `amake` will (re)build the target file. [Note: The term *(re)build* is used in this section to indicate that a file will be *built* (created) if it does not exist, or *rebuilt* (updated) if it does exist.]

As mentioned above, the Absoft `amake` program operates based on rules that are: built-in to the program, specified by the user, or a combination of both. The program uses information from the following sources to determine whether a particular file needs to be (re)built and, if so, how this will be done:

- A description file supplied by the user that specifies:
  (a) dependency relationships between targets and prerequisites, and
  (b) the commands needed to (re)build the target file.

- File names and the date/time each file was last modified.

- A set of default rules that define how files are (re)built based on the relationships between their suffixes.

Using Macros

Before discussing how a description file is created and used, it is necessary to have some understanding of how macros are used with `amake`. The term *macro*, as used here, refers to a symbol or character string that substitutes for something else, such as a set of commands. Macros are very useful in defining dependency relationships.

Advantages of using macros

The `amake` tool allows you to define macros, either within the description file itself, or as arguments on the `amake` command line. By using macros, you can:

- Represent recurring strings, such as file names or commands, in simplified form, reducing redundancy and thus, file size.

- Improve the consistency, readability, and maintainability of your description files.
- Allow for variation in the value of a macro from one (re)build to the next, and for values to be changed globally by redefining the corresponding macro.

Defining macros

A *macro definition* is made up of three basic elements: a name, followed by an equal sign, followed by a symbol or string that defines what the macro represents (in description files, usually a command string). You invoke a macro by placing a $ symbol immediately before the name and enclosing the name in either parentheses ( ) or braces { }. [Exception: A name of only one character can be invoked without being enclosed in parentheses or braces.] By convention, macro names are written in uppercase characters, but any combination of upper or lower case letters or other non-reserved characters is acceptable. The following are examples of valid macro definitions and their corresponding invocations:

Macro Definition	Macro Invocation
`DEBUGOPT = -g`	`$(DEBUGOPT)`
`SRCFILES = one.f two.f`	`$(SRCFILES)`
`OBJFILES = one.obj two.obj`	`$(OBJFILES)`
`ALLFILES = $(SRCFILES) $(OBJFILES)`	`$(ALLFILES)`
`RESFILES = $(RCFILES:.rc=.res)`	`$(RESFILES)`

The last example invokes the two previous macros within the definition, producing a list of the two FORTRAN source files and two object files as follows:

```
one.f two.f one.obj two.obj
```

The order of precedence for macro definitions is (from highest to lowest): the `amake` command line, the description file, and the default definitions.

Special macros

The `amake` utility includes a set of special-purpose macros that you may find useful in building your description files and rules. The most commonly-used are:

Macro	Function
$@	Represents the *full name of the current target*—for use only on a (re)build command line. (When building a library it represents the name of the library.)
$*	Represents the *base name of the current target*—for use only on a (re)build command line.
$<	Represents a *current prerequisite*—for use only on a (re)build command line.
$$@	Represents the *base name of the current target*—for use only on a dependency line.
$?	Represents a *list of prerequisites* that have been changed more recently than the current target—for use only on a (re)build command line.

Other special macros that are provided with Absoft `amake` include:

Macro	Function
`MAKE`	Used for recursive makes—that is, when a make command is included as part of a description file.
`MAKEFLAGS`	Sets the command-line options available to make—usually defined as an environment variable (see **Environment Variables** later in this section).
`SUFFIXES`	Contains the default list of suffixes for the `.SUFFIXES` special target (see **Special Targets** later in this section).

Cautions in using macros

In addition to being aware of the order of precedence for macro definitions (see above) you should use caution in defining and using macros for the following reasons:

- A description file macro should be defined *before* the first time it is used in a dependency block.
- A macro should be defined only *once* within a description file.
- Macros may not be recursive—a macro may not directly or indirectly reference itself\.
- If you reference an undefined macro, `amake` will assign it a null string and *no error message will be given*.

- While other characters are acceptable, it is advisable to use upper-case characters for macro names and to avoid characters that have special meanings in the operating system environment.

Using Description Files

The relationships between target files and their prerequisite files are specified in a *description file* which is called either `makefile` or `Makefile` by default (in that order). This file contains one or more *dependency blocks*, each consisting of the following elements:

- The target file name followed by a colon.
- The prerequisite file names (if any) following the colon.
- White space (a tab or spaces) followed by the commands needed to rebuild the target file.

[Note: Description files are also commonly referred to as *makefiles*. The term *description file* is used in this section for the sake of consistency.]

Working with dependency blocks

The general form of a dependency block is:

```
target: prerequisite1 prerequisite2...
     command(s) to (re)build target
```

For readability and ease of maintenance, we recommend that you:

- Place the target file name, colon, and prerequisite file name(s) on the first line and command(s) on the second line whenever possible; and
- Use a tab rather than spaces to precede commands.

For example, the first line of the following block:

```
test: a.f95 b.f95 lib.lib
     f95 -o test.exe a.f95 b.f95 lib.lib
```

specifies that the target `samp.exe` is dependent on the prerequisites `a.f95`, `b.f95`, and `lib.lib`. If any of the three prerequisites have been updated at the same time or after the target, test`.exe` will be rebuilt automatically using the command specified on the second line. The first line is referred to here as the *dependency line* and the second as the *(re)build command line*, or simply, the *command(s)*. [Note: The term *(re)build command line* in this context applies only to dependency blocks and should not be confused with the *make command line* discussed later in this section.]

If desired, the entire dependency block can be placed on one line by including a semicolon after the last prerequisite file name. The example above would look like:

```
test: a.f95 b.f95 lib.lib; f95 -o test.exe a.f95 b.f95 lib.lib
```

If a line exceeds the maximum length allowed on your system, or you wish to shorten it and continue it onto the next line, you can use the continuation character for your environment. Using the example above, the backslash character (\) must be the last character on the first line as follows:

```
test: a.f95 b.f95 lib.lib; f95 -o test.exe a.f95 \
b.f95 lib.lib
```

Defining a target more than once

There may also be times when you will need to define the same target more than once within the same description file. This can be done using the *double-colon* feature of Absoft `amake`. This allows you to define two different sets of prerequisites (and the associated (re)build commands) for the same target. This feature is particularly useful in updating archive libraries. For example:

```
graph.lib:: vertex.f95
     $(F95) /c /g -DDEBUG vertex.f95
     lb /out:$@ $@ vertex.obj
     del vertex.obj

graph.lib:: edge.f95
     $(F95) /c /O edge.f95
     lb /out:$@ $@ edge.obj
     del edge.obj
```

In this example, two different sets of commands are passed to the Fortran 90/95 compiler during the process of building the library `graph.lib`.

Using include directives

An `include` directive can be used to include a text file within a description file. Such a text file could consist of macro definitions, dependency blocks, or any other components you would include as part of a description file. An `include` directive consists of the word `include`, left-justified, followed by one or more spaces or tabs, followed by the name of the file that is to be included at that point in the description file. For example:

```
include mymacros.txt
```

Included files are processed before the next line in the current description file. They can also be nested.

A sample description file

The following is an example of a simple description file:

```
# program name
NAME = util

# set FLAGS for command line
F95FLAGS = /g
LDFLAGS =

SRCS = util.f95 build.f95 parse.f95 tstring.f95
OBJS = util.obj build.obj parse.obj tstring.obj
PROG = $(NAME)

$(PROG): $(OBJS)
     $(F95) $(F95FLAGS) $(OBJS) /o $(PROG) $(LDFLAGS)

util.obj: util.f95 util.inc tstring.inc decl.inc

build.obj: build.f95 util.inc tstring.inc decl.inc

parse.obj: parse.f95 util.inc tstring.inc decl.inc

tstring.obj: tstring.f95 tstring.inc
```

Explanation:

- Lines beginning with a pound sign (#) are interpreted as comments.
- Lines containing an equal sign (=) are macro definitions; macros should be defined before they are used in a dependency block. (See **Defining macros** and **Cautions in using macros** earlier in this section).
- The lines containing a colon are dependency lines.
- Lines indented under dependency lines are (re)build commands.
- A dependency line and a set of (re)build commands together constitute a dependency block.

Although the order of these components may not affect the operation of `amake`, we suggest that you follow the format shown above in creating and maintaining your description files, that is: *macro definitions*, followed by *user-defined suffix rules*, followed by *dependency blocks*—with each definition, rule, or block separated by a blank line.

Using Dependency Rules

The `amake` utility uses a set of internal rules, commonly referred to as *dependency rules* or *suffix rules* to determine how to (re)build a particular target file. These rules determine file relationships based on filename suffixes. Absoft `amake` looks for dependency rules in two locations:

1. a default file that is automatically read by `amake`, and

2. your description file.

Rules specified in a description file *always* override the corresponding default rules.

The default rules

The default dependency rules (or suffix rules) automatically handle the common file transformations that `amake` performs, such as compiling source files to produce object files. Without these default rules, you would have to specify all file relationships in a description file; this would tend to become very complex and redundant in a large software development project. The default rules are located in:

```
c:\absoft15.0\bin\default.amk.
```

The following is a list of the default dependency rules included with Absoft `amake` for Fortran 90/95, FORTRAN 77, and C files. The macros shown within these rules are pre-defined in the `default.amk` file. [Note: The numbers on the left are not part of the rules and are included for reference only.]

Default Rules for Fortran 90/95 files

```
(1)    .f95:
             $(F95) $(F95FLAGS) $(LDFLAGS) -o $@ $<

(2)    .f95.obj:
             $(F95) $(F95FLAGS) -c $*.f95

(3)    .f95.lib:
             $(F95) $(F95FLAGS) -c $*.f95
             $(LIB) $(LIBFLAGS) $@ $*.obj
             $(RM) $*.obj

(4)    .rc.res
             $(RC) $RCFLAGS) $<
```

Explanation:

(1) Compiles a Fortran 90/95 source file into an executable target.

(2) Creates an object file from a Fortran 90/95 source file.

(3) Compiles a Fortran 90/95 source file into an object file and then adds it to a library (.lib) file.

(4) Creates a resource file from a resource script.

Default Rules for FORTRAN 77 files

```
(1)    .f:
             $(F77) $(FFLAGS) $(LDFLAGS) -o $@ $<

(2)    .f.obj:
             $(F77) $(FFLAGS) -c $*.f

(3)    .f.lib:
             $(F77) $(FFLAGS) -c $*.f
             $(LIB) $(LIBFLAGS) $@ $*.obj
             $(RM) $*.obj

(4)    .rc.res
             $(RC) $RCFLAGS) $<
```

Explanation:

(1) Compiles a FORTRAN 77 source file into an executable target.

(2) Creates an object file from a FORTRAN 77 source file.

(3) Compiles a FORTRAN 77 source file into an object file and then adds it to a library (.lib) file.

(4) Creates a resource file from a resource script.

Default Rules for C files

```
(1)    .c:
             $(CC) $(CFLAGS) $(LDFLAGS) $< -o $@

(2)    .c.obj:
             $(CC) $(CFLAGS) -c $<

(3)    .c.i:
             $(CC) $(CFLAGS) -P $<

(4)    .c.lib:
             $(CC) -c $(CFLAGS) $<
             $(LIB) $(LIBFLAGS) $@ $*.obj
             $(RM) $*.obj
```

Explanation:

(1) Compiles a C source file into an executable target.

(2) Creates an object file from a C source file.

(3) Creates an intermediate (.i) file by running a C source file through the C preprocessor.

(4) Compiles a C source file into an object file and then adds it to a library (.lib) file.

Creating your own rules

In general, it is best to rely on the default dependency rules as much as possible. There will be times, however, when you may need to modify the behavior of `amake` by creating your own dependency rules. There are two possible ways to do this:

- Include dependency rules in your description file, or
- Modify the file of default rules by adding your own rule(s), or deleting/changing existing rule(s).

We recommend that you use the first alternative if possible, and avoid modifying the default rules unless absolutely necessary. Since rules in a description file *always* override any corresponding default rules, the first alternative should be sufficient for virtually any circumstance. [Caution: Unless you are replacing an existing default rule, it is advisable to avoid using suffixes that are pre-defined in `amake` to avoid conflicts with the default rules.]

The following is an example of a user-specified dependency rule included in the description file discussed earlier in this section:

```
# program name
NAME = util

# set FLAGS for command line
F95FLAGS = /g
LDFLAGS =

SRCS = util.f95 build.f95 parse.f95 tstring.f95
OBJS = util.obj build.obj parse.obj tstring.obj
PROG = $(NAME)

.f95.obj:
     $(F95) $(F95FLAGS) /c $<
     copy $< c:\usr\workdir

$(PROG): $(OBJS)
     $(F95) $(F95FLAGS) $(OBJS) /o $(PROG) $(LDFLAGS)

util.obj: util.f95 util.inc tstring.inc decl.inc

build.obj: build.f95 util.inc tstring.inc decl.inc

parse.obj: parse.f95 util.inc tstring.inc decl.inc

tstring.obj: tstring.f95 tstring.inc
```

The user-supplied rule:

```
.f95.obj:
     $(F95) $(F95FLAGS) -c $<
     copy $< c:\usr\workdir
```

will override the corresponding default rule in the `default.mk` file:

```
.f95.obj:
     $(F95) $(F95FLAGS) -c $<
```

Rather than following the default rule for creating an object file from a Fortran 90/95 source file, the new suffix rule will override the default to invoke the Fortran compiler and copy the resulting object file to the working directory. (The default rule *only* invokes the Fortran 90/95 compiler.)

amake Usage and Syntax

The `amake` command accepts options, description file names, macro definitions, and target file names as arguments in the form:

```
make [options] [description file] [macros] [target(s)]
```

Arguments specified on the `amake` command line override any corresponding definitions found in a description file or in the default dependency rules.

`amake` command-line options may specified with either a dash (`-`) or a slash (`/`):

`/d` Lists the prerequisites for each dependency block that caused `amake` to rebuild a target. All prerequisites that are newer than the target are displayed. Useful for determining why certain (re)build commands are executed.

`/D` Displays the contents of a description file as it is read by the `amake` program.

`/e` Causes environment variables to override macros defined in a description file. By default, user-defined macros override environment variables (see **Environment Variables** below).

`/f` Takes an argument in the form *filename* which specifies the name of a description file to be used in place of the default name `makefile`. A file name consisting of a dash (-) uses the standard input rather than *filename* as input. If there are no `/f` arguments, the program will search (by default) for a file named `makefile` or `Makefile` in the current directory.

`/i` Ignores error codes returned by commands. This is equivalent to using the `.IGNORE` special target in a description file (see **Special Targets** below). Useful in situations when it is not necessary that certain commands execute successfully.

`/k` This option stops processing on the current entry when an error occurs, but continues processing on other branches of the dependency tree that do not depend on the current entry.

`/n`	Displays all commands, but does not execute them. (Command lines beginning with an @ character are also displayed.) Useful in debugging/testing description files.
`/p`	Prints a complete list of macro definitions, dependency blocks, and suffix rules.
`/q`	Returns a zero or nonzero status code depending on whether the target is or is not up-to-date, respectively. Useful when `amake` is called from a script or tool that requires the current target.
`/r`	Does not use the default rules (i.e., does not read in the `default.mk` file). Useful for situations where you want to completely isolate the environment in which `amake` operates.
`/s`	Does not print command lines before executing. This is equivalent to using the `.SILENT` special target in a description file.
`/t`	*Touches* the target files (assigning them the current date/time) without executing the commands to (re)build them. Used to bypass the (re)build process for particular targets—should be used with caution.

Any command-line arguments other than options, description file names, or macros are assumed to be the names of targets to be (re)built; these are evaluated in left-to-right order. If there are no such arguments, the first target in the description file whose name does not begin with a period is rebuilt (see below).

