CHAPTER 1 INTRODUCTION ......................................................................................................................... 1

Introduction to Absoft Pro Fortran .................................................................................................................... 1
Absoft Fortran 90/95 ........................................................................................................................................ 1
Absoft FORTRAN 77 .................................................................................................................................... 1
Absoft C/C++ ................................................................................................................................................ 2

Conventions Used in this Manual ................................................................................................................... 2

Road Maps ...................................................................................................................................................... 2
Fortran Road Maps ....................................................................................................................................... 2

Year 2000 Problem ...................................................................................................................................... 3
Fortran 90/95 DATE_AND_TIME Subroutine .......................................................................................... 4
Unix Compatibility Library .......................................................................................................................... 4

CHAPTER 2 USING THE ABSOFT EDITOR ................................................................................................. 5

The Absoft Editor .............................................................................................................................................. 5
Text Selection ............................................................................................................................................... 6
Using Compilers .......................................................................................................................................... 6
Pop-up menus .............................................................................................................................................. 6

Creating New Source Files .......................................................................................................................... 7
Manipulating Windows ................................................................................................................................ 8

Using the Editor Menus ................................................................................................................................. 8
Application Menu ....................................................................................................................................... 8
Preferences .................................................................................................................................................. 9
Format ......................................................................................................................................................... 9
Environment ............................................................................................................................................... 11
Tools ............................................................................................................................................................ 13
File Menu ................................................................................................................................................... 13
New…((FILE) ......................................................................................................................................... 13
Open…(FILE) ......................................................................................................................................... 14
Open Recent ............................................................................................................................................. 14
Open Selected ......................................................................................................................................... 14
Open Complement .................................................................................................................................. 14
Close ((FILE) ......................................................................................................................................... 14
Close All .................................................................................................................................................. 14
Save (FILE) ............................................................................................................................................. 14
Save As ..................................................................................................................................................... 14
Save All ................................................................................................................................................... 15
Revert ....................................................................................................................................................... 15
Print Setup…(FILE) ............................................................................................................................... 15
Print…(FILE) ......................................................................................................................................... 15
Page Setup ............................................................................................................................................... 15
Edit Menu .................................................................................................................................................. 16
CHAPTER 3 USING THE COMPILERS .................................................. 25

Compiling Programs .................................................................................. 25

Using the Command Line .......................................................................... 25

File Name Conventions ............................................................................... 26

Fortran User Guide
Table of Contents

Using the Absoft Developer Tools Interface

Working with Projects ................................................................. 27
Options Dialog ............................................................................. 28
  Target Tab – General Options .................................................. 29
    Target Type ........................................................................... 29
    MRWE Events (-N9) ................................................................. 30
    Link Large Data Stubs ............................................................ 30
    Use 32-bit branches (-N11) ....................................................... 30
    Runtime Stack Trace (-et) ......................................................... 30
  Options ..................................................................................... 31
  Object File(s) Directory ............................................................ 32
  Target Directory ........................................................................ 32
  Target Filename .......................................................................... 32
  Target Tab – FPU Options ....................................................... 33
    FPU Rounding Mode ............................................................... 33
    FPU Exception Handling ....................................................... 34
    Don’t generate FMA instructions (-Q51) ................................. 34
  Other Target Options ............................................................... 34
    Generate Assembly Language (-S) .......................................... 35
    Generate Relocatable Object (-c) ............................................. 35
    Library Specification (-l) ......................................................... 35
    Library Path Specification (-L) ............................................... 35

Absoft Developer Tools Interface

Application Menu ....................................................................... 36
  About Tools ............................................................................. 36
  Preferences .............................................................................. 36
  Hide Tools ............................................................................... 37
  Quit Tools ............................................................................... 37
File Menu .................................................................................. 37
  New ....................................................................................... 37
  Open ..................................................................................... 37
  Open Recent ........................................................................... 37
  Close .................................................................................... 38
  Save ...................................................................................... 38
  Save As .................................................................................. 38
  Revert To Saved ...................................................................... 38
  Page Setup .............................................................................. 38
  Print ...................................................................................... 38
  New File ................................................................................ 38
Edit Menu .................................................................................. 38
  Undo ..................................................................................... 38
  Redo ..................................................................................... 38
  View Previous ......................................................................... 39
Configure Menu ........................................................................ 39
  Add/Remove File(s) ............................................................... 39
  Set Include Paths ................................................................... 40
  Set Project Options ............................................................... 40
  File Options ........................................................................... 40
  Remove File Options ............................................................. 41
  Remove All File Options ......................................................... 41
  MRWE Preferences ............................................................... 41
  Set Default Options ............................................................. 41
Tools Menu ............................................................................... 42
  Search ............................................................................... 42
  Build ................................................................................... 42
  Rebuild All ........................................................................... 42

Fortran User Guide
## Table of Contents

- Update Dependencies.................................................................................................................. 42
- Check Syntax ................................................................................................................................. 42
- Compile ........................................................................................................................................ 42
- Edit ............................................................................................................................................... 42
- Preprocess ................................................................................................................................... 43
- Clean ......................................................................................................................................... 43
- Stop .......................................................................................................................................... 43
- Execute....................................................................................................................................... 43
- Debug......................................................................................................................................... 43
- Profile........................................................................................................................................ 43
- Terminal .................................................................................................................................... 43
- Generate Makefile......................................................................................................................... 43

**Window Menu** .......................................................................................................................... 44
- Hide Toolbar ................................................................................................................................. 44
- Customize Toolbar ......................................................................................................................... 44
- Tile Horizontally ....................................................................................................................... 44
- Tile Vertically ............................................................................................................................ 44
- Project ....................................................................................................................................... 44
- Output ....................................................................................................................................... 45

**Help Menu** ................................................................................................................................ 46
- Tools Help ................................................................................................................................ 46
- Hide ToolTips ............................................................................................................................... 46

## Absol Fortran 95 Options

### General - F95 Options
- Warning level (-znn) .................................................................................................................. 47
- Error Handling (-dq and -ea) ........................................................................................................ 48
- Max Internal Handle (-T nn) ......................................................................................................... 48
- Temporary string size (-t nn) ......................................................................................................... 48
- Warn of Non-Standard usage (-en) ................................................................................................ 49
- Suppress warnings (-w) ................................................................................................................ 49
- Suppress Warning number(s) (-Znn) ............................................................................................ 49
- Use System Module Files (-SysModFiles) .................................................................................. 49
- Set Module Paths (-p path) ........................................................................................................... 49
- Quiet (-q) .................................................................................................................................. 49
- Verbose (-v) ................................................................................................................................. 49
- Procedure Trace (-B80) ................................................................................................................ 50
- Output Version number (-V) ........................................................................................................ 50
- Default Recursion (-eR) ................................................................................................................ 50

### Compatibility - F95 Options
- Disable compiler directive (-xdirective) ..................................................................................... 51
- INTEGER and LOGICAL sizes (-in) .............................................................................................. 52
- Character Argument Parameters (-YCFRL={0|1}) ..................................................................... 52
- Demote Double Precision to Real (-dp) ....................................................................................... 52
- Promote REAL and COMPLEX (-N113) ...................................................................................... 52
- One trip DO loops (-ej) .............................................................................................................. 52
- Static storage (-s) ...................................................................................................................... 52
- Check Array Boundaries (-Rb) .................................................................................................... 52
- Check Array Conformance (-Rc) .................................................................................................. 53
- Check Substrings (-Rs) ................................................................................................................ 53
- Check Pointers (-Rp) ................................................................................................................... 53
- Pointers Equivalent to Integers (-YPEI={0|1}) .......................................................................... 53
- External Symbol Suffix (-YEXT_SFX=string) ............................................................................ 54

### Format - F95 Options
- Free-Form (-f free) ..................................................................................................................... 54
- Fixed-Form (-f fixed) .................................................................................................................. 55
- Alternate Fixed form (-f alt_fixed) .......................................................................................... 55

---

**Fortran User Guide**
<table>
<thead>
<tr>
<th>Table of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed line length (-W nn) ................................................................. 55</td>
</tr>
<tr>
<td>YEXT_NAMES={ASIS</td>
</tr>
<tr>
<td>Escape Sequences in Strings (-YCSLASH=1) ........................................ 55</td>
</tr>
<tr>
<td>No Dot for Percent (-YNDFP=1) .......................................................... 56</td>
</tr>
<tr>
<td>Common Block - F95 Options ............................................................... 56</td>
</tr>
<tr>
<td>COMMON Block Name Prefix (-YCOM_PFX=string) .................................... 56</td>
</tr>
<tr>
<td>COMMON Block Name Suffix (-YCOM_SFX=string) .................................... 57</td>
</tr>
<tr>
<td>COMMON Block Name Character Case (-YCOM_NAMES={UCS</td>
</tr>
<tr>
<td>Absoft Fortran 90/95 Compiler Directives .......................................... 58</td>
</tr>
<tr>
<td>NAME Directive...................................................................................... 59</td>
</tr>
<tr>
<td>FREE[FORM] Directive ........................................................................... 59</td>
</tr>
<tr>
<td>FIXED Directive..................................................................................... 59</td>
</tr>
<tr>
<td>NOFREEFROM Directive .......................................................................... 60</td>
</tr>
<tr>
<td>PACK[ON] Directive ............................................................................. 60</td>
</tr>
<tr>
<td>PACKOFF Directive ................................................................................ 61</td>
</tr>
<tr>
<td>STACK Directive ................................................................................... 61</td>
</tr>
<tr>
<td>Absoft FORTRAN 77 Options .................................................................. 61</td>
</tr>
<tr>
<td>General - F77 Options ........................................................................... 62</td>
</tr>
<tr>
<td>Max Internal Handle (-T nn) ................................................................. 62</td>
</tr>
<tr>
<td>Temporary string size (-T nn) ............................................................... 62</td>
</tr>
<tr>
<td>Suppress Warnings (-w) ........................................................................ 63</td>
</tr>
<tr>
<td>Warn of non-ANSI Usage (-N32) ......................................................... 63</td>
</tr>
<tr>
<td>Warns for Undeclared Variables (-N114) ............................................. 63</td>
</tr>
<tr>
<td>Assume Pointer Aliases Exist (-B19) .................................................. 63</td>
</tr>
<tr>
<td>Quiet (-q) .............................................................................................. 63</td>
</tr>
<tr>
<td>Show Progress (-v) ............................................................................... 63</td>
</tr>
<tr>
<td>Procedure Trace (-B80) ........................................................................ 64</td>
</tr>
<tr>
<td>Condition compilation (-x) ................................................................... 64</td>
</tr>
<tr>
<td>Control - F77 Options .......................................................................... 65</td>
</tr>
<tr>
<td>Compiler Directives (-Dname=value) ..................................................... 65</td>
</tr>
<tr>
<td>Compatibility - F77 Options ............................................................... 66</td>
</tr>
<tr>
<td>Integer Sizes (-i2 and -i8) ................................................................. 66</td>
</tr>
<tr>
<td>Vax/Mainframe Compatibility ............................................................. 67</td>
</tr>
<tr>
<td>Folding to Lower Case (-f) ................................................................. 67</td>
</tr>
<tr>
<td>Static Storage (-s) ............................................................................... 67</td>
</tr>
<tr>
<td>Use Record Lengths in I/O (-N3) ......................................................... 67</td>
</tr>
<tr>
<td>RECL Defines 32-bit Words (-N51) ...................................................... 67</td>
</tr>
<tr>
<td>Folding to Upper Case (-N109) ............................................................ 67</td>
</tr>
<tr>
<td>One-Trip DO Loops (F66) (-d) .............................................................. 68</td>
</tr>
<tr>
<td>Zero Extend INTEGER*1 (-N102) ......................................................... 68</td>
</tr>
<tr>
<td>Append underscore to names (-N15) ................................................... 68</td>
</tr>
<tr>
<td>Miscellaneous - F77 Options ............................................................. 69</td>
</tr>
<tr>
<td>Double Precision Transcendentals (-N2) ............................................. 69</td>
</tr>
<tr>
<td>Sign Extend BYTE() &amp; WORD() (-N7) ............................................... 70</td>
</tr>
<tr>
<td>DATA Variables are Static (-N1) ......................................................... 70</td>
</tr>
<tr>
<td>Evaluate Left-to-Right (-N20) ............................................................. 70</td>
</tr>
</tbody>
</table>
Preprocessor – C/C++ Options ............................................................................................................................................ 77

General – C/C++ Options ............................................................................................................................................ 78

Other F77 Options .............................................................................................................................................. ..................... 74

Other F77 Options .............................................................................................................................................. ..................... 74

C/C++ Options................................................................................................................................................. 77

Promote REAL and COMPLEX (-N113) ................................................................................................................... 70
Escape Sequences in Strings (-K) ............................................................................................................................... 70
Allows CASE Without DEFAULT (-N4) ..................................................................................................................... 70
Allows UNIT= Without FMT= (-N16) ........................................................................................................................ 70
Pack STRUCTURE Elements (-N33) ........................................................................................................................... 70

Format - F77 Options........................................................................................................................................... 71

Do not search standard system directories .......................................................................................... 82

Undefines ( ... ) ............................................................................................................................................. 82

Defines ( ... ) ...................................................................................................................................................... 82

Exception Handling ( ... ) ....................................................................................................................................... 82

Include Tree to Stderr ( ... ) .................................................................................................................................... 82

Procedure Trace ( ... )........................................................................................................................................ 82

Quiet ( -q ) ......................................................................................................................................................... 82

No Alias Optimizations (-N22) ............................................................................................................................ 82

Treat All Warnings as Errors (-w1) ......................................................................................................................... 82

Treat Anachronisms as Errors (-w2) ........................................................................................................................ 82

Suppress All Warnings (-w31) ................................................................................................................................. 82

Supress Warnings (-w16) ...................................................................................................................................... 82

Supress Informationals (-w8) ................................................................................................................................. 82

Supress Anachronisms (-w41) ............................................................................................................................... 82

Suppress Template Warnings (-w1) ........................................................................................................................ 82

Treat Anachronisms as Errors (-w52) ................................................................................................................... 82

Treat All Warnings as Errors (-wabort) .................................................................................................................. 82

No Alias Optimizations (-N19) ............................................................................................................................ 82

Quiet ( -q ) ......................................................................................................................................................... 82

Verbos (-v) .......................................................................................................................................................... 82

Procedure Trace (-N124)......................................................................................................................................... 82

Verbos Templates (-pV) ......................................................................................................................................... 82

Include Tree to Stderr (-H) .................................................................................................................................... 82

No inlines (-inline none) ..................................................................................................................................... 82

Exception Handling (-except on/off) ..................................................................................................................... 82

Preprocessor – C/C++ Options........................................................................................................................................ 81

Defines (-D name=[value]) ................................................................................................................................... 82

Undefines (-U name) ............................................................................................................................................ 82

Do not search standard system directories .......................................................................................... 82

Preprocess files only ........................................................................................................................................ 82

Fortran User Guide
CHAPTER 4 PORTING CODE

Porting Code from VAX

Compile Time Options and Issues

Runtime Issues

Porting Code from IBM VS FORTRAN

Compile-time Options and Issues

Run-time Issues

Porting Code From Microsoft FORTRAN (PC version)

Compile-time Options and Issues

Porting Code from Sun Workstations

Porting Code from the NeXT Workstation

Porting Code from the IBM RS/6000 Workstation

Porting Code from Intel 386/486/Pentium Computers

Porting Code To/From Other Macintosh Systems

Language Systems Fortran

Other Absoft Compilers

Other Porting Issues

Memory Management

Dynamic Storage

Static Storage
CHAPTER 5 THE MACINTOSH RUNTIME WINDOW ENVIRONMENT ...... 105

Using MRWE ..................................................................................................................... 105
The MRWE Window ............................................................................................................ 105
How Your Program And MRWE Work Together ................................................................. 107
Working With Text in MRWE ............................................................................................... 107
Using The MRWE Default Menus ....................................................................................... 107
File Menu ............................................................................................................................... 107
Save (S) ................................................................................................................................. 108
Save As .................................................................
Page Setup ............................................................................................................................ 108
Print Window (P) .................................................................................................................... 108
Quit (Q) ................................................................................................................................... 108
Edit Menu ............................................................................................................................... 108
Undo (Z) .................................................................................................................................. 108
Cut (X) .................................................................................................................................... 108
Copy (C) .................................................................................................................................. 108
Paste (V) ................................................................................................................................. 109
Clear ....................................................................................................................................... 109
Font and Size Menus ............................................................................................................. 109

Programming with MRWE .................................................................................................. 109
Program Organization: Fortran VS. Macintosh ................................................................. 109
MRWE Event Loop Operation .............................................................................................. 111
Customizing Menus .............................................................................................................. 112
Adding Menus ....................................................................................................................... 112
Special Characters ................................................................................................................ 112
Menus and the READ statement .......................................................................................... 113
Removing a Menu or Menu Item .......................................................................................... 114
Menu Response Routines and mrwe_DoMenu ................................................................. 114
Adding Checkmarks to Menus ............................................................................................. 115
Enabling/Disabling Menu Items ........................................................................................... 115
Launching OTHER APPLICATIONS .................................................................................. 116
Apple Events ......................................................................................................................... 117
Apple Event Target .............................................................................................................. 117
Apple Event Class and ID ..................................................................................................... 118
Extra Information in an Apple Event .................................................................................... 118
Receiving Apple Events ...................................................................................................... 121
Error Codes Returned from Apple Event Routines ............................................................ 124
Other Examples of Apple Events ........................................................................................ 124
Sending a request to the Finder .......................................................................................... 124
Using other standard Apple Events ................................................................................... 125
Sending information between MRWE applications ........................................................... 125
Scripting ................................................................................................................................. 125
Further Information About Apple Events ........................................................................... 126
Creating Multiple Windows ............................................................................................... 126
## Table of Contents

- **Compiler Options** .................................................................................................................. 159
- **Profiling** ............................................................................................................................... 159
- **CHAPTER 8 USING THE FX DEBUGGER** ............................................................................... 161
  - **Introduction** ......................................................................................................................... 161
    - About Fx ............................................................................................................................... 161
    - About Fx Interfaces ............................................................................................................... 162
    - About the Examples in this Manual ..................................................................................... 162
  - **Tutorial** ............................................................................................................................... 163
    - Launching the Debugger ........................................................................................................ 163
    - Setting a Breakpoint ............................................................................................................. 166
    - Executing Programs .............................................................................................................. 167
    - Exiting from the Fx Debugger .............................................................................................. 170
  - **Fx Interface Reference** ....................................................................................................... 171
    - Using the Fx Interface .......................................................................................................... 171
    - Fx Main Window .................................................................................................................. 172
    - Fx Debugger Application Menu ........................................................................................... 174
    - File Menu ............................................................................................................................ 177
    - Edit Menu ............................................................................................................................ 180
    - Find Submenu ...................................................................................................................... 180
    - Selection Menu ................................................................................................................... 181
    - Debug Menu ....................................................................................................................... 183
    - Program Menu .................................................................................................................... 185
    - Breakpoints Submenu ......................................................................................................... 188
    - Monitors Submenu ............................................................................................................... 190
    - WatchValues Submenu ...................................................................................................... 193
    - Window Menu ..................................................................................................................... 196
    - Help Menu .......................................................................................................................... 196
    - Command Text Field .......................................................................................................... 198
  - **Running FX** ......................................................................................................................... 199
    - Customizing the Debugger .................................................................................................. 200
    - Preparing Your Program For Debugging ............................................................................. 202
    - Starting an Fx Debugging Session ....................................................................................... 203
    - Restarting A Debugging Session ......................................................................................... 205
    - Debugging with Core Files ............................................................................................... 206
    - Examining Source Code .................................................................................................... 207
    - Recreating a Debugging Session ......................................................................................... 209
  - **Executing Programs in Fx** .................................................................................................. 211
    - Executing Single Statements .............................................................................................. 211
    - Using Breakpoints ............................................................................................................... 212
    - Executing Programs .......................................................................................................... 216
    - Animating Program Execution ........................................................................................... 216
    - Calling Program Procedures ............................................................................................. 217
  - **Working With Program Variables** .................................................................................... 220
    - Displaying Program Variables ........................................................................................... 220
    - Modifying Variables .......................................................................................................... 224
    - Finding Overwritten Variables ......................................................................................... 224

---

Fortran User Guide
APPENDIX A  ABSOFT COMPILER OPTION GUIDE ................................................................. 307

Absoft Pro Fortran Compiler Options ....................................................................................................................... 307
FPU Control Options ......................................................................................................................................................... 308
PowerPC Processor Specific Options ............................................................................................................................... 308
Fortran 90/95 Control Options ........................................................................................................................................ 308
Fortran 90/95 Optimization Options ................................................................................................................................ 309
Fortran 90/95 Source Format Options ............................................................................................................................ 309
Fortran 90/95 Compatibility Options .............................................................................................................................. 309
FORTRAN 77 Control Options ......................................................................................................................................... 310

Fortran User Guide
Table of Contents

FORTRAN 77 Optimization Options ................................................................. 312
FORTRAN 77 Source Format Options ............................................................... 312
FORTRAN 77 Compatibility Options ............................................................... 312

APPENDIX B  TERMINAL PROGRAMMING .............................................. 315

APPENDIX C  FX CONTROL VARIABLES ............................................... 317

APPENDIX D  ASCII TABLE ........................................................................... 321

APPENDIX E  BIBLIOGRAPHY ......................................................................... 325

Fortran 90/95 ......................................................................................................... 325

FORTRAN 77 ......................................................................................................... 325

APPENDIX F  TECHNICAL SUPPORT ............................................................ 327
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 Classic Macintosh, Macintosh OS X, Windows, 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. Absoft compilers support most popular inter-language calling conventions implemented on Macintosh OS X systems, providing compatibility with existing libraries and object files, simplifying porting efforts.

This User Guide explains the operation of Absoft Fortran 90/95, Absoft FORTRAN 77, Absoft C/C++, and the Fx™ debugger on the Macintosh OS X operating system for the PowerPC families of processors. In the event you have licensed only one of these compilers, please refer only to the appropriate section(s) and disregard the others. All compilers operate in a similar manner, share a common tool set, and are link compatible. A brief summary of each compiler appears below.

Absoft Fortran 90/95

A complete, optimizing ANSI Fortran 90/95 implementation with 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 interfacing with FORTRAN 77 and C Programming Language libraries.

Absoft FORTRAN 77

Refined over 20 years, with emphasis on porting legacy code from workstations. Absoft Fortran 77 is full ANSI 77 incorporating 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 that 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 and C/C++ so existing, extended FORTRAN 77 routines can be easily compiled and linked with new Fortran 90/95 or C/C++ code.
Introduction

Absoft C/C++

A combination of ANSI C, K&R C and C++ workstation class compilers. Absoft C/C++ is supplied to facilitate mixed Fortran/C/C++ development. The entire Win32 API is supported and DLLs can be created from C and C++.

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 are used for emphasis and book titles.
- Unless otherwise indicated, all numbers are in decimal form.
- FORTRAN examples appear in the following form:

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

ROAD MAPS

Although this manual contains all the information needed to build programs with Absoft Pro Fortran on Macintosh OS X, there are a number of other manuals that describe Fortran 90/95 and FORTRAN 77 in further detail. The road map in this chapter will guide you to these manuals for introductory or advanced reference. The bibliography in appendices lists further information about each manual.

Fortran Road Maps

The Absoft implementation of Fortran 90/95 is detailed in the online manual, Fortran 90 Concise Reference, in the Documentation directory of the Pro Fortran CDROM. FORTRAN 77 is detailed in the online manual, FORTRAN 77 Language Reference Manual, also in the Documentation directory of the Pro Fortran CDROM. A discussion of floating point precision is at the end of the chapter Porting Code. Figures 1-1 shows additional manuals that can be used for referencing the FORTRAN language and internal math operations.

Fortran User Guide
YEAR 2000 PROBLEM

All versions of Absoft Pro Fortran products for Macintosh, Power Macintosh, Windows 95/98, Windows NT, Linux, and UNIX will 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. This means the version of Absoft Pro Fortran that you already have will continue to operate correctly. No patches or version updates are required.

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 90/95 **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 *Fortran 90 Concise Reference* 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 manual **Absoft Compatibility Libraries** for information on their format and use.
CHAPTER 2

Using the Absoft Editor

This chapter describes how to use the Absoft Editor 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.