Special Targets

In addition to the options listed above, the following *special targets* can be used in a dependency block (rule) to further customize the behavior of `amake`:

`.DEFAULT`	Used when there is no target name specified or default rule for building a target file. A set of pre-defined commands are invoked by the `.DEFAULT` target.
`.DONE`	This target and its prerequisites are processed after all other targets have been (re)built.
`.IGNORE`	Ignores all error codes; equivalent to the `/i` option on the make command line.
`.INIT`	This target and its prerequisites are processed before any other targets are (re)built.

`.SILENT`	Executes commands, but does not send them to the standard output; equivalent to the `/s` option on the make command line.
`.SUFFIXES`	Used to add dependency rules to the default rules (specify `.SUFFIXES` as the target followed by the suffixes to be added as the prerequisites), or to delete the default rules entirely (specify `.SUFFIXES` as the target without prerequisites).

Dummy Files

There may be times when you will want to run `amake` without actually (re)building a target or when you need to force a target to be (re)built regardless of when the last modification was made to a prerequisite. You can do this by using *dummy files*—i.e., specifying one or more filenames in your description file that do not represent an actual file, but that cause the behavior of `amake` to change. Often, this can be used to bypass the established dependency tree and force `amake` to behave in a desired manner.

The most common type of dummy filename is a *dummy target*. For example:

```
clobber :
     del *.obj
```

will execute the commands on the second line without (re)building any files.

Environment Variables

Each time you run `amake`, the environment variables that exist at that time are read and added to the existing macro definitions. Essentially, environment variables are handled in the same manner as macros by `amake`. As briefly described earlier in this section, the `MAKEFLAGS` variable (also sometimes referred to as the `MAKEFLAGS` macro) defines the command-line options available to `amake` and is usually defined as an environment variable; the `MAKEFLAGS` environment variable is read and processed prior to any options specified on the `amake` command line.

When you run `amake`, the following order of precedence is followed (from highest to lowest priority):

1. command-line arguments
2. description file entries (definitions)
3. environment variables
4. default dependency rules

If you invoke the `/e` command-line option, priority levels 2 and 3 are reversed so that the order of precedence becomes:

1. command-line arguments
2. environment variables
3. description file entries (definitions)
4. default dependency rules

Example: Rebuilding an Executable File

Generally, in a software development environment, you would run the `amake` utility whenever there is a need to update an executable file, such as after changes have been made to source files or libraries. To summarize the operation of `amake`, the program:

1. Searches for a description file called `makefile` (or, if that name does not exist, `Makefile`) by default, or another name assigned through the `/f` option.
2. Checks dependencies in a bottom-up manner, establishing relationships between targets and their prerequisites and building a dependency tree in the process.
3. (Re)builds target files when they are out-of-date with respect to their prerequisites according to commands specified in the description file, the default rules, or both.

Using our sample description file, `amake` will: read in the macro definitions, check the syntax of all entries, and (re)build the executable file `util` based on the `.f95.obj` suffix rule and the dependency blocks (lines) following it:

```
# program name
NAME = util

# set FLAGS for command line
F95FLAGS = /g
LDFLAGS =

SRCS = util.f95 build.f95 parse.f95 tstring.f95
OBJS = util.obj build.obj parse.obj tstring.obj
PROG = $(NAME)

.f95.obj:
     $(F95) $(F95FLAGS) /c $<
     copy $< c:\usr\workdir

$(PROG): $(OBJS)
     $(F95) $(FFLAGS) $(OBJS) /o $(PROG) $(LDFLAGS)

util.obj: util.f95 util.inc tstring.inc decl.inc

build.obj: build.f95 util.inc tstring.inc decl.inc

parse.obj: parse.f95 util.inc tstring.inc decl.inc

tstring.obj: tstring.f95 tstring.inc
```

Error Handling and Cautions

The following is a list of common errors you may encounter while using `amake` and possible reasons for their occurrence.

Syntax Errors

Error Message	Explanation
`Badly formed macro`	Incorrect syntax for a macro definition—often, the macro name is missing.
`Improper macro`	An error occurred during macro expansion. Often, the problem is a missing parenthesis or bracket.
`Macro too long ...`	A macro name is too long; cannot be longer than 100 characters.
`Rules must be after target`	Occurs when a line beginning with a space or tab has been encountered before a dependency line in a description file.

Other Common Errors

Error Message	Explanation
`Cannot open file`	The description file specified in an `include` directive could not be found or was not accessible. (See **Using include directives** earlier in this section.)
`Don't know how to make target`	There is no target entry in a description file, none of the default rules apply, and there is no `.DEFAULT` rule.
`Too many options`	The `amake` program has exceeded the allocated space while processing command-line options or a target list.
`Too many rules defined for target`	Multiple sets of rules have been defined for a target; targets may only have one set of rules.
`Unexpected end of line seen`	The colon in a dependency line is missing.

Cautions

In addition to handling the errors described above, particular caution should be exercised as follows when running `amake`:

- Use of the `/t` (touch) or `/i` (ignore errors) options can be destructive in the way that they override the normal behavior of `amake` (see **amake Usage and Syntax** earlier in this section). These options should be used with great care and, if possible, tested first before being used with actual files. The `/t` option, in particular, can save considerable time by "updating" files without (re)building them, but it erases the file relationships that would normally be established.

- Unforeseen problems can arise by changing default rules or variables, such as the `MAKEFLAGS` environment variable. It is best not to change these default values but, if this must be done, caution is advisable.

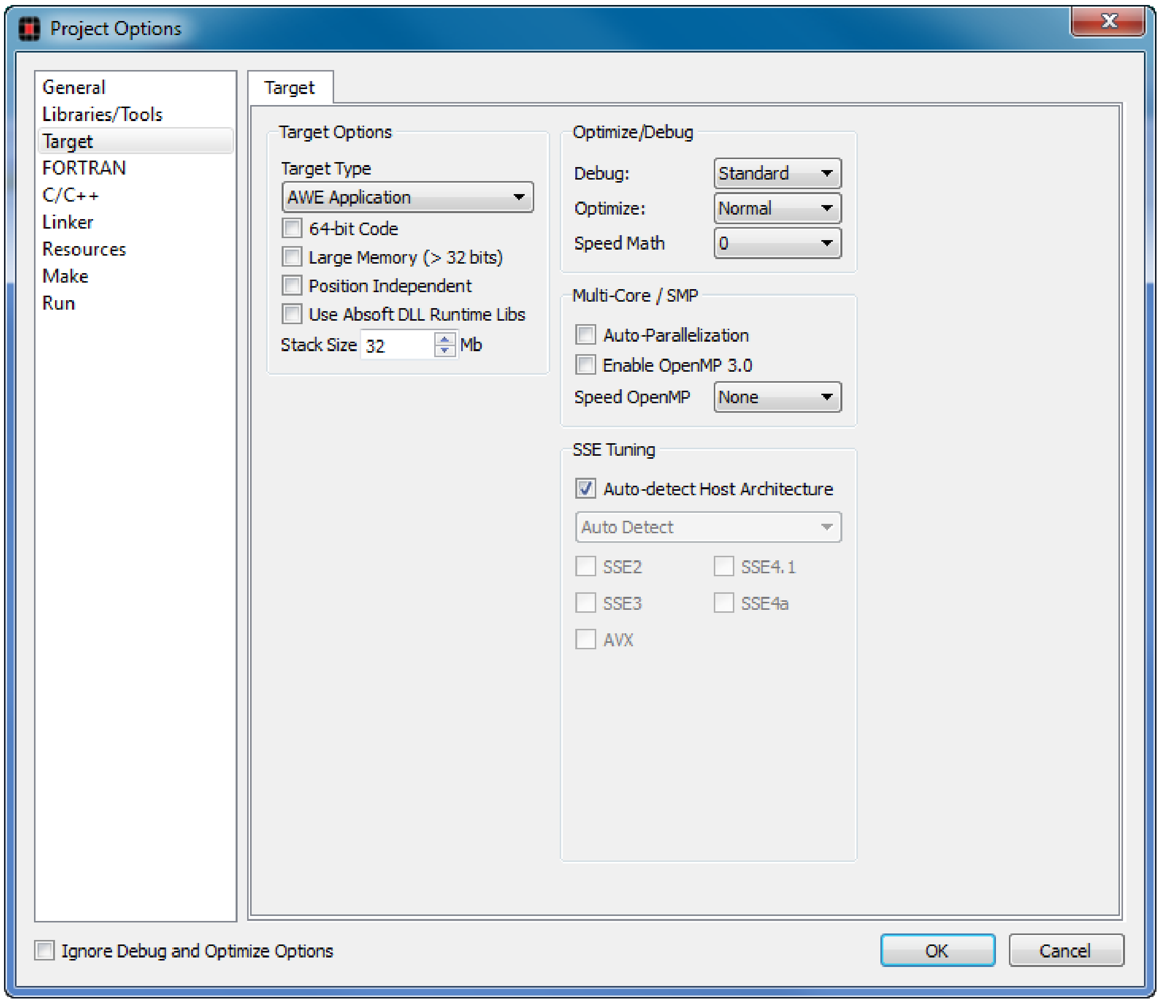
- Caution should be used when defining and using macros, especially when macros are to be invoked recursively and when using any of the special pre-defined macros described earlier in this section.

CHAPTER 8

The Absoft Window Environment

This chapter describes AWE, the Absoft Window Environment. AWE provides an alternate executable format to a simple terminal application. AWE supplies a windowed application for program input and output with the ability to save and print the output. In addition, you can open new windows and communicate with them through normal Fortran `READ` and `WRITE` statements.

An Absoft Window Environment application is selected by choosing a **Target Type** of **AWE Application** from the **Target** pane of the Options dialog. An AWE application can also be selected from the command line with **–awe** option.



Target Options
Target Type: AWE Application
[ ] 64-bit Code
[ ] Large Memory (> 32 bits)
[ ] Position Independent
[ ] Use Absoft DLL Runtime Libs
Stack Size 32 Mb

Optimize/Debug
Debug: Standard
Optimize: Normal
Speed Math: 0

Multi-Core / SMP
[ ] Auto-Parallelization
[ ] Enable OpenMP 3.0
Speed OpenMP: None

SSE Tuning
[✓] Auto-detect Host Architecture
Auto Detect
[ ] SSE2 [ ] SSE4.1
[ ] SSE3 [ ] SSE4a
[ ] AVX

[ ] Ignore Debug and Optimize Options

AWE PREFERENCES

When an AWE application is selected, AbsoftTools automatically adds the file `AWE_Prefences.f95` to the project. This file contains functions and subroutines that set the default settings for the behavior of the AWE application. You can alter the default behavior by simply editing this file. The procedures are:

```
Integer function AWE_getStackSize()
```

This routine specifies the stack size for an AWE application. The default stack size is 32 megabytes.

```
logical function AWE_getMdiMode()
```

This function controls whether windows opened in AWE will appear inside a single "frame" window or whether they open as individual windows. The default is to open windows inside the frame.

```
logical function AWE_getShowMaximized()
```

This function can be used to open the AWE window already maximized. The default is `.false.`.

```
logical function AWE_promptSaveOnExit()
```

This function controls whether AWE prompts to save the output window(s) at program exit. If this prompt is disabled, the contents of the window(s) will be lost if not explicitly saved. The default is to display a prompt to save the output.

```
integer function AWE_getMainWindowWidth()
```

This function controls the initial width of the window. The default is 1024.

```
integer function AWE_getMainWindowHeight()
```

This function controls the initial height of the window. The default is 768.

```
integer function AWE_defaultFontSize()
```

This function controls the height of the font use in the window. The default is 10.

```
subroutine AWE_defaultFontFamily(family)
```

This subroutne controls the family of the font use in the window. The default is `“Sans”`.

```
logical function AWE_autoSave()
```

This function controls whether the window text is automatically saved when the program exits. If this function returns `.true.`, the content of any windows will be automatically saved to files with the names of the windows. The default is `.false.`.

```
logical function AWE_showDefaultOutputWindow()
```

This function controls whether default AWE window is shown or not. If you only want to show a plot or a canvas without the default text window, set this to `.false.`. The default is `.true.`. Note that if this window is not shown, input/output to the system device is not available.

OPENING ADDITIONAL TEXT WINDOWS

Additional text windows can be created with the Fortran `OPEN` statement setting the `ACCESS=` specifier to:

```
      "window [,height, width]"
```

The optional arguments, *`height`* and *`width`*, are integers which specify the dimensions of the window in pixels. The window title will be the argument of the `FILE=` specifier of the `OPEN` statement. For example:

```
      OPEN(15, FILE=”my window”, ACCESS=”window, 800, 400”)
```

DETERMINING WHEN A WINDOW CLOSES

If your program needs to know when a window closes (perhaps by the user clicking in the close box for it), add a subroutine named `AWE_windowDidClose` to your program. It will be called when a window closes with the name of the window as a `CHARACTER` argument.

```
      SUBROUTINE AWE_windowDidClose(WindowName), &
                  BIND(C,NAME=”_AWE_windowDidClose”)
      CHARACTER(LEN=*) WindowName
      END SUBROUTINE AWE_windowDidClose
```

AWE MENUS

You can add your own menus and callback subroutines to an AWE application. After adding all of your menu commands and connecting them to callback subroutines, you exit your program normally. Then, when a menu command is chosen, your callback subroutine is entered. This section describes the functionality that is available. The interfaces indicate these are integer functions. They do not return any useful information and the result can be discarded.

```
interface
  subroutine AWE_addMenu (unit, title, text, callback)
  integer(kind=any) :: unit
  character(len=*) :: title, text
  external :: callback
  end subroutine AWE_addMenu
end interface
```

`unit` is Fortran unit number used to open the window. The unit number of the default input/output window is -2. `title` is the name of the menu and `text` is the name of the menu command. `callback` is the name of a subroutine in your program that is called when the menu command is selected.

To add an accelerator to a menu use the ampersand character (&) before the letter in the menu command (the `text` variable) that will be the accelerator. For example:

```
CALL AWE_addMenu(MyUnit, "My Menu", "&A Command", MyCallbackA)
```

If an accelerator is used with the name of menu (the `title` argument), then all calls to `AWE_addMenu` must specify the accelerator. For example:

```
CALL AWE_addMenu(MyUnit, "&My Menu", "&A Command", MyCallbackA)
CALL AWE_addMenu(MyUnit, "&My Menu", "&B Command", MyCallbackB)
```

```
interface
  subroutine AWE_setItemCheckable(unit, title, text, flag)
  integer(kind=any) :: unit
  character(len=*) :: title, text
  logical(kind=4) :: flag
  end subroutine AWE_setMenuItemCheckable
end interface
```

`unit` is Fortran unit number used to open the window. The unit number of the default input/output window is -2. `title` is the name of a previously added menu and `text` is the name of a previously added menu command. The menu command will be checkable if `flag` is `.true.`.

```
interface
  logical function AWE_isMenuItemChecked(unit, title, text)
  integer(kind=any) :: unit
  character(len=*) :: title, text
  end function AWE_ isMenuItemChecked
end interface
```

`unit` is Fortran unit number used to open the window. The unit number of the default input/output window is -2. `title` is the name of a previously added menu and `text` is the name of a previously added menu command. The functions returns `.true.` if the menu item is checked.

```
interface
  subroutine AWE_setItemChecked(unit, title, text, flag)
  integer(kind=any) :: unit
  character(len=*) :: title, text
  logical(kind=4) :: flag
  end subroutine AWE_setItemChecked
end interface
```

`unit` is Fortran unit number used to open the window. The unit number of the default input/output window is -2. `title` is the name of a previously added menu and `text` is the name of a previously added menu command. The menu command must also have been specified in a previous `AWE_setMenuItemCheckable` reference. The menu command will be checked if `flag` is `.true.`. The menu command will be unchecked if `flag` is `.false.`.

```
interface
  subroutine AWE_menuItemEnable(unit, title, text, flag)
  integer(kind=any) :: unit
  character(len=*) :: title, text
  logical(kind=4) :: flag
  end subroutine AWE_menuItemEnable
end interface
```

`unit` is Fortran unit number used to open the window. The unit number of the default input/output window is -2. `title` is the name of a previously added menu and `text` is the name of a previously added menu command. The menu command will be enabled if `flag` is `.true.`. The menu command will be disabled if `flag` is `.false.`.

SPREAD SHEETS

You can create spread sheet windows in AWE to display rank 2 arrays. Subroutines are provided to open, close, read, write, and label spread sheets. Menu commands, described above, can be added to an AWE program to manipulate the data in the spread sheet.

```
interface
  subroutine AWE_CreateSpreadsheet(unit, title, rows, columns)
  integer(kind=4) :: unit
  character(len=*) :: title
  logical(kind=4) :: rows, columns
  end subroutine AWE_CreateSpreadsheet
end interface
```

This subroutine creates a spread sheet window. `unit` is the value used to identify the spread sheet in subsequent spread sheet write, read, and close subroutine references, described next. `title` is the title that will be displayed in the spread sheet window. `rows` and `columns` are the number of rows and columns respectively in the spread sheet.

```
interface
  subroutine AWE_setHorizontalHeaderLabels(unit, labels)
  integer(kind=4) :: unit
  character(len=*), dimension(:) :: array
  end subroutine AWE_setHorizontalHeaderLabels
end interface
```

This subroutine is used to set the horizontal labels of the spread sheet. `unit` is the value that was used to identify the spread sheet when it was opened. `labels` is a rank 1 `character` array used to label the columns.

```
interface
  subroutine AWE_setVerticalHeaderLabels(unit, labels)
  integer(kind=4) :: unit
  character(len=*), dimension(:) :: array
  end subroutine AWE_setVerticalHeaderLabels
end interface
```

This subroutine is used to set the vertical labels of the spread sheet. `unit` is the value that was used to identify the spread sheet when it was opened. `labels` is a rank 1 `character` array used to label the rows.

```
interface
  subroutine AWE_writeSpreadsheet(unit, array)
  integer(kind=4) :: unit
  any, dimension(:,:) :: array
  end subroutine AWE_writeSpreadsheet
end interface
```

This subroutine is used to transfer data from an array in your program to the spread sheet. `unit` is the value that was used to identify the spread sheet when it was opened. `array` can be any type or kind. Its shape must match the number of rows and columns specified when the spreadsheet was opened.

```
interface
  subroutine AWE_readSpreadsheet(unit, array)
  integer(kind=4) :: unit
  any, dimension(:,:) :: array
  end subroutine AWE_readSpreadsheet
end interface
```

This subroutine is used to read the spread sheet data into an array in your program. `unit` is the value that was used to identify the spread sheet when it was opened. `array` should be the same type and kind used in writes to the spread sheet. The shape of `array` must match the number of rows and columns specified when the spreadsheet was opened.

```
interface
  subroutine AWE_closeSpreadsheet(unit)
  integer(kind=4) :: unit
  end subroutine AWE_closeSpreadsheet
end interface
```

This subroutine closes the spread sheet window. `unit` is the value that was used to identify the spread sheet when it was opened.

ALERT BOXES

An alert box can be displayed with the following function:

```
interface
  subroutine AWE_alertBox(title, text)
  character(len=*) :: title, text
  end subroutine AWE_alertBox
end interface
```

`title` is used as the title of the alert box `text` is the text that will be displayed in it.

PLOTS

AWE can be used to create several different types of plots: XY plots, contour plots, polar charts, bar charts, and pie charts. The plots can be printed and saved to PNG (Portable Network Graphics) format files. Default values are supplied for all label and color parameters allowing you to quickly display your data. Each of these parameters is easily customized giving you the flexibility to produce professional looking plots.

AWE plots use RGB colors extensively. Although default values are always supplied, you can replace them with any RGB value you want. The Appendix, *AWE RGB Colors*, in this manual lists a number of predefined values supplied in the `AWE_Interfaces` module that are available to you by simply using their symbolic names.

An example of using AWE to create each of the three types of plots in provided in the Absoft examples directory.

Pie Charts

Only three subroutine calls are needed to create, display and close pie charts. An AWE derived type called `AWE_PieChart` is used with each subroutine to specify the pie chart. It is defined as follows:

```
        TYPE AWE_PieChart
          INTEGER, PRIVATE :: id
          CHARACTER*(128) :: title = "Pie Chart"
          INTEGER :: chartBackgroundColor = z'FFE8E8E8'
          INTEGER :: chartTextColor = AWE_black
          CHARACTER*(128), ALLOCATABLE :: legendNames(:)
          INTEGER, ALLOCATABLE :: legendColors(:)
        END TYPE
```

`id` is the internal pie chart identifier. AWE will automatically assign a unique value when the pie chart is created.

`title` it the pie chart title. It is centered and displayed on top of the chart.

`chartBackgroundColor` is the RGB background color the chart is displayed on.

`chartTextColor` is the RGB color of the legend text.

`legendNames` is an array of the names used for legends. The legend is displayed vertically on the right side of the chart.

`legendColors` is an array of the RGB colors used for chart wedges.

To create a pie chart, declare an instance of `AWE_PieChart`, supply the parameters you want to customize and call the `AWE_createPiechart` subroutine with the `AWE_PieChart` variable as an argument:

```
        USE AWE_Interfaces
        TYPE(AWE_PieChart) :: piechart
        CALL AWE_createPiechart(piechart)
```

To display the pie chart, call the `AWE_writePiechart` subroutine with the `AWE_PieChart` variable used to create the pie chart and a rank 1 array with your data. The rank 1 array can be either `INTEGER(KIND=4)`, `INTEGER(KIND=8)`, or `REAL` of any `KIND`.

```
        REAL, DIMENSION(3) :: array=[3,4,5]
        CALL AWE_writePiechart(piechart, array)
```

To close the pie chart, call the `AWE_closePiechart` subroutine with the `AWE_PieChart` variable used to create the pie chart:

```
        CALL AWE_closePiechart(piechart)
```

To save a pie chart to a file, call the `AWE_savePiechart` subroutine the `AWE_PieChart` variable used to create the chart and the name of the file to save it to. The format will automatically be determined by the file name extension. The default is PNG. PNG and BMP are supported on all operating systems.

```
        CALL AWE_savePiechart(piechart,”chart.png”)
```

Bar Charts

Only three subroutine calls are needed to create, display and close bar charts. An AWE derived type called `AWE_BarChart` is used with each subroutine to specify the bar chart. It is defined as follows:

```
        TYPE AWE_BarChart
          INTEGER, PRIVATE :: id
          CHARACTER*(128) :: title = "Title"
          CHARACTER*(128) :: xAxisName = "xAxis"
          CHARACTER*(128) :: yAxisName = "yAxis"
          INTEGER :: chartBackgroundColor = z'FFE8E8E8'
          CHARACTER*(128), ALLOCATABLE :: legendNames(:)
          INTEGER, ALLOCATABLE :: legendColors(:)
        END TYPE
```

`id` is the internal bar chart identifier. AWE will automatically assign a unique value when the bar chart is created.

`title` it the bar chart title. It is centered and displayed on top of the chart.

`xAxisName` is the name of the X axis. It is centered and displayed on bottom of the chart.

`yAxisName` is the name of the Y axis. It is centered and displayed on the left side of the chart.

`chartBackgroundColor` is the RGB background color the chart is displayed on.

`legendNames` is an array of the names used for legends. The legend is displayed vertically on the right side of the chart.

`legendColors` is an array of the RGB colors used for chart bars.

To create a bar chart, declare an instance of `AWE_BarChart`, supply the parameters you want to customize and call the `AWE_createBarchart` subroutine with the `AWE_BarChart` variable as an argument:

```
        USE AWE_Interfaces
        TYPE(AWE_BarChart) :: barchart
        CALL AWE_createBarchart(barchart)
```

To display the bar chart, call the `AWE_writeBarchart` subroutine with the `AWE_BarChart` variable used to create the bar chart and a rank 2 array with your data. The rank 2 array can be either `INTEGER(KIND=4)`, `INTEGER(KIND=8)`, or `REAL` of any `KIND`. The first dimension is the number of bars per interval and second dimension is the intervals.

```
        REAL, DIMENSION(3,3):: array
        array = RESHAPE([4,2,8,4,3,6,1,8,2], [3,3])
        CALL AWE_writeBarchart(barchart, array)
```

To close the bar chart, call the `AWE_closeBarchart` subroutine with the `AWE_BarChart` variable used to create the bar chart:

```
        CALL AWE_closeBarchart(barchart)
```

To save a bar chart to a file, call the `AWE_saveBarchart` subroutine the `AWE_BarChart` variable used to create the chart and the name of the file to save it to. The format will automatically be determined by the file name extension. The default is PNG. PNG and BMP are supported on all operating systems.

```
        CALL AWE_saveBarchart(barchart,”chart.png”)
```

XY Plots

Only three subroutine calls are needed to create, display and close XY plots. Two AWE derived types are used are used to specify the appearance of XY plots. The first one is used for titles, and axis scaling. It is defined as follows:

```
        TYPE AWE_XYPlot
          INTEGER, PRIVATE :: id
          CHARACTER*(128) :: title = "Title"
          CHARACTER*(128) :: xAxisName = "xAxis"
          CHARACTER*(128) :: yAxisName = "yAxis"
          INTEGER         :: chartBackgroundColor = z'FFE8E8E8'
          INTEGER         :: xAxisScaleType = AWE_ScaleType_Linear
          REAL(KIND=8)    :: xAxisScaleUB = undefined
          REAL(KIND=8)    :: xAxisScaleLB = undefined
          REAL(KIND=8)    :: xAxisScaleStep = undefined
          INTEGER         :: yAxisScaleType = AWE_ScaleType_Linear
          REAL(KIND=8)    :: yAxisScaleUB = undefined
          REAL(KIND=8)    :: yAxisScaleLB = undefined
          REAL(KIND=8)    :: yAxisScaleStep = undefined
          INTEGER         :: showHorizontalGridLines = .TRUE.
          INTEGER         :: showVerticalGridLines = .TRUE.
        END TYPE
```

`id` is the internal XY plot identifier. AWE will automatically assign a unique value when the XY plot is created.

`title` is the XY plot title. It is centered and displayed on top of the chart.

`xAxisName` is the name of the X axis. It is centered and displayed on bottom of the plot.

`yAxisName` is the name of the Y axis. It is centered and displayed on the left side of the plot.

chartBackgroundColor is the RGB background color the plot is displayed on.

xAxisScaleType specifies the scale type of the X axis. The two possible values are AWE_ScaleType_Linear and AWE_ScaleType_Logarithmic.

xAxisScaleUB specifies the upper bound of the X axis. The default is the upper bound of the x data.

xAxisScaleLB specifies the lower bound of the X axis. The default is the lower bound of the x data.

xAxisScaleStep specifies the step increments of the X axis. The default is an increment appropriate to the range of the x data.

yAxisScaleType specifies the scale type of the Y axis. The two possible values are AWE_ScaleType_Linear and AWE_ScaleType_Logarithmic.

yAxisScaleUB specifies the upper bound of the Y axis. The default is the upper bound of the y data.

yAxisScaleLB specifies the lower bound of the Y axis. The default is the lower bound of the y data.

yAxisScaleStep specifies the step increments of the Y axis. The default is an increment appropriate to the range of the y data.

showHorizontalGridLines enables or disables horizontal grid lines. The default is .TRUE..

showVerticalGridLines enables or disables vertical grid lines. The default is .TRUE..

Multiple curves can plotted on a single XY plot. The color, style, and label for each curve is specified in a unique AWE_XYPlot_Data derived type:

```
TYPE AWE_XYPlot_Data
  CHARACTER*(128) :: curveLabel = "Label"
  INTEGER :: curveColor = AWE_steel_blue
  INTEGER :: curveWidth = 1
  INTEGER :: plotSymbolColor = AWE_crimson
  INTEGER :: plotSymbolSize = 6
  INTEGER :: plotSymbolStyle = AWE_PlotSymbol_NoSymbol
  INTEGER :: fittedCurve = AWE_InvertedCurve
END TYPE AWE_XYPlot_Data
```

curveLabel is the label used to identify the curve. It is listed on the right side of the plot.

curveColor is the RGB color used to draw the curve.

curveWidth is the size of the pen used to draw the curve.

`plotSymbolColor` is the RGB color used to draw the symbol at the data points of the plot. See `plotSymbolStyle` below.

`plotSymbolSize` is the size of the symbol drawn at the data points of the plot. See `plotSymbolStyle` below.

`plotSymbolStyle` is the style of the symbol drawn at the data points of the plot. It can be one of the following:

```
AWE_PlotSymbol_NoSymbol          AWE_PlotSymbol_Ellipse
AWE_PlotSymbol_Rect              AWE_PlotSymbol_Diamond
AWE_PlotSymbol_Triangle          AWE_PlotSymbol_DTriangle
AWE_PlotSymbol_UTriangle         AWE_PlotSymbol_LTriangle
AWE_PlotSymbol_RTriangle         AWE_PlotSymbol_Cross
AWE_PlotSymbol_XCross            AWE_PlotSymbol_HLine
AWE_PlotSymbol_VLine             AWE_PlotSymbol_Star1
AWE_PlotSymbol_Star2             AWE_PlotSymbol_Hexagon
```

`fittedCurve` can be either `AWE_InvertedCurve` or `AWE_FittedCurve`.

To create an XY plot, declare an instance of `AWE_XYPlot`, supply the parameters you want to customize and call the `AWE_createXYPlot` subroutine with the `AWE_XYPlot` variable as an argument:

```
USE AWE_Interfaces
TYPE(AWE_XYPlot) :: xyplot
CALL AWE_createXYPlot(xyPlot)
```

To plot an XY curve, call the `AWE_writeXYPlot` subroutine with the `AWE_XYPlot` variable used to create the plot, a rank 2, shape `(2,:)` array, and an instance of `AWE_XYPlot_Data` variable. The rank 2 array can be either `INTEGER(KIND=4)`, `INTEGER(KIND=8)`, or `REAL` of any `KIND`.

```
TYPE(AWE_XYPlot_Data) :: plotData
REAL, DIMENSION(2,5) :: array
array = RESHAPE([2,5,4,3,6,7,8,5,10,11], [2,5])
CALL AWE_writeXYPlot(xyplot, array, plotData)
```

To close the XY plot, call the `AWE_closeXYPlot` subroutine with the `AWE_XYPlot` variable used to create the plot:

```
CALL AWE_closeXYPlot(xyPlot)
```

To clear an XY plot, call the `AWE_clearXYPlot` subroutine with the `AWE_XYPlot` variable used to create the plot:

```
CALL AWE_clearXYPlot(xyPlot)
```

To save a plot to a file, call the `AWE_saveXYPlot` subroutine the `AWE_XYPlot` variable used to create the plot and the name of the file to save it to. The format will automatically be determined by the file name extension. The default is PNG. PNG and BMP are supported on all operating systems.

```
        CALL AWE_saveXYPlot(xyplot,”plot.png”)
```

Contour Plots

A contour plot is a two-dimensional representation of a three dimensional dat. Two AWE derived types are used are used to specify the appearance of contour plots. The first one is used for titling the plot. It is defined as follows:

```
      TYPE AWE_ContourPlot
        INTEGER, PRIVATE id
        CHARACTER*(128) :: title = "Title"
        LOGICAL         :: showHorizontalGridLines = .FALSE.
        LOGICAL         :: showVerticalGridLines   = .FALSE.
        LOGICAL         :: showImageBlend          = .TRUE.
        LOGICAL         :: showContourLines        = .FALSE.
      END TYPE
```

`title` is the contour plot title. It is centered and displayed on top of the plot.