With the Absoft Editor, you can edit multiple files at the same time, launch a compiler, and return to the editor to correct syntax errors detected by the compiler. The Absoft Editor is a Macintosh OS X program.

THE ABSOFT EDITOR

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).
6 Using the Absoft Editor

Text Selection

Text may be selected for copying or deleting in two different manners. For small amounts
of text, you can drag the cursor over the text while holding the left mouse button down.
For larger amounts of text, click the left mouse button at the beginning of the selection,
hold the Shift key down, and the click the left mouse button again at the end of the
selection.

Using Compilers

The editor can be used to run a compiler to either check the syntax of the source file or
compile it into an executable application. Default compiler options are set with the
Preferences control, described later in this chapter. Specific options for individual files
can be set either in the Source Info dialog for the file (also described later) or in the
Option Toolbar.

The Option Toolbar appears below the editor’s menu bar:

![Compiler Options]

You can choose from a predefined set of options or type in your own custom set for the
file.

Pop-up menus

Holding down the control key and clicking the mouse button with the cursor positioned
over an open file window will display a pop-up menu of context sensitive commands.

Cut

This command removes the selected text from the front-most window and
places it on the clipboard. Text on the clipboard may be pasted into other
windows.

Copy

The Copy command copies the selected text from the front-most window and
places it on the clipboard. Text on the clipboard may be pasted into other
windows.

Paste

This command replaces the selected text in the front-most window with the
text on the clipboard. If no text is selected in the front-most window, the
clipboard text is inserted at the insertion point.

Fortran User Guide
CREATING NEW SOURCE FILES

To create a new source file, choose the New command from the File menu. If Display new file dialog box is checked on the Format tab of the Preferences property sheet (See Preferences below) the following dialog will be presented, allowing you to specify the type of new file to create:

Otherwise, the default file type will be determined by the Use as new file default format setting on the Format tab of the Preferences property sheet.

The window will be untitled (it will have the name “Untitled”) until the first time you save it. At that time you will be asked to name the file. Text can be entered and edited using the same basic editing techniques that you use with any Macintosh-based text
Using the Absoft Editor

Editor or word processor. You can cut and paste text within the window and move the cursor to enter text at any line.

Manipulating Windows

One or more file windows can be open at any time in the Absoft Editor, allowing you to easily cut and paste text between files.

When working with multiple windows, it is important to note that the Absoft Editor distinguishes between the active window (front-most window) and any inactive windows. Editing commands initiated from the menus will affect or insert text in the active (front-most) window. If you want to know which window is your active window, check the Window menu. In the Window menu, the active window will have a check mark next to it.

USING THE EDITOR MENUS

The rest of this chapter describes the editor commands in the Application, File, Edit, View, Tools, Window, and Help menus. The name of the command is given, followed by its keyboard equivalent (if any) and a description of its function.

Application Menu

The Application menu contains the standard commands for Macintosh OS X applications. It also includes the Preferences command which is used to configure the Absoft Editor to your specific editing needs. Preferences consist of three separate property tabs: Format, Environment, and Tools.
Preferences...

Format

The Format pane contains the following controls:

Language

This selection box is used to establish the source language for making the settings in the next sections. The default choices are any of Plain Text, Fortran 77, Fortran 90/95, C, C++, Makefile, or Ini. Additional, languages may be added by editing the appropriate files in ./Resources/General Resources. See the Color section below for more information.

Syntax Colors
Use this section to change the default use of color, and the actual color in the text, of Comments, Intrinsics, Keywords, and literal Strings for the selected language. Languages may define up to eight unique colors, and eight or more keyword lists. Some additional color syntax settings for a given language may be viewed, if available, by using the up and down buttons. Additional, user defined keywords may be added by editing the associated resource files located in ./Resources/General Resources/Languages.

Disable all color syntax

Checking this check box disables syntax coloring for the selected language.

Use as new file default format

This check box controls the default formatting for new files. For example, if you want new (and unnamed) files to default to the Fortran 90/95 format specifications that you have established, choose Fortran 90/95 from the Language selection box and then check this check box.

Display new file dialog box

This checkbox enables the new file type dialog box. Check this check box if you would like to interactively choose the type of file and formatting each time a new file is created. The default file type chosen with the Use as new file default format setting will be the initial file type selected in this dialog box.

Auto wrap at column

This option is available when either Fortran 90/95 or Fortran 77 are specified in the Language selection. If you place a check in this box, the editor will automatically wrap and continue your source line if you type past the specified column. The character you choose will be placed in column six. The default character is the ‘&’ character.

Default Properties

This section controls the default property settings for all files of the currently selected language type (see Show Info in the Edit menu discussed later in this chapter). You can choose to have Line Format, Coding Form, and Line Numbers shown by default in the text window with your file. (Coding Form is available only for Fortran 90/95 and Fortran 77.) You can also set the default Tab Size and whether or not you want Auto Indent enabled.

The drop down list box is used to confirm or change the interpretation that the Absoft editor applies to the format of the source file.

Use the Show Info command in the Edit menu to set properties for individual source files.
Environment

The **Environment** pane has the following controls:

**Prompt for file at startup**

Check this check box if you want the Absoft Editor to prompt you for the name of a file to open rather than starting up with a blank screen or automatically creating a new file.

**Always open new window**

Check this check box if you want the Absoft Editor to automatically create a new file rather than starting up by prompting you for the name of a file to open or starting up with a blank screen.

**Save before launching tools**

Check this check box if you want the Absoft Editor to save changes to any open files before starting a compiler.
Prompt before saving files

Check this check box if you want the Absoft Editor to issue a prompt, allowing you to confirm or cancel saving changes to any open files before starting a compiler.

Always preserve backup copy

Check this check box if you want the Absoft Editor to maintain a backup copy of the file before every Save command. The backup file will have the extension .aeb.

Text Selection Filters

The controls in this section establish the logic used for selecting text elements. These controls allow you to select text based on the semantics and syntax rules of the programming language of the source file.

Font Settings

Use this button to select the text font and size you wish to use.

Cache Settings

The editor maintains certain information about each file that you open including the current cursor position, the tab size, etc. Use this Settings button to specify how long this information should be maintained since the last time the file was opened.
Tools

The text boxes in this section allow you to set the path to the tool that you prefer to use for each of the source file types. You can also supply default options for individual tools with these text boxes.

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 the Absoft Editor operates.

New… (⌘N)

This command creates a new window for entering and editing text. The window will be untitled (it will have the name “Untitled”) until the first time you save it.
Open…(⌘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 window that contains that file will be brought to the front of the editor.

Open Recent

Up to 16 files will appear in this list. They represent the file that have been most recently opened in the Absoft Editor. They are listed as a convenience for quickly opening files for editing.

Open Selected

This command opens the file selected in an include statement. It may be a Fortran or a C include statement.

Open Complement

This command opens the complement of the current programming language file. If a C or a Fortran source file is open, the header or include file with the same root name will be opened. If a header or an include file is opened, the source file with the same root name will be opened.

Close (⌘W)

This command closes the file displayed in the front-most window. If any unsaved changes had been made to the text, you will be asked to save it.

Close All

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

Save (⌘S)

Choose this command to save the text in the front-most editor window. The first time you save, you will be asked to provide a name and a path for the file. Thereafter, each time you save, the changes will automatically be written to this file. If no changes have been made, this menu command is dimmed and unavailable.

Save As…

Use the Save As command to save the text in the front-most editor window to a different file. A standard file save dialog will appear, allowing you to specify the name of file. The front-most editor window becomes the newly named file.
Save All

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

Revert

The Revert command restores the text in the editor window to its previously saved state.

Print Setup…(⌘ P)

Use this command to display the standard Print Setup dialog for printing.

Print… (⌘ P)

This command prints the front-most window to the currently selected printer.

Page Setup…

Use this command to open the Page Setup dialog:
This dialog allows you to specify a page header and footer. Click on the right-arrow buttons to insert the text you wish to display in the Header and the Footer: File Name, Page Number, Date, and Time. This dialog also allows you to specify the document margins.

**Edit Menu**

The commands in the Edit menu are used for performing standard editor functions, such as Cut, Copy, and Paste.

**Undo** ($\text{⌘} \text{Z}$)

Use this command to undo changes made in the front-most window.

**Redo** ($\text{⌘} \text{Z}$)

Use this command to redo commands that were undone with the Undo command.

**Cut** ($\text{⌘} X$)

This command removes the selected text from the front-most window and places it on the clipboard. Text on the clipboard may be pasted into other windows.

**Copy** ($\text{⌘} C$)

The Copy command copies the selected text from the front-most window and places it on the clipboard. Text on the clipboard may be pasted into other windows.

**Paste** ($\text{⌘} V$)

This command replaces the selected text in the front-most window with the text on the clipboard. If no text is selected in the front-most window, the clipboard text is inserted at the insertion point.

**Clear**

Similar to cut, this command removes the selected text from the front-most window, but does not place a copy on the clipboard.

**Select All** ($\text{⌘} A$)

Use this command to select all of the text in the file displayed in the front-most window.
Using the Absoft Editor

Find

This command displays the Find sub-menu with commands for finding and replacing text within the file.

Find (F)

Use this command to open the Find dialog for locating specified text within the front-most window.

The controls in the Find dialog are used as follows:

Find

Enter the text string you wish to locate here. A history of previous uses of this command is maintained, allowing you to select strings that you located earlier.

Replace

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

Ignore Case

Check this box to find text occurrences in your source file that match your specified text regardless of capitalization and case.

Find Next (F)

This command repeats the last Find command in the front-most window by searching forwards in the file.

Find Previous (Shift F)

This command repeats the last Find command in the front-most window by searching backwards in the file.

Fortran User Guide
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 \texttt{F2} sets (or unsets) a bookmark. In other words, \texttt{F2} toggles a bookmark.

A bookmark appears as a small flag at the beginning of the line. Pressing the \texttt{F2} key alone moves the insertion caret to the next bookmarked line in the file. Holding the Shift key down and pressing the \texttt{F2} key moves the insertion caret to the previous bookmarked line in the file. Holding the Shift key down and typing \texttt{F2} clears all bookmarks in the file.

Bookmarks Menu

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

\textbf{Toggle Bookmark (F2)}

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

\textbf{Previous Bookmark (F2)}

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

\textbf{Next Bookmark (F2)}

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

\textbf{Clear File’s Bookmarks (Shift+F2)}

Use this command to remove all bookmarks in the file.

\textbf{Clear All Bookmarks (Shift+F2)}

Use this command to remove all bookmarks in all files.

Format Menu

The commands in the Format menu provide you with a number of useful commands for managing and formatting your source files.

Fortran User Guide
Show Info… (⌘⌘)

The Show Info dialog is used to control certain characteristics of the display of the file in the front-most window. It also allows you set specific commands and options if you use the Absoft Editor to compile the source file. The characteristics that can be controlled include:

**Options**

If the Absoft Editor was not started from the Absoft Developer Tools Interface, this field contains the tool option settings for the type of file as established on the Tools page of the Preferences dialog (see Preferences in the File menu, discussed earlier in this chapter, for more information).

If the Absoft Editor was started from the Absoft Developer Tools Interface, this field contains the compiler settings for the project and source file.

**Language**

Use this selection box to establish or change the source language setting for this file.

**Format**

This drop down list box is used to confirm or change the interpretation that the Absoft editor applies to the format of the source file.
Line Style

This selection box controls how end-of-record (end-of-line) characters are interpreted. Lines in DOS files end with a carriage return/line feed pair. Macintosh lines end with a carriage return only. Files on a Unix system have lines terminated by line feeds only.

The file must be saved before changes to this parameter will take affect.

Auto Indent

This check box enables automatic spacing of new lines to the column where the previous line started.

Coding Form

FORTRAN 77 specifies the purpose of individual columns within a source statement record. A check in this check box will color certain columns to make them easier to identify. The specific columns are 6, 72-80, and 132-140. For more information on source record fields, refer to the section FORTRAN 77 ANSI Standard in The Fortran 77 Program chapter of the FORTRAN 77 Language Reference Manual.

Read Only

A check in this check box will prevent you from making any inadvertent changes to a file that you wish to maintain as read only.

Line Numbers

A check in this check box will display the line numbers of the source lines in the file.

Tab Size

The value in this edit box controls how many spaces a tab is expanded to. Note that for column oriented languages such as FORTRAN 77, the value in this field should agree with how the compiler interprets tab characters.

Insert Continuation (\texttt{I})

Use this command to insert a Fortran continuation line immediately after the line on which the cursor is positioned. The cursor will be repositioned to the next character position following the continuation character. The continuation character used is defined on the Format page of the Preferences dialog. See Preferences in the Application menu, described earlier for more information.
Go to Line (L)

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

![Goto dialog](image)

Match Brackets (}{}

When editing any file, this command may be used to find the matching closing character for the opening character next to the cursor on the line where the cursor is positioned. The characters it will match are: (), {}, and [] – parentheses, braces, and brackets.

Shift Left ([)

Use this command to shift selected text to the left by one tab stop.

Shift Right (])

Use this command to shift selected text to the right by one tab stop.

Comment

This command inserts an exclamation mark ('!') in column one of the current line or the selected lines.

Uncomment

This command deletes an exclamation mark ('!') from column one of the current line or the selected lines.

Tools Menu

The commands in the Tools menu are used to invoke various functions of the language compilers you use. If the compiler issues any diagnostics, they will be reported to you in the Errors window. Errors are indicated with a red circle (stop sign) next to the diagnostic message and warnings are indicated with an inverted yellow triangle (caution sign). You can double-click on any line in this window to go directly to the corresponding line in the Absoft editor window. The Errors window is automatically opened only if the compiler issues warnings or errors.
Compile (⌘Y)

Use this command to compile the file in the front-most window. The tool that you have selected for the file type will be started with the command line shown in the Option toolbar.

Check Syntax (⌘K)

This command invokes the check syntax phase of the compiler that you have selected for the type of file in the front-most window.

Stop (⌘.)

Use this command to stop the compiler started with either the Check Syntax or Compile command.

Errors

This sub-menu contains commands for moving to the previous or next error or warning in the source file.

Previous Error (⌘D)

If you compiled your source file with the Compile command in the Tools menu and error diagnostics were issued by the compiler, you can use this command to go to the previous error in the source file.

Next Error (⌘E)

If you compiled your source file with the Compile command in the Tools menu and error diagnostics were issued by the compiler, you can use this command to go to the next error in the source file.

Window Menu

The commands in this menu allow you to arrange the open windows in the Absoft Editor and to bring a specific window to the front.

Close Window (⌘W)

This command closes the front-most window in the Absoft Editor. It is the same as the Close command in the File menu.

Zoom Window

This command maximizes the front-most window.
Minimize Window (⌘M)

This command minimizes the front-most window and places it on the docking bar.

Hide Toolbar

This command removes the toolbar from the front-most window.

Customize Toolbar...

This menu command opens the Toolbar editor that allows you to customize the toolbar.

Tile Horizontally

Use this command to arrange your windows horizontally based on the order in which they were opened.

Tile Vertically

This command is used to arrange your windows vertically based on the order in which they were opened.

Cascade

This command is used to arrange your windows in a cascaded fashion.

Window list

Selecting the name of a window in this list will bring it to the front, restoring it to its previous size if it has been minimized.
Help Menu

The commands in this menu are used to obtain help on the listed topics.

Tools Help

Use this command to view Absoft Developer Tools Interface application online help files.

Hide ToolTips

Use this command to disable pop-up tooltips for all windows and controls in the Absoft Developer Tools Interface for the remainder of the session.
CHAPTER 3

Using the Compilers

This chapter describes how to use the Absoft Fortran 90/95 and FORTRAN 77 compilers to create executable files on the Mac OSX operating system. Beginning with an overview of invoking the compilers, this chapter explains how to compile a small number of Fortran source files into an executable application. Next the Absoft Developer Tools Interface, AbsoftTools, application is described with detailed descriptions of options and compiler settings.

COMPILING PROGRAMS

Three methods of compiling programs are available: a traditional command line, the Absoft Developer Tools Interface, and makefiles. The command line and the Absoft Developer Tools Interface application are discussed in this chapter. Makefiles and the Absoft make utility, amake, are described in the chapter Building Programs.

Source file names and compiler options are selected with the mouse pointer in the Absoft Developer Tools Interface application. 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 Terminal application shell and set a number of environment variables that assist and control the use of the compilers.

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

    f95 [options] files
    f77 [options] files

The various options are described in the specific compiler options sections later in this chapter.
FILE NAME CONVENTIONS

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

<table>
<thead>
<tr>
<th>File Type</th>
<th>Default form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free format Fortran 90/95 source files</td>
<td>file.f90 or file.f95</td>
</tr>
<tr>
<td>Free format Fortran 90/95 preprocessor files</td>
<td>file.F90 or file.F95</td>
</tr>
<tr>
<td>Fixed format Fortran 90/95 source files</td>
<td>file.f</td>
</tr>
<tr>
<td>Fixed format Fortran 90/95 preprocessor files</td>
<td>file.F</td>
</tr>
<tr>
<td>C language source files</td>
<td>file.c</td>
</tr>
<tr>
<td>Assembly language source files</td>
<td>file.s</td>
</tr>
<tr>
<td>Relocatable object files</td>
<td>file.o</td>
</tr>
</tbody>
</table>

Acceptable inputs to \texttt{f77} are:

<table>
<thead>
<tr>
<th>File Type</th>
<th>Default form</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORTRAN 77 source files</td>
<td>file.f or file.for</td>
</tr>
<tr>
<td>FORTRAN 77 preprocessor files</td>
<td>file.F or file.FOR</td>
</tr>
<tr>
<td>C language source files</td>
<td>file.c</td>
</tr>
<tr>
<td>Assembly language source files</td>
<td>file.s</td>
</tr>
<tr>
<td>Relocatable object files</td>
<td>file.o</td>
</tr>
</tbody>
</table>

File names that do not have one of these default forms are passed to the linker. It is assumed that the C compiler (\texttt{cc}), assembler (\texttt{as}), and linker (\texttt{ld}) are installed on the system and use standard command line syntax.

Output file names take the form:

<table>
<thead>
<tr>
<th>File Type</th>
<th>Default form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly language source files</td>
<td>file.s</td>
</tr>
<tr>
<td>Relocatable object files</td>
<td>file.o</td>
</tr>
<tr>
<td>Precompiled module file</td>
<td>file.mod</td>
</tr>
<tr>
<td>Executable object files</td>
<td>a.out</td>
</tr>
</tbody>
</table>

USING THE ABOFF DEVELOPER TOOLS INTERFACE

The Absoft Developer Tools Interface, AbsoftTools, is started by double clicking on the AbsoftTools icon, the equivalent alias icon in the Finder, or by clicking on the AbsoftTools application icon in the Dock.
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 File menu New… command to create a new project. The New Project dialog will appear as shown below:

![New Project Dialog](image)

Name is the name that will be applied to the project file, the associated makefile, and the default name of the executable program. Location is the directory where the project file and the associated makefile are created. Use the button labeled with an ellipsis to open a directory browser to change the directory. Use the Create a directory of the same name first check box to automatically create a subdirectory with the supplied name before creating the actual project files. The Option Set drop-down menu is used to choose a specific set of pre-determined options for the project. (See Set Default Options, described later in this chapter, for information on configuring default option sets for new projects.) The File Set drop-down menu is used to select a default set of source files to create for the project. The Empty Project set creates a project with no source files by default. The New Fortran 95 Project and the New Fortran 77 Project sets create projects containing a new source file with the same name as the project file in the project directory. The New MRWE Application menu item creates all the source files necessary for a Macintosh Runtime Window Environment application, including a Fortran 95 file, a custom Info.plist file and the resource description files.

After you click the OK button, if you selected a new MRWE application, the MRWE Application Preferences dialog will appear. This dialog contains controls that allow you to customize how MRWE handles text and text saving. It also allows you to specify what your application does at the end of execution: quit immediately or pause automatically.
After you click the OK button to accept these values, the project options panel will be displayed, allowing any of the global compiler options, Plugin options, or Build options to be set or changed. After these are established, begin selection of the files that will constitute the project by choosing the Add/Remove File(s) command from the Configure menu.

Options Dialog

The Options dialog appears automatically after you confirm the settings for a New Project. The Options dialog can be invoked in several manners: by selecting the Set Project Options command to configure options for the current project, using the Set Default Options command to configure a set of default options used in the creation of new projects, or using the Set File Options command to configure options for the selected source file.

The Options dialog has these features:

- Any button that contains ellipses, such as Set Module Path(s)..., will bring up additional dialogs.
- As options are checked in the dialog, they will be added to the Options text field in the lower portion of the current options pane.
Using the Compilers 29

The Options panel displayed with the Set Project Options command also has tabs for the individual compilers and the linker. Within the Options dialog, you can select options to modify the way compiled applications function. When the Build command is invoked, a makefile is constructed and the source code will be compiled and linked into an executable program, ready to run. If any errors occur, they will be listed in the Output pane, under the Build tab. See the chapter Building Programs for information on makefile commands and using amake.

Target Tab – General Options

The Target tab controls two option subsets that apply to all of the files for a project: General and FPU. Choose the desired subset from the Options Subset drop-down menu. General options are described first:

Target Type

The radio buttons and check boxes in this area of the Target tab control the type of application produced by the compilation process. By default, a stand-alone terminal application (the Terminal Application button) is generated. This type of application runs in the Terminal application command line shell. Use -cons to select this option on a command line.
An application linked with MRWE (the **MRWE Application** button) creates an application with a Macintosh style interface (see the chapter, **Macintosh Programming** for more information). Use `-mrwe` to select this option on a command line.

Use the `-plainappl` option (selected with the **Carbon Application** button) when you are creating an application with an interface that you supply. This type of application will have neither an MRWE interface nor a Terminal application shell interface – you are responsible for the interface presented to the user. The option includes all of the standard Carbon API frameworks.

Specify the **Static Library** option if you are creating a static library (see **Linking Programs** and **Creating Libraries** in the chapter titled **Building Programs**).

**MRWE Events**  **(-N9)**

An application, with the Macintosh Runtime Window Environment (MRWE), may be terminated by pressing Command-Period (``-``). The checks for `%%-` are normally done at every Fortran I/O statement. The `-N9` option will generate code that checks for `%%-` more frequently. Using this option will slow program execution slightly, but the application will remain friendly during long computations that do not use I/O.

**Link Large Data Stubs**

The Carbon and Cocoa development APIs have initialization routines that are called during application startup. For console applications, these are replaced by symbols in the data segment. This imposes a limit on the size of a console application's data segment. The Link Large Data Stubs option removes this limit by supplying stub routines located in the text section.

**Use 32-bit branches (-N11)**

The program counter relative branch instructions of the PowerPC microprocessor are limited to signed 16-bit offsets. By default, the Absoft compilers issue these single instruction branches within program units. In some instances of large source files, the range of this branch instruction may be exceeded, resulting in an error diagnostic being issued by the assembler.

The `-N11` compiler option should be used to overcome this limitation. This option should only be used when the assembler issues diagnostics as it will cause more code to be generated than is usually necessary.

**Runtime Stack Trace (-et)**

Use this option to enable a diagnostic stack trace back if a catchable error occurs at runtime. The file name, line number, and list of calling procedures will be displayed at the point of the error. Errors that can be caught include memory access violations, I/O errors, and integer division by zero.

**NOTE:** This option requires Macintosh OS X 10.2 or later.
Options

The Optimize and Debug drop-down menus are used to control the production of object code. Selecting one of the available optimization choices in the Optimize menu enables compiler optimizations.

The Optimize options control compile time optimizations to generate an application with code that executes more quickly. Absoft Pro Fortran 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. Any optimization option automatically enables –cpu:host (see CPU below).

The Basic (–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.

The Normal (–O2) option enables optimizers that can 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. Note: -O2 is equivalent to the deprecated –O option.

The Advanced (–O3) option enables advanced optimizers that can substantially rearrange the code generated for a program. The optimizations include loop transformations and sophisticated scheduling. This option is not usable with debugging options.

When None (Debug) is selected in the Optimize drop-down menu, the Debug options menu is enabled allowing you to select the level of symbolic information to produced for debugging with Fx (see the chapter Using the Fx Debugger):

Standard produces an object file containing typical debugging information with entry points, line numbers, and program symbols. This is the standard debugging option and is selected on the command line with the -g switch.