`showHorizontalGridLines` enable or disable horizontal grid lines. The default is `.FALSE`.

`showVerticalGridLines` enable or disable vertical grid lines. The default is `.FALSE`.

`showImageBlend` enable or disable display of contour color map. The default is `.TRUE`.

`showContourLines` enable or disable display of contour lines for the contour levels. The default is `.FALSE`.

The second derived type supplies the color map, thresholds, and axis limits.

```
      TYPE AWE_ContourPlot_Data
        CHARACTER*(128) :: rightAxisName = "Label"
        INTEGER, ALLOCATABLE :: mapColors(:)
        REAL(KIND=KIND(1.0d0)), ALLOCATABLE :: mapThresholds(:)
        REAL(KIND=KIND(1.0d0)), ALLOCATABLE :: contourLevels(:)
        REAL(KIND=KIND(1.0d0)) xmin, xmax, ymin, ymax, zmin, zmax
      END TYPE AWE_ContourPlot_Data
```

`rightAxisName` is the name of the Z axis and is displayed on the right side of the plot.

`mapColors` is an array of colors used to define three dimensional depths.

`mapThresholds` is an array of z axis color thresholds. The thresholds must be a vector greater than 1 and range from 0.0 - 1.0. The z values are mapped into the thresholds to produce the color.

`contourLevels` is an array of values that determine the placement of contour lines when they are enabled.

`xmin, xmax, ymin, ymax, zmin,` and `zmax` are the axes limits.

To create a contour plot, declare an instance of `AWE_ContourPlot`, supply the parameters you want to customize and call the `AWE_createContourPlot` subroutine with the `AWE_ContourPlot` variable as an argument:

```
        USE AWE_Interfaces
        TYPE(AWE_ContourPlot) :: ContourPlot
        CALL AWE_createContourPlot(ContourPlot)
```

To plot contour, call the `AWE_writeContourPlot` subroutine with the `AWE_ContourPlot` variable used to create the plot, a double precision function name to generate the z axis data, and an instance of `AWE_ContourPlot_Data` variable..

```
        TYPE(AWE_ContourPlot_Data) :: plotData
        REAL(KIND=KIND(1.0d0), EXTERNAL :: ContourCallback
        CALL AWE_writeContourPlot(ContourPlot, ContourCallback, plotData)
```

`ContourCallback` is a pure, double precision function that takes two double precision arguments, x and y, and returns z as its result:

```
        INTERFACE
          PURE REAL(KIND=KIND(1.0d0) FUNCTION ContourCallback (x,y)
            REAL(KIND=KIND(1.0d0), INTENT(IN) :: x,y
          END FUNCTION ContourCallback
        END INTERFACE
```

NOTE: `ContourCallback` is called from a separate thread and cannot be used to reliably create and/or share data with other routines in your program as there is no way to ensure synchronization.

To close the contour plot, call the `AWE_closeContourPlot` subroutine with the `AWE_ContourPlot` variable used to create the plot:

```
        CALL AWE_closeContourPlot(ContourPlot)
```

To clear a contour plot, call the `AWE_clearContourPlot` subroutine with the `AWE_ContourPlot` variable used to create the plot:

```
        CALL AWE_clearContourPlot(ContourPlot)
```

To save a plot to a file, call the `AWE_saveContourPlot` subroutine the `AWE_ContourPlot` variable used to create the plot and the name of the file to save it to. The format will automatically be determined by the file name extension. The default is PNG. PNG and BMP are supported on all operating systems.

```
        CALL AWE_saveContourPlot(ContourPlot,”plot.png”)
```

Polar Plots

Only three subroutine calls are needed to create, display and close polar plots. Two AWE derived types are used are used to specify the appearance of polar plots. The first one is used for the plot title and background. It is defined as follows:

```
TYPE AWE_PolarPlot
  INTEGER, PRIVATE id
  CHARACTER*(128) :: title = "Title"
  INTEGER :: chartBackgroundColor = AWE_dark_blue
END TYPE PolarPlot
```

`id` is the internal polar plot identifier. AWE will automatically assign a unique value when the polar plot is created.

`title` it the polar plot title. It is centered and displayed on top of the chart.

`chartBackgroundColor` is background color of the polar axis.

The color, style, and label for each plot is specified in a unique `AWE_PolarPlot_Data` derived type:

```
TYPE AWE_PolarPlot_Data
  CHARACTER*(128) :: curveLabel = "Label"
  INTEGER :: curveColor      = AWE_yellow
  INTEGER :: curveWidth      = 1
  INTEGER :: plotSymbolColor = AWE_red
  INTEGER :: plotSymbolSize  = 3
  INTEGER :: plotSymbolStyle = AWE_PlotSymbol_NoSymbol
END TYPE AWE_PolarPlot_Data
```

`curveLabel` is the label used to identify the curve. It is listed on the bottom of the plot.

`curveColor` is RGB color used to draw the curve.

`curveWidth` is the size of the pen used to draw the curve.

`plotSymbolColor` is the RGB color used to draw the symbol at the data points of the plot. See `plotSymbolStyle` below.

`plotSymbolSize` is the size of the symbol drawn at the data points of the plot. See `plotSymbolStyle` below.

`plotSymbolStyle` is the style of the symbol drawn at the data points of the plot. It can be one of the following:

```
AWE_PlotSymbol_NoSymbol          AWE_PlotSymbol_Ellipse
AWE_PlotSymbol_Rect              AWE_PlotSymbol_Diamond
AWE_PlotSymbol_Triangle          AWE_PlotSymbol_DTriangle
AWE_PlotSymbol_UTriangle         AWE_PlotSymbol_LTriangle
AWE_PlotSymbol_RTriangle         AWE_PlotSymbol_Cross
AWE_PlotSymbol_XCross            AWE_PlotSymbol_HLine
AWE_PlotSymbol_VLine             AWE_PlotSymbol_Star1
AWE_PlotSymbol_Star2             AWE_PlotSymbol_Hexagon
```

To create a polar plot, declare an instance of `AWE_PolarPlot`, supply the parameters you want to customize and call the `AWE_createPolarPlot` subroutine with the `AWE_PolarPlot` variable as an argument:

```
        USE AWE_Interfaces
        TYPE(AWE_PolarPlot) :: plot
        CALL AWE_createPolarPlot(plot)
```

To plot an polar plot, call the `AWE_writePolarPlot` subroutine with the `AWE_PolarPlot` variable used to create the plot, a rank 2, shape `(2,:)` array, and an instance of `AWE_PolarPlot_Data` variable. The rank 2 array can be either `INTEGER(KIND=4)`, `INTEGER(KIND=8)`, or `REAL` of any `KIND`.

```
        TYPE(AWE_PolarPlot_Data) :: plotData
        REAL, DIMENSION(2,600) :: array
        CALL AWE_writePolarPlot(plot, array, plotData)
```

To close the polar plot, call the `AWE_closePolarPlot` subroutine with the `AWE_PolarPlot` variable used to create the plot:

```
        CALL AWE_closePolarPlot(plot)
```

To clear a polar plot, call the `AWE_clearPolarPlot` subroutine with the `AWE_PolarPlot` variable used to create the plot:

```
        CALL AWE_clearPolarPlot(plot)
```

To save a plot to a file, call the `AWE_savePolarPlot` subroutine the `AWE_PolarPlot` variable used to create the plot and the name of the file to save it to. The format will automatically be determined by the file name extension. The default is PNG. PNG and BMP are supported on all operating systems.

```
        CALL AWE_savePolarPlot(plot,”plot.png”)
```

CANVASES

A canvass provides a drawing surface and graphics primitives that can be used to create a free form drawing. Among the many primitives provided are line, rectangle, arc, are polygon, and commands. A number of derived types are used with canvases. They will be described first.

Canvas Derived Types

The AWE_Canvas type is used to specify the canvas for creation and all subsequent calls.

```
TYPE AWE_Canvas
  INTEGER, PRIVATE :: id
  CHARACTER(LEN=128) :: title = ""
  INTEGER :: width  = 500
  INTEGER :: height = 500
  INTEGER :: backgroundColor = AWE_white
END TYPE AWE_Canvas
```

`id` is the internal canvas identifier. AWE will automatically assign a unique value when the canvas is created.

`title` is the canvas window title.

`Width` and `height` are the dimensions of the canvas. The defaults are 500 x 500.

`backgroundColor` is the color of the drawing background.

An `AWE_CanvasPen` defines the style of the pen used with various other drawing commands.

```
TYPE AWE_CanvasPen
  REAL    :: penWidth  = 0.0
  INTEGER :: penStyle  = CanvasPenStyle_SolidLine
  INTEGER :: capStyle  = CanvasPenCapStyle_SquareCap
  INTEGER :: joinStyle = CanvasPenJoinStyle_BevelJoin
  INTEGER :: penColor  = AWE_black
END TYPE AWE_CanvasPen
```

`penWidth` is the width of the pen.

`penStyle` is the style of the pen. The following styles are available:

```
CanvasPenStyle_NoPen             CanvasPenStyle_SolidLine
CanvasPenStyle_DashLine          CanvasPenStyle_DotLine
CanvasPenStyle_DashDotLine       CanvasPenStyle_DashDotDotLine
```

`capStyle` defines the shape of the end cap of line segment drawn with the pen. The following styles are available:

```
CanvasPenCapStyle_SquareCap      CanvasPenCapStyle_FlatCap
CanvasPenCapStyle_RoundCap
```

`joinStyle` defines the shape of joint between two line segments drawn with the pen. The following styles are available:

```
CanvasPenJoinStyle_BevelJoin     CanvasPenJoinStyle_MiterJoin = 1
CanvasPenJoinStyle_RoundJoin
```

`penColor` defines the color of the pen.

An `AWE_CanvasBrush` defines the style of the brush used with various other drawing commands.

```
        TYPE AWE_CanvasBrush
          INTEGER :: brushColor = AWE_black
        END TYPE AWE_CanvasBrush
```

`brushColor` defines the color of the brush.

An `AWE_Point` defines a point on the canvas.

```
        TYPE AWE_Point
          REAL x,y
        END TYPE AWE_Point
```

`x` and `y` are the coordinates of the point.

An `AWE_Size` defines two dimensional space.

```
        TYPE AWE_Size
          REAL width,height
        END TYPE AWE_Size
```

`width` and `height` are the dimensions.

An `AWE_Line` defines a line on the canvas.

```
        TYPE AWE_Line
         TYPE(AWE_Point) :: start, end
        END TYPE AWE_Line
```

`start` and `end` are the end points of the line. They are given as instances of an `AWE_Point` type.

An `AWE_Rect` defines a rectangle on the canvas.

```
        TYPE AWE_Rect
          TYPE(AWE_Point) :: origin
          TYPE(AWE_Size) :: size
        END TYPE AWE_Rect
```

`origin` is given an `AWE_Point` and is the origin of the rectangle.

`size` is given an `AWE_Size` and is the width and height of the rectangle.

An AWE_Font defines a font for use with text drawing routines on the canvas.

```
        TYPE AWE_Font
          CHARACTER(LEN=64) :: familyName = "Sans"
          INTEGER :: pointSize = 12
          INTEGER :: weight    = AWE_FontWeight_Normal
          LOGICAL :: italic    = .false.
        END TYPE AWE_Font
```

familyName defines the font family of the font.

pointSize defines the size of the font.

weight defines the weight of the font. The following weights are available:

```
   AWE_FontWeight_Light             AWE_FontWeight_Normal
   AWE_FontWeight_DemiBold          AWE_FontWeight_Bold
   AWE_FontWeight_Black
```

italic specified if the font is be in *italics*. The default is .false..

Canvas Routines

The following routines are used to create, close, and draw on the canvas.

To create a canvas, declare an instance of AWE_Canvas, supply the parameters you want to customize and call the AWE_createCanvas subroutine with the AWE_Canvas variable as an argument:

```
        USE AWE_Interfaces
        TYPE(AWE_Canvas) :: canvas
        CALL AWE_createCanvas(canvas)
```

To draw lines on a canvas, use the AWE_canvasDrawLines subroutine.

```
        USE AWE_Interfaces
        TYPE(AWE_Canvas) :: canvas
        TYPE(AWE_Line) :: lines(:)
        TYPE(AWE_CanvasPen), OPTIONAL :: pen
        CALL AWE_createDrawLines(canvas, lines, pen)
```

Note that lines is an array of AWE_Line types. If the optional pen argument is omitted, the last pen used will be supplied.

To draw an arc on a canvas, use the `AWE_canvasDrawArc` subroutine.

```
        USE AWE_Interfaces
        TYPE(AWE_Canvas) :: canvas
        TYPE(AWE_Rect) :: rect
        REAL startAngle, spanAngle
        TYPE(AWE_CanvasPen), OPTIONAL :: pen
        CALL AWE_canvasDrawArc(canvas, rect, startAngle, spanAngle, pen)
```

`rect` defines the bounding rectangle. `startAngle` specifies the beginning of the arc in degrees. `spanAngle` defines the arc in degrees. If the optional `pen` argument is omitted, the last pen used will be supplied.

To draw rectangles on a canvas, use the `AWE_canvasDrawRects` subroutine.

```
        USE AWE_Interfaces
        TYPE(AWE_Canvas) :: canvas
        TYPE(AWE_Rect) :: rects(:)
        TYPE(AWE_CanvasPen), OPTIONAL :: pen
        TYPE(AWE_CanvasBrush), OPTIONAL :: brush
        CALL AWE_canvasDrawRects(canvas, rects, pen, brush)
```

Note that `rects` is an array of `AWE_Rect` types. If the optional `pen` and/or `brush` arguments are omitted, the last pen and/or brush used will be supplied.

To draw rounded rectangles on a canvas, use the `AWE_canvasDrawRoundedRects` subroutine.

```
        USE AWE_Interfaces
        TYPE(AWE_Canvas) :: canvas
        TYPE(AWE_Rect) :: rects(:)
        REAL :: xradius, yradius
        TYPE(AWE_CanvasPen), OPTIONAL :: pen
        TYPE(AWE_CanvasBrush), OPTIONAL :: brush
        CALL AWE_canvasDrawRoundedRects(canvas, rects, xradius,&
                                        yradius, pen, brush)
```

Note that `rects` is an array of `AWE_Rect` types. `xradius` and `yradius` control the rounded corners. If the optional `pen` and/or `brush` arguments are omitted, the last pen and/or brush used will be supplied.

To draw a polygon on a canvas, use the `AWE_canvasDrawPolygon` subroutine.

```
        USE AWE_Interfaces
        TYPE(AWE_Canvas) :: canvas
        TYPE(AWE_Point) :: points(:)
        TYPE(AWE_CanvasPen), OPTIONAL :: pen
        TYPE(AWE_CanvasBrush), OPTIONAL :: brush
        CALL AWE_canvasDrawPolygon(canvas, points, pen, brush)
```

If the optional `pen` and/or `brush` arguments are omitted, the last pen and/or brush used will be supplied.

To draw a chord on a canvas, use the AWE_canvasDrawChord subroutine.

```
        USE AWE_Interfaces
        TYPE(AWE_Canvas) :: canvas
        TYPE(AWE_Rect) :: rect
        REAL startAngle, spanAngle
        TYPE(AWE_CanvasPen), OPTIONAL :: pen
        TYPE(AWE_CanvasBrush), OPTIONAL :: brush
        CALL AWE_canvasDrawChord(canvas, rect, startAngle, spanAngle, &
                                 pen, brush)
```

rect defines the bounding rectangle. startAngle specifies the beginning of the arc in degrees. spanAngle defines the arc in degrees. If the optional pen and/or brush arguments are omitted, the last pen and/or brush used will be supplied.

To draw a pie segment on a canvas, use the AWE_canvasDrawPie subroutine.

```
        USE AWE_Interfaces
        TYPE(AWE_Canvas) :: canvas
        TYPE(AWE_Rect) :: rect
        real startAngle, spanAngle
        TYPE(AWE_CanvasPen), OPTIONAL :: pen
        TYPE(AWE_CanvasBrush), OPTIONAL :: brush
        CALL AWE_canvasDrawPie(canvas, rect, startAngle, spanAngle, &
                               pen, brush)
```

rect defines the bounding rectangle. startAngle specifies the beginning of the arc in degrees. spanAngle defines the arc in degrees. If the optional pen and/or brush arguments are omitted, the last pen and/or brush used will be supplied.

To draw an ellipse on a canvas, use the AWE_canvasDrawEllipse subroutine.

```
        USE AWE_Interfaces
        TYPE(AWE_Canvas) :: canvas
        TYPE(AWE_Rect) :: rect
        TYPE(AWE_CanvasPen), OPTIONAL :: pen
        TYPE(AWE_CanvasBrush), OPTIONAL :: brush
        CALL AWE_canvasDrawEllipse(canvas, rect, pen, brush)
```

rect defines the bounding rectangle. The ellipse will be sized to fit the rectangle. If the optional pen and/or brush arguments are omitted, the last pen and/or brush used will be supplied.

To draw text on a canvas, use the AWE_canvasDrawText subroutine.

```
        USE AWE_Interfaces
        TYPE(AWE_Canvas) :: canvas
        TYPE(AWE_Rect) :: rect
        CHARACTER(LEN=*) text
        INTEGER, optional :: flags
        TYPE(AWE_Font) :: font
        INTEGER, OPTIONAL:: textColor = AWE_black
        CALL AWE_canvasDrawText(canvas, rect, text, flags, font, &
                               textColor)
```

`rect` defines the bounding rectangle. The ellipse will be sized to fit the rectangle. If the optional `textColor` argument is omitted, it defaults to `AWE_black`. The `flag` arguments can be a combination of:

```
AWE_TextFlag_None                 AWE_TextFlag_AlignLeft
AWE_TextFlag_AlignRight           AWE_TextFlag_AlignHCenter
AWE_TextFlag_AlignJustify         AWE_TextFlag_AlignTop
AWE_TextFlag_AlignBottom          AWE_TextFlag_AlignVCenter
AWE_TextFlag_AlignCenter          AWE_TextFlag_TextDontClip
AWE_TextFlag_TextSingleLine       AWE_TextFlag_TextExpandTabs
AWE_TextFlag_TextShowMnemonic     AWE_TextFlag_TextWordWrap
AWE_TextFlag_TextIncludeTrailingSpaces
```

To display a picture on a canvas, use the `AWE_canvasDrawPicture` subroutine.

```
    USE AWE_Interfaces
    TYPE(AWE_Canvas) :: canvas
    TYPE(AWE_Rect) :: rect
    CHARACTER(LEN=*) filename
    CALL AWE_canvasDrawPicture(canvas, rect, filename)
```

`rect` defines the bounding rectangle.

To close the canvas, call the `AWE_closeCanvas` subroutine with the `AWE_Canvas` variable used to create the canvas:

```
    CALL AWE_closeCanvas(canvas)
```

To clear a canvas, call the `AWE_clearCanvas` subroutine with the `AWE_Canvas` variable used to create the canvas:

```
    CALL AWE_clearCanvas(canvas)
```

To save a canvas to a file, call the `AWE_saveCanvas` subroutine the `AWE_ContourPlot` variable used to create the canvas and the name of the file to save it to. The format will automatically be determined by the file name extension. The default is PNG. PNG and BMP are supported on all operating systems.

```
   CALL AWE_saveCanvas(canvas,”canvas.png”)
```

DIALOGS

Modal dialogs can be easily created and displayed using AWE. A modal dialog requires the user to interact with it before returning to the main program flow. A typical use of an AWE dialog is to establish initial program parameters; however, they can be displayed at any point in the program where you need them.

There are three steps to working with AWE dialogs: create the dialog, add items to the dialog, and display the dialog. Dialog items automatically arranged vertically in the dialog box in the order they are added.

Creating an AWE dialog

Before you can add items to a dialog you must create it. This done with the `AWE_createDialog` subroutine call:

```
        USE AWE_Interfaces
        TYPE(AWE_FormDialog) :: dialog
        CALL AWE_createDialog(dialog)
```

The derived type `AWE_FormDialog` is defined as follows:

```
        TYPE AWE_FormDialog
          INTEGER, PRIVATE :: dialogID
          CHARACTER(LEN=128) title
        END TYPE
```

`dialogID` is the internal dialog identifier. AWE will automatically assign a unique value when the dialog is created.

`title` is the text that will be displayed in the dialog title bar.

Adding Items to an AWE dialog

Dialog items are displayed in a dialog in the order they are added. Items are added to a dialog with the `AWE_addToDialog` subroutine call:

```
        CALL AWE_addToDialog(item, dialog)
```

`item` is the dialog item to add (see below).

`dialog` is the instance of the `AWE_FormDialog` derived type used to create the dialog.

The following sections describe the items and how to add them to a dialog.

Dialog Labels

Static text is added to a dialog with an `AWE_FormLabel` derived type:

```
        TYPE AWE_FormLabel
          INTEGER, PRIVATE :: id
          CHARACTER(LEN=128) :: text = ""
        END TYPE
```

`id` is the internal dialog item identifier. AWE will automatically assign a unique value when the item is added to the dialog.

`text` is the text that will be displayed in the dialog box.

```
        USE AWE_Interfaces
        TYPE(AWE_FormDialog) :: dialog
        TYPE(AWE_FormLabel) :: label
        dialog%title = "title"
        Label%text = "label"
        CALL AWE_createDialog(dialog)
        CALL AWE_AddToDialog(label, dialog)
```

Dialog Combo Box

A combo box is a type of drop down menu with a title and selections. It is added to a dialog with an `AWE_FormComboBox` derived type:

```
        TYPE AWE_FormComboBox
          INTEGER, PRIVATE :: id
          CHARACTER (LEN=128) :: title = ""
          CHARACTER (LEN=128), ALLOCATABLE, DIMENSION(:) :: items
          INTEGER selected = 1
        END TYPE
```

`id` is the internal dialog item identifier. AWE will automatically assign a unique value when the item is added to the dialog.

`title` is the text that will be displayed to the left of the combo box.

`items` is the array of items text that will be displayed in the combo box menu.

`selected` contains the array index of the selected combo box item when the dialog is dismissed. The default is 1, but can be preset to any of the menu items by setting `selected` to the desired value before the item is added to the dialog.

```
        USE AWE_Interfaces
        TYPE(AWE_FormDialog) :: dialog
        TYPE(AWE_FormComboBox) :: comboBox
        dialog%title = "title"
        ALLOCATE(comboBox%items(3))
        comboBox%title = "label"
        comboBox%items(1) = "Item1"
        comboBox%items(2) = "Item2"
        comboBox%items(3) = "Item3"
        CALL AWE_createDialog(dialog)
        CALL AWE_AddToDialog(comboBox, dialog)
```

Dialog Check Box

A check box is a dialog item with a title and a checkable box. It is added to a dialog with an `AWE_FormCheckBox` derived type:

```
        TYPE AWE_FormCheckBox
          INTEGER, PRIVATE :: id
          CHARACTER (LEN=128) :: title = ""
          LOGICAL checked = .false.
        END TYPE
```

`id` is the internal dialog item identifier. AWE will automatically assign a unique value when the item is added to the dialog.

`title` is the text that will be displayed to the left of the check box.

`checked` indicates the state of the check box item when the dialog is dismissed. A value of `.true.` means the box is checked and a value of `.false.` means it is unchecked. The default is unchecked. To preset the state to checked, set `checked=.true.` before adding the item to the dialog.

```
USE AWE_Interfaces
TYPE(AWE_FormDialog) :: dialog
TYPE(AWE_FormCheckBox) :: checkBox
dialog%title = "title"
checkBox%title = "label"
CALL AWE_createDialog(dialog)
CALL AWE_AddToDialog(checkBox, dialog)
```

Dialog Text Edit Box

A text edit box is a dialog item with a title and a box where text can be entered. It is added to a dialog with an `AWE_FormLineEdit` derived type:

```
TYPE AWE_FormLineEdit
  INTEGER, PRIVATE :: id
  CHARACTER (LEN=128) :: title = ""
  CHARACTER (LEN=128) :: text = ""
  CHARACTER (LEN=128) :: placeholder = ""
END TYPE
```

`id` is the internal dialog item identifier. AWE will automatically assign a unique value when the item is added to the dialog.

`title` is the text that will be displayed to the left of the box.

`text` is the variable that any text entered by the user will be returned in. If this variable is initialized before adding the item to the dialog, it will be displayed in the dialog as the default value.

`placeholder` is text that will be displayed, grayed out, in the box. As soon as the cursor is placed in the text box, `placeholder` disappears. It is often useful as a prompt. If `text` is initialized, `placeholder` is ignored.

```
USE AWE_Interfaces
TYPE(AWE_FormDialog) :: dialog
TYPE(AWE_FormLineEdit) :: lineEdit
dialog%title = "title"
LineEdit%title = "label"
CALL AWE_createDialog(dialog)
CALL AWE_AddToDialog(LineEdit, dialog)
```

Dialog Radio Buttons

Radio buttons are a dialog item with a title and an array of buttons representing a single choice that may be made. They are added to a dialog with an `AWE_FormRadioButtons` derived type:

```
TYPE AWE_FormRadioButtons
  INTEGER, PRIVATE :: id
  CHARACTER (LEN=128) :: title = ""
  CHARACTER (LEN=128), ALLOCATABLE, DIMENSION(:) :: items
  INTEGER :: selected = 1
END TYPE
```

`id` is the internal dialog item identifier. AWE will automatically assign a unique value when the item is added to the dialog.

`title` is the text that will be displayed above the buttons.

`items` is an array of character variables containing the text for the buttons. There will be as many buttons as there are array elements.

`selected` returns the index of the selected button when the dialog is dismissed. It may be initialized to any value between `1` and the dimension extent of `items`. The default initialization is `1`, the first radio button.

```
USE AWE_Interfaces
TYPE(AWE_FormDialog) :: dialog
TYPE(AWE_FormRadioButtons) :: radioButtons
dialog%title = "title"
radioButtons%title = "Model"
allocate (radioButtons%items(3))
radioButtons%items(1) = "Small"
radioButtons%items(1) = "Medium"
radioButtons%items(1) = "Large"
CALL AWE_createDialog(dialog)
CALL AWE_AddToDialog(radioButtons, dialog)
```

Dialog File Selection Box

A file selection box item in a dialog consists of a title, a text box, and a file selection browse button. It is used to select files or directories. A file name can be entered directly in the text box. If the browse button is used, then the selection from the file selection dialog will be entered in the box automatically. A file selection box item is added to a dialog with an `AWE_FormFileDialog` derived type:

```
TYPE AWE_FormFileDialog
  INTEGER, PRIVATE :: id
  CHARACTER (LEN=128) :: title = ""
  CHARACTER (LEN=128) :: defaultDirectory = ""
  CHARACTER (LEN=128) :: placeholder = ""
  CHARACTER (LEN=128) :: text = ""
  INTEGER :: chooseMode = AWE_FormFileDialog_chooseExistingFile
END TYPE
```

`id` is the internal dialog item identifier. AWE will automatically assign a unique value when the item is added to the dialog.

`title` is the text that will be displayed to the left of the box.

`defaultDirectory` is used to set a default directory for the file selection dialog that is shown when the browse button is used.

`text` is the variable that any text entered by the user will be returned in. If this variable is initialized before adding the item to the dialog, it will be displayed in the dialog as the default value.

`placeholder` is text that will be displayed, grayed out, in the box. As soon as the cursor is placed in the text box, `placeholder` disappears. It is often useful as a prompt. If `text` is initialized, `placeholder` is ignored.

`chooseMode` determines the type of file selection dialog that is shown when the browse button is used. Three types are available:

```
AWE_FormFileDialog_chooseExistingFile
AWE_FormFileDialog_chooseDirectory
AWE_FormFileDialog_chooseNewFiles
```

Add a file selection box item to a dialog:

```
USE AWE_Interfaces
TYPE(AWE_FormDialog) :: dialog
TYPE(AWE_ FormFileDialog) :: fileDialog
dialog%title = "title"
fileDialog%title = "label"
CALL AWE_createDialog(dialog)
CALL AWE_AddToDialog(fileDialog, dialog)
```

Display an AWE Dialog

To display an AWE dialog, use the `AWE_showDialog` function:

```
INTEGER :: result
result = AWE_showDialog(dialog)
```

If `result==0`, the **Cancel** button was clicked. If `result==1`, the **OK** button was clicked.

TIMERS

A timer schedules a subroutine in your program to be executed at specified intervals. A derived type is used to define the timer interval and if the timer is to fire only once or be rescheduled after it goes off.

```
        TYPE AWE_Timer
          INTEGER, PRIVATE :: id
          INTEGER :: interval = 100
          LOGICAL :: singleShot = .false.
        END TYPE AWE_Timer
```

`id` is the internal timer item identifier. AWE will automatically assign a unique value when the timer is scheduled.

`interval` is the timer interval in milliseconds.

`singleShot` is used to indicate if the timer is be rescheduled after it goes off. The default is `.false.`.

To create a timer, declare an instance of `AWE_Timer`, supply the parameters you want to customize and call the `AWE_createTimer` subroutine with the `AWE_Timer` variable and the name of the subroutine to be called as arguments:

```
        USE AWE_Interfaces
        TYPE(AWE_Timer) :: timer
        EXTERNAL callBack
        CALL AWE_createTimer(timer, callBack)
```

If the main program has exited, the timer is started immediately. Otherwise, the timer is queued and will start as soon as the main program exits or a `STOP` statement is executed.

To cancel a timer, call the `AWE_destroyTimer` subroutine with the `AWE_Timer` variable used to create the timer:

```
        SUBROUTINE AWE_destroyTimer(timer)
          TYPE(AWE_Timer) :: timer
        END SUBROUTINE
```

CHAPTER 9

Interfacing With Other Languages

This chapter discusses interfacing Absoft Pro Fortran with the C Programming Language and assembly language, debugging programs, and profiling executables. Although Fortran programs can call C functions easily with just a `CALL` statement, the sections below should be read carefully to understand the differences between argument and data types.

INTERFACING WITH C

Absoft FORTRAN 77 is designed to be fully compatible with the Microsoft C/C++ compilers. The linker can be used to freely link C modules with FORTRAN main programs and vice versa. However, some precautions must be taken to ensure proper interfacing. Data types in arguments and results must be equivalent and some changes to the linking procedure must be made. All of these rules are detailed below. Be sure to follow them closely, or the results will be both unpredictable and invalid.

FORTRAN Data Types in C

Declarations for FORTRAN data types and the equivalent declarations in C are as follows:

FORTRAN	C
LOGICAL*1 l	unsigned char l;
LOGICAL*2 m	unsigned short m;
LOGICAL*4 n	unsigned long n;
CHARACTER*n c	char c[n];
INTEGER*1 i	char i;
INTEGER*2 j	short j;
INTEGER*4 k	int k;
	long k;
INTEGER*8 l	long long l;
REAL*4 a	float a;
REAL*8 d	double d;
COMPLEX*8 c	struct complx {
	float x;
	float y;
	};
	struct complx c;
COMPLEX*16 d	struct dcomp {
	double x;
	double y;
	};
	struct dcomp d;
RECORD ...	struct ...