Full forces the compiler to place information in the debugger symbol tables for all structures whether they have associated storage or not. Normally, the compiler does not place information in the debugger symbol tables for structures that are only declared, but never have storage associated with them. This keeps the symbol tables to a manageable size when include files are used to make structure declarations. This option is selected on the command line with the -g and –N111 switches.
Use the **CPU** options to target object code to a specific type of processor. This option is selected on the command line with `–cpu:type` switch. The recognized **type** arguments are:

<table>
<thead>
<tr>
<th>Default</th>
<th>G3 unless optimization is enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>automatically establishes <strong>type</strong> based on the processor in the machine that the program is compiled with</td>
</tr>
<tr>
<td>G5</td>
<td>G5 processor</td>
</tr>
<tr>
<td>G4</td>
<td>G4 processor</td>
</tr>
<tr>
<td>G3</td>
<td>G3 processor</td>
</tr>
</tbody>
</table>

The **Enable Profiling** option (`-P` from the command line) causes your program to be instrumented for runtime profiling with the `gprof` tool. For information on using the Mac OS X profiler, see the Mac OS X manual page for `gprof`.

The next two options in this section, **Use UNIX Library** and **Use VAX/VMS Library**, automatically include these compatibility libraries in your project. The compatibility libraries are described in the *Absoft Support Libraries* manual that is included Pro Fortran.

Use of the `-Q51` option will cause the compiler not to use floating-point multiply-add type of instructions. Since there is no rounding performed between the multiplication and addition during the execution of these instructions, numeric results will vary depending on where they are used.

The last option in this section, **Use Altivec**, controls the use of optimized vector libraries for the PowerPC G4 processor. If your program is written in Fortran 90/95, uses the IMSL libraries, or uses the LAPACK libraries, special vector libraries will be linked to your program that take advantage of the single precision vector unit in the G4 processor.

**Object File(s) Directory**

This is the directory where all object files will be created and maintained.

**Target Directory**

This is the directory where the executable file (or makefile target) will be placed.

**Target Filename**

The name of the generated executable file can be typed here. If no name is specified, the default is to produce an executable file called `a.out`. The executable file will be placed in the directory specified in the **Target Directory** text box. This option is specified on the command line as `–o name`, where `name` is the name the executable file will be given.
Target Tab – FPU Options

The FPU option subset provides control over the operation of the Floating-Point Unit of the processor.

FPU Rounding Mode

These radio buttons control the rounding method used by the floating point unit. The default method is the IEEE P754 default state: Round to nearest. The rounding method can be controlled on the command line with the option:

```
-round=mode
```

where mode is one of:

- NEAREST
- DOWN
- UP
- TOZERO
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:

```
-trap=exception[,exception,...]
```

where `exception` is one or more of:

```
INVALID
DIVBYZERO
OVERFLOW
UNDERFLOW
INEXACT
ALL
```

This option requires the Runtime Stack Trace (-et) option to be enabled to provide an exception handler.

Don’t generate FMA instructions (-Q51)

Use of the -Q51 option will cause the compiler not to use floating-point multiply-add type of instructions. Since there is no rounding performed between the multiplication and addition during the execution of these instructions, numeric results will vary depending on where they are used.

Other Target 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.
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 Macintosh OS X 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.

Library Specification (-l)

Specifying the `-lname` option will cause the linker to search the library file `libname.a`.

Library Path Specification (-L)

The `-Lpath` option will cause the linker to search the specified directory named in `path` for library files given with succeeding `-l` options.

ABSOFT DEVELOPER TOOLS INTERFACE

The following sections describe the menu commands available in the Absoft Developer Tools Interface (AbsoftTools) application.
Application Menu

About Tools

This dialog displays various information about the application and its environment, including the version number.

Preferences…

This command opens the Preferences dialog for editing and customizing the way the Absoft Developer Tools Interface application operates. The options are divided into several groups:

Preferences

The Startup Options check boxes control what action (if any) the Absoft Developer Tools Interface application performs upon startup. If the Prompt for project at startup box is checked, the application will open a standard file dialog for selecting an existing project file. If the Always open new project box is checked, the application will open a standard file dialog for creating a new project file, each time the application is launched.

The radio buttons in the Default Directory Options area control the master directory used for creating new projects. It can be set to either the current working directory, or the directory indicated by an environment variable. The default environment variable is AB SofT and points to the Pro Fortran installation directory.

The radio buttons under Tool Options allow you to specify the type of notification you wish to receive from the various developer tools.
Check the **Auto-add module file paths** box to automatically pass the path of any module that has been incorporated into the project to the appropriate tool.

If the **Auto-add include file paths** box is checked, the path of any include or header file that has been incorporated into the project is automatically passed to the appropriate compiler, as if the **Set Include Paths** command in the **Configure** menu had been used to specify these directories.

The **Single project window** control toggles modes between single window projects and dual project and output window style projects in the Absoft Developer Tools Interface application.

Use the **Editor** menu and the **Connect via Services** menu to set the editor application to launch for source file editing, and the inter-process communication style used to talk with the specified editor, respectively. Services are currently only provided by the **TextEdit+** application or the **Absoft Editor**. Selecting **BBEdit** from the pop-up menu will allow complete two-way inter-process communication between the retail version of BBEdit for Mac OS X and Absoft Developer Tools Interface via a set of custom AppleEvents.

**Hide Tools**

Hide the Absoft Developer Tools Interface application.

**Quit Tools**

Exit the Absoft Developer Tools Interface application.

**File Menu**

**New**

Use this command to create a new project file. All file selections, include paths, and option settings are reset to default values. See **Working with Projects**, above, for details on creating a new project.

**Open**

This command is used to open a previously saved Absoft Developer Tools Interface project file with all file selections and option settings restored from that session.

**Open Recent**

Use the menu items on the submenu to open recently used project files.
Close

This command closes the current project. If you have a previously unsaved project already open, you will be prompted to save it before continuing.

Save

Use this command to save your current file selections and option settings so that they can be restored at a later time.

Save As

The Save As command saves your files selections and option settings to a different file than the one currently in use. This command is useful for making a copy of the current settings before making changes.

Revert To Saved

The Revert To Saved command restores the previously saved state of the current project file.

Page Setup...

Use this command to display the standard Page Setup dialog box for printing.

Print

The Print command prints the active output tab. You can print the entire window by pressing the control key while issuing this command.

New File

Use this command to create a new source file for the current project. The specified file will be added to the project file and opened using the default source file editor application.

Edit Menu

Undo

Use this command to undo the last editing action, if possible.

Redo

Use this command to reverse the last editing undo action, if possible.
View Previous

The View menu contains two commands: **Previous** and **Next**. If error or warning diagnostics are issued by any the developer tools in the Build tab of the output pane, you can use the Previous command to go to the previous error in the source file. Use the Next command to go to the next error in the source file. These menu items can also be used when the Search tab is active to go to the next match in the list.

Configure Menu

The primary menu for managing projects is the Configure Menu. This menu is used to specify the files that comprise the project, paths to search for include files, and the tool options used to build the application or library.

Add/Remove File(s)

Use this command to open the file selection dialog. Select the project’s source files, external object files, and libraries here by clicking either the Add Files... button to add additional files or the Delete Files button to remove the selected files. Note that the type of file displayed can be limited with the Add files as drop-down menu. If the source files you are adding will be processed with the VAST preprocessor (see Plug-ins later in this chapter), they must be selected and added to the project with the appropriate Add files as setting.

To add multiple files at once you can either shift-click an extended selection range of files or command-click to select a non-contiguous range of files from the standard file dialog. To add files that may otherwise be inaccessible, hold down the Shift or Option key while selecting the Add/Remove File(s) menu item.
The up and down pointing arrows above the **File(s) Added** list box are used to change the compilation order of the source files and to set the order in which external object files and library files are presented to the linker.

You can also add include and header files to the project. The paths to these files will automatically be supplied to the appropriate compiler if the **Auto-add include file paths** box is checked in Preferences.

The **Add file paths relative to “…”** check box can be used to toggle relative file paths on or off as desired. The default is to use file paths relative to the current project file. The current state of the check box will be used for all newly added files or paths.

**Set Include Paths**

Use this command to select additional directory paths to be searched for include and header files, in the same manner that source files are added to the project. This is the `-I` option on the command line and is used to supply a comma separated list of directory paths which are prepended to file names used with the Fortran `INCLUDE` statement or the C/C++ `#include` directive.

```
-Ipath[,path...]
```

The paths are prepended in the order presented with the `-I` option when the include file is not first found in the local directory and when it is not itself an absolute path (a full file specification). Paths supplied with the `-I` option are searched before the path specified with the `ABSOFT` shell variable.

The **Add file paths relative to “…”** check box can be used to toggle relative file paths on or off as desired. The default is to use file paths relative to the current project file. The current state of the check box will be used for all newly added files or paths.

**Set Project Options**

This command opens the **Options** dialog introduced earlier. The remaining tabs available in this dialog and the options they control will be discussed in detail in the following sections.

**File Options**

The **File Options** command allows you to set individual options for a specific project source file and tool. The menu command becomes active after you first select a source file in the **Files** tab. Option choices made will affect only the file name displayed in the options dialog title bar. Options set using this command will override any options set by the **Set Project Options** command.
Remove File Options

The **Remove File Options** command allows you to remove individual options for a specific file. The menu command becomes active after you first select a source file in the **Files** tab.

Remove All File Options

The **Remove All File Options** command allows you to remove all individual source file options for the entire project file.

MRWE Preferences...

If you have created an MRWE application (see **Working with Projects** earlier in this chapter), this menu selection will display the **MRWE Application Preferences** dialog. With this dialog you can modify how your application handles text and text saving. You can also modify the text font and size as well as whether your application pauses or exits when it has completed.

Set Default Options

Use this command to control the default options that are set when a new project is created. Several default settings were established when Absoft Pro Fortran was installed, including: **None**, **Debug**, and **Optimize**. Other common configurations are also supplied.

![Default Options Dialog](image)

**Set Default Options**

Click on the settings you want to use when a new project is created. You can edit the individual settings by clicking on the **Edit** button. The Options dialog (described earlier in this chapter) will appear giving you complete control over all of the general, compiler, linker options. You can also add your own default settings by clicking on the **New** button. Delete an option set with the **Delete** button.
Tools Menu

This menu provides access to the various developer tools you will work with: compilers, editors, debuggers, profilers, preprocessors, and makefiles.

Search

Use this command to search for strings of text in files.

Build

This is the primary command for building and updating your project. When you have finished adding the files to your project with the Add/Remove File(s) command in the Configure menu and have set all of the options, use this command to compile and link your program or library. This command is also used when you change options or edit files. It will recompile only those files that have been changed.

Rebuild All

Use this command to completely rebuild your project. All of the files will be processed regardless of whether they have been modified since the last build or not.

Update Dependencies

This command is used to force the Absoft Developer Tools Interface to rescan all source and include files for build dependencies.

Check Syntax

The Check Syntax command will check syntax only for the file currently selected in the Files tab of the project window.

Compile

The Compile command will compile the file currently selected in the Files tab of the project window.

Edit

Use this command to open for editing the file currently selected in the Files tab of the project window.
Preprocess

The **Preprocess** command is available only if you have the VAST parallel or VAST Altivec preprocessor installed. It executes the installed multi-processor preprocessor and leaves the intermediate file on your disk so that you can examine it.

Clean

Use this command to delete the executable and all of the object files in your project. It will also delete any .rsrcc files if your project incorporates resource description files (.r). For a complete list of what files will be deleted see the appropriate section of the associated makefile in the **Makefile** tab of the Output pane.

Stop

The Stop command terminates the currently executing tool.

Execute

Use this command to execute or run your program after it has finished building. Standard input can be redirected from a file if necessary in the Execute dialog. Standard Output for the executable will be shown in the Execute tab of the output pane by default.

Debug

Use this command to debug your program with the Absoft Fx debugger. See the chapter **Using the Fx Debugger** for more information.

Profile

The **Profile** command will run your program and produce execution time statistics. Note that you must first have produced a special instrumented executable by selecting the **Enable Profiling** option on the **Target** tab of the **Options** dialog.

Terminal

This command opens a Terminal and sets the current working directory to that of your project.

Generate Makefile

Use this command to create a makefile that can be used with the Absoft make utility, amake. See the chapter **Building Programs** for information on makefile commands and using amake.
Window Menu

The commands in this menu allow you to arrange the open windows in the Absoft Developer Tools Interface, manage the tool bar, and to bring a specific window to the front.

Hide Toolbar

Use this command to show or hide the toolbar.

Customize Toolbar...

This command displays the standard OS X toolbar dialog for customizing toolbars.

Tile Horizontally

Select this command to tile all open Absoft Developer Tools Interface applications windows in a horizontal fashion.

Tile Vertically

Select this command to tile all open Absoft Developer Tools Interface applications windows in a vertical fashion.

Project

Use this command to activate or re-open the Project window for the current project file. This menu item is only enabled if the Single Window Mode option has been turned off in the Preferences dialog.

The toolbar is displayed across the top of the Project window, or above the Project pane, if the Single Window Mode option is turned on. The toolbar provides quick mouse access to many tools used in the Absoft Developer Tools Interface.

Build  Build the program using the selected options.
Clean  Delete all current object files, intermediate files, and the executable file for the active program.
Run    Execute the program.
Debug  Debug the program.
Profile Profile the program.

Fortran User Guide
Search  Search for text or expression.
Stop     Halt execution of current developer tool.
Plus     Add new source file to project.
Minus    Remove selected file from project.

Output

Use this command to activate or re-open the Output window for the current project file. This menu item is only enabled if the Single Window Mode option has been turned off in the Preferences dialog.

The Output window, or pane, contains several tabs. Build shows the latest build or compile results for the current project file. Search shows the latest search results for the current project. Execute displays standard output when the program is launched from the Absoft Developer Tools Interface application. Makefile show the associated makefile for the current project document. Profile displays data files produced by the profiler.
Using the Compilers

Left  Display previous item in source file editor.

Right  Display next item in source file editor.

Help Menu

Tools Help

Use this command to view Absoft Developer Tools Interface application online help files.

Hide ToolTips

Use this command to disable pop-up tooltips for all windows and controls in the Absoft Developer Tools Interface for the remainder of the session.

Fortran User Guide
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 dialog. The Fortran 90/95 options fall into four categories: General, Compatibility, Format, and Common Block.

For quick reference, the options listed in the sections that follow are in the order in which they appear in the option tabs. Each option is listed with the corresponding option letter(s) and a short description. When an option is checked in the Absoft Developer Tools Interface application, the same letters will appear in the in the corresponding Options box.

**General - F95 Options**

When this subset of the F95 options tab is selected, options for controlling various aspects of compiling the Fortran 90/95 programs are available. Click on the box next to the option to add the option for compiling.

---

**F95 General Options**

---
Warning level (-znn)

Use the -znn 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 -znn 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.

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 100000x10nn 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 1024x10nn bytes.
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) (-Znn)

Use the -Znn 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 -znn option.

Use System Module Files (-SysModFiles)

Checking this box automatically adds the f90includes include directory to the search path for modules.

Set Module Paths (-p path)

The Absoft Fortran 90/95 compiler will automatically search the local directory for precompiled module files. Use this command to open the file selection dialog to add additional search paths. Paths specified are searched in a position dependent order. 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.

Note: there must be a space between the option (-p) and the path.

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.
**Procedure Trace (-B80)**

Specifying the `-B80` option will cause the compiler to generate code to write the name of the currently executing procedure to standard out. This option is useful for tracing program execution and quickly isolating execution problems.

**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.
Compatibility - F95 Options

When this subset of the F95 options tab is selected, options for enhancing the compatibility of Fortran 90/95 programs with other programming languages are available. Click on the box next to the option to add the option for compiling.

**Disable compiler directive (\(\texttt{-x\ directive}\))**

The \(\texttt{-x}\) option is used to disable compiler directive in the source file. \texttt{directive} may be any of the following:

- NAME
- FIXED
- FREE
- STATIC

See the section \textit{Absoft Fortran 90/95 Compiler Directives}, later in this chapter, for more information on using compiler directives in your source code.
INTEGER and LOGICAL sizes (-i)

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

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

Use the \texttt{-YCFRL=1} option to force the compiler to pass CHARACTER arguments in a manner that is compatible with \texttt{g77} and \texttt{f2c} protocols. Use the \texttt{-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.

Demote Double Precision to Real (-dp)

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

Promote REAL and COMPLEX (-N113)

Without an explicit length declaration, single precision REAL and COMPLEX data types default to thirty-two bits or four bytes (\texttt{KIND=4}) and sixty-four bits or eight bytes (\texttt{KIND=8}), respectively. The \texttt{-N113} option is used to promote these to their double precision equivalents (\texttt{KIND=8}). This option does not affect variables which appear in type statements with explicit sizes (such as \texttt{REAL \ (KIND=4)} or \texttt{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 \texttt{-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 \texttt{-s} options is used to allocate local variables statically, even if \texttt{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 \texttt{ALLOCATE} statement and local variables in recursive procedures.

Check Array Boundaries (-Rb)

Fortran User Guide
When the \texttt{-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 (**\texttt{-Rc}\ **)**

The \texttt{-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 (**\texttt{-Rs}\ **)**

When the \texttt{-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 (**\texttt{-Rp}\ **)**

Use \texttt{-Rp} compiler option is used to generate additional program code to insure that Fortran 90 style \texttt{POINTER} references are not null.

**Pointers Equivalent to Integers (**\texttt{-YPEI=0|1}\ **)**

This option controls whether or not the compiler will allow or accept 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:

```fortran
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:

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

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.

This subset of the F95 options tab displays options for controlling how Fortran 90/95 interprets the format of source files. These options allow Absoft Fortran 90/95 to accept older or variant extensions of Fortran source code from other computers such as mainframes.

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".

For Fortran User Guide
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 upper case. Set this option to LCS to emit names entirely in lower 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.

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 9)
\v Vertical Tab (VT, ASCII 11)
\xh[0][h] Hexadecimal, up to 2 digits
\o[o[0]] 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.

Common Block - F95 Options

This subset of the F95 options tab displays options for controlling how Fortran 90/95 treats common block names.

F95 Common Block Options

COMMON Block Name Prefix (-YCOM_PFX=string)

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

Fortran User Guide
COMMON Block Name Suffix (-YCOM_SFX=string)

The -YEXT_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={UCS | LCS})

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

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).

Assume Pointer Aliases Exist (-B19)

The -B19 option is selected when more than one symbolic name is used to reference a variable’s memory location. This can occur when pointers are used, when variables in COMMON are also passed as arguments, or when two dummy arguments are the same actual argument.

Note: Standard FORTRAN should not require this option, but the use of extensions may dictate its use. Performance loss should be expected when this option is selected.

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.

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 upper case (UCS), regardless of the how they appear in the source code. Set this option to LCS to fold variable names to lower case. Set this option to ASIS to force variable names to be processed exactly as they appear in the source program.
Symbol 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 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 options, which may appear after the -YALL_NAMES option to control an individual symbolic name type.

Cache Control (-YDEALLOC= {MINE | ALL | CACHE})

This option is used to control the underlying runtime memory management associated with the Fortran 95 ALLOCATE and DEALLOCATE statements. By default the runtime caches memory which has been deallocated (CACHE). Specifying MINE will cause all user allocated memory to be returned via a call to free(2) when a call to DEALLOCATE is executed. Specifying ALL will cause all user allocated memory to be returned via a call to free(2) and return any compiler allocated memory that has been cached. The tradeoff is minimizing memory use (ALL/MINE) versus speed of execution (CACHE).

Ignore CDEC$ directives (-YNO_CDEC)

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

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.

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. 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.

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

If you want to specify more than one compiler directive on a line, separate each directive with a comma.

**NAME Directive**

The `NAME` directive allows you to specify a case-sensitive external name in a Fortran program. You can use this directive, for example, when writing calls to C routines. 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[FORM] Directive**

The `FREE` or `FREEFORM` 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.
NOFREEFROM Directive

The `NOFREEFORM` directive is the same as the `FIXED` directive (see above) and specifies that the source code in the program unit is written in the fixed source form.

FIXEDFORMLINESIZE Directive

The `FIXEDFORMLINESIZE` directive specifies the line length for fixed-form source code. The format of the `FIXEDFORMLINESIZE` directive is:

```
!DIR$ FIXEDFORMLINESIZE:{72|80|132}
```

ATTRIBUTES Directive

The `ATTRIBUTES` directive can be used to apply special attributes to simplify passing variables between Fortran 90/95 and other languages. The format of the `ATTRIBUTES` directive is:

```
!DIR$ ATTRIBUTES attr-list::sym-list
```

where: `attr-list` is a comma separated list of attributes from the following set.

```
ALIAS
C
REFERENCE
STDCALL
VALUE
```

`sym-list` is a comma separated list of symbols.

The `ALIAS` attribute takes the form of

```
ALIAS:external
```

where: `external` is the external name of the procedure.

PACK[ON] Directive

The `PACK` or `PACKON` directive specifies that sequenced structure fields be aligned on byte even byte or word (four-byte) boundaries. The default is 1 (byte). The format for this compiler directive is:

```
!DIR$ PACK [= {1|2|4}]`

Fortran User Guide
The packing directives affect the current program unit being compiled (if there is one), or the next program unit (when there is no current program unit). The packing directive is reset to the default (PACKOFF) after the end of each program unit. A packing directive affects only derived-types found below the directive in the source code.

**PACKOFF Directive**

The PACKOFF directive returns structure field alignment to the default for the machine architecture which is alignment on the most efficient boundary for the data type. The format for this compiler directive is:

```
!DIR$ PACKOFF
```

**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 give you a great deal of control over the compilation and execution of FORTRAN 77 programs. Select the Set Project Options command in the Configure menu to access the Options dialog. The options for FORTRAN 77 fall into several categories: General, Control, Compatibility, Miscellaneous, Format, and Common Block.

For quick reference, the options listed in the sections that follow are in the order in which they appear on the F77 tab of the options dialog. Each option is listed with the corresponding option letter(s) and a short description. When an option is checked in the Absoft Developer Tools Interface application, the same letters will appear in the in the corresponding Options box.
General - F77 Options

These options control the general characteristics of the FORTRAN 77 components of the program being built.

F77 General Options

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.

**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.

**Warnings for Undeclared Variables (-N114)**

If the `IMPLICIT NONE` statement appears in a program unit, the compiler will issue an error diagnostic whenever it encounters an undeclared variable. If you specify the `-N114` option, the compiler will issue a warning diagnostic.

**Assume Pointer Aliases Exist (-B19)**

The `-B19` option is selected when more than one symbolic name is used to reference a variable’s memory location. This can occur when pointers are used, when variables in `COMMON` are passed as arguments, or when two dummy arguments are the same actual argument.

**Note:** Standard FORTRAN should not require this option, but the use of extensions may dictate its use. Performance loss should be expected when this option is selected.

**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.
Procedure Trace (-B80)

Specifying the -N option will cause the compiler to generate code to write the name of the currently executing procedure to standard out. This option is useful for tracing program execution and quickly isolating execution problems.

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 easily 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 of The FORTRAN 77 Program chapter of the Absoft FORTRAN 77 Language Reference Manual.
Control - F77 Options

When this subset of the F77 tab is selected, a dialog is shown for setting compiler directives:

![F77 Control Options](image)

**Compiler Directives (-Dname=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 77 Program* of the *Absoft FORTRAN 77 Language Reference Manual*. 
Compatibility - F77 Options

This subset of the F77 tab of the options dialog displays compatibility options for compiling FORTRAN programs. 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.

F77 Compatibility Options

Integer Sizes (-i2 and -i8)

Without an explicit length declaration, INTEGER and LOGICAL data types default to thirtytwo 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.
Vax/Mainframe Compatibility

The VAX and mainframe compatibility switches may be quickly turned on by clicking in the **VAX/Mainframe Compatibility** check box. Selecting this option is the same as specifying `-f -s -N3 -N51` from the command line.

**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 lowercase 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. The `-N1` compiler option causes the definition of variables initialized in `DATA` statements to be maintained after the execution of a `RETURN` or `END` statement. This option should be used for compatibility with VAX and other FORTRAN environments.