1. On 64-bit systems, `long` is equivalent to `INTEGER*8`.

The storage allocated by the C language declarations will be identical to the storage allocated by the corresponding FORTRAN declaration. There are additional cautions when passing FORTRAN strings to C routines. See **Passing Strings to C** later in this chapter for more information.

Passing arguments Between C and FORTRAN

The Absoft FORTRAN 77 compiler uses the calling conventions of the C language. Therefore, a FORTRAN routine may be called from C without being declared in the C program and vice versa, if the routine returns all results in parameters. Otherwise, the function must be typed compatibly in both program units. In addition, care must be taken to pass compatible parameter types between the languages. Refer to the table in the previous section.

By default, Fortran external names are emitted with all lowercase letters and a single trailing underscore.

Reference Parameters

By default, all FORTRAN arguments to routines are passed by reference, which means pointers to the data are passed, not the actual data. Therefore, when calling a FORTRAN procedure from C, pointers to arguments must be passed rather than values. Both integer and floating point values may be passed by reference. Consider the following example:

```
SUBROUTINE sub(a_dummy,i_dummy)
REAL*4 a_dummy
INTEGER*4 i_dummy

WRITE (*,*) 'The arguments are ',a_dummy, ' and ',i_dummy
RETURN
END
```

The above subroutine is called from FORTRAN using the `CALL` statement:

```
a_actual = 3.3
i_actual = 9
CALL sub(a_actual, i_actual)
END
```

However, to call the subroutine from C, the function reference must explicitly pass pointers to the actual parameters as follows:

```
main()
{
float a_actual;
int i_actual;

      a_actual = 3.3;
      i_actual = 9;
      sub_(&a_actual,&i_actual);
}
```

Note that the values of the actual parameters may then be changed in the FORTRAN subroutine with an assignment statement or an I/O statement.

When calling a C function from FORTRAN with a reference parameter, the C parameters are declared as pointers to the data type and the FORTRAN parameters are passed normally:

```
PROGRAM convert_to_radians
WRITE (*,*) 'Enter degrees:'
READ (*,*) c
CALL c_rad(c)
WRITE (*,*) 'Equal to ',c,' radians'
END
```

```
void c_rad_(c)
float *c;
{
      float deg_to_rad = 3.14159/180.0;
      *c = *c * deg_to_rad;
}
```

Value Parameters

Absoft FORTRAN 77 provides the intrinsic function `[%]VAL` for passing value parameters. Although there is generally no need to pass a value directly to a FORTRAN procedure, these functions may be used to pass a value to a C function:

```
WRITE (*,*) 'Enter an integer:'
READ (*,*) i
CALL c_fun(VAL(i))
END
```

```
void c_fun_(i)
int i;
{
      printf ("%d is ", i);
      if (i % 2 == 0)
            printf ("even.\n");
      else
            printf ("odd.\n");
}
```

The value of `i` will be passed directly to `c_fun`, and will be left unaltered upon return.

Value parameters can be passed from C to FORTRAN with use of the `VALUE` statement. The arguments that are passed by value are declared as `VALUE`.

```
void c_fun()
{
void fortran_sub_();
int i;

      fortran_sub(i);
}


SUBROUTINE fortran_sub_(i)
VALUE i
```

```
      ...
      END
```

Note that C will pass all floating point data as double precision by default.

Indirection (the LOC Function)

The `[%]LOC` function is provided to give one level of indirection. The argument to `[%]LOC` must be a scalar name, an array name or the name of an external procedure. The function returns the address of its argument as a 32-bit integer.

This example illustrates the use of `LOC` to pass an array. Note that this is a one-dimensional array. Due to the different ordering used by C and FORTRAN for arrays, multi-dimensional arrays cannot be freely passed and indexed between the languages:

```
INTEGER*4 ia(10)

CALL c_fun(LOC(ia))
WRITE(*,*) ia
END
```

```
void c_fun_(i)
int *i[10];
{
int j;

      for(j=0; j<10; j++)
            (*i)[j] = j;
}
```

Function Results

In order to obtain function results in FORTRAN from C language functions and vice versa, the functions must be typed equivalently in both languages: either `INTEGER`, `REAL`, `DOUBLE PRECISION`, `RECORD`, or `POINTER`. All other data types must be returned in reference parameters. The following are examples of the passing of function results between FORTRAN and C.

A Call to C from FORTRAN

```
PROGRAM callc
INTEGER*4 cmax, A, B

WRITE (*,*) 'Enter two numbers:'
READ (*,*) A, B
WRITE (*,*) 'The largest of', A,' and', B,' is', cmax(A,B)
END
```

```
int cmax_(x,y)
int *x,*y;
{
      return( (*x >= *y) ? *x : *y );
}
```

A Call to FORTRAN from C

```
main()
{
float qt_to_liters_(), qt;

      printf ("Enter number of quarts:\n");
      scanf ("%f",&qt);
      printf("%f quarts = %f liters.\n", qt, qt_to_liters_(&qt));
}
```

```
REAL*4 FUNCTION qt_to_liters(q)
REAL*4 q

qt_to_liters = q * 0.9461
END
```

Passing Strings to C

FORTRAN strings are a sequence of characters padded with blanks out to their full fixed length, while strings in C are a sequence or array of characters terminated by a null character. Therefore, when passing FORTRAN strings to C routines, eliminate the extra blanks and terminate them with a null character. The following FORTRAN expression will properly pass the FORTRAN string `anystring` to the C routine `CPrint`:

```
PROGRAM cstringcall
character*255 string
string = 'Moscow on the Hudson'
CALL cprint(TRIM(string)//CHAR(0))
END
```

```
void cprint_(anystring)
char *anystring[];
{
      printf ("%s\n",anystring);
}
```

This example will neatly output "`Moscow on the Hudson`". If the `TRIM` function was not used, the same string would be printed, but followed by 235 blanks. If the `CHAR(0)` was omitted, C would print characters until a null character was encountered, whenever that might be.

In Absoft FORTRAN, the **-K** option may be used to allow embedded escape sequences in strings. The sequence for a null "`\0`" may be used to pass string constants as argument:

```
character*15 Fstring
CALL cprint("string constant\0") ! null terminated string

Fstring = "string constant"      ! blank padded string
CALL cprint(TRIM(Fstring)//"\0") ! append a null
```

You can also take advantage of the method Absoft Fortran compilers employ to pass the lengths of `CHARACTER` arguments on the stack. After the end of the formal argument list, the lengths of any `CHARACTER` arguments in the list are passed by value as 32-bit integers. The lengths are passed in the order they appear in the argument list and only `CHARACTER` argument lengths are passed. For example:

```
main()
{
int i = 1;
double d = 3.0;
void fortran_sub_();

      fortran_sub_(&i, "two", &d, 3);
}
```

```
SUBROUTINE fortran_sub(i, s, d)
INTEGER i
CHARACTER*(*) s
DOUBLE PRECISION d

PRINT *, i, s, d
END
```

Naming Conventions

Global names in FORTRAN include procedure names and COMMON block names, both of which are significant to 31 characters. All global procedure names are folded to lower case and have a single underscore ("_") appended unless the compiler character case and symbol decoration options are used. All COMMON block names are folded to lower case and have the characters "_C" prepended unless the compiler character case and symbol decoration options are used. All other symbols in FORTRAN are manipulated as addresses or offsets from local labels and are invisible to the linker.

Procedure Names

Names of functions and subroutines in Fortran programs will appear in the assembly language source output or object file records with their names folded to lower case and with a single underscore ("_") appended. Symbolic names in the C language are case sensitive, distinguishing between upper and lower case characters. To make FORTRAN code compatible with C, use the –YEXT_NAMES=ASIS and –YEXT_SFX="" options, the !DIR$ NAME directive, or the BIND attribute.

Accessing COMMON Blocks from C

COMMON block names are global symbols formed in Absoft Pro Fortran folding the name of the common block to lower case and then prepending the characters "_C" to the name of the COMMON block. The elements of the COMMON block can be accessed from the C language by declaring an external structure using this name.

For example:

```
COMMON /comm/ a,b,c
```

can be accessed with the global declaration:

```
extern struct {
            float a;
            float b;
            float c;
            } _Ccomm;
```

Declaring C Structures In FORTRAN

If there are equivalent data types in FORTRAN for all elements of a C structure, a `RECORD` can be declared in FORTRAN to match the structure in C:

```
C                              FORTRAN

struct str {                   STRUCTURE /str/
char c;                          CHARACTER c
long l;                          INTEGER*4 l
float f;                         REAL*4 f
double d;                        REAL*8 d
};                             END STRUCTURE
struct str my_struct;          RECORD /str/ my_struct
```

By default, the alignment of the C structure should be identical to the FORTRAN `RECORD`. Refer to the **Specification and DATA Statements** chapter of the *Absoft Fortran Language Reference Manual* for more information on the FORTRAN `RECORD` type.

Appendix A

Absoft Compiler Option Guide

This appendix summaries the options for the Absoft Fortran 90/95, FORTRAN 77, and C/C++ compilers. Refer to the chapter, **Using the Compilers** for detailed descriptions of the options

ABSOFT COMPILER OPTIONS

Option	***Effect***
-c	suppresses creation of an executable file — leaves compiled files in object code format
-cpp	always run C pre-processor regardless of file extension
-no-cpp	never run C pre-processor
-g	generates symbol information for Fx3™.
-I	used to supply a comma separated list of directory paths which are prepended to file names used with the `INCLUDE` statement
-O1	enables a group of basic optimizations which cause most code to run faster and enables optimizations that do not rearrange your program
-O2	enables a group of moderate optimizers that can rearrange the code generated for a program
-O3	enables a group of advanced optimizers including the IPA (Inter-Procedural Analyzer) linker that can substantially rearrange the code generated for a program
-O4	enables a group of advanced optimizers that can substantially rearrange the code generated for a program
-m64	directs the compiler to produce a 64-bit executable file
-o *name*	directs the compiler to produce an executable file called ***name*** where ***name*** is a Windows file name
-cpu:*type*	specifies the target processor where ***type*** is one of **486**, **p5**, **p6**, **p7**, **athlon**, or **host**

FLOATING POINT UNIT CONTROL OPTIONS

-OPT=*roundoff*	changes the rounding mode of the FPU to ***roundoff***.
-TENV=*exception*	enables FPU exception trapping for ***exception***.

FORTRAN 90/95 GENERAL OPTIONS

-en	causes the compiler to issue a warning whenever the source code contains an extension to the Fortran 90/95 standard.
-w	suppresses listing of all compile-time warning messages.
-Z*n*	suppresses messages by message number.
-q	suppresses any messages printed to standard output during the compilation process
-v	directs the compiler to print status information as the compilation process proceeds
-z*n*	suppresses messages by message level.
-dq	continue compilation if more than 100 errors are encountered.
-ea	causes the f95 compiler to abort the compilation process on the first error that it encounters.
-V	causes the f95 compiler to display its version number.
-eR	default recursion
-T*n*	changes the number of handles used internally by the compiler.
-t*n*	this option increases the default temporary string size to `1024x10`^{n} bytes.
-p path	specify module search path

FORTRAN 90/95 COMPATIBILITY OPTIONS

-x*directive*	disable compiler ***directive*** in the source file.
-i*n*	changes the default storage length of `INTEGER` data types to ***n*** bytes which can be either 2 or 8
-dp	causes variables declared in a `DOUBLE PRECISION` statement and constants specified with the `D` exponent to be converted to the default real kind.
-ej	causes all `DO` loops to be executed at least once, regardless of the initial value of the iteration count.
-N113	changes `REAL(KIND=4)` and `COMPLEX(KIND=4)` data types without explicit length declaration to `REAL(KIND=8)` and `COMPLEX(KIND=8)`
-s	allocate local variables statically
-Rb	enables array boundary checking
-Rc	enables array conformance checking
-Rs	enables substring checking
-Rp	enables null pointer checking
-YCFRL=	controls how `CHARACTER` arguments are passed to `FUNCTION` and `SUBROUTINE` programs
-YPEI=	allows Cray-style pointers to be equivalent to integers

FORTRAN 90/95 FORMAT OPTIONS

-YEXT_NAMES	controls the case of external procedure names
-YEXT_PFX=	establishes the prefix of external procedure names
-YEXT_SFX=	establishes the suffix of external procedure names

-f*form*	sets the ***form*** of the source file to `fixed`, `free`, `alt_fixed`.
-W*n*	sets the line length of source statements accepted by the compiler in Fixed-Form source format
-N26	force the compiler to consider the byte ordering of all unformatted files to be big-endian by default
-N27	force the compiler to consider the byte ordering of all unformatted files to be little-endian by default
-YCSLASH=	directs the compiler to transform certain escape sequences marked with a ‘\’ embedded in character constants
-YNDFP=	disallow the use of a ‘.’ as a structure field separator. The default value is 0 or false
-TMS7D	recognize Microsoft style compiler directives beginning with a ‘$’ in column 1

FORTRAN 90/95 MISCELLANEOUS OPTIONS

-YCOM_NAMES	controls the case of `COMMON` block names
-COM_PFX=	establishes the prefix of `COMMON` names
-YCM_SFX=	establishes the suffix of `COMMON` names
-h, -H, -U	loop unrolling control options
-safefp	disable floating point optimizations in numerically sensitive codes

FORTRAN 77 GENERAL OPTIONS

-w	suppresses listing of all compile-time warning messages
-N32	directs the compiler to issue a warning whenever the source code contains an extension to the ANSI FORTRAN 77 standard
-q	suppresses any messages printed to standard output during the compilation process
-v	directs the compiler to print status information as the compilation process proceeds
-C	generates code to check that array indexes are within array bounds — file names and source code line numbers will be displayed with all run time error messages
-x	replaces any occurrence of `X` or `D` in column one with a blank character: allows a restricted form of conditional compilation
-T	used to change the number of handles used internally by the compiler.
-t	modifies the default temporary string size to *`nn`* bytes from the default of 1024 bytes

FORTRAN 77 CONTROL OPTIONS

-D	used to define conditional compilation variables from the command line **(-D *name*[=value])** — if *`value`* is not present, the variable is assigned the value of 1

FORTRAN 77 COMPATIBILITY OPTIONS

-f	folds all symbolic names to lower case
-s	forces all program storage to be treated as static
-N109	folds all symbolic names to UPPER CASE
-i*n*	changes the default storage length of `INTEGER` data types to ***n*** bytes which can be either 2 or 8
-N113	changes `REAL` and `COMPLEX` data types without explicit length declaration to `DOUBLE PRECISION` and `DOUBLE COMPLEX`

FORTRAN 77 FORMAT OPTIONS

-8	directs the compiler to accept source code written in Fortran 90 Free Source Form
-V	directs the compiler to accept VAX Tab-Format source code
-W	directs the compiler to accept statements which extend beyond column 72 up to column 132
-N26	force the compiler to consider the byte ordering of all unformatted files to be big-endian by default
-N27	force the compiler to consider the byte ordering of all unformatted files to be little-endian by default
-K	directs the compiler to transform certain escape sequences marked with a ‘\’ embedded in character constants

FORTRAN 77 COMMON OPTIONS

-N22	append trailing underscore to `COMMON` block names
-N25	export `COMMON` block names in DLLs
-N110	do not mangle `COMMON` block names with leading “_c”

FORTRAN 77 OTHER OPTIONS

-h, -H, -U	loop unrolling control options

Appendix B

ASCII Table

ASCII codes 0 through 31 are control codes that may or may not have meaning on Windows. They are listed for historical reasons and may aid when porting code from other systems. Codes 128 through 255 are extensions to the 7-bit ASCII standard and the symbol displayed depends on the font being used; the symbols shown below are from the Times New Roman font. Most of these characters may be typed with keystrokes; use the `Key Caps` desk accessory to determine which keystrokes to use. The Dec, Oct, and Hex columns refer to the decimal, octal, and hexadecimal numerical representations.

Character	Dec	Oct	Hex	Description
NULL	0	000	00	null
SOH	1	001	01	start of heading
STX	2	002	02	start of text
ETX	3	003	03	end of text
ECT	4	004	04	end of trans
ENQ	5	005	05	enquiry
ACK	6	006	06	acknowledge
BEL	7	007	07	bell code
BS	8	010	08	back space
HT	9	011	09	horizontal tab
LF	10	012	0A	line feed
VT	11	013	0B	vertical tab
FF	12	014	0C	form feed
CR	13	015	0D	carriage return
SO	14	016	0E	shift out
SI	15	017	0F	shift in
DLE	16	020	10	data link escape
DC1	17	021	11	device control 1
DC2	18	022	12	device control 2
DC3	19	023	13	device control 3
DC4	20	024	14	device control 4
NAK	21	025	15	negative ack
SYN	22	026	16	synch idle
ETB	23	027	17	end of trans blk
CAN	24	030	18	cancel
EM	25	031	19	end of medium
SS	26	032	1A	special sequence
ESC	27	033	1B	escape
FS	28	034	1C	file separator
GS	29	035	1D	group separator
RS	30	036	1E	record separator
US	31	037	1F	unit separator
	32	040	20	space
!	33	041	21	exclamation
"	34	042	22	quotation mark
#	35	043	23	number sign
$	36	044	24	dollar sign
%	37	045	25	percent sign
&	38	046	26	ampersand
'	39	047	27	apostrophe
(	40	050	28	opening paren
)	41	051	29	closing paren
*	42	052	2A	asterisk
+	43	053	2B	plus
,	44	054	2C	comma
-	45	055	2D	minus
.	46	056	2E	period
/	47	057	2F	slash
0	48	060	30	zero
1	49	061	31	one
2	50	062	32	two
3	51	063	33	three
4	52	064	34	four
5	53	065	35	five
6	54	066	36	six
7	55	067	37	seven
8	56	070	38	eight
9	57	071	39	nine
:	58	072	3A	colon
;	59	073	3B	semicolon
<	60	074	3C	less than
=	61	075	3D	equal
>	62	076	3E	greater than
?	63	077	3F	question mark

Character	Dec	Oct	Hex	Description
@	64	100	40	commercial at
A	65	101	41	upper case letter
B	66	102	42	upper case letter
C	67	103	43	upper case letter
D	68	104	44	upper case letter
E	69	105	45	upper case letter
F	70	106	46	upper case letter
G	71	107	47	upper case letter
H	72	110	48	upper case letter
I	73	111	49	upper case letter
J	74	112	4A	upper case letter
K	75	113	4B	upper case letter
L	76	114	4C	upper case letter
M	77	115	4D	upper case letter
N	78	116	4E	upper case letter
O	79	117	4F	upper case letter
P	80	120	50	upper case letter
Q	81	121	51	upper case letter
R	82	122	52	upper case letter
S	83	123	53	upper case letter
T	84	124	54	upper case letter
U	85	125	55	upper case letter
V	86	126	56	upper case letter
W	87	127	57	upper case letter
X	88	130	58	upper case letter
Y	89	131	59	upper case letter
Z	90	132	5A	upper case letter
[	91	133	5B	opening bracket
\	92	134	5C	back slash
]	93	135	5D	closing bracket
^	94	136	5E	circumflex
_	95	137	5F	underscore
`	96	140	60	grave accent
a	97	141	61	lower case letter
b	98	142	62	lower case letter
c	99	143	63	lower case letter
d	100	144	64	lower case letter
e	101	145	65	lower case letter
f	102	146	66	lower case letter
g	103	147	67	lower case letter
h	104	140	68	lower case letter
i	105	151	69	lower case letter
j	106	152	6A	lower case letter
k	107	153	6B	lower case letter
l	108	154	6C	lower case letter
m	109	155	6D	lower case letter
n	110	156	6E	lower case letter
o	111	157	6F	lower case letter
p	112	160	70	lower case letter
q	113	161	71	lower case letter
r	114	162	72	lower case letter
s	115	163	73	lower case letter
t	116	164	74	lower case letter
u	117	165	75	lower case letter
v	118	166	76	lower case letter
w	119	167	77	lower case letter
x	120	170	78	lower case letter
y	121	171	79	lower case letter
z	122	172	7A	lower case letter
{	123	173	7B	opening brace
|	124	174	7C	vertical bar
}	125	175	7D	closing brace
~	126	176	7E	tilde
	127	177	7F	delete
□	128	200	80	
□	129	201	81	
‚	130	202	82	
ƒ	131	203	83	
„	132	204	84	
…	133	205	85	
†	134	206	86	
‡	135	207	87	
ˆ	136	210	88	
‰	137	211	89	
Š	138	212	8A	
‹	139	213	8B	
Œ	140	214	8C	
□	141	215	8D	
□	142	216	8E	
□	143	217	8F	
□	144	220	90	
‘	145	221	91	
’	146	222	92	
“	147	223	93	
”	148	224	94	
•	149	225	95	
–	150	226	96	
—	151	227	97	
˜	152	230	98	
™	153	231	99	
š	154	232	9A	
›	155	233	9B	
œ	156	234	9C	
□	157	235	9D	
□	158	236	9E	
Ÿ	159	237	9F	
	160	240	A0	
¡	161	241	A1	
¢	162	242	A2	
£	163	243	A3	
¤	164	244	A4	
¥	165	245	A5	
¦	166	246	A6	
§	167	247	A7	
¨	168	250	A8	
©	169	251	A9	
ª	170	252	AA	
«	171	253	AB	
¬	172	254	AC	
-	173	255	AD	
®	174	256	AE	
¯	175	257	AF	
°	176	260	B0	
±	177	261	B1	
²	178	262	B2	
³	179	263	B3	
´	180	264	B4	
µ	181	265	B5	

Character	Dec	Oct	Hex
¶	182	266	B6
·	183	267	B7
¸	184	270	B8
¹	185	271	B9
º	186	272	BA
»	187	273	BB
¼	188	274	BC
½	189	275	BD
¾	190	276	BE
¿	191	277	BF
À	192	300	C0
Á	193	301	C1
Â	194	302	C2
Ã	195	303	C3
Ä	196	304	C4
Å	197	305	C5
Æ	198	306	C6
Ç	199	307	C7
È	200	310	C8
É	201	311	C9
Ê	202	312	CA
Ë	203	313	CB
Ì	204	314	CC
Í	205	315	CD
Î	206	316	CE
Ï	207	317	CF
Ð	208	320	D0
Ñ	209	321	D1
Ò	210	322	D2
Ó	211	323	D3
Ô	212	324	D4
Õ	213	325	D5
Ö	214	326	D6
×	215	327	D7
Ø	216	330	D8
Ù	217	331	D9
Ú	218	332	DA
Û	219	333	DB
Ü	220	334	DC
Ý	221	335	DD
Þ	222	336	DE
ß	223	337	DF
à	224	340	E0
á	225	341	E1
â	226	342	E2
ã	227	343	E3
ä	228	344	E4
å	229	345	E5
æ	230	346	E6
ç	231	347	E7
è	232	350	E8
é	233	351	E9
ê	234	352	EA
ë	235	353	EB
ì	236	354	EC
í	237	355	ED
î	238	356	EE
ï	239	357	EF
ð	240	360	F0
ñ	241	361	F1
ò	242	362	F2
ó	243	363	F3
ô	244	364	F4
õ	245	365	F5
ö	246	366	F6
÷	247	367	F7
ø	248	370	F8
ù	249	371	F9
ú	250	372	FA
û	251	373	FB
ü	252	374	FC
ý	253	375	FD
þ	254	376	FE
ÿ	255	377	FF

Appendix C

Bibliography

REFERENCES ON THE FORTRAN 90/95 LANGUAGE

Michael Metcalf and John Reid, *FORTRAN 90/95 explained*, Oxford University Press (1996)

Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, and Brian T. Smith, *Fortran Top 90*, Unicomp, Inc, Albuquerque, NM (1994)

American National Standard Programming Language Fortran 90, X3.198-1992, ANSI, 1430 Broadway, New York, N.Y. 10018

REFERENCES ON THE FORTRAN 77 LANGUAGE

These books and manuals are useful references for the FORTRAN language and the floating point math format used by Absoft Fortran 77 on Windows.

Page, Didday, and Alpert, *FORTRAN 77 for Humans*, West Publishing Company (1983) **Highly recommended for beginners**

Kruger, Anton, *Efficient FORTRAN Programming*, John Wiley & Sons, Inc. (1990) **Highly recommended for beginners**

Loren P. Meissner and Elliot I. Organick, *FORTRAN 77*, Addison-Wesley Publishing Company (1980)

Harry Katzan, Jr., *FORTRAN 77*, Van Nostrand Reinhold Company (1978)

J.N.P. Hume and R.C. Holt, *Programming FORTRAN 77*, Reston Publishing Company, Inc. (1979)

Harice L. Seeds, *FORTRAN IV*, John Wiley & Sons (1975)

Jehosua Friedmann, Philip Greenberg, and Alan M. Hoffberg, *FORTRAN IV, A Self-Teaching Guide*, John Wiley & Sons, Inc. (1975)

James S. Coan, *Basic FORTRAN*, Hayden Book Company (1980)

Brian W. Kernighan and P.J. Plauger, *Software Tools*, Addison-Wesley Publishing Company (1976)

Brian W. Kernighan and P.J. Plauger, *The Elements of Programming Style,* McGraw-Hill Book Company (1978)

American National Standard Programming Language FORTRAN, X3.9-1978, ANSI, 1430 Broadway, New York, N.Y. 10018

COMPUTER, *A Proposed Standard for Binary Floating-Point Arithmetic*, Draft 8.0 of IEEE Task P754, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720 (1981)

M. Abramowitz and I.E. Stegun, *Handbook of Mathematical Functions*, U.S. Department of Commerce, National Bureau of Standards (1972)

Fortran Forum, Association for Computing Machinery. Phone: 1-212-869-7440.

Fortran Journal, Fortran Users Group. Phone: 1-714-441-2022.

REFERENCES ON THE C/C++ PROGRAMMING LANGUAGES

Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice Hall (1988)

Samuel P Harbison and Guy L. Steele Jr., *C: A Reference Manual*, Prentice Hall (1987)

Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley (1991)

Margaret A. Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley (1990)

Stanley B. Lippman, *C++ Primer,* Addison-Wesley (1991)

Ira Pohl, *C++ for C Programmers*, Benjamin/Cummings (1989)

James T Smith, *C++ For Scientists and Engineers*, McGraw-Hill (1991)

Scott Meyers, *Effective C++*, Addison-Wesley (1992)

American National Standard Programming Language C, X3.159-1989, ANSI, 1430 Broadway, New York, N.Y. 10018

REFERENCES ON WINDOWS PROGRAMMING

These books are suggested reading for learning how to program in the Win32 API for Windows. Most of these books are available in book stores.

Microsoft Win32 Programmer's Reference, Volumes 1-5, Mircosoft Press (1993)

Charles Petzold, *Programming Windows 3.1*, Mircosoft Press (1992)

Jerry Richter, *Advanced Windows, The Developer's Guide to the Win32 API for Windows NT 3.5 and Windows 95,* Mircosoft Press (1995)