**Use Record Lengths in I/O (-N3)**

If the `-N3` compiler option is used, record length information will be included for sequential, unformatted files as if the “`BLOCK=-1`” specifier were implicitly included in all appropriate `OPEN` statements. See the chapter Input/Output and FORMAT Specification in the *Absoft FORTRAN 77 Language Reference Manual* for more information about the `BLOCK=-1` specifier. This option should be used for compatibility with VAX and other FORTRAN environments.

**RECL Defines 32-bit Words (-N51)**

If the `-N51` compiler option is used, the “`RECL`” specifier will be interpreted as the number of 32 bit words in a record for `UNFORMATTED`, `DIRECT` access files. Without this option, `RECL` defines the number of bytes in a record. This option should be used for compatibility with VAX and other FORTRAN environments.

**Folding to Upper Case (-N109)**
By default, the compiler considers upper and lowercase 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.

**Zero Extend INTEGER*1 (-N102)**

Normally, INTEGER*1 variables are sign extended when they are loaded from memory, providing for integers which range from -128 to 127. In order to provide compatibility with other implementations of FORTRAN, the -N102 option can be used to direct the compiler to zero extend these variables when they are loaded, making them essentially unsigned entities with a range of 0-255.

**Append underscore to names (-N15)**

Use of the -N15 option will cause the compiler to define SUBROUTINE and FUNCTION names with a trailing underscore. This option can be used to avoid name conflicts with the system libraries or to interface with other FORTRAN environments.
Miscellaneous - F77 Options

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

F77 Miscellaneous Options

Double Precision Transcendentals (-N2)

The -N2 option causes the compiler to use double precision or double complex transcendental intrinsic functions, overriding single precision and complex type specifications. This provides an extra bit of precision in some cases and is compatible with some C environments.
Sign Extend BYTE() & WORD() (-N7)

The -N7 compiler option causes the compiler to extend the sign of a value returned from the intrinsic functions BYTE and WORD. This option has no effect on the memory assignment statements described in the chapter Expressions and Assignment Statements of the Absoft FORTRAN 77 Language Reference Manual.

DATA Variables are Static (-N1)

The -N1 compiler option causes all variables initialized with DATA statements to be stored as static variables.

Evaluate Left-to-Right (-N20)

When two or more operators of equal precedence appear consecutively in an arithmetic expression, the -N20 forces the compiler to evaluate the operators from left to right (except for the exponentiation operators), regardless of whether it is the most efficient method or not.

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).

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 77 Program of the Absoft FORTRAN 77 Language Reference Manual for more information on the escape sequences that are supported.

Allows CASE Without DEFAULT (-N4)

By default, a run-time error is reported if a CASE DEFAULT statement is not present in a block CASE structure when a match is not found. The -N4 causes control of execution to be transferred to the statement following the END SELECT statement when a CASE DEFAULT statement is not present and no match is found, preventing such a run-time error.

Allows UNIT= Without FMT= (-N16)

If the -N16 compiler option is used, the format specifier FMT= may be omitted in an I/O statement when the unit specifier UNIT= is present.
Pack STRUCTURE Elements (-N33)

Normally, the fields in a STRUCTURE are aligned based on the standard for C. This may cause spaces to be left between structure elements and space to be added to the end of a structure. The -N33 option will cause structure fields to be “packed” — allocated with no space between them. Certain calls to Windows require structures to be packed in this manner. You can also use the conditional compilation directives $PACK and $PACKOFF to control the packing of individual structures (see the section Conditional Compilation Directives in the chapter The FORTRAN 77 Program of the Absoft FORTRAN 77 Language Reference Manual). The use of this option may cause misaligned storage locations.

Format - F77 Options

For compatibility with other FORTRAN environments and to provide more flexibility, the compiler can be directed to accept source code which 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 77 Program of the Absoft FORTRAN 77 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 77 Program of the Absoft FORTRAN 77 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.

IBM VS Free Form (-N112)

Use of the -N112 option causes the compiler to accept source code in the form specified by IBM VS Free 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.

Pad Source Lines (-N115)

Use the -N115 option to pad source lines to column 72 with spaces (or 132 with the -W option). By default, the compiler considers only the characters actually present in the source file. This option is useful when porting certain legacy programs that depend on the compiler reading source records as card images.

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.

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. The CONVERT specifier in the OPEN statement may be used to override this setting for individual files.
COMMON Block - F77 Options

Several options are available pertaining to the usage of common blocks in FORTRAN 77 source code.

Align COMMON Variables (-N34)

If a COMMON block is defined in a manner which causes a misaligned storage location, the -N34 option can be used to insert space to eliminate the misalignment. This option may invalidate your code if the same COMMON block is defined differently in different program units.

Set COMMON Block Name (-N22)

The -N22 option is used to change the scheme the compiler employs for generating global names for COMMON blocks. The default is to prepend the characters “_C” to the COMMON block name. This option causes the compiler to append a single underscore (_) instead. See also the -N110 option below.
Don’t Mangle COMMON Block Name (-N110)

The -N110 option prevents the compiler from mangling (changing) the global names for COMMON blocks. The default is to prepend the characters “_C” to the COMMON block name so that it does not conflict with other global names such as external procedure names. This option causes the compiler to emit the COMMON block name exactly as it appears in source.

Other F77 Options

The following options are not available on the F77 option pages but may be used with the command line interface or the make facility (See the chapter, Building Programs).

Suppress alignment warnings (-A)

The compiler normally will issue a warning message if a variable is aligned on a boundary that does not match its size. Misligned storage locations slow down memory access and can cause difficulty if you attempt to port the program to other computers. Use the -A option to suppress the listing of this type of warning only.

Check Syntax Only (-N52)

The -N52 option runs only the front end of the compiler. No object or executable files are created.

Character Argument Parameters (-N90)

Use the -N90 option to force the compiler to pass CHARACTER arguments in a manner that is compatible with g77 and f2c protocols. The default is to pass CHARACTER arguments in a manner that is compatible with Absoft Compilers on other platforms.

BLOCK DATA Code Section (-N116)

BLOCK DATA subprograms generate data initialization records only in the object file. Since they do not generate code references, they cannot effectively be used in libraries; the linker will not include them, as they resolve no references. Use the -N116 option to force the compiler to generate empty, stub routines that will allow the linker to use the associated data initialization records.
DATA treated as constants (-N5)

The -N5 compiler option enables the optimizer to propagate as constants those variables initialized in DATA statements that are not redefined. For example:

<table>
<thead>
<tr>
<th>Original code:</th>
<th>Becomes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBROUTINE DIVIDE(A) INTEGER A(50), B, C</td>
<td>SUBROUTINE DIVIDE(A) INTEGER A(50), B, C</td>
</tr>
<tr>
<td>DATA B, C/100, 10/</td>
<td>DATA B, C/100, 10/</td>
</tr>
<tr>
<td>DO I=1, 50</td>
<td></td>
</tr>
<tr>
<td>A(I) = A(I)/B-I*C</td>
<td>DO I=1, 50</td>
</tr>
<tr>
<td>END DO</td>
<td>A(I) = A(I)/100-I*10</td>
</tr>
<tr>
<td>RETURN</td>
<td>END DO</td>
</tr>
<tr>
<td>END</td>
<td>RETURN</td>
</tr>
</tbody>
</table>

This option is automatically turned on with the -O option for basic optimizations.

Function decomposition (-N18)

The -N18 compiler option causes intrinsic functions to be decomposed in line wherever possible. For example:

<table>
<thead>
<tr>
<th>Original code:</th>
<th>Becomes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>I = MOD (J, K)</td>
<td>I = (J-((J/K)*K))</td>
</tr>
</tbody>
</table>

This option is automatically turned on with the -O option for basic optimizations.

Evaluate Constant Functions (-N41)

Use the -N41 compiler option to direct the compiler to evaluate FORTRAN 77 intrinsic functions whose arguments are constant expressions. This option is automatically turned on with the -O option for basic optimizations.
Loop unrolling (-U and -h \textit{nn} and -H \textit{nn})

The Absoft Fortran 77 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 which execute a large number of iterations, while the benefit is small for loops which iterate only a few times. Due to this, only innermost loops are considered for unrolling. The -h \textit{nn} option will cause the compiler to unroll your innermost loops \textit{nn} times, where \textit{nn} is any power of two. The -H \textit{nn} option will cause the compiler to consider loops containing \textit{nn} or fewer statements for unrolling. When the -O option is used, the default is to only consider loops of a single line and unroll them four times. Using the -U option is equivalent to using -h 2 -H 10, causing innermost loops of ten or fewer lines to be unrolled twice. 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:

\begin{verbatim}
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=4,N-(MOD(N,4)),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
\end{verbatim}

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 \texttt{DO} loops, unrolling loops will invalidate your program.

Align \textsc{structure} fields to one byte boundaries (-N56)

The -N56 option will force structure fields to be completely “packed”. No padding will occur between fields. The use of this option may cause misaligned storage locations.

Align \textsc{structure} fields to two byte boundaries (-N57)

The -N57 option will force structure fields to be aligned to 2-byte boundaries. Fields beginning at odd addresses will be aligned to the next even address. The use of this option may cause misaligned storage locations for fields greater than 2 bytes in length.
Align STRUCTURE fields to four byte boundaries (-N58)

The –N58 option will force structure fields to be aligned to 4-byte boundaries. Fields not beginning at a modulo 4 address will be aligned to the next 4-byte boundary. The use of this option may cause misaligned storage locations for fields greater than 4 bytes in length.

Align STRUCTURE fields to eight byte boundaries (-N59)

The –N59 option will force structure fields to be aligned to 8-byte boundaries. Fields not beginning at a modulo 8 address will be aligned to the next 8-byte boundary.

C/C++ OPTIONS

The compiler options detailed in this section give you a great deal of control over the compilation and execution of C/C++ programs with either the Absoft compiler or the GNU compiler. The choice of specific compiler is made on the Build tab of the Options dialog described at the end of this chapter.

This section of the manual describes the options of the Absoft C/C++ compiler only. The GNU options in the Absoft Developer Tools Interface application are provided as a convenience; for more information on using the Mac OS X C/C++ compiler, see the Macintosh OS X manual page for cc.

Select the Set Project Options command in the Configure menu to access the Options dialog. The C/C++ options fall into three categories: General, Preprocessor, and Format. For quick reference, the options listed in the sections that follow are in the order in which they appear in the dialog. Each option is listed with the corresponding option letter(s) and a short description. When an option is checked in the Absoft Developer Tools Interface application, the same letters will appear in the Options box.
General – C/C++ Options

These options control some general characteristics of the C/C++ components of the program being built. For information on using specific options of the Mac OS X C/C++ compiler, see the Mac OS X manual page for `cc`.

![C/C++ General Options](image)

**Max Internal Handle (-T n)**

This option is used to change the number of handles used internally by the compiler. Under most conditions, the default value of 10000 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.

**Max Errors (-maxerr n)**

This option sets the maximum number of errors that can be emitted before the compiler aborts. Setting a reasonable number, such as 20, prevents the compiler from issuing hundreds of meaningless error messages when a brace or semi-colon is missing.
Diagnostic Messages

The level of diagnostic reporting can be controlled with the next seven options. The Absoft C/C++ compiler categorizes diagnostic messages as follows:

- **error**: non-recoverable syntactic error such as a missing brace or semi-colon
- **warning**: the usage or construction may be an error or non-portable, but is allowed on the assumption that you know what you are doing
- **informational**: the element is legal but may be the cause of a subtle bug
- **anachronism**: currently allowed construction which may not be supported in a future release
- **template**: show the point of instantiation of a template function when that instantiation caused an error

### Suppress All Warnings (-w31)

Checking this box suppresses the listing of all diagnostic warning messages. Various levels of warnings may be suppressed by checking the individual warning boxes as discussed below.

### Suppress Warnings (-w16)

Checking this box suppresses the listing of most warning messages. The only diagnostic warnings still reported are informational and anachronisms.

### Suppress Informationals (-w8)

This check box controls the listing of informational warning messages such as using a member name in a derived class that was defined in the base class, referencing a function which does not have a prototype, or defining a variable which is never used.

### Suppress Anachronisms (-w2)

Checking this box suppresses the listing of anachronism warning messages in C++ compilations such as a trailing comma in a parameter list, a temporary used for non-const reference, or the use of a count in `delete[]`. 
Suppress Template Warnings (-w1)

This option suppresses warnings issued when an error occurs while instantiating a template function.

Treat Anachronisms as Errors (-wp)

This check box causes the usage of C++ anachronisms to be treated as errors. This option should always be used when developing new code to avoid the use of deprecated features.

Treat All Warnings as Errors (-wabort)

This check box causes any warning condition to be treated as an error.

No Alias Optimizations (-N19)

The -N19 option is selected to prevent the compiler from performing address optimizations when pointer aliases are present.

Quiet (-q)

The Absoft C/C++ compiler normally displays information to standard output as it compiles an application. Enabling the -q option will suppress any messages printed to standard output. Errors will still be printed to the standard diagnostic output, however.

Verbose (-v)

Enabling the -v option will cause ACC to display the commands it is sending to the compiler and linker.

Procedure Trace (-N124)

Specifying the –N124 option will cause the compiler to generate code to write the name of the currently executing procedure to standard out. This option is useful for tracing program execution and quickly isolating execution problems.

Verbose Templates (-ptv)

This option causes the compiler to print a message to stderr whenever a template is instantiated.

Include Tree to Stderr (-H)

An include file tree will be printed to stderr when the -H option is enabled.

No inlines (-inline none)

This option prevents the compiler from inlining functions.
Exception Handling (-except on|off)

This option enables C++ exception handling. It allows you to use the try/catch/throw constructs in your program. All functions that are called directly or indirectly from a try block must be compiled with the -except on option. Only specify this option when you are actually using exception handling as there is a considerable overhead to its use.

Preprocessor – C/C++ Options

This subset of the C/C++ tab of the options dialog displays options for controlling the preprocessor stage of the C and C++ compilation process.
Defines (-D name[=value])

Use these edit and list text boxes to define the names and optional values of preprocessor variables. The -D option is used to define preprocessor 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.

Undefines (-U name)

Use these edit and list text boxes to undefine the names preprocessor variables. The -U option is used to undefine preprocessor variables from the command line.

Do not search standard system directories

Do not search the standard system directories for header files (-nostdinc). Only search directories provided by the –I option.
Preprocess files only

The **Preprocess files only** option (-E) directs the compiler to place the output of the C preprocessor in a specified file, or to **stdout**.

Format – C/C++ Options

This subset of the C/C++ tab of the options dialog displays options used to specify the language syntax format of the source files.

C/C++ Format Options

**C++ (-c++)**

Enabling the **-c++** option allows you to use the namespaces and runtime type information features ANSI C++ in your source code.

**ANSI C (-A)**

Set the **-A** option for source which conforms to the ANSI X3.159-1989 standard for the C programming language.
K and R C (-K)

The -K option is set with the radio button titled K and R C and should be used with older style Kernighan and Ritchie C. This is the default option.

LINKER OPTIONS

The linker options detailed in this section give you control over certain aspects of the program linking process. Select the Set Program Options command in the Configure menu to access the Options dialog.

For quick reference, the options listed in the sections that follow are in the order in which they appear in the dialog boxes. For information on using specific options of the Mac OS X Linker, see the Mac OS X manual pages for ld. When an option is checked, the same letters will appear in the Options box.

General - Link Options

There is only one set of options for the linker in the Options dialog:
Produce map file

Produce a load map, which lists all the segments and sections. The list will contain the address where each input file’s section appears in the output file and the section’s size.

Suppress Warnings

Use this option to suppress all linker warning messages.

Verbose

This option traces the progress of the linker, and indicates which symbols are being resolved.

Report Duplicate Symbols

Use this option to treat multiply defined symbols as warnings instead of errors. The first such symbol is used for linking. Other symbols of the same name may be accessed by local references.

Add Framework(s)... 

This button displays a file browser that allows you add system frameworks to your project. On a command line use \texttt{–framework name}.

Passing Options To The Linker

For ease of use within the Macintosh OS X environment, many of the options that are available to the system linker are also available to the \texttt{f77} and \texttt{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 \texttt{–S} or \texttt{–c} options are specified, all intermediate files (relocatable objects and/or assembly source) will be deleted. See the system documentation on \texttt{ld} for more information regarding these options. In brief, the options are as follows:

Undefine A Symbol (\texttt{–u})

Specifying the \texttt{–usymbol\_name} option will enter \texttt{symbol\_name} as an undefined symbol to the linker.

Linker Options (\texttt{–X} and \texttt{–Xlinker})

Use the \texttt{–X option} switch to pass an option directly to the linker. The FORTRAN 77 or Fortran 90/95 driver will pass \texttt{option} to the linker. If you want to pass an options which takes an argument, use the \texttt{–X option} twice.

For C and C++, the option is name \texttt{–Xlinker}. 
Other Link Options

For more information on using specific options of the Mac OS X linker, see the Mac OS X manual page for *ld*.

**PLUG-INS**

The **Plug-ins** tab of the Options dialog provides a way to integrate any additional tools that you may have purchased into your project. Select the **Set Project Options** command in the **Configure** menu to access the Options dialog.

Two types of plug-ins are available: libraries and preprocessors. The check boxes for these items are enabled only if the product was purchased and installed.
IMPORTANT: Source files that are processed by a VAST preprocessor must be added to the project using the VAST Preprocessor file type. Use the Add files as drop-down menu select the type of preprocessor as required. See the section describing Add/Remove File(s) discussed earlier in this chapter.

VAST

Pacific-Sierra Research Corporation develops the VAST pre-processors. They are pre-processor and library combinations for multi-processor and Altivec Mac OS X installations and can be used with FORTRAN 77 or Fortran 90/95 programs. Click the Settings tab to display a dialog for specifying additional preprocessor options.

Documentation for the VAST preprocessors is provided in electronic format on the Absoft Pro Fortran CDROM.

IMPORTANT: The VAST preprocessors do not support the use of automatic static storage (-s option) by either Absoft Fortran 90/95 or FORTRAN 77.
The VAST preprocessors can be invoked from a command line or within a custom makefile by using the supplied compiler drivers, pf90 and pf77, for Fortran 90/95 and FORTRAN 77, respectively.

**IMSL Library**

The IMSL check box controls the use of the Visual Numerics IMSL Math and Statistics libraries. Complete documentation is provided on the Pro Fortran CDROM. Place a check in this box to automatically link against the library.

**LAPACK Library**

These libraries contain the basic LAPACK and LAPACK90 libraries obtained from [www.netlib.org](http://www.netlib.org). LAPACK is used for the most common problems in numerical linear algebra including linear equations, linear least squares, eigenvalue, and singular value problems. LAPACK90 is the Fortran 90/95 interface for LAPACK. Source code for these libraries is supplied with Pro Fortran.

**UNIX Library**

The Unix library supplied routines compatible with those provided by Sun Microsystems and other Unix based Fortran compilers. Documentation on the routines in this library is available in the *Compatibility Libraries* manual supplied with Pro Fortran. Source code to all library routines is supplied.

**VAX/VMS Library**

The VMS library has a few additional routines with calling conventions that match VAX FORTRAN. Documentation on the routines in this library is available in the *Compatibility Libraries* manual supplied with Pro Fortran. Source code to all library routines is supplied.

None of the routines in this library are part of the ANSI FORTRAN 77 or Fortran 90/95 standards and should be used with caution if portability between platforms is a concern.
Altivec Libraries

This controls the use of optimized vector libraries for the PowerPC G4 processor. If your program is written in Fortran 90/95, uses the IMSL libraries, or uses the LAPACK libraries, special vector libraries will be linked to your program that take advantage of the single precision vector unit in the G4 processor. The command line equivalent is `-altivec`.

BUILD OPTIONS

Use this tab of the Options dialog to specify the tools that are used to build an application. The Tools section specifies the compilers, linker, make facilities, and any additional developer tools. The default path to these tools is the BIN directory of the main Absoft directory defined by the environment variable `ABSOFT` (e.g. `Applications\Absoft\Bin`). Paths specified by the environment variable `PATH` will also be searched. Use fully qualified paths for tools not residing in these directories.

Build Options

The Prebuild and Postbuild edit boxes in the Custom Build section can be used to specify files containing additional macros, rules, dependencies, and commands to be inserted into
the makefile. Examine the Makefile tab in the output window (discussed later in this chapter) for more information.

The check boxes at the bottom of this property page allow you to:

1. Discontinue the build process after the first error is encountered
2. Prevent the build process from updating file dependencies
3. Display file dependency scan progress
4. Specify that only the Fortran 90/95 compiler be used for all FORTRAN language compilations.
5. All normal C and C++ file extensions will be handled by the C++ compiler only
6. The GNU C/C++ compile will be used for all C/C++ files
7. Indicate that no default libraries names are to be supplied to the linker
8. Clean up output files using wildcard characters instead of deleting each output file individually.
CHAPTER 4

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 77 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 Macintosh OS X specific issues about porting code.

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

- f      Fold all symbols to lower case.
- 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 the Macintosh 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 the Macintosh. 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, Fx, 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. For a complete list of VAX extensions, refer to Appendix H. 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

Key Unsupported VAX FORTRAN Extensions

• Quad-precision floating point math
• Absoft Fortran 77 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 a group of compiler options collectively referred to as the “VAX compatibility options”, listed below. When using the Commando interface for the compiler, they may be invoked by a single check box.

- \texttt{f} Fold all symbols to lower case.
- \texttt{s} Force all program storage to be treated as static and initialized to zero.
- \texttt{N3} Include record length information for \texttt{SEQUENTIAL, UNFORMATTED} files.
- \texttt{N51} Interpret the \texttt{RECL} specifier as the number of 32-bit words in a record.

VAX-compatible time, date, and random number routines are available by linking with the file \texttt{vmslib.o} in the FLibraries folder. They are:

\begin{verbatim}
DATE subroutine returns current date as \texttt{CHARACTER*9}
IDATE subroutine returns current date as 3 \texttt{INTEGER*4}
TIME subroutine returns current time as \texttt{CHARACTER*8}
SECONDS subroutine returns seconds since midnight
RAN function returns random number
\end{verbatim}

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

\begin{verbatim}
/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 Absoft Fortran 77 automatically accepts an unlimited number of continuation lines
/CROSSREFERENCE no equivalent
/DEBUG -g to generate debugging information
/D_LINES -x to compile lines with a “D” or “X” in column 1
/DIAGNOSTICS append \texttt{\textgreater{}} \texttt{filename} to the \texttt{f77} command line to create a file containing compiler warning and error messages (type Option-\textgreater{} for the \texttt{\textgreater{}} character)
\end{verbatim}
Porting Code

/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 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 a symbol table dump may be generated with the -D option

/MACHINE_CODE -S to generate an assembly source file that can be assembled

/OBJECT no equivalent

/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

The tab size on the Macintosh may be different than the VAX. You can set the tab size for a file by pressing ⌘-Y while editing a file and typing the tab size for the file. For more information about tab size, see the section Tab Size later in this chapter.

Runtime Issues

If the program is having problems with I/O, make sure you are using the -N3 and -N51 options described in detail in sections Use record lengths in I/O and RECL Defines 32-bit words in chapter Using the Compiler.

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 which 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

Fortran User Guide
Key Unsupported VS FORTRAN Extensions

- Quad-precision floating point math
- 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 these compiler options:

- `-f` Fold all symbols to lower case
- `-s` Force all program storage to be treated as static and initialized to zero
- `-N3` Include record length information for `SEQUENTIAL`, `UNFORMATTED` files

Run-time Issues

If the program is having problems with unformatted I/O, make sure you are using the `-N3` option described in detail in the chapter Using the Compiler.

PORTING CODE FROM MICROSOFT FORTRAN (PC VERSION)

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)
- Unlimited number of continuation lines
- AUTOMATIC statement
- STRUCTURE, RECORD, UNION, MAP statements
- SELECT CASE statements
- DO WHILE loops
- INCLUDE statement
- OPEN statement displays standard file dialog when using `FILE=""`
- Conditional compilation statements
Key Unsupported Microsoft FORTRAN Extensions

- Metacommands
- MS-DOS specific intrinsic functions
- INTERFACE TO statement

Compile-time Options and Issues

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

- **-f** Fold all symbols to lower case
- **-s** Force all program storage to be treated as static and initialized to zero
- **-N3** Include record length information for SEQUENTIAL, UNFORMATTED files

If you use the Microsoft FORTRAN I/O specifier FORM='BINARY' to read and write binary sequential files with no internal structure, do not use the **-N3** option which includes record length within sequential, unformatted files.