Appendix D

AWE RGB Colors

AWE plots use RGB colors extensively. Although default values are always supplied, you can replace them with any RGB value you want. This appendix lists a number of predefined values supplied in the `AWE_Interfaces` module that are available to you by simply using their symbolic names.

```
INTEGER, PARAMETER :: AWE_maroon = z'800000'
INTEGER, PARAMETER :: AWE_dark_red = z'8B0000'
INTEGER, PARAMETER :: AWE_brown = z'A52A2A'
INTEGER, PARAMETER :: AWE_firebrick = z'B22222'
INTEGER, PARAMETER :: AWE_crimson = z'DC143C'
INTEGER, PARAMETER :: AWE_red = z'FF0000'
INTEGER, PARAMETER :: AWE_tomato = z'FF6347'
INTEGER, PARAMETER :: AWE_coral = z'FF7F50'
INTEGER, PARAMETER :: AWE_indian_red = z'CD5C5C'
INTEGER, PARAMETER :: AWE_light_coral = z'F08080'
INTEGER, PARAMETER :: AWE_dark_salmon = z'E9967A'
INTEGER, PARAMETER :: AWE_salmon = z'FA8072'
INTEGER, PARAMETER :: AWE_light_salmon = z'FFA07A'
INTEGER, PARAMETER :: AWE_orange_red = z'FF4500'
INTEGER, PARAMETER :: AWE_dark_orange = z'FF8C00'
INTEGER, PARAMETER :: AWE_orange = z'FFA500'
INTEGER, PARAMETER :: AWE_gold = z'FFD700'
INTEGER, PARAMETER :: AWE_dark_golden_rod = z'B8860B'
INTEGER, PARAMETER :: AWE_golden_rod = z'DAA520'
INTEGER, PARAMETER :: AWE_pale_golden_rod = z'EEE8AA'
INTEGER, PARAMETER :: AWE_dark_khaki = z'BDB76B'
INTEGER, PARAMETER :: AWE_khaki = z'F0E68C'
INTEGER, PARAMETER :: AWE_olive = z'808000'
INTEGER, PARAMETER :: AWE_yellow = z'FFFF00'
INTEGER, PARAMETER :: AWE_yellow_green = z'9ACD32'
INTEGER, PARAMETER :: AWE_dark_olive_green = z'556B2F'
INTEGER, PARAMETER :: AWE_olive_drab = z'6B8E23'
INTEGER, PARAMETER :: AWE_lawn_green = z'7CFC00'
INTEGER, PARAMETER :: AWE_chartreuse = z'7FFF00'
INTEGER, PARAMETER :: AWE_green_yellow = z'ADFF2F'
INTEGER, PARAMETER :: AWE_dark_green = z'006400'
INTEGER, PARAMETER :: AWE_green = z'008000'
INTEGER, PARAMETER :: AWE_forest_green = z'228B22'
INTEGER, PARAMETER :: AWE_lime = z'00FF00'
INTEGER, PARAMETER :: AWE_lime_green = z'32CD32'
INTEGER, PARAMETER :: AWE_light_green = z'90EE90'
INTEGER, PARAMETER :: AWE_pale_green = z'98FB98'
INTEGER, PARAMETER :: AWE_dark_sea_green = z'8FBC8F'
INTEGER, PARAMETER :: AWE_medium_spring_green = z'00FA9A'
INTEGER, PARAMETER :: AWE_spring_green = z'00FF7F'
INTEGER, PARAMETER :: AWE_sea_green = z'2E8B57'
INTEGER, PARAMETER :: AWE_medium_aqua_marine = z'66CDAA'
INTEGER, PARAMETER :: AWE_medium_sea_green = z'3CB371'
INTEGER, PARAMETER :: AWE_light_sea_green = z'20B2AA'
INTEGER, PARAMETER :: AWE_dark_slate_gray = z'2F4F4F'
INTEGER, PARAMETER :: AWE_teal = z'008080'
```

```
      INTEGER, PARAMETER :: AWE_dark_cyan = z'008B8B'
      INTEGER, PARAMETER :: AWE_aqua = z'00FFFF'
      INTEGER, PARAMETER :: AWE_cyan = z'00FFFF'
      INTEGER, PARAMETER :: AWE_light_cyan = z'E0FFFF'
      INTEGER, PARAMETER :: AWE_dark_turquoise = z'00CED1'
      INTEGER, PARAMETER :: AWE_turquoise = z'FF40E0D0'
      INTEGER, PARAMETER :: AWE_medium_turquoise = z'48D1CC'
      INTEGER, PARAMETER :: AWE_pale_turquoise = z'AFEEEE'
      INTEGER, PARAMETER :: AWE_aqua_marine = z'7FFFD4'
      INTEGER, PARAMETER :: AWE_powder_blue = z'B0E0E6'
      INTEGER, PARAMETER :: AWE_cadet_blue = z'5F9EA0'
      INTEGER, PARAMETER :: AWE_steel_blue = z'4682B4'
      INTEGER, PARAMETER :: AWE_corn_flower_blue = z'6495ED'
      INTEGER, PARAMETER :: AWE_deep_sky_blue = z'00BFFF'
      INTEGER, PARAMETER :: AWE_dodger_blue = z'1E90FF'
      INTEGER, PARAMETER :: AWE_light_blue = z'ADD8E6'
      INTEGER, PARAMETER :: AWE_sky_blue = z'87CEEB'
      INTEGER, PARAMETER :: AWE_light_sky_blue = z'87CEFA'
      INTEGER, PARAMETER :: AWE_midnight_blue = z'191970'
      INTEGER, PARAMETER :: AWE_navy = z'000080'
      INTEGER, PARAMETER :: AWE_dark_blue = z'00008B'
      INTEGER, PARAMETER :: AWE_medium_blue = z'0000CD'
      INTEGER, PARAMETER :: AWE_blue = z'0000FF'
      INTEGER, PARAMETER :: AWE_royal_blue = z'4169E1'
      INTEGER, PARAMETER :: AWE_blue_violet = z'8A2BE2'
      INTEGER, PARAMETER :: AWE_indigo = z'4B0082'
      INTEGER, PARAMETER :: AWE_dark_slate_blue = z'483D8B'
      INTEGER, PARAMETER :: AWE_slate_blue = z'6A5ACD'
      INTEGER, PARAMETER :: AWE_medium_slate_blue = z'7B68EE'
      INTEGER, PARAMETER :: AWE_medium_purple = z'9370DB'
      INTEGER, PARAMETER :: AWE_dark_magenta = z'8B008B'
      INTEGER, PARAMETER :: AWE_dark_violet = z'9400D3'
      INTEGER, PARAMETER :: AWE_dark_orchid = z'9932CC'
      INTEGER, PARAMETER :: AWE_medium_orchid = z'BA55D3'
      INTEGER, PARAMETER :: AWE_purple = z'800080'
      INTEGER, PARAMETER :: AWE_thistle = z'D8BFD8'
      INTEGER, PARAMETER :: AWE_plum = z'DDA0DD'
      INTEGER, PARAMETER :: AWE_violet = z'EE82EE'
      INTEGER, PARAMETER :: AWE_magenta = z'FF00FF'
      INTEGER, PARAMETER :: AWE_orchid = z'DA70D6'
      INTEGER, PARAMETER :: AWE_medium_violet_red = z'C71585'
      INTEGER, PARAMETER :: AWE_pale_violet_red = z'DB7093'
      INTEGER, PARAMETER :: AWE_deep_pink = z'FF1493'
      INTEGER, PARAMETER :: AWE_hot_pink = z'FF69B4'
      INTEGER, PARAMETER :: AWE_light_pink = z'FFB6C1'
      INTEGER, PARAMETER :: AWE_pink = z'FFC0CB'
      INTEGER, PARAMETER :: AWE_antique_white = z'FAEBD7'
      INTEGER, PARAMETER :: AWE_beige = z'F5F5DC'
      INTEGER, PARAMETER :: AWE_bisque = z'FFE4C4'
      INTEGER, PARAMETER :: AWE_blanched_almond = z'FFEBCD'
      INTEGER, PARAMETER :: AWE_wheat = z'F5DEB3'
      INTEGER, PARAMETER :: AWE_corn_silk = z'FFF8DC'
      INTEGER, PARAMETER :: AWE_lemon_chiffon = z'FFFACD'
      INTEGER, PARAMETER :: AWE_light_golden_rod_yellow = z'FAFAD2'
      INTEGER, PARAMETER :: AWE_light_yellow = z'FFFFE0'
      INTEGER, PARAMETER :: AWE_saddle_brown = z'8B4513'
      INTEGER, PARAMETER :: AWE_sienna = z'A0522D'
      INTEGER, PARAMETER :: AWE_chocolate = z'D2691E'
      INTEGER, PARAMETER :: AWE_peru = z'CD853F'
      INTEGER, PARAMETER :: AWE_sandy_brown = z'F4A460'
      INTEGER, PARAMETER :: AWE_burly_wood = z'DEB887'
```

```
INTEGER, PARAMETER :: AWE_tan = z'D2B48C'
INTEGER, PARAMETER :: AWE_rosy_brown = z'BC8F8F'
INTEGER, PARAMETER :: AWE_moccasin = z'FFE4B5'
INTEGER, PARAMETER :: AWE_navajo_white = z'FFDEAD'
INTEGER, PARAMETER :: AWE_peach_puff = z'FFDAB9'
INTEGER, PARAMETER :: AWE_misty_rose = z'FFE4E1'
INTEGER, PARAMETER :: AWE_lavender_blush = z'FFF0F5'
INTEGER, PARAMETER :: AWE_linen = z'FAF0E6'
INTEGER, PARAMETER :: AWE_old_lace = z'FDF5E6'
INTEGER, PARAMETER :: AWE_papaya_whip = z'FFEFD5'
INTEGER, PARAMETER :: AWE_sea_shell = z'FFF5EE'
INTEGER, PARAMETER :: AWE_mint_cream = z'F5FFFA'
INTEGER, PARAMETER :: AWE_slate_gray = z'708090'
INTEGER, PARAMETER :: AWE_light_slate_gray = z'778899'
INTEGER, PARAMETER :: AWE_light_steel_blue = z'B0C4DE'
INTEGER, PARAMETER :: AWE_lavender = z'E6E6FA'
INTEGER, PARAMETER :: AWE_floral_white = z'FFFAF0'
INTEGER, PARAMETER :: AWE_alice_blue = z'F0F8FF'
INTEGER, PARAMETER :: AWE_ghost_white = z'F8F8FF'
INTEGER, PARAMETER :: AWE_honeydew = z'F0FFF0'
INTEGER, PARAMETER :: AWE_ivory = z'FFFFF0'
INTEGER, PARAMETER :: AWE_azure = z'F0FFFF'
INTEGER, PARAMETER :: AWE_snow = z'FFFAFA'
INTEGER, PARAMETER :: AWE_black = z'000000'
INTEGER, PARAMETER :: AWE_dim_gray = z'696969'
INTEGER, PARAMETER :: AWE_gray = z'808080'
INTEGER, PARAMETER :: AWE_dark_gray = z'A9A9A9'
INTEGER, PARAMETER :: AWE_silver = z'C0C0C0'
INTEGER, PARAMETER :: AWE_light_gray = z'D3D3D3'
INTEGER, PARAMETER :: AWE_gainsboro = z'DCDCDC'
INTEGER, PARAMETER :: AWE_white_smoke = z'F5F5F5'
INTEGER, PARAMETER :: AWE_white = z'FFFFFF'
```

Appendix E

speed_math option

The **-speed_math=*n*** option enables aggressive math optimizations that may improve performance at the expense of accuracy. Valid arguments for ***n*** are 0-11. The following table describes the effect of each level:

n	effect
0	enable wrap around optimization
1	allow relational operator folding; may cause signed integer overflow
2	enable partial redundancy elimination for loads and stores
3	enable memory optimization for functions without aliased arrays
4	inline NINT and related intrinsics with limited-domain algorithm use fast_powf in libm instead of powf
5	use multiplication and square root for exp() where faster
6	allow optimizations that reassociate floating point operators
7	*see notes below*
8	allow use of reciprocal instruction; convert a/b to a*(1/b)
9	use fast algorithms with limited domains for complex norm and divide use x*rsqrt(x) for sqrt(x) on machines where faster dead casgn function elimination
10	use AMD ACML library if applicable
11	allow relational operator folding; may cause unsigned integer overflow use IEEE rounding instead of Fortran rounding for NINT intrinsics use IEEE rounding instead of Fortran rounding for ANINT intrinsics

NOTES:

A. Departure from strict rounding is applied at 3 levels: level 1 is applied at ***n=5***, level 2 is applied at ***n=7***, and level 3 is applied at ***n=10***.
B. Conformance to IEEE-754 arithmetic rules is relaxed at 2 levels: level 1 is applied at ***n=6***, level 2 is applied at ***n=10***
C. At ***n=10***, the loop unrolling constraints are modified: loop size is increased to 7000, limit is increased to 9, minimum iteration is decreased to 200.

Appendix F

Technical Support

The Absoft Technical Support Group will provide technical assistance to all registered users. They will *not* answer general questions about operating systems, operating system interfaces, graphical user interfaces, or teach the FORTRAN language. For further help on these subjects, please consult this manual and any of the books and manuals listed in the bibliography.

Before contacting Technical Support, please study this manual and the *Fortran User Guide* to make sure your problem is not covered here. Specifically, look at the chapter **Using The Compilers** in the *Fortran User Guide* and the **Error Messages** appendices of both manuals. To help Technical Support provide a quick and accurate solution to your problem, please include the following information in any correspondence or have it available when calling.

Product Information:

Name of product .
Version number.
Serial number.
Version number of the operating system.

System Configuration:

Hardware configuration (hard drive, etc.).
System software release (i.e. 4.0, 3.5, etc).
Any software or hardware modifications to your system.

Problem Description:

What happens?
When does it occur?
Provide a small (20 line) reproducible program or step-by-step example if possible.

Contacting Technical Support:

Address:	Absoft Corporation Attn: Technical Support 2075 West Big Beaver Road, Suite 250 Troy, MI 48084

Technical Support:	(248) 220-1191	9am - 3pm EST
FAX	(248) 220-1194	24 Hours
email	support@absoft.com	24 Hours
World Wide Web	http://www.absoft.com	

APPENDIX G

Visual Basic DLLs

This appendix describes how to use the Absoft Fortran compiler and linker to create DLLs that are callable from Microsoft Visual Basic™. A DLL, Dynamic Link Library, is a library of routines that are callable at runtime from any application that conforms to the Windows API. The following discussion applies primarily to 32-bit DLLs. See the section at the end for 64-bit DLL notes.

CREATING THE FORTRAN DLL

Write your FORTRAN source code in the usual manner, declaring the program unit as either a subroutine or a function. Insert the additional keyword `STDCALL` before each `SUBROUTINE` or `FUNCTION` keyword. Adhere to the normal FORTRAN `CALL/RETURN` sequences and argument passing rules. For example, consider the following subroutine

```
subroutine DegreeSin(input, output)

implicit none

stdcall DegreeSin
double precision input
double precision output

output = dsind(input)

return

end
```

This subroutine simply computes the double precision sin in degrees of the input argument and returns the result in the output argument. If you are using the Developer Tools, set the target type to DLL in the Project menu with the Default Tool Options menu command. If you are compiling from the command line, use the –dll option which instructs the compiler to produce a DLL. You must use the option the compiler option preserve the case of external names (-YEXT_NAMES=ASIS) so that the case of the routine name is preserved. You should also use the use the option to prevent the compiler from automatically appending an underscore to external names (-YEXT_SFX=””) or add an underscore in your Visual Basic code.

```
f95 –dll -YEXT_NAMES=ASIS –YEXT_SFX=”” DegreeSin.f
```

CREATING THE VISUAL BASIC CODE

Note: The default argument passing method for Visual Basic .NET is different from what was in previous versions. Arguments are now passed *ByVal* by default. Note below that *ByRef* must be specified.

Complete documentation about calling Fortran DLLs from Microsoft Visual Basic can be found in the *Microsoft Visual Basic Programmer's Guide*. This section will describe the basics of referencing a Fortran Subroutine or Function.

The first step is to declare the Fortran subprogram in your Basic program. The declaration for the subroutine discussed in the previous section would be:

```
Imports System.Runtime.InteropServices
Imports System.Text


Public Class Form1

...

<DllImport("c:\...\DegreeSin.dll")> _
    Public Shared Sub DegreeSin@8( _
    ByRef inval As System.Double, ByRef outval As System.Double)

End Sub
```

The string following the `Lib` keyword should be the path to the DLL.

The actual reference to the Fortran subroutine is:

```
Dim inval As Double
Dim outval As Double
  .
  .
  .
Call DegreeSin(inval, outval)
```

PASSING VISUAL BASIC ARRAYS TO A DLL

The Microsoft Visual Basic Programmer's Guide describes how to pass arrays to DLLs. Visual Basic passes entire arrays using OLE Automation argument protocols. Absoft F77/F95/C/C++ expect CDECL arguments. The Visual Basic manual section explains how to pass the address of the first argument of the array. Basically:

```
   Declare ... lParam as Any

   Dim array(100)

   Call DLL(array(0))
```

PASSING VISUAL BASIC STRINGS TO A DLL

Visual Basic strings are maintained in a data structured referred to as a BSTR which is not compatible with other languages in a DLL. However, you can pass a null terminated, C programming language string in Visual Basic. The declaration and usage would take the following form:

```
Imports System.Runtime.InteropServices
Imports System.Text


Public Class Form1

...

<DllImport("c:\test\vbstring\test.dll")> _
    Public Shared Sub getString@4(ByVal text As System.String)
End Sub

Private Sub Button1_Click()
Dim text As String
text = "hello, world"
Call getString(text)
End Sub
```

The key is to pass the string by value (ByVal).

At the FORTRAN end:

```
      subroutine getString(p_theString)

      use windows

      implicit none

      stdcall getString

C FORTRAN expects the string lengths to be passed
C after the formal argument list as values. Since
C Visual Basic does not do this and because this is
C a STDCALL procedure, we need to accept the argument
C as a general pointer to a string passed by value.

      integer p_theString; value p_theString

C Local variables
      character*1024 string ! longer than expected pointer (p_string,
string)
      pointer (p_string, string)
      character*1024 temp, title
      integer length, p_temp, p_title, i

C The Visual Basic string is passed as a null terminated
C C string. The first thing we have to do is find the
C null to determine the length of the string.
      p_string = p_theString
      length = index(string, char(0))
      if ((length .lt. 2) .or. (length .gt. 1024)) return

C Copy the string to a local (and safe) variable.
```

```
C Initialize the message box title string
      temp = string(1:length-1)
      title = "FORTRAN DLL"

C Null terminate the strings for the call to "Message Box".
      temp = trim(temp)//char(0)
      title = trim(title)//char(0)

C Create pointers to the strings so they can be
C passed by value to the Win32 API function "MessageBox".
      p_temp = loc(temp)
      p_title = loc(title)
      i = MessageBox(0, p_temp, p_title, MB_OK)


C Null terminate the string so that we can get the length
      temp ="Goodbye World"//char(0)

C Get the length of the string
      length = index(temp, char(0))

C Copy the string character by character
       do i=1,length-1
      string(i:i)=temp(i:i)
      end do
      string(length:length)=char(0)

      end
```

Use the following commands to build the DLL (assuming the FORTRAN source file is "test.f"):

```
f95 -dll -YEXT_NAMES=ASIS –YEXT_SFX=”” test.f
```

64-BIT DLL NOTES

The `STDCALL` declaration should be omitted from the Fortran source code. 64-bit code does not use the `STDCALL` call/return convention.

Because the `STDCALL` call/return convention and its associated name mangling is not used for 64-bit code, the Visual Basic declarations for the above examples should be:

```
<DllImport("c:\...\DegreeSin.dll")> _
    Public Shared Sub DegreeSin( _
    ByRef inval As System.Double, ByRef outval As System.Double)

<DllImport("c:\test\vbstring\test.dll")> _
    Public Shared Sub getString(ByVal text As System.String)
```