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 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** -U for loop unrolling optimization; -R for loop invariant removal
- **$MESSAGE** no equivalent
- **$PACK** use $PACKON and $PACKOFF
- **$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

Fortran User Guide
If code ported from MS-DOS does not compile because of many errors, the end-of-line characters within the file may not match the return character (decimal 13 or Control-M) the Macintosh expects. To remove extraneous linefeeds from files transferred from MS-DOS which have both a linefeed and a return character, use the Replace... item in the Find menu and type control-J for the string to find and nothing for the string to replace.

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 the Macintosh. The -N15 option may be used to append underscores to routine names.

Key Supported Sun FORTRAN Extensions

- NAMELIST; 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

Key Unsupported Sun FORTRAN Extensions

- Quad-precision floating point math

Absoft has a compiler for Sparc-based SunOS systems. It has the same features and language extensions as Absoft Fortran 77. The compilers are 100% source-compatible.

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 the Macintosh OS X with PowerPC. 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.

PORTING CODE FROM INTEL 386/486/PENTIUM COMPUTERS

Absoft Pro Fortran is available for the Intel Pentium systems including Windows 95, Windows 98, and Windows/NT. It has the same optimizations and language extensions as Absoft Pro Fortran for the Macintosh with PowerPC. The compilers are 100% source compatible.

PORTING CODE TO/FROM OTHER MACINTOSH SYSTEMS

Language Systems Fortran

Absoft Fortran 77 and Language Systems Fortran share many extensions implemented in other compilers. In addition, Absoft Fortran 77 automatically supports most of the Language Systems Fortran specific language extensions. Below is a list of key Language Systems extensions that are supported and a list of those that are not supported. For a complete list of Language Systems extensions and their usage, refer to Appendix N.

Key Supported Language Systems Fortran Extensions

- STRING declaration statement
- POINTER declaration statement
- LEAVE control statement
- GLOBAL, CGLOBAL, and PBLOBAL statements
- CEXTERNAL and PEXTERNAL statements
- INT1, INT2, INT4, and JSIZEOF intrinsic functions

Key Unsupported Language Systems Fortran Extensions

- variables in FORMAT statements
- Language Systems Fortran compiler directives
Other Absoft Compilers

Over the past 15 years, Absoft has offered several different compilers for a number of Macintosh environments. This section outlines some of the differences between these products.

MacFortran       This 68000 compiler supported ANSI FORTRAN 77 and compiled programs directly from the Finder without using MPW. Although it lacked optimizations and support for many of the extensions in Absoft Pro Fortran for Macintosh with PowerPC, it compiled very fast and was easy to use.

MacFortran/020   This 68000 compiler was the same as MacFortran but it could also produce faster code for 68020 and 68030 systems that incorporated a floating point unit.

MacFortran II    This 68000 compiler is very similar to Absoft Pro Fortran for Macintosh with PowerPC. It supports many of the same optimizations and extensions, but is designed for 68000 based Macintoshes.

OTHER PORTING ISSUES

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

- Memory Management
- Tab Character Size
- Naming Conventions
- Numeric Precision
- File and Path Names
- Floating Point Math Control

Memory Management

Local variables and temporary values are stored in the stack frame. All other storage is allocated statically in the data and/or bss sections.

Dynamic Storage

Storage for variables local to a function or a subroutine is allocated in the stack frame. As a result, local variables are undefined when execution of a function or subroutine begins and become undefined again when execution terminates. This can cause difficulties in two areas.

First, problems may arise when porting Fortran applications from environments that statically allocate all memory; the application may except variables to retain their definition status across procedure references. However, it produces applications that
make more effective use of memory and provides the ability to call functions and subroutines recursively. The next section describes how to declare static storage space.

Second, the Macintosh OS X stack is limited to 512 KB and large arrays allocated in the stack frame may overflow the stack. You can increase the stack size with the `ulimit` command (ulimit is a bash command - the csh equivalent to `ulimit -s` is `limit stack`) to raise the stack size limit:

```
# ulimit -s
512
# ulimit -s 32768
# ulimit -s
32768
```

### Static Storage

There are three ways to define static storage in Fortran. The first two allow static variables to be defined selectively and are either placing them in `COMMON` blocks or using the `SAVE` statement. The third method, using the `-s` compiler option, forces all program storage to be treated as static. Static memory is allocated out of the data and/or bss sections and remains defined for as long as the application runs. In addition, all static storage will be initialized to zero when the application begins execution.

### Naming Conventions

Global names in Fortran include all procedure names and `COMMON` block names, both of which are significant to 31 characters. All global names in Absoft FORTRAN 77 are case sensitive unless one of the compiler character case options has been selected. All global names in Absoft Fortran 90/95 are upper case unless one of the compiler character case options has been selected. All other symbols 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 exactly as they are stated in the Fortran source code with a leading underscore (“_”) prepended. This is identical to how the C Programming Language represents symbolic names on Macintosh OS X.

If a FORTRAN 77 subroutine is defined as:

```
SUBROUTINE SUB("
  
RETURN
END
```
It will be defined in assembly language as:

```
.text
.globl _sub
_sub:
.
.
blr
```

COMMON Block Names

The convention in Absoft Pro Fortran us to precede the name given in the COMMON statement with the characters “_C”. BLANK common uses the characters _blank.

For example, the COMMON block declaration:

```
COMMON /the_block/ a, b, c
```

Eill produce the following assembler directive:

```
.comm _Cthe_block, 0x0000000c
```

File and Path Names

When the compiler encounters the Fortran INCLUDE statement, it takes the CHARACTER constant immediately following as a file name, searches for the file, and, if the file is found, copies its contents into the source file. If an absolute or relative path name is specified, the compiler will search only that path. If only a file name is given, the compiler will first look for the file in the current directory. It will then search any directory defined by the environment variable F77INCLUDES. Additional search paths may be specified with the –I compiler option.

Tab Character Size

The compiler assumes a standard tab size of eight spaces. This is the default for most editors. When the compiler encounters a tab character (ASCII 9) during compilation, it is replaced with the appropriate number of spaces for alignment to the next tab stop. By setting the environment variable TABSIZE, the tab size used by the compiler can be changed. The following command line for the Bourne shell will set the tab size for the compiler to four spaces:

```
TABSIZE=4
export TABSIZE
```
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 \texttt{RECL=} in an \texttt{OPEN} statement as 32-bit words instead of bytes.

-f90nlexts

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

\begin{verbatim}
$ONE
A(1)=1,2,3,4
$END
\end{verbatim}

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. 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 variable 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 and 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
### Porting Code

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FE_OVERFLOW</td>
<td>overflow</td>
</tr>
<tr>
<td>FE_INVALID</td>
<td>invalid argument</td>
</tr>
</tbody>
</table>

**NOTE** At the time this manual was written Macintosh OS X does not support floating exception handling.
CHAPTER 5

The Macintosh Runtime Window Environment

This chapter discusses the Macintosh Runtime Window Environment (MRWE), a special feature of Absoft Pro Fortran which gives your Fortran 90/95 and FORTRAN programs a Macintosh interface with windows and menus. Usually, when you want to create applications with windows and menus, you need to know how to use the Macintosh Toolbox and the Apple user-interface guidelines. When you create an application using MRWE, your program will automatically exhibit these features without the need for intensive Macintosh programming. MRWE is convenient and flexible to use.

USING MRWE

MRWE is a collection of pre-compiled Fortran routines contained in the library file \texttt{libmrwe.a} that you can link with your program. Applications generated by Absoft Pro Fortran are usually linked with this library unless explicitly select a different \texttt{Target Type}. The Fortran source code for MRWE is in the \texttt{FExamples} folder and can be used as an example of how to call Macintosh toolbox routines.
The Absoft Developer Tools Interface allows you to select the format of the executable file. An executable setting of MRWE Application (the default) will cause your program to be built as a stand-alone, double-clickable application with MRWE providing a window and menus interface. If your source code includes its own event loop or relies on the Carbon library even loop and calls Toolbox routines, you should disable MRWE by choosing Carbon Application for the target type.

The MRWE Window

When you launch a program that has been linked with the MRWE library, you will see a blank window on the screen and, above it, a menu bar, as shown below. This window is where MRWE displays standard output from Fortran programs and is where all standard input is typed. It looks like a terminal display, but allows you to see text that has scrolled off the screen.

Standard input and output are preconnected to the Fortran I/O units 5, 6, 9 and *. Any of these except * may be connected instead to a file by specifying the unit in an OPEN statement. After closing the file on that unit, the unit will be reconnected to standard input and output.
How Your Program And MRWE Work Together

When your program is launched, control is first passed to MRWE, which sets up a Macintosh environment with menus and a window. After the window appears, the application begins executing your Fortran program. By default MRWE is inactive until an I/O statement occurs. You cannot pull down menus or terminate the application at this point. Your Macintosh is completely dedicated to executing your Fortran program at maximum speed.

During any input and output statements (i.e. READ, WRITE, ENCODE, etc.), you can terminate the application with a Command-Period (\texttt{\textasciitilde}-). When the program is waiting for you to type standard input, you also have complete control of the menus and normal mouse actions. If you want the ability to terminate your application while it is calculating, or want to run other applications, select the \texttt{-N9} compiler option before compiling the program. The \texttt{-N9} compiler option turns on the \textit{timer interval} feature. Your program will be interrupted at timed intervals to check for Command-Period termination.

Working With Text in MRWE

The MRWE that appears is named after the application (truncated to 24 characters) with “output” appended. Text can be entered from the keyboard.

You can only type into the window when a READ statement is currently active for standard input (i.e. one of the preconnected units), and you can only type on the last line of the window. Any text that was already in the window when the READ statement began cannot be modified. You can, however, copy any text in the window to the clipboard. Also, when using the extended keyboard, the Home, End, Page Up, and Page Down keys can be used for scrolling. To insert an end-of-file character, type either Command-Return or Command-Enter.

The MRWE window has a text limit of 30K. When this limit is reached, the oldest text in the window is automatically deleted to make room for new output. If you prefer, you can specify that MRWE save all text to a file, indicating your preference when the project is created or by later modifying the \texttt{MRWEPrefs.r} resource file.

Using The MRWE Default Menus

An MRWE application automatically has several menus built-in. Following is a description of the commands performed by each item within these default menus.

File Menu

The File menu contains commands for saving and printing the text in the window.
Save (§S)

This command saves the MRWE text to a text file. The window title is used for the file name, and if the file already exists, it is overwritten. The window title is the name of the application with the word “output” appended. To automatically save you can indicate your preference when the project is created or by later modifying the MRWEPrefs.r resource file.

Save As…

This command saves the window text to a text file. It displays a standard file dialog prompting for a file name in which to save the text. The default file name is the name of the application (truncated to 24 characters) with “output” appended. If the file already exists, you will be prompted to overwrite it.

Page Setup…

Choosing this command displays the standard page setup dialog box for the printer.

Print Window…(§P)

To print the contents of the window to the printer, use this menu item. A dialog box is displayed showing the printing status. When a block of text is selected, this menu item changes to Print Selection… and prints just the selected text rather than the entire window.

Quit (§Q)

This command stops execution of the application by executing a STOP statement, thus closing all open Fortran units.

Edit Menu

The Edit menu contains the standard editing commands for cutting, pasting, and copying text.

Undo (§Z)

This command is always disabled unless a desk accessory is being used.

Cut (§X)

This command removes the selected text in the window and places it on the Clipboard. Text on the Clipboard may be pasted into other applications. Like other editing commands, this command is only available during a READ statement, and any text that was already in the window before the READ statement began cannot be cut.
Copy (⌘C)

The Copy command places the selected text from the window onto the Clipboard and leaves the text of the window unchanged. Text on the Clipboard may be pasted into other applications.

Paste (⌘V)

This command replaces the selected text of the window with the text on the Clipboard. If no text is selected in the window, the Clipboard text is inserted at the insertion point. Like other editing commands, this command is only available during a `READ` statement, and any text that was already in the window before the `READ` statement began cannot be pasted into.

Clear

This command clears the selected text. Like other editing commands, this command is only available during a `READ` statement.

Font and Size Menus

The font and the size menus list available fonts and font sizes. They allow you to change the appearance of text in a window to improve viewing. A checkmark beside an item in these menus shows the current setting for the frontmost window.

PROGRAMMING WITH MRWE

The sections that follow discuss features of MRWE that improve your program’s user interface and allow it to communicate with other programs. Menus can be easily customized and messages can be sent to other programs. Before discussing these features, a better understanding of the differences between a typical Macintosh application and a typical Fortran program is necessary.

Program Organization: Fortran VS. Macintosh

Usually a program has the following structure: first, it initializes its variables, possibly reading data from files, then it calculates, then it outputs the results. Some of these steps may be repeated, but the basic order of execution is linear (see the figure below). Once a program begins, there is typically very little interaction with the user.
The Macintosh, on the other hand, presents a very different, highly interactive environment to the user. The interface provided on a Macintosh between the program and the user allows for control and data selections to be made in a graphical manner as well as the more traditional textual methods. The Macintosh communicates user requests to the program through a mechanism known as an event that describes an action such as making a menu selection, clicking the mouse button, or typing a key on the keyboard. After the program receives an event it processes it, carrying out whatever action is required.

These actions are usually directed by an event loop, in which the program requests from the Macintosh any events pending for the program, carries out the directed action, and then repeats the process (see figure 8-5). When no events are pending, the program will carry out whatever other procedures it was designed to do. Some programs may simply be idle at this stage if they are designed entirely to react to user input (such as a drawing program).

Unfortunately, there is much more to Macintosh programming than this simple description implies, and it would be necessary to add a great deal of programming to even the simplest Fortran program to add a Macintosh look and feel to it. MRWE can add most of these features to your program and provides facilities that allow you to add additional features without delving into the details of Macintosh programming.
MRWE Event Loop Operation

As the preceding discussion indicates, an event loop is a necessary component of a Macintosh program, and one is built into the MRWE routines. To use it, you must provide subprograms that this event loop can call to respond to user requests. This section will describe how to use this built-in event loop.

Just one call to the subroutine mrwe_EventLoop will begin this mechanism, making your program behave automatically in response to user actions.

```fortran
EXTERNAL recur, term
CALL mrwe_EventLoop(recur, term)
```

where: `recur` is the name of a routine that will be called repeatedly while the event loop executes. This routine should execute quickly and return so that the program remains responsive to the user. Specify zero if you do not want to use a recurrent routine.

`term` is the name of a routine that you want to call when the program terminates. Specify zero if you do not want to use a termination routine.

When using these routines, remember to declare them as external in the subprogram in which `mrwe_EventLoop` is called. After calling `mrwe_EventLoop`, all events will be processed appropriately and, if a menu item is chosen, the subprogram you have associated with it will be called. Before making this call, you will probably want to set up menu items and tell MRWE which routine will respond to particular menu items. This process will be explained in the next section on Customizing Menus.

Note that `mrwe_EventLoop` never returns to the procedure from which it was called. It loops endlessly, processing the events sent by the user. Your program will only end when one of the following occurs: the user chooses to quit, one of your menu response routines issues a STOP statement, or there is a runtime error. [Caution: `mrwe_EventLoop` must not be called more than once.]
Customizing Menus

This section describes how to customize menus to suit your application. Menus and menu items can be added, deleted, or modified easily using MRWE.

Adding Menus

Menus can be added to your program at any time, but if you call mrwe_EventLoop you should probably install some menus and items first. This is because after calling mrwe_EventLoop, the user will control most of what the program does next. To install a menu item, call the function mrwe_AddMenu specifying the menu name, the item name, and the name of the menu response routine that should be called when the item is chosen:

```fortran
INTEGER*4 mrwe_AddMenu
EXTERNAL Routine
itemId = mrwe_AddMenu(menuName, itemName, Routine)
```

where: `menuName` is a CHARACTER expression that specifies the menu that the item will appear in. If the menu does not already appear in the menu bar, it will be created and will appear to the right of the other menus.

`itemName` is a CHARACTER expression that specifies the name of the item to be added to the bottom of the menu. If the item already exists, then only the `Routine` can be changed.

`Routine` is the name of the menu response routine to be called when the item is chosen, and should be declared external in the routine from which `mrwe_AddMenu` is called.

`itemId`, the function result, is an INTEGER*4 value. If an error occurs, `itemId` is -1; otherwise, it is a value that uniquely identifies the menu item. The `itemId` is a composite of the menu number and item number. If an EQUIVALENCE is done of `itemId` to an array of two INTEGER*2 elements, the first element is the menu number and the second element is the item number.

Special Characters
Certain characters have special meanings in menu item names. If you specify any of the characters listed below when creating the item, you must include those characters in exactly the same order when referring to the item by name.

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>When followed by the letters B, I, U, O, or S, sets the style of a menu item to bold, italic, underline, outline, or shadow, respectively.</td>
</tr>
<tr>
<td>/</td>
<td>When followed by a character, invokes the item from the keyboard by pressing the command key and the character together.</td>
</tr>
<tr>
<td>(</td>
<td>Disables the menu item. This is not recommended—instead, see “Enabling/Disabling Menu Items” later in this section.</td>
</tr>
<tr>
<td>!</td>
<td>When followed by a character, puts that character to the left of the item; also, see “Adding Checkmarks to Menus” later in this section.</td>
</tr>
<tr>
<td>-</td>
<td>Makes the item a line separator when it is the first character in the menu item name; typically used as (- to disable the separator.</td>
</tr>
</tbody>
</table>

Special Characters in Menu Item Names

**Menus and the READ statement**

The menu items that you install in the menu bar by calling `mrwe_AddMenu` will be available to a user of your program after your program calls `mrwe_EventLoop` and as soon as your menu response routines have finished responding to any menu items that have been chosen. The user will also be able to pull down the menus while your program is in the middle of executing a READ statement on standard input (units *, 5, 6, or 9), but the menu items will be disabled. This is because menu response routines must not execute Fortran I/O statements while the program is in the middle of a READ statement on Standard Input.

If you know that your menu response routine and every routine that it calls does not execute any Fortran I/O statements, you can enable the menu item even during a READ on Standard Input. To do this, add a @ character to the beginning of the name of the menu item when you install it with `mrwe_AddMenu`. 
Removing a Menu or Menu Item

To remove a menu item or an entire menu, use the function `mrwe_RemoveMenu`:

```fortran
INTEGER mrwe_RemoveMenu
iresult = mrwe_RemoveMenu(itemID)
```

where: `itemID` is an INTEGER*4 expression that specifies the item to be removed from the specified menu. It is the same value that was returned by `mrwe_AddMenu` when the item was first installed. If you specify 0 for the item component of `itemID`, the entire menu will be removed.

`iresult` is an INTEGER. The function will return a value of 0 if successful, or -1 if there was an error.

**Important:** When a menu item is removed, any items that come after it will be given new item numbers. For example, in a menu with four items: if item #2 is removed, then item #3 becomes item #2, and item #4 becomes item #3. Any variable containing an `itemID` for an item after the one removed will no longer indicate the proper item. Note also that while item numbers are renumbered when an item is removed, menu numbers do not change when a menu is removed.

Menu Response Routines and `mrwe_DoMenu`

When you choose a menu item while running an MRWE application, the routine that performs that function (the “response routine”) is executed. This type of routine should be declared as:

```fortran
INTEGER FUNCTION Routine(itemID)
INTEGER*4 itemID
```

where: `itemID` is the same value that was returned by `mrwe_AddMenu` when the item was first installed. This value can be helpful if more than one item uses the same response routine.

Your response routine is called by the function `mrwe_DoMenu`, which is normally called by `mrwe_DoMouseDown` whenever a mouse-down event occurs with the cursor in the menu bar at the top of the screen. The `mrwe_DoMenu` function can also be called explicitly at any time during the program when you want to execute the routine associated with a menu command. The `mrwe_DoMenu` routine will return one of the following two results: the result of the response routine, or -1 if the item could not be found. Call the `mrwe_DoMenu` function as follows:

```fortran
INTEGER mrwe_DoMenu
iresult = mrwe_DoMenu(itemID)
```
where: `itemID` is an INTEGER expression. It is the same value that was returned by `mrwe_AddMenu` when the item was first installed. It is a composite of the menu and item numbers.

`iresult` is the INTEGER returned by the response routine `mrwe_DoMenu` called. If you declare your response routines as functions, they can return a value through `mrwe_DoMenu`. If you declare them as subroutines, you can call `mrwe_DoMenu` as a subroutine.

### Adding Checkmarks to Menus

Checkmarks can be placed beside a menu item after it is created by calling the routine `mrwe_MenuItemCheckmark`:

```fortran
CALL mrwe_MenuItemCheckmark (itemID, state)
```

where: `itemID` is an INTEGER*4 expression that specifies the menu item where the checkmark will be placed. It is the same value that was returned by `mrwe_AddMenu` when the menu item was installed.  
`state`, is a LOGICAL expression. If `state = .TRUE.`, a checkmark will be placed next to the item; if `state = .FALSE.`, the checkmark will be removed.

### Enabling/Disabling Menu Items

The function `mrwe_MenuItemEnable` is used to enable or disable a menu item:

```fortran
INTEGER mrwe_MenuItemEnable
iresult = mrwe_MenuItemEnable(itemID, state)
```

where: `itemID` is an INTEGER*4 expression that specifies the menu item which will be enabled or disabled. It is the same value that was returned by `mrwe_AddMenu` when the item was first installed.  
`state` is a LOGICAL expression. If the `state = .TRUE.`, the menu item will be enabled; if `state = .FALSE.`, the item will be disabled.

`iresult` is an INTEGER. The function will return a value of 0 if successful, or -1 if there was an error.

When installing an item by calling `mrwe_AddMenu`, you can also disable it by adding the special character ( before the item name. However, this method will not work correctly with the `mrwe_MenuItemEnable` function and is not recommended. Instead, install the item as enabled, and then explicitly disable it using `mrwe_MenuItemEnable`. 

`Fortran User Guide`
Further Information About Menus

For further information on menu features, see Inside Macintosh I, “The Menu Manager”. To access these features, use the menu number and item number in the itemID returned by mrwe_AddMenu. If the routine expects a MenuHandle, this INTEGER*4 value can be derived using the function GetMHandle, passing to it the menu number as a VAL2:

```fortran
RECORD /ItemID/ sdItem ! defined in the file MRWE.inc
EXTERNAL DefaultSettings
INTEGER*4 h_menu

sdItem.both=mrwe_AddMenu('Settings','Use Defaults',DefaultSettings)
h_menu = GetMHandle(VAL2(sdItem.menu))
CALL SetItemMark(VAL4(h_menu), VAL2(sdItem.item),VAL1('•'))
```

Launching OTHER APPLICATIONS

From your Fortran program you can launch other Macintosh applications using the function mrwe_LaunchApp as defined below. You can also make an application that is already running become the front-most, active application. [Note: This function can only be used with System 7.]

```fortran
INTEGER mrwe_LaunchApp
iresult=mrwe_LaunchApp (name,front,useMin)
```

where: `name` is a CHARACTER expression that specifies the application to be launched and can be expressed in one of the following three ways:

1) `process=programName` Identifies the application to be launched; specify the path if necessary.

2) `creator=XXXX` Launches an application using its creator type. However, if there is more than one program with the same creator type, the system will launch the first one it finds.

3) `self` Allows an MRWE application to refer to itself. You can use this to bring your program to the front (see example below).
The Macintosh Runtime Window Environment

front is a LOGICAL expression that determines whether the specified application should be the frontmost application. If front=.TRUE., the application will become the frontmost application. If the application is not currently running, it will be launched in front of all other applications. If the application is already running, it will become the frontmost application. If front=.FALSE., the position of the application will remain unchanged. The menu bar at the top of the screen shows the menus of the frontmost application.

useMin is a LOGICAL expression that refers to the amount of memory needed to run the program. If useMin=.TRUE., then the program will be launched if the minimum amount of memory specified is available. If useMin=.FALSE., then the program is launched only if the preferred amount is available in the system. These values can be viewed and changed in the Finder by selecting the application’s icon and choosing Get Info in the File menu. If the program is already running, useMin will be ignored.

iresult is an INTEGER error code indicating problems that may prevent launching the application. (See the table later in this chapter for a list of error codes.)

The following statement shows how to launch ResEdit:

    CALL mrwe_LaunchApp('creator=RSED',.TRUE.,.FALSE.)

ResEdit is specified by using its creator ID 'RSED'. The value of front is .TRUE., so the program is launched in front of all other running applications. Because useMin = .FALSE., ResEdit will only be launched if the preferred amount of memory is available.

The next example shows how to make your Fortran program the frontmost application:

    CALL mrwe_LaunchApp('self',.TRUE.,.FALSE.)

Apple Events

Apple Events is a System 7 feature that allows Macintosh applications to communicate with each other. By using this feature, you can have your applications send messages to and receive messages from other applications. MRWE allows you to use both the standard messages defined by Apple Computer and to define your own messages. This section will describe how these events (or messages) are classified, what information you must specify to transmit an event successfully, and examples of how to send and receive various events.

Apple Event Target
The application to which you send an Apple Event is called the *target* and is specified similarly to the way *name* is specified when launching an application (see “Launching Other Applications” above):

<table>
<thead>
<tr>
<th>Target Specification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>process=programName</td>
<td>Sends the message to the application whose name is specified; include the path if necessary.</td>
</tr>
<tr>
<td>creator=XXXX</td>
<td>Sends the message to an application whose Creator ID is specified.</td>
</tr>
<tr>
<td>sender</td>
<td>Sends the message to the application that sent the last Apple Event received by your application.</td>
</tr>
<tr>
<td>browser</td>
<td>Shows the Browser dialog box, allowing the user at runtime to select any currently running application to send the Apple Event to.</td>
</tr>
<tr>
<td>browser=prompt</td>
<td>Shows the Browser dialog box with a custom prompt message instead of Choose a program to link to.</td>
</tr>
<tr>
<td>self</td>
<td>Sends a message to itself.</td>
</tr>
<tr>
<td>same</td>
<td>Sends an event to the same target to which the most recent event was sent.</td>
</tr>
</tbody>
</table>

Target Specification for Apple Events

**Apple Event Class and ID**

The *class* and *ID* of an event indicate the action to be carried out—what the sender wants the target application to do. The *class* is a category of events that perform related functions. For example, Finder events make the Finder perform specific tasks such as restarting or shutting down the computer. The *class* and *eventID* arguments are specified with CHARACTER*4 values. For Finder events, the class is indicated by the abbreviation FNDR. The event ID identifies the specific event being sent, such as “rest” for *restart*. (See the table later in this section for a list of common Apple Events).

**Extra Information in an Apple Event**

Some Apple Events require additional information in order to complete a message. For example, an Open Document or Print Document event must specify *which* document to act on; this is called extra information. Events can be classified according to the kind of extra information required, if any. The following table shows the three kinds of extra information supported by MRWE and how they are represented as INTEGER values.
<table>
<thead>
<tr>
<th>Kind</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>signal</td>
<td>0</td>
<td>A signal event needs no extra information. It signals a request to do something or indicates that something has been done. For example, an application can tell another application to quit.</td>
</tr>
<tr>
<td>document</td>
<td>1</td>
<td>A document event operates on a document. The name of the document being operated on must be specified. For example, one application can tell another application to print a document.</td>
</tr>
<tr>
<td>text</td>
<td>2</td>
<td>A text event includes a character string. Two MRWE applications can pass information back and forth using text events.</td>
</tr>
</tbody>
</table>

Kind of Extra Information for an Apple Event

**Typical Apple Events**

Apple Computer has defined many Apple Events and has specified the class, event ID, and kind of extra information associated with each event. An application that supports the Apple Events feature can send or receive at least four basic events called Required Apple Events: Open Application, Open Document, Print Document, and Quit Application. MRWE supports some of the standard Apple Events and also allows you to define your own events. It allows you to send these events to other programs and to specify routines that will respond if one of these events is received by your application (see Receiving Apple Events later in this section). Required Apple Events is a subset of the Core class of Apple Events—the latter is identified by the abbreviation “aevt”.

Following is a table of common Apple Events, their class, event ID, and extra information specifications, and a brief description of what they do.
<table>
<thead>
<tr>
<th>Event name</th>
<th>Class</th>
<th>Event ID</th>
<th>Kind of extra info</th>
<th>What the event does</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open Document</td>
<td>aevt</td>
<td>odoc</td>
<td>document</td>
<td>Tells the target application to open the specified document.†</td>
</tr>
<tr>
<td>Print Document</td>
<td>aevt</td>
<td>pdoc</td>
<td>document</td>
<td>Tells the target application to print the specified document.†</td>
</tr>
<tr>
<td>Open Application</td>
<td>aevt</td>
<td>oapp</td>
<td>signal</td>
<td>Tells the receiving application it has just been opened; usually causes an untitled document to appear.†</td>
</tr>
<tr>
<td>Quit Application</td>
<td>aevt</td>
<td>quit</td>
<td>signal</td>
<td>Tells the receiving application to begin the process of quitting.</td>
</tr>
<tr>
<td>Do Script</td>
<td>misc</td>
<td>dosc</td>
<td>text</td>
<td>Tells an application to perform actions specified in a scripting language.</td>
</tr>
<tr>
<td>Restart</td>
<td>FNDR</td>
<td>rest</td>
<td>signal</td>
<td>Tells the Macintosh to restart.*</td>
</tr>
<tr>
<td>Shutdown</td>
<td>FNDR</td>
<td>shut</td>
<td>signal</td>
<td>Tells the Macintosh to shut down.*</td>
</tr>
<tr>
<td>Sleep</td>
<td>FNDR</td>
<td>slep</td>
<td>signal</td>
<td>On a portable Macintosh, puts the computer in low-power mode.*</td>
</tr>
<tr>
<td>Empty Trash</td>
<td>FNDR</td>
<td>empt</td>
<td>signal</td>
<td>Tells the Finder to empty the trash.*</td>
</tr>
<tr>
<td>About</td>
<td>aevt</td>
<td>abou</td>
<td>signal</td>
<td>Tells the Finder to show its “About This Macintosh” window.*</td>
</tr>
<tr>
<td>Show Clipboard</td>
<td>FNDR</td>
<td>shcl</td>
<td>signal</td>
<td>Tells the Finder to show its clipboard window.*</td>
</tr>
</tbody>
</table>

† When an application is first launched, it will immediately receive one of these three events. Open Application is only sent when neither Open Document nor Print Document will be sent upon launch. When MRWE launches an application, it automatically sends an Open Application event.

* These Apple Events only work if sent to the Finder on the Macintosh running the application. Also, they all work the same as the corresponding Finder menu items.

Common Apple Event Specifications

Sending Apple Events

An Apple Event can be sent from one application to another as follows:

```fortran
INTEGER mrwe_SendAE
irez=mrwe_SendAE(class,eventID,extraKind,target,extraInfo)
```

where: `class`, a CHARACTER*4 expression, is a category of related events, such as `FNDR` for Finder events.

Fortran User Guide
eventID, a CHARACTER*4 expression, identifies which event to send, such as 'odoc' for an Open Document event or 'quit' for a Quit Application event.

extraKind, an INTEGER, specifies the kind of extra information, if any, that needs to be sent along with the event. Its value can be 0 for a signal event, 1 for a document event, or 2 for a text event.

target, a CHARACTER expression, identifies the application that the event is sent to.

eextraInfo, a CHARACTER expression, is the extra information needed for certain kinds of Apple Events.

iresult is an INTEGER error code indicating problems that may occur while sending the event, such as an inability to send the event or the target application’s inability to receive the event properly. (See the table for a list of error codes.)

For example, you could send an Apple Event from an MRWE application to any application which understands Apple Events, and tell it to print a certain text file. To have your program tell Microsoft Word™ to print the document called results, you would use the following statement:

\[
\begin{align*}
\text{ires} & = \text{mrwe\_SendAE('aevt','pdoc',1,'creator=MSWD','results')} \\
\end{align*}
\]

It only takes one function call to send an event. The first argument identifies the target application (Microsoft Word), the second and third arguments define the class and event ID, and the fourth and fifth arguments indicate that the event is a document event and specify the name of the file.

Receiving Apple Events

MRWE provides a mechanism for receiving Apple Events, but it is up to you to determine how your program will react when it receives a event. When using Apple Events, you must specify routines that respond to particular events, just as you would specify a response routine when adding menu items (see “Customizing Menus” earlier in this chapter). For example, you may write one routine to respond to Open Document events and another to respond to Print Document events. You need to tell MRWE which events you want to respond to and the name of the routine that will handle the event. You do this by installing an Apple Event response routine using the function mrwe\_AddAppleEvent.

\[
\begin{align*}
\text{LOGICAL mrwe\_AddAppleEvent} \\
\text{EXTERNAL Routine} \\
\text{iresult=mrwe\_AddAppleEvent(class,eventID,extraKind,Routine)} \\
\end{align*}
\]

where: class, a CHARACTER*4 expression, identifies the category of the event to be received.
eventID, a CHARACTER*4 expression, identifies which event within the class will be received, such as 'odoc' for an Open Document event or 'quit' for a Quit Application event.

extraKind, an INTEGER, specifies the kind of extra information, if any, that will be received along with the event. Its value can be 0 for a signal event, 1 for a document event, or 2 for a text event.

Routine is the name of the response routine MRWE will call when the event is received, and should be declared external in the routine from which mrwe_AddAppleEvent is called.

lresult is a LOGICAL value indicating whether the response routine was installed. If lresult=.TRUE., the routine was installed properly. If lresult=.FALSE., the response was not installed because Apple Events are not supported by System 6.

For example, a routine that handles Print Document events could be installed with the following call, where PrintDoc is the name of a function that will be called every time the application receives a Print Document event.

    lres = mrwe_AddAppleEvent('aevt', 'pdoc', 1, PrintDoc)

Response routines should be declared like this for signal events:

    INTEGER FUNCTION Routine(class, eventID)
    CHARACTER class*4, eventID*4

or like this for a response routine that responds to document or text Apple Events:

    INTEGER FUNCTION Routine(class, eventID, extraInfo)
    CHARACTER class*4, eventID*4, extraInfo(*)

For a document event, extraInfo is the path and filename of the document to act on. For a text event, it is a text string. The INTEGER value that your function should return is an error code. It should return 0 if everything went well, or one of the error codes shown in error code table to indicate to the sender that a problem occurred while responding to the event. There is one response routine that is built into MRWE to handle Quit Application events, called mrwe_QuitAE. Normally, there should be no need to change this routine.

When your application is launched by the Finder under System 7, the Finder will send either an Open Application event, an Open Document event, or a Print Document event. The following table shows several methods you can use to launch an application and the corresponding Apple Event your application will receive:
<table>
<thead>
<tr>
<th><strong>Method of launch</strong></th>
<th><strong>Apple Events received</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Double-click the application icon or single-click it and choose Open from the File menu.</td>
<td>An Open Application event.</td>
</tr>
<tr>
<td>Double-click a document icon.</td>
<td>An Open Document event, if successful. The creator of the document must match that of the application (Mrwe by default). Also, if there is more than one application with the same creator, the application that is launched may not be the one you expect.</td>
</tr>
<tr>
<td>Select the application icon and the icons of one or more documents, then double click on any selected icon or choose Open in the File menu.</td>
<td>An Open Document event for each document whose creator matches the application’s creator.</td>
</tr>
<tr>
<td>Select the icons of one or more documents, then drag them to the application icon.</td>
<td>An Open Document event for each document.</td>
</tr>
<tr>
<td>Select the application icon and the icons of one or more documents, then choose Print in the File menu.</td>
<td>A Print Document event for each document whose creator matches the application’s creator. Immediately after the Print Document events, a Quit Application event will be received.</td>
</tr>
<tr>
<td>Launch the application from another application.</td>
<td>Probably an Open Application event, although it depends on the application doing the launch.</td>
</tr>
</tbody>
</table>

**Apple Events Received Upon Launch**
Error Codes Returned from Apple Event Routines

Apple Event routines return an INTEGER error code. If the value returned is 0, no error occurred; if it is not 0, an error occurred either in sending the Apple Event, or in the target application receiving the event. Following is a list of error codes and their meanings:

<table>
<thead>
<tr>
<th>Error code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No error occurred in either sending or receiving the event.</td>
</tr>
<tr>
<td>-108</td>
<td>Not enough memory available to complete the action. If sending an Apple Event to an application that is not currently running, the preferred amount of memory is not available in which to launch it.</td>
</tr>
<tr>
<td>-128</td>
<td>The user of the application pressed Cancel in the Browser dialog.</td>
</tr>
<tr>
<td>-1701</td>
<td>The wrong kind was specified for an Apple Event, or the extra information in the event could not be extracted properly.</td>
</tr>
<tr>
<td>-1712</td>
<td>No response from the target application within the limit of seven seconds.</td>
</tr>
<tr>
<td>-1717</td>
<td>No handler installed to handle Apple Events of that class and eventID. Handlers should be installed before the call to mrwe_EventLoop is made.</td>
</tr>
<tr>
<td>-5553</td>
<td>The operating system does not support the requested feature. For example, System 6 does not support either Apple Events or using mrwe_LaunchApp to launch other applications.</td>
</tr>
</tbody>
</table>

Other Examples of Apple Events

Following are some additional examples of common events you might wish to transmit between applications:

Sending a request to the Finder

If you are running lengthy processes overnight or over the weekend, you may want to turn off your machine after all processes are completed. To do this, simply send the Finder a Shut Down Apple Event. This will turn off the computer if the Shut Down item in the Finder's Special menu can turn off the system. When sending an Apple Event to the Finder, you must properly specify the path of the System Folder that contains the Finder. For example, if your hard disk is named “HD”, you could send a Shut Down event with the following statement:
Using other standard Apple Events

To cause another application (which supports Apple Events) to open a particular file, you would send that application an Open Document event specifying the file:

\[
\text{ires} = \text{mrwe\_SendAE('aevt','odoc',1,'creator=MSWD','test')}
\]

To allow your application to respond to Open Document events, install a response routine to handle Open Document events:

\[
\text{ires} = \text{mrwe\_AddAppleEvent('aevt','odoc',1,OpenDocProc)}
\]

Sending information between MRWE applications

One application could send a document event to another application telling it to process the file in a certain way. When the second application finishes, it can send a signal event back to the sender to indicate completion.

Another way to send messages is through text events. The class and eventID are not limited to the examples shown in the table earlier in this section. They can be defined as any four characters you choose to distinguish different events. Following is an example of how to send information through text events. The sending application would include:

\[
\begin{align*}
\text{CHARACTER*30 internalFile} \\
\text{WRITE (internalFile,*) a,b,c} \\
\text{ires=mrwe\_SendAE('mrwe','guas',2,'process=SecondApp',internalFile)}
\end{align*}
\]

and the receiving application would include:

\[
\begin{align*}
\text{EXTERNAL GuasProc} \\
\text{ires=mrwe\_AddAppleEvent('mrwe','guas',2,GausProc)} \\
\text{END} \\
\text{INTEGER FUNCTION GausProc(class,type,internalFile)} \\
\text{CHARACTER*4 class,type} \\
\text{CHARACTER*(*) internalFile} \\
\text{READ (internalFile,*) a,b,c}
\end{align*}
\]

Scripting

Scripting refers to text commands that an application can interpret and execute. An Apple Event called Do Script sends text commands from one application to another that has a scripting language, allowing the scriptable application to be controlled. It is not enough for an application to have a scripting language; it must also accept the Do Script Apple Event.
ires = mrwe_SendAE('misc','dosc',2,'creator=WILD','Answer "Press ' & 'OK to flash the screen." with "Flash" '//CHAR(13)//'Flash')

**Further Information About Apple Events**

MRWE supports sending and receiving only certain kinds of Apple Events. This is because the information carried in an Apple Event can be very diverse and complex. Some programs may use Apple Events to carry information in a form MRWE cannot recognize. To learn more about Apple Events, consult *Inside Macintosh VI*, “Apple Events Manager” (Chapter 6), and the *Apple Events Registry*, published by Apple Computer, Inc.

**Creating Multiple Windows**

You can easily open additional windows in MRWE. While the standard window described at the beginning of this chapter shows characters in the standard input and output units, additional windows can show characters written to other units. By specifying `ACCESS="window"` in an `OPEN` statement, you open a window connected to the unit specified in the `OPEN` statement. Any read or write functions associated with that unit will appear in a window. The `ACCESS` specifier can be expressed in the following three ways:

```
ACCESS="window"
ACCESS="window, height, width"
ACCESS="window, height, width, top_edge, left_edge"
```

where: `height` and `width` define the dimensions of the window in pixels (not including the title bar of the window). Both arguments should be expressed as positive integers.

`top_edge` and `left_edge` define the distance in pixels from the top of the screen to the top of the window and from the left edge of the screen to the left edge of the window, respectively. Both arguments are expressed as signed integers (either positive or negative).

The following is a simple example of how to use the `OPEN` statement to create an additional window in an MRWE application:

```
OPEN (7,FILE='Second Window',ACCESS='window, 200, 360')
WRITE (7,*) 'This text will appear in the second window'
READ (7,*) ! This is like a PAUSE statement, but in the new window
CLOSE (7)
```

The first line creates a window titled 'Second Window' and connects it to unit 7. The size will be 200 pixels tall and 360 pixels wide. Since the location of the window is not specified, the window will open using the MRWE default for window location. The
The Macintosh Runtime Window Environment

second line writes the statement to the window. The READ statement in the third line will function like a PAUSE statement and wait for the user to press the Return or Enter key. When either key is pressed, the program will continue to the fourth line, which will close the window.

Showing Alert Messages

Macintosh applications often announce messages when a problem occurs by showing an Alert dialog box:

![Alert Dialog Box Example](image)

Show Alert Box Example

To show an Alert dialog box, use:

```fortran
CALL mrwe_ShowAlert(button, message)
```

where: `button` is an INTEGER whose value indicates the contents of the button that dismisses the Alert.

`message` is a CHARACTER string containing the text of the message to be displayed.

Following are the possible values of the `button` argument and the corresponding buttons that will appear in the dialog box:

<table>
<thead>
<tr>
<th>Value</th>
<th>Button</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Ok</td>
</tr>
<tr>
<td>1</td>
<td>Cancel</td>
</tr>
<tr>
<td>2</td>
<td>Continue</td>
</tr>
<tr>
<td>3</td>
<td>Quit (causes the program to stop)</td>
</tr>
<tr>
<td>4</td>
<td>Ok (widens the dialog box by 15 percent)</td>
</tr>
</tbody>
</table>

“Show Alert” Buttons
SetMrwePrefs

The MRWE interface to your Fortran program can be easily customized to meet your needs by specifying the text characteristics and how the application behaves when it quits. The initial attributes of the MRWE interface are established when you create or modify the options of a project that includes MRWE. Additionally, the SetMrwePrefs program allows you to modify these characteristics after the application has been built.

SetMrwePrefs is located in the /Applications/Absoft/bin. Use the finder to navigate to this folder and double click on the SetMrwePrefs application. Use the Open command in the file menu to select the program that you want to customize. The following dialog window will open:

![SetPrefs Dialog Box](image)

SetPrefs Dialog Box
Figure 8-7

Below is a brief description of the options that are available to you within the SetPrefs Tool. Options are grouped according to the way they appear in the SetPrefs dialog box and are followed by the appropriate command line arguments.

Applications to affect

Before you can choose any customization options in the dialog box, you must identify the applications whose attributes will be affected. You can select either all future applications, or one or more specific applications.

All future applications ((AbsoftLibraries)MRWE.o)
Check this box if you want to change the MRWE.o library; so that any future applications linked with MRWE will have the new attributes you have selected.

Select Applications… *(ApplicationNames)*

Click on this button to open a file selection dialog box that allows you to choose one or more applications. The attributes that you select will apply to these applications when you click the Setprefs button.

**End of execution**

These options control the actions of the MRWE window before the application stops executing.

**Pause** *(–pause)*

Click the checkbox if you want MRWE to pause after the program ends, allowing you to read output in the window. By default, MRWE immediately quits.

**Save Text** *(–saveOnClose never|prompt|always)*

When MRWE quits, text in the window can be saved. If this option is never, text will not be saved. If this option is prompt, a dialog will ask whether text should be saved. To automatically save text without a prompt, use always.

**Window characteristics**

These options affect only the standard MRWE window, not windows opened with the ACCESS=“window” specifier in an OPEN statement.

**Window name** *(–savename name)*

This is the name used in the title bar of the window. By default, it is the name of the application (often called Link.out) with the suffix 'output'.

**Close box** *(–closebox)*

Click the checkbox to give the MRWE window a close box. If the window has a close box, clicking it causes the program to stop.

**Window Size** *(–windowsize normal|maximize)*

When MRWE starts, this specifies the size of the window. If the normal is selected, the window is the default size. If maximize is selected, the window is maximized.
Text characteristics

These options control the appearance of text characters in the window. You may prefer a text style different than the default for easier reading. Also, the text style of the window controls the text style when you print from MRWE.

Tab Size (-tabsize size)

This is the tab modulo size MRWE. If the value is greater than 20 or less than 0, the value is the default, 8. If the value is 0, tabs are passed as is to the application.

Font (-font fontname)

This specifies the font to be used in the window; the default font is Monaco.

Font Size (-fsize fontsize)

Type in a font size in points or select one from the pop-up menu; the default is 9.

Line Space (-linespace points)

This option defines the amount of space between lines in points; the default is 0.

No Font Menus (-nofontmenus)

Click the checkbox to use Font and Size menus in the application; the default state is on.

Memory requirements

These options set the application’s memory requirements, which are shown in the Get Info dialog box in the Finder.

Preferred (-sizepref KiloBytes)

Number of kilobytes of memory the program initially asks the system for. In the Finder's Get Info dialog box, this is the “Suggested size” and will initially be used for the “Current size”, although you can change this.

Minimum (-sizemin KiloBytes)

This is the minimum number of kilobytes required to run the application. The application will not be launched if this minimum amount of memory is not available.

Timer interval

Fortran User Guide
If you turn on the *timer interval* feature with the `-N9` compiler option, your program will be interrupted at timed intervals to check for Command-Period termination and to allow other programs enough time to run smoothly.

**Timer (`-timer 60ths-of-a-second`)**

The timer interval is expressed as the number of 60ths of a second that MRWE will let Fortran code process exclusively. After the interval has expired, MRWE will allocate processing time to other active programs. By default, the interval is set at 30 (1/2 second).
CHAPTER 6

Building Programs

This chapter covers the specifics of building Fortran 90/95 and FORTRAN 77 programs, including a discussion of the 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.

The Components of an Application

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

Working with Resources

A resource is one of the most important concepts in Macintosh programming. A resource is a collection of information used by the Macintosh OS X operating 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 Macintosh program, $rez$, included with the Apple developer tools, is a resource compiler that creates new resources based on a textual resource description file.

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: $f95$ or $f77$ – these commands control both components of the compiler (front and back ends), the system assembler ($as$) and the linker ($ld$) (see the section below on Linking Programs). The features of the $f95$ and $f77$ 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...]

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 a.out. To compile hello.f with the static local storage option, and generate an application named welcome, enter:

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

The option, -o name, specifies the name of the executable file overriding the default name of a.out. 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.o and Goodbye.o. 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 commands, any set of options may be passed directly to the linker. To do this, append the following option to the compiler command:

```
-X opts
```

The argument opts is a string enclosed in quotes to be passed to the linker. For example, -X -v will pass the -v 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.

\[ \text{Fsplit [option...]} \ [\text{file...}] \]

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

\[ \text{Fsplit largefile.f} \]

A procedure includes block data, function, main, program, and subroutine program declarations. The procedure, proc, is put into file \( \text{proc.f} \) with the following exceptions:

- An unnamed main program is placed in \( \text{MAIN.f} \).
- An unnamed block data subprogram is placed in a file named \( \text{blockdataNNN.f} \), 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 \text{fsplit} command was executed. The tab size is pulled from the environment variable \text{TABSIZE} if it exists, otherwise, a tab size of 8 is used. Options for the command are:

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

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

Fortran User Guide
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 easily, 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 simply 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:

<table>
<thead>
<tr>
<th>Macro Definition</th>
<th>Macro Invocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEBUGOPT = -g</td>
<td>$(DEBUGOPT)</td>
</tr>
<tr>
<td>SRCFILES = one.f two.f</td>
<td>$(SRCFILES)</td>
</tr>
<tr>
<td>OBJFILES = one.o two.o</td>
<td>$(OBJFILES)</td>
</tr>
<tr>
<td>ALLFILES = $(SRCFILES) $(OBJFILES)</td>
<td>$(ALLFILES)</td>
</tr>
</tbody>
</table>

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.o two.o

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:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$@</td>
<td>Represents the <em>full name of the current target</em>—for use only on a (re)build command line. (When building a library it represents the name of the library.)</td>
</tr>
<tr>
<td>$*</td>
<td>Represents the <em>base name of the current target</em>—for use only on a (re)build command line.</td>
</tr>
<tr>
<td>$&lt;</td>
<td>Represents a <em>current prerequisite</em>—for use only on a (re)build command line.</td>
</tr>
<tr>
<td>$$@</td>
<td>Represents the <em>base name of the current target</em>—for use only on a dependency line.</td>
</tr>
<tr>
<td>$?</td>
<td>Represents a <em>list of prerequisites</em> that have been changed more recently than the current target—for use only on a (re)build command line.</td>
</tr>
</tbody>
</table>
Other special macros that are provided with Absoft amake include:

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

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 libstat.a
   f95 -o test a.f95 b.f95 libstat.a
```
specifies that the target `test` is dependent on the prerequisites `a.f95`, `b.f95`, and `libstat.a`. If any of the three prerequisites have been updated at the same time or after the target, `test` 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 libstat.a; f95 -o test a.f95 b.f95 libstat.a
```

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 libstat.a; f95 -o test a.f95 \b.f95 libstat.a
```

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:

```
test: a.f95 b.f95 libstat.a
   f95 -o test a.f95 b.f95 libstat.a
```

Fortran User Guide
libgraph.a:: vertex.f95  
  $(F95) -c -g -DDEBUG vertex.f95
  ar -r libgraph.a vertex.o
  rm vertex.o

libgraph.a:: edge.f95  
  $(F95) -c -O edge.f95
  ar -r libgraph.a edge.o
  rm edge.o

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

**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
OBJJS = util.o build.o parse.o tstring.o
PROG = $(NAME)

$(PROG): $(OBJJS)
  $(F95) $(F95FLAGS) $(OBJJS) -o $(PROG) $(LDFLAGS)

util.o: util.f95 util.inc tstring.inc decl.inc
build.o: build.f95 util.inc tstring.inc decl.inc
parse.o: parse.f95 util.inc tstring.inc decl.inc
tstring.o: 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:

/usr/absoft/bin/default.mk.

The following is a list of the default dependency rules included with Absoft amake for Fortran 90/95 and FORTRAN 77 files. The macros shown within these rules are pre-defined in the default.mk 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) .f90:
    $(F95) $(F95FLAGS) $(LDFLAGS) -o $@ $<
(2) \texttt{.f90.o:} \\
\hspace{1cm} $(F95) \ (F95FLAGS) -c \ *\texttt{.f90}$

(3) \texttt{.f95:} \\
\hspace{1cm} $(F95) \ (F95FLAGS) \ (LDFLAGS) -o \ @ \ <$

(4) \texttt{.f95.o:} \\
\hspace{1cm} $(F95) \ (F95FLAGS) -c \ *\texttt{.f95}$

Explanation:
(1) Compiles a Fortran 90 source file into an executable target.
(2) Creates an object file from a Fortran 90 source file.
(3) Compiles a Fortran 95 source file into an executable target.
(4) Creates an object file from a Fortran 95 source file.

Default Rules for FORTRAN 77 files

(1) \texttt{.f:} \\
\hspace{1cm} $(F77) \ (FFLAGS) \ (LDFLAGS) -o \ @ \ <$

(2) \texttt{.f.o:} \\
\hspace{1cm} $(F77) \ (FFLAGS) -c \ *\texttt{.f}$

(1) \texttt{.for:} \\
\hspace{1cm} $(F77) \ (FFLAGS) \ (LDFLAGS) -o \ @ \ <$

(2) \texttt{.for.o:} \\
\hspace{1cm} $(F77) \ (FFLAGS) -c \ *\texttt{.for}$

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 executable target.
(4) Creates an object file from a FORTRAN 77 source 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 \texttt{amake} by creating your own dependency rules. There are two possible ways to do this:

\begin{itemize}
\item Include dependency rules in your description file, or
\item Modify the file of default rules by adding your own rule(s), or deleting/changing existing rule(s).
\end{itemize}
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 =

SOURCES = util.f95 build.f95 parse.f95 tstring.f95
OBJECTS = util.o build.o parse.o tstring.o
TARGET = $(NAME)

.f95.o:
  $(F95) $(F95FLAGS) /c $<
  cp $< /home/usr/workdir

$(TARGET): $(OBJECTS)
  $(F95) $(F95FLAGS) $(OBJECTS) /o $(TARGET) $(LDFLAGS)

util.o: util.f95 util.inc tstring.inc decl.inc
build.o: build.f95 util.inc tstring.inc decl.inc
parse.o: parse.f95 util.inc tstring.inc decl.inc
tstring.o: tstring.f95 tstring.inc
```

The user-supplied rule:
```
.f95.o:
  $(F95) $(F95FLAGS) -c $<
  cp $< /home/usr/workdir
```

will override the corresponding default rule in the default.mk file:
```
.f95.o:
  $(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:

**Fortran User Guide**
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 are specified with a dash (-):

- **-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:

```make
    clobber :
        rm *.o
```

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.o suffix rule and the dependency blocks (lines) following it:

```fortran
# program name
NAME = util

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

SRCS = util.f95 build.f95 parse.f95 tstring.f95
OBJJS = util.o build.o parse.o tstring.o
PROG = $(NAME)

.f95.o:
  $(F95) $(F95FLAGS) /c $<
  cp $< /home/usr/workdir

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

util.o: util.f95 util.inc tstring.inc decl.inc
build.o: build.f95 util.inc tstring.inc decl.inc
parse.o: parse.f95 util.inc tstring.inc decl.inc
tstring.o: 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

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

### Other Common Errors

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

### 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 \texttt{-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 \texttt{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 7

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 Pro Fortran is designed to be fully compatible with the implementation of the standard C Programming Language provided on Macintosh OS X. 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. The case of global symbols C is significant. The symbolic names of external procedure must match in case and decoration.

Fortran Data Types in C

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOGICAL*1 l</td>
<td>unsigned char l;</td>
</tr>
<tr>
<td>LOGICAL*2 m</td>
<td>unsigned short m;</td>
</tr>
<tr>
<td>LOGICAL*4 n</td>
<td>unsigned long n;</td>
</tr>
<tr>
<td>CHARACTER*n c</td>
<td>char c[n];</td>
</tr>
<tr>
<td>INTEGER*1 i or BYTE i</td>
<td>char i;</td>
</tr>
<tr>
<td>INTEGER*2 j</td>
<td>short j;</td>
</tr>
<tr>
<td>INTEGER*4 k</td>
<td>int k;</td>
</tr>
<tr>
<td>REAL*4 a</td>
<td>float a;</td>
</tr>
<tr>
<td>REAL*8 d</td>
<td>double d;</td>
</tr>
<tr>
<td>COMPLEX*8 c</td>
<td>struct complx {</td>
</tr>
<tr>
<td></td>
<td>float x;</td>
</tr>
<tr>
<td></td>
<td>float y;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>struct complx c;</td>
</tr>
<tr>
<td>COMPLEX*16 d</td>
<td>struct dcomp {</td>
</tr>
<tr>
<td></td>
<td>double x;</td>
</tr>
<tr>
<td></td>
<td>double y;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>struct dcomp d;</td>
</tr>
</tbody>
</table>
The storage allocated by the C language declarations will be identical to the storage allocated by the corresponding Fortran declaration.

There are additional precautions when passing Fortran strings to C routines. See the section **Passing Strings to C** later in this chapter for more information.

**Required Compiler Options**

FORTRAN 77 code should be compiled with the following options:

- `-f` fold symbols to lower case
- `-s` use static storage
- `-N15` append trailing underscores to global names

Fortran 90 code should be compiled with the following options:

- `-YEXT_NAMES=LCS` fold symbols to lower case
- `-s` use static storage
- `-YEXT_NAMES=_` append trailing underscores to global names

C code does not have to be compiled with any special options for the C compiler.

**Rules for Linking**

When linking Fortran and C programs, the `f77` or `f90` compiler driver should be used so that the appropriate Fortran and C libraries are included in the final application. The following command will compile the file `f1.f` with the FORTRAN 77 compiler and the file `c1.c` with the C compiler. It will then link the two resulting object files along with `o1.o` and the appropriate libraries to generate an executable application named `exec`:

```
f77 -o exec f1.f c1.c o1.o
```

**Passing Parameters Between C and Fortran**

The Absoft Pro Fortran compilers use the same calling conventions as the C programming 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 earlier in this chapter.
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:

```fortran
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:

```fortran
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:

```c
int main()
{
    float a_actual;
    int i_actual;
    void SUB();

    a_actual = 3.3;
    i_actual = 9;
    SUB(&a_actual, &i_actual);
    return 0;
}
```

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:

```fortran
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 Pro Fortran provides the intrinsic function %VAL() for passing value parameters. Function interfaces may also be used to specify which arguments to pass by value. Although it is generally pointless to pass a value directly to a Fortran procedure, these functions may be used to pass a value to a C function. The following is an example of passing a 4-byte integer:

    WRITE (*,*) 'Enter an integer:'
    READ (*,*) i
    CALL C_FUN(VAL(i))
    END

    void C_FUN()
    { }
    
    void FORTRAN_SUB();
    int i;
    FORTRAN_SUB(i);
    }
    
    SUBROUTINE FORTRAN_SUB(i)
    VALUE i
    ...
    END

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 simply 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, and that the only Fortran data type that cannot be passed by value is CHARACTER.
Array Parameters

One-dimensional arrays can be passed freely back and forth as both language implementations pass arrays by reference. However, since C and Fortran use different row/column ordering, multi-dimensional arrays cannot be easily passed and indexed between the languages.

```fortran
INTEGER ia(10)

CALL C_FUN(ia)
WRITE (*,*) ia
END
```

```c
void C_FUN(i)
int i[];
{
 int j;
 for(i=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`, or `DOUBLE PRECISION`. All other data types must be returned in reference parameters. The following are examples of the passing of function results between Fortran and C. The names are case-sensitive, so trying to call `cmax`, for example, will result in an error at link time.

A call to C from Fortran

```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
```

```c
int CMAX (x,y)
int *x,*y;
{
   return( (*x >= *y) ? *x : *y );
}
```

A call to Fortran from C

```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 of characters terminated by a null character. Therefore, when passing Fortran strings to C routines, you should terminate them with a null character. The following Fortran expression will properly pass the Fortran string string to the C routine CPRINT:

```fortran
PROGRAM cstringcall
  character*255 string
  string = 'Moscow on the Hudson'
  CALL CPRINT(TRIM(string)//CHAR(0))
END
```

```c
void CPRINT (anystring)
char *anystring;
{
    printf("%s\n",anystring);
}
```

This example will neatly output "Moscow on the Hudson". If the TRIM function were not used, the same string would be printed, but followed by 235 blanks. If the CHAR(0) function was omitted, C would print characters until a null character was encountered, whenever that might be.
You can also take advantage of the string length arguments that Fortran passes. After the end of the formal argument list, Fortran passes (and expects) the length of each CHARACTER argument as a 32-bit integer value parameter. For example:

```fortran
SUBROUTINE FPRINT(string)
    character*(*) string
    print *, string
END
```

```c
#include <string.h>
int main()
{
    char string[] = {"Moscow on the Hudson");
    void FPRINT(char *, int);

    FPRINT(string, strlen(string));
    return 0;
}
```

### Calling Fortran math routines

All of the Fortran intrinsic math functions which return values recognized by the C Programming Language can be called directly from C as long as the Fortran run time library, `libf77math.a`, is linked to the application.

Taking the intrinsic function names in lower case and adding two underscores to the beginning forms the names of the functions that can be called.

The following example calls the Fortran intrinsic function `SIN` directly from C:

```c
main()
{
    float sin_of_a, a, __sin();

    a = 3.1415926/6;
    sin_of_a = __sin(a);
}
```

### Naming Conventions

Global names in FORTRAN include procedure names and COMMON block names, both of which are significant to 31 characters. All global names are case sensitive, meaning the compiler recognizes the difference between upper and lower case characters. Use of the `-f` option will fold global names to lower case, while the `-N109` option will fold global names to upper case. 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 as they were typed in the source code with an underscore (“_”) prefix character attached. Symbolic names in the C language are case sensitive, distinguishing between upper and lower case characters. To make FORTRAN code compatible with C, avoid using the \texttt{-f} or \texttt{-N109} options when compiling the FORTRAN source code.

Accessing COMMON blocks from C

COMMON block names are global symbols formed in Absoft Pro Fortran by prepending the characters \texttt{"_C"} to the name of the COMMON block. The elements of the COMMON block can be accessed from C by declaring an external structure using this name. For example,

\begin{verbatim}
COMMON /comm/ a,b,c
\end{verbatim}

can be accessed with the C declaration:

\begin{verbatim}
extern struct {
    float a;
    float b;
    float c;
} _CCOMM;
\end{verbatim}

Declaring C Structures in Absoft Pro 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:

\begin{verbatim}
C
struct str {  
    char c;  
    long l;  
    float f;  
    double d; 
};
struct str my_struct;
\end{verbatim}

\begin{verbatim}
FORTRAN
STRUCTURE /str/ 
    CHARACTER c  
    INTEGER*4 l  
    REAL*4  f  
    REAL*8 d  
END STRUCTURE
RECORD /str/  my_struct
\end{verbatim}

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 FORTRAN 77 Language Reference Manual for more information on the FORTRAN RECORD type.

INTERFACING WITH ASSEMBLY LANGUAGE

If you are interested in interfacing to Fortran through assembly language, refer to the Apple document, OS\textunderscore X\_PPC\_Runtime\_Conventions.pdf, supplied with the Macintosh OS X developer tools.

Fortran User Guide
DEBUGGING

Debugging a Fortran program is accomplished with the Absoft source-level debugger, Fx™. This is a multi-language, windowed debugger designed especially for the Intel based Linux computers. The operation of the debugger is detailed in the chapter, Using the Fx Debugger. The following paragraphs describe the compiler options and resources necessary to prepare a program for debugging.

Compiler Options

The -g compiler option directs the compiler to add symbol and line number information to the object file. This option should be enabled for each source file that you will want to have source code displayed while debugging. It is not required for files that you are not interested in.

It is recommended that all optimization options be disabled while debugging. This is because the optimizers can greatly distort the appearance and order of execution of the individual statements in your program. Code can be removed or added (for loop unrolling), variables may be removed or allocated to registers (making it impossible to examine or modify them), and statements may be executed out of order.

PROFILING

The Macintosh OS X operating system includes the libraries and tools necessary to obtain procedure level profiles of your application. You simply create an instrumented version of your application (see Compiler Options below) and then execute it. The file gmon.out will automatically be created. Use gprof to display and analyze the results.

Compiler Options

The -P compiler option directs the compiler to add the symbol information to the object file necessary to profile an application. Enabling this option will allow the application to report the number of times a particular subroutine is called or a function is referenced.

All other options that you would normally use should be enabled, including optimization.
Chapter 8

Using the Fx Debugger

INTRODUCTION

When the state of the art user interface consisted of a card reader and line printer and the majority of programs were written in assembly language for mainframe computers, debugging programs involved long hours examining source listings and core dumps. Other debugging techniques included inserting code to print the values of variables and trace the flow of program execution, staring off into space, and blaming the hardware. As both hardware and software technology progressed, the need for more sophisticated debugging tools led to the development of interactive assembly and source level debuggers.

About Fx

Although interactive debuggers have existed for a number of years, a surprising number of programmers do not use them during program development. Many may have tried to use one and found the experience so frustrating that they returned to more primitive debugging techniques. Others, particularly FORTRAN and assembly language programmers, may have found that available debuggers fail to provide adequate support for their language of choice.

Absoft has created Fx to meet the needs of both the novice and experienced programmer. Fx provides both standard debugging capabilities, such as breakpoints and variable display, and includes advanced features like variable monitors and dynamic windows. FORTRAN programmers will find that Fx supports their language, and assembly language users will appreciate the support provided for machine language debugging.

Some people enjoy using a point and click graphical interface; others may find it a distraction, or not have access to the necessary hardware. Fx provides interfaces to suit both of these needs. When complete access to a computer is available, Fx allows you to take advantage of a graphic interface and still retain the convenience of a command line. When you are debugging on a character based terminal, Fx provides a command line interface and provides windowing capabilities where possible.
About Fx Interfaces

This manual has been structured to allow you to begin using the Fx graphical interface as quickly as possible. The interface is based on the Apple Macintosh OS X user environment. Once Fx is installed on your machine it can be started in one of two ways, either it can be launched from double clicking its freestanding or docked icon, or it can be launched from a Terminal window. From the Terminal, you can begin debugging a simple program by entering "open –a Fx.app" and pressing the Return key.

About the Examples in this Manual

The Fx interface for Mac OS X supports command line entry. You should keep in mind that there may be more convenient ways to specify a given command. Although the examples in this manual use the complete spelling of each command, all commands can be entered by specifying the minimum number of characters required to distinguish the command. For example, the quit command can be abbreviated to q. When two commands begin with the same sequence of characters, the one which is more commonly used will be executed. For example, the step command can be abbreviated to s, while the search command can be abbreviated to sea.
TUTORIAL

If you have never used a source level debugger before, you will be surprised by how useful a debugger can be in isolating problems in a program. This tutorial introduces the basics of Fx and provides step-by-step instructions that cover the following topics:

- **Launching the Debugger**
  
  Describes how to start a debugging session and introduces the basic features of the Fx Debugger interface.

- **Setting Breakpoints**
  
  Provides an introduction to using breakpoints to analyze sections of code.

- **Executing Programs**
  
  Describes how to run a program under Fx. and introduces different menu selections for working with variables while executing the program.

- **Quitting the Debugger**
  
  Describes how to exit from Fx.

**Launching the Debugger**

This section discusses how to start Fx and load a program for a debugging session. It describes how to specify the name of the program to debug, the name of a core file, and the directories that contain the source code for the program. Although Fx can be launched from an icon, it is recommended that you launch it from a shell while using this tutorial.

**The Tutorial Program**

The tutorial program, wrdcnt, is a program that counts the number of words, lines, and characters in a text file. The program is used to demonstrate the basic features of the Fx Debugger.

Before launching Fx, the executable version of the tutorial program must be created in the shell using the make command. The Fx Help Library includes the tutorial source file and a makefile that builds the tutorial program. To get a copy of these files and create the tutorial program, at the Terminal shell prompt enter:

\[
\begin{align*}
\text{ar xv $ABSOFT/lib/Fxhelp.a wrdcnt.c Makefile} \\
\text{amake}
\end{align*}
\]

After the tutorial program has been compiled, the Fx Debugger is launched from the shell prompt to begin a debugging session by entering the command Fx:

\[
\begin{align*}
\text{open –a Fx.app}
\end{align*}
\]
Using the Initialize Fx Dialog

Launching the debugger will make the Initialize Fx dialog appear. To make the dialog the active, or key window, click inside of it. To see information on the features of the dialog Control-click anywhere in the dialog. If, for some reason, you need to quit from the debugger at this point, click the Quit button.

![Initialize Fx dialog](image)

The four text fields: **Executable File**, **Core File**, **Source Path(s)**, and **Working Directory** allow you to type in the names of the files and directories the debugger will use.

**Executable File Text Field**

The **Executable File** text field names the program to be debugged. The name of tutorial program, `wrdcnt`, will be entered into this text field.

**Core File Text Field**

This text field allows you to specify a core file. When an operating system creates a core file of a program, Fx can use this file to easily determine the location in your code where an error occurred. For the purpose of this tutorial, a core file will not be used.

**Source Path(s) Text Field**

The **Source Path(s)** text field lists the directories that contain the source files that were compiled to produce the program you are debugging.

**Working Directory Text Field**

The **Working Directory** text field allows you to specify the directory that will be your program's current directory while you are debugging with Fx.

**Loading the Program Into the Debugger**

The tutorial program is loaded into the debugger by using the four text fields in the Initialize Fx dialog. Since Fx has supplied suitable defaults for the **Source Path(s)** and
Working Directory, you only need to enter the name of the executable file in the Executable File field:

- wrdcnt
- click the OK button

When the program is loaded, the source code will be shown in the main window, the Fx Debugger window. This window is where you will spend much of your time, executing commands necessary to debug your program.

![Fx Debugger window]

The Status Field is located under the window's Title Bar; it shows where you are in the program, the name of the source file, the program unit, and the line number. The name of the executable file appears in the title bar.
Underneath the Status Field is the Source Code pane, which shows the program's source code. By pressing the arrows on the scroll bar, different sections of the source code can be viewed. The Command Output pane is located under the Source Code pane and displays the output from certain commands (like printing the values of variables). It also has a scroll bar to view output. These panels can be re-sized by clicking on the divider, indicated by the small grabber icon, and dragging in the desired direction.

The Command Text Field is available for invoking Fx commands without using the menus. To invoke a command from this field, type in the command and press the Return key.

Located along the bottom of the Fx Debugger window are the User Configurable buttons which invoke Fx commands directly from the window.

Setting a Breakpoint

Breakpoints allow you to suspend execution of your program at a specific location. You can set breakpoints on source lines, procedures and even individual machine instructions. Before we execute the program, we need to set two breakpoints: one at a source line and another inside a specific procedure.

Setting a Breakpoint on a Line

If you double or triple-click with the left mouse button anywhere on a line of source code, a section of text is selected, and the Selection menu items and any corresponding User Configurable buttons are updated to reflect the new status of the debugger. For illustration purposes, however, this tutorial will concentrate only on the menu items in the Selection menu itself, but the same operations described could also be accomplished using the equivalent buttons. The Selection menu contains items for setting breakpoints, printing variables, and viewing procedure code.

For the tutorial, the Toggle Break item will be used to set a breakpoint at line 40.

✓ triple-click on line 40
✓ from the Selection menu choose Toggle Break menu item
Fx will mark the location of the breakpoint with the letter "B" in the **Source Code** pane. When this line is encountered during program execution, the program will stop.

**Setting a Breakpoint within a Procedure**

The **Break In** menu item on the Selection menu sets a breakpoint within a procedure. The second breakpoint we will set is within the procedure **GetCounts** which is called from line 63. This breakpoint will stop the program when **GetCounts** is called.

- double-click **GetCounts** in the **Source Code** pane
- choose **Break In** from the **Selection** menu

In general when selecting text, be careful that you select only the item desired. If any other characters are selected, the command will result in an error.

Now that the two breakpoints are set, the program can be executed to explore other features of the debugger.

**Executing Programs**

The Fx Debugger can execute a program in different ways. The Run menu item is used to execute a program and is located in the Debug menu. This menu item invokes a dialog in which program arguments can be specified to the command.

![Debug menu](image)

**Using the Run Menu Item**

To execute the **wrdcnt** program, a file needs to be specified so the program can count words, lines, and characters. After choosing the **Run** command from the **Debug** menu, this file can be specified as an argument in the **Run Program** dialog. For the tutorial, the number of lines, words and characters in the **wrdcnt.c** source file will be counted.

- choose the **Run** command from the **Debug** menu
- in the **Run Program** dialog, enter **wrdcnt.c** in the **Program Argument** field
- click the OK button
The wrdcnt program will execute to the first breakpoint we set, which is located at line 40 in the source code.

**Printing Values of Variables**

The values of program variables can be tracked during program execution and used as an aid to debugging a program. The **Print** command in the Selection menu will display the values of variables in the **Command Output** pane:

- select the variable `argc` in line 40
- choose **Print** from the Selection menu

The number of arguments that have been passed is shown in the **Command Output** pane. Now, execute the program to the next breakpoint:

- click the **Continue** button

After execution stops, note the **Status Field** under the Title Bar. It will show the location where you are currently working, including the line number, unit and file name.

**Executing Single Statements**

To execute one statement at a time, use the **Step Into** button in the main window. This button will execute each statement, including any procedures that the statement contains. The **Step Over** button, will also execute one statement, but if the line contains a procedure, the debugger will not stop program execution until the procedure stops executing. At this point in the tutorial, you should be at line 134.

- click the **Step Into** button to reach line 139
- click the **Step Over** button

If you executed line 139 by mistake and want to return to line 139, press the **Return** button to complete execution of the procedure.

**Dereferencing a Variable**

The **Print** menu item in the Selection submenu dereferences C pointer variables:

- select the variable `f`
- choose **Print** from the Selection menu

The number of words, lines and characters counted to this point is shown in the **Command Output** pane.
Watching Variables

Often you will need to watch the value of a variable to determine if a procedure or function is overwriting it. The Watch menu item in the Selection menu allows you to watch the values of variables as the program executes:

- select \texttt{f->line\_count} in line 135
- choose Watch from the Selection menu
- select \texttt{f->word\_count} in line 136
- choose Watch from the Selection menu
- click the Step Over button to execute source lines. The values will be printed as the program executes

Removing Watched Variables

To remove the variables that you are watching, use the Watch Variables window:

- select \texttt{f->line\_count} in the Watch Variables window
- click the Unwatch button
- click the Unwatch All button to remove all variables

![Watch Variables window]

Changing Variables

Changing the value of a variable is done through the Change menu item in the Selection menu:

- select \texttt{f->word\_count}
- choose Change from the Selection menu
- enter 100 in the Value field
- click the OK button
Use the **Print** menu selection to see the new value of the variable:

- select `f->word_count`
- choose **Print** from the Selection submenu

**Exiting from the Fx Debugger**

You can terminate a debugging session at any time by choosing **Quit Fx Debugger** from the Fx Debugger application menu. Fx will terminate the program you are debugging and return you to the shell prompt:

- choose **Quit Fx Debugger** from the Fx Debugger menu
FX INTERFACE REFERENCE

This chapter concentrates on the specific features of the Fx interface to enable you to get the most out of the debugger for your needs.

- Using the Fx Interface
  Describes using the Fx Debugger interface and its window: buttons, menus, and lists.

- Reference Section
  Provides a reference guide to the features of the debugger.

Using the Fx Interface

The Fx graphic interface provides a complete windowed environment for source level and assembly language debugging. The interface is based on Apple's Macintosh OS X user environment, and its behavior is designed to be consistent with other Macintosh OS X applications.

If you are not familiar with the Macintosh OS X interface, you may want to become familiar with the basic information provided in the documentation that came with your machine before proceeding.

The Fx graphic interface permits all commands to be selected from within the debugger windows. Most commands will be executed from the main window, the Fx Debugger window. Commands can be invoked by pressing buttons, using menu items or pop-up lists. The following are tips when working the interface:

- Any menu item that is followed by an ellipse (“…”) after the item name invokes a window or dialog.

- A menu item may have a keyboard alternative, a short-cut to accessing the menu item from the keyboard. The alternative is defined by using the specified character in the menu name with the Command key. For example, to display the Source window, press the Command and Option keys along with the letter 's' on the keyboard.

- A default button is displayed in color and pulses which identifies that button as the window's default action, which can be invoked by pressing the Return key.

- If a button is associated with a text field, a Return may cause the button to act as if it had been clicked.

- Clicking in a window makes a window the key window and in most cases the main window, and it brings the window to the front of it tier on the screen.

Fortran User Guide
An action is first associated with the key window.

A key window receives characters from the keyboard. If the key window is a dialog and it cannot handle the action, the action is then associated with the main window.

A main window contains the target for control actions.

A key window receives characters from the keyboard. If the key window is a dialog and it cannot handle the action, the action is then associated with the main window.

Fx Main Window

The main window, the Fx Debugger window contains buttons and text fields to execute debugging commands and view the source code:

![Fx Debugger window](image)

**Title Bar**

The Title Bar displays the pathname of the program you are debugging.

---

**Fortran User Guide**
Status Field

Under the Title Bar is the Status Field that shows information about the program you are debugging: the line number, file name, and program unit.

User Configurable Push Buttons

A block of space for frequently used commands is provided at the bottom of the window that you can configure to include your favorite commands through the Preferences dialog. To activate a button, point to the button with the cursor and press the mouse button.

Navigating Through Text

The window also contains two scrollable panes. The first is the Source Code pane, which allows you to view the program code by using the scrollers. The second is the Command Output pane that displays any output, such as variable values. The text in this panel will consume memory as it collects information, but memory can be conserved by limiting the number of lines automatically saved in this and other output panels via the Preferences dialog. The divider, located between these two panes, and indicated by the small grabber icon, allows you to resize the panels as needed.

Command Text Field

The bottom of the window contains a text field labeled, “Command”. Almost all commands in Fx have a command line equivalent that can be entered and executed from this text field. Also there are some infrequently used commands in Fx that do not have a menu or button equivalent, and can only be executed from this field.

The next sections describe all the features of the Fx Debugger. A general overview of menus, buttons and fields is given, followed by a description of the menu items.

The arrow symbol represents a submenu that is attached to a controlling menu item. To open a submenu, position the cursor within the content of the controlling menu item. The submenu contains a group of like menu items. To execute a menu item, drag the cursor onto an item, and click the mouse button.

Some menu items may invoke windows and dialogs. Any of these windows or dialogs may be canceled without executing the command by clicking in their respective close box. Help can be obtained for any window by Control-clicking within the content of a window, to obtain more information about that window's commands.
Fx Debugger Application Menu

The Fx Debugger application menu has menu items that invoke debugger commands or submenus.

![Fx Debugger menu]

About Fx Debugger

The About Fx Debugger item invokes a dialog that shows the version number and application icon of the Fx Debugger installed.

Preferences...

The Preferences menu item invokes a dialog that allows you to customize the debugger to suit your own need. Among the items under your control are output buffer sizes, some Fx control variables such as the Tab size, Monitor step size, Case Sensitivity, and Expression Language, and the User Configurable Buttons of the main window. Use the pop-up list at the top of the Preferences dialog to switch between the various panel contents. Use the Default Settings button to restore the factory settings for a displayed group of preference items.

Control Variable Preferences

This group of preferences contains four panes that allow you to modify various debugger control variables to customize the way you view and work with source code.
**Expression Language**

This pane has three selections, **Automatic, C, or FORTRAN**, to choose the language in which expressions are evaluated. The default is set to **Automatic**: the expression language will automatically correspond to the current source code file.

**Case Sensitivity**

This pane allows you to set the case sensitivity for variable and procedure names, since some compilers may fold all upper case letters into lower case when compiling code.

**Monitor Step Size**

For details on the functionality of this pane, see the section "Set Step Size," later in this chapter.

**Tab Size**

This pane allows you to specify the number of spaces to expand tab characters.
User Configurable Button Preferences

This group of preferences controls the what command push buttons equivalents will appear in the main window. See the section, *Customizing the Debugger* for more information.
Buffer Length Preferences

These preference items control the lengths of various debugger buffers. See the section, *Customizing the Debugger* for more information.

![Preference Settings Window](image)

**Buffer Length**

**Hide Fx Debugger**

This menu item hides all the Fx Debugger windows.

**Quit Fx Debugger**

This menu item closes the Fx Debugger interface, and returns you to the shell if the debugger was launched from a terminal window.

**File Menu**

The File menu contains general menu items that control the Fx graphical interface and debugging sessions. From this menu a debugging session can be logged or reinitialized, in addition to other session controls.

![File Menu](image)
Load

The Load menu item restarts a debugging session without having to shut down. This is useful when different source paths need to be specified for a program already running, or when you need to load a new program. When the Load menu item is invoked the following window appears:

This dialog is used to specify the Executable File, Core File, Source Path(s), and the Working Directory of the program you wish to debug. The four text boxes will retain their original arguments if left unchanged. Pressing the Default button will cause the debugger to attempt to locate an a.out executable and a core file for the current working directory. To exit from the dialog without executing the reinitialization press the Cancel button. If you need help or additional information about the dialog, Control-click in the window.

Log Submenu

The Log submenu contains menu items used in creating a separate file of debugger commands.
Log...
Invokes a dialog to create and name a log file. The name of the file is entered into the Log File text field. When the Log Commands toggle button is pressed, the debugger will record all subsequent commands into the specified file. Use the Default button to restore the default file name to the Log File field.

Toggle Logging On / Toggle Logging Off
Use this menu item to toggle logging on or off to the file specified in the Log File text field.

Read Log File...
This menu item invokes a selection dialog allowing you to choose a file of debugging commands created using the Log menu item. When the Open button is pressed, the commands from the log file are executed.

List File...
When selected, the List File... menu item invokes a file selection dialog to examine a file without leaving the debugger. To view a file, highlight the file and press the Open button.
Edit Menu

Besides containing the standard Edit menu commands, the **Edit Menu** also contains a few menu items for controlling the appearance of the **Source Code** pane.

![Edit menu](image)

Find Submenu

The Find submenu contains menu items for locating text or strings.

![Find submenu](image)

**Find...**

The **Find...** menu item opens a dialog for entering a string that is then searched for in current window. The **Find...** command is case sensitive.

![Find dialog](image)

**Find Next**

Use this menu item to find the next occurrence of the string specified with the **Find...** command.

**Enter Selection**

This menu item will enter any currently selected text in the active window as the string for the **Find...** command.

**Fortran User Guide**
Show Current
When selected, this command displays the next line to be executed in the Source Code pane.

Toggle Source
This command toggles the source code view between high-level language and assembly language.

Selection Menu
The Selection Menu contains items for setting breakpoints, viewing code, printing variables, and changing values. Use these menu items in cooperation with scroll views and panes such as the Source Code pane, any List File window scroll view, the Memory Window Output pane, or the Stack window scroll view, etc..., to perform operations on the current selection. Most Selection menu items are also available as choices for the User Configurable buttons.

<table>
<thead>
<tr>
<th>Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toggle Break</td>
</tr>
<tr>
<td>Print</td>
</tr>
<tr>
<td>Print*</td>
</tr>
<tr>
<td>Watch</td>
</tr>
<tr>
<td>Watch*</td>
</tr>
<tr>
<td>Change...</td>
</tr>
<tr>
<td>Format...</td>
</tr>
<tr>
<td>Break In</td>
</tr>
<tr>
<td>View Code</td>
</tr>
</tbody>
</table>

Toggle Break
This menu item sets or deletes breakpoints in the program. This item can be used only in conjunction with selections in the Source Code pane.

Print
Prints the value of a selected variable to the Command Output pane.

Print*
This menu item dereferences a pointer and prints the output to the Command Output pane.

Watch
The Watch menu item sets a watch variable on the selection and activates the Watch Variables window, from which you can watch the value of variables change dynamically during program execution.
**Watch**

The **Watch** menu item dereferences a selected pointer, sets a watch variable on the dereferenced items, and activates the **Watch Variables** window, from which you can observe the value of variables change dynamically during program execution.

**Change...**

The **Change** menu item invokes the **Change** dialog. In this dialog, entering the desired new value in the Value text field, and pressing the OK button will change the value of the selected symbol.

![Change dialog](image)

**Format...**

This menu item contains two pop-up lists that control how the output will be formatted in regards to the data type and size. The **Format** dialog determines the output format for the **Command Output** pane, the **Symbols Window Output** pane, and the **Memory Window Output** pane, based upon which is currently the debugger’s main window. Press and drag along the **Type Pop-Up List** to select the data type to be used. Press and drag along the **Size Pop-Up List** to select the size of the item to be output.

![Format dialog](image)

**Break In**

This menu item sets a breakpoint within a function or procedure. When set the breakpoint will halt program execution at the first executable statement in the procedure.

**View Code**

Allows you to view the code of a procedure in the **Source Code** pane. To return to the position of the next executable statement, use the **Show Current** command.
Debug Menu

The Debug menu contains menu items that execute program code during a debugging session.

![Debug Menu](image)

**External Procedure**

The **External Procedure** menu item allows procedures and functions within the program to be specified and executed out of sequence. Press the **OK** button to execute the procedure or function.

![External Procedure dialog](image)

**Run**

The **Run** menu item is used to start or restart execution of a program. It can be used to pass arguments to the program, such as redirecting the standard input and output while the program executes.

![Run Program dialog](image)

**Restart**

The Restart command restarts program execution from the beginning of the program.
Kill

The **Kill** menu item terminates the current processes being debugged without exiting from the debugger. This menu item is useful when you need to recompile a program without leaving the debugger. It is not necessary to use the **Kill** menu item when you quit from the debugger or restart program execution with the **Run** menu item: both the **Quit Fx Debugger** and the **Run** menu items kill the current process before they execute.

Continue

The **Continue** command resumes program execution until a breakpoint is encountered, an error occurs, or the program runs to completion.

Return

The **Return** command resumes program execution until the current procedure returns to its calling procedure.

Step Into

The **Step Into** command executes single statements. When pressed, the next line of code will be executed. If the source line contains a function or subroutine call, execution will stop at the first executable line of the function or subroutine.

Step Over

This command also executes single statements. However, it does not follow subroutine or function references, like the **Step Into** command, but rather stops on the next source line of the current procedure.

Until...

Executes the program to a certain location. This menu item will show the following dialog when selected.

![Execute To dialog](image)

The **Execute To** dialog can be used to run a program to a certain location in the code. When a location is specified in the **Location** text field, the debugger sets a temporary breakpoint, when the **OK** button is pressed. The **Disable Breakpoints** check box tells the debugger to ignore all other breakpoints.
Walk

The **Walk** menu item allows other debugger commands to be repeatedly issued by the debugger itself. The command, or a series of commands, to be invoked is specified in the **Walk Command(s)** text field. Commands should be separated with a semi-colon.

After the **Walk** command is issued with the **Start** button, the debugger will continue to execute the commands in the **Walk Command(s)** text field, until the **Stop** button is clicked. To stop the walk command at any time, press and hold down the **Stop** button until the debugger halts. Use the **Speed** slider to set the speed at which debugger commands are issued, from one to ten, with one being the quickest and ten the slowest.

Program Menu

The commands in the **Program** menu are used to open windows for displaying additional information such as memory dumps in various formats, assembly language register values, and stack frame traces.
Memory...

This menu item is used to display memory addresses beginning at a specific location. The address is specified in the Address text field, and listed in the Memory Window Output pane when the Display button is pressed.

Memory Window Output Pane

Memory locations are displayed in this buffered scrolling pane, which can be limited in size to conserve memory via the Preferences... menu item.

Address Text Field

The address should be entered into this text field. The address expression entered must evaluate to an address of a memory location in the program being debugged. For more information on proper syntax for this text field, see the section Working With Program Variables.

Format... Button

Use this button to invoke the Format dialog. The Format dialog determines how memory will be displayed by controlling the type of date structure and the size of each memory location displayed in the Memory Window Output pane, see the section on the Selection Menu for more information on the Format dialog.

Count Text Field

This text field specifies the number of memory locations displayed in the Memory Window Output pane.

Display

The Display button uses the current format options and text field arguments, and displays the specified memory location in the Memory Window Output pane.

Fortran User Guide
Source

The Source menu item invokes the Source window.

![Source window]

The Source window lists all the source files for the program that have been compiled with the -g option. When a source file is clicked, the functions within the file are listed in the second column of the Source Browser pane.

View

The View button will list the selected code in the Source Code pane of the main window. The same action can be accomplished by double-clicking on either a source file or function.

Break In

The Break In button sets a breakpoint at the first executable statement in the selected function.

List File

The List File button displays the code for the selected source file in a separate Fx Debugger window.

Sorting Order of Functions

Click the Alphabetical radio button to have the source file functions sorted by alphabetical ordering. Click the Numerical radio button to have the functions sorted as they would appear by line number in the source file.
Variables
The Variables menu item invokes the Symbols window shown below. The Symbols window dynamically displays local, static and global variables, and allows the programmer to easily view variable types and values. Symbols can be selected in the Symbol Browser at the top of the window. The selected variable's type is shown in the Type pane. Output for this window is directed to the Output pane. For more information on using the Symbols window, see the section Working With Program Variables.

Breakpoints Submenu
The Breakpoint submenu contains menu items for setting, listing and deleting breakpoints. Breakpoints are analogous to stop signs: when a breakpoint is encountered during program execution, the program suspends execution and returns control to the debugger. They are useful for analyzing minute or large sections of code and can be set at a particular line, subroutine or function. Breakpoints are covered in detail in the section Executing Programs.
Show Breakpoints...

The Show Breakpoints menu item invokes the Breakpoints window. The Breakpoints window displays all the current breakpoints and automatically updates to reflect the current state of the debugger. It is responsible for all operations involving breakpoints, including setting a breakpoint at a given location, deleting a specific breakpoint, deleting all breakpoints, associating other debugger commands with a breakpoint, setting the skip count of a break point, and viewing the source code for a given breakpoint.

![Breakpoints window](image)

Breakpoints window

**Breakpoint Browser**

All active breakpoints, along with any associated commands, the skip count, and a count of the number of times each breakpoint has been encountered without halting program execution are all shown in the Breakpoint Browser pane at the top of the Breakpoints window. Use this browser to view and select active breakpoints. The browser is used in combination with the text fields and buttons to modify and delete existing breakpoints.

**Location Text Field**

The location of the breakpoint is entered into this text field. The location of a breakpoint can be at any valid address expression, a line number, or the first executable source line in a procedure. For more information on how to specify a breakpoint location, see the section Executing Programs.

**Associate Commands Text Field**

Specifies the debugger commands to be executed when the breakpoint stops program execution. When execution stops at the specified breakpoint, the commands will be executed.
Skip Count Text Field

The skip count refers to the number of times the breakpoint is to be ignored before program execution is stopped. If a skip count is not specified, a value of zero will be used—program execution will stop the first time the breakpoint is encountered.

Set

The Set button places a breakpoint in a given location.

Delete

The Delete button is used to remove breakpoints from a program. To delete a breakpoint from the list in the Breakpoint Browser, highlight the breakpoint and press the Delete button.

Delete All

The Delete All button and Delete All menu item remove all breakpoints from a program.

View

The View button displays the code for the selected breakpoint in the Source Code pane of the main window.

Monitors Submenu

The Monitors submenu contains menu items for setting, removing, and listing variable monitors. Invoking a monitor is similar to setting a breakpoint, with a notable exception. When a monitor is set, it will not halt execution at a specific location. Instead execution halts when a relational expression specified by the programmer evaluates true.
Show Monitors...

The Show Monitors menu item invokes the Monitors window. The Monitors window displays all the current variable monitors and automatically updates to reflect the current state of the debugger. It is responsible for all operations involving variable monitors, including setting, removing a specific monitor, removing all monitors.

![Monitors window](image)

Monitor Browser

All active variable monitors are shown in the Monitor Browser at the top of the Monitors window. The Monitor Browser displays active variable monitors by id and expression. Use this browser to view and select active monitors. The browser is used in combination with the text fields and buttons to modify and remove existing monitors.

Variable Text Field

Specifies the name of the variable where the monitor will be set.

Condition Pop-Up List

Determines the operator used to compare the variable with the value specified in the Value text field.

Value Text Field

Contains the value of the variable at which execution will stop.

Set

The Set button installs a new monitor based upon the relational expression specified by the Variable and Value text fields, and the Operator Pop-Up. A monitor prompts the Fx Debugger to stop execution when a variable changes value.
Remove
The Remove button removes a previously installed monitor. To remove a monitor, select an active monitor in the Monitor Browser and press the Remove button.

Remove All
The Remove All button and Remove All menu item remove all monitors from a program.

Set Step Size...
The Monitor's Step Size pane determines how Fx will execute the Continue command when working with monitors. The Set Step Size... menu item invokes the appropriate Preferences dialog that contains the Control Variable settings. From this panel the Monitor's Step Size can then be set.

Step Into Source Statement
The Step Into Source Statement button specifies that when the Continue command is invoked, the next statement is executed and the value of the monitor will be checked.

Step Over Source Statement
The Step Over Source Statement button specifies that when the Continue command is invoked, the next statement is executed and the value of the monitor will be checked. If the statement that is executed contains any procedures, they are treated as part of the statement line.

Monitor's Step Size pane

Fortran User Guide
Instruction Step Into

The **Step Into Assembly Instruction** item specifies that when the **Continue** command is invoked while working in assembly level, the next instruction is executed and the value of the monitor will be checked.

Instruction Step Over

The **Step Over Assembly Instruction** item specifies when the **Continue** command is invoked while working in assembly level, the next instruction executes and the value of the monitor will be checked. If the instruction is a call to a procedure, it will be treated as a single statement.

WatchValues Submenu

The **Watch Values** submenu contains items that allow you to view or change values of program and debugger symbols and variables.

<table>
<thead>
<tr>
<th>Show Watch Values...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change...</td>
</tr>
<tr>
<td>Unwatch All</td>
</tr>
</tbody>
</table>

Watch Values submenu

Show Watch Values...

The **Show Watch Values** menu item invokes the **Watch Variables** window, which dynamically displays all watch variable values during program execution. It is responsible for all operations involving watch variables including setting a watch variable, deleting a watch variable, or deleting all watch variables. The **Watch Variables** window can also be used in conjunction with other Fx Debugger dialogs to provide services such as changing the value of a watch variable symbol, or changing the format of a watch variable value.
Watch Variable Browser

All active watch variables are shown in the Watch Variable Browser at the top of the Watch Variables window. The Watch Variable Browser displays a watch variable's ID number, the watch variable symbol, and the symbol's value. Use this browser to view and select active watch variables. The browser is used in combination with the text fields and buttons to modify and delete existing watch variables.

Watch

The Watch button creates a new watch instance on the specified symbol. Use the Format dialog to specify the format for the watch variable's value.

Unwatch

The Unwatch button is used to remove a watch variable from a program. To delete a watch variable from the list in the Watch Variable browser, highlight any watch variable with the same ID number and press the Unwatch button.

Unwatch All

The Unwatch All button and Unwatch All menu item remove all breakpoints from a program.

Change... Button

Use the Change button to invoke the Change dialog with the currently selected watch variable's symbol and value passed as parameters for the Symbol and Value text fields.

Format...

Use the Format dialog to specify the format for a new watch variable.

Change ... Menu Item

Use the Change menu item to invoke the Change dialog from the menu without a selected symbol.
Registers

The Registers selection is used to display the contents of the machine registers. Once the Dynamic Updates button is clicked the window will automatically display the registers as the program executes.

Stack...

This menu item invokes the Stack window that displays all the procedures previously executed. Once the Dynamic Updates button is clicked the window will automatically display the procedures as the program executes.

ASCII Table

This menu item invokes a table of ASCII characters and their corresponding hexadecimal numbers.
File Status
Displays information about all connected and preconnected FORTRAN units. This menu item is active only when debugging programs compiled with Absoft compilers.

Main Window
This menu item activates or reopens the Fx Debugger main window for the current debug session.

Window Menu
The Window menu contains menu items to control the placement and positioning of windows.

Help Menu
The Help menu contains menu items to access or disable help during a debugging session.
Help...

The **Help** menu item invokes the Help window that displays general information on how to use the application and its commands. Use the **Help Browser** to select help on a specific item.

![Help window](image)

**Topics Item**
Activates the Help window information on topics of interest.

**Commands Item**
Activates the Help window with information on commands available in Fx.

**Windows Item**
Activates the Help window with information on the windows and dialogs available in Fx. This help is also available by Control-clicking in any application window.

**Hide ToolTips**
The Hide ToolTips menu item disables all tool tips in the application that temporarily appear when the mouse hovers for a few seconds over the location of a control, such as a button or text field in a dialog.
Command Text Field

Although Fx provides an excellent windowed interface to execute debugging commands from buttons and menus, commands can also be executed by using the Command text field, located at the bottom of the Fx Debugger main window. Although most of the commands are available to the user in the menus or with push buttons, this text field can be used as a quick alternative to issue a command to the debugger.

Using the Command Text Field

To use the Command Text Field, enter the name of the command and press the Return key. A command can be executed with arguments, and must follow the syntax format in the Command Reference section. The last command executed from this text field can be re-executed, if the Return key is pressed.
RUNNING FX

This section introduces the basics of running the Fx Debugger to debug programs. The following topics are covered in this section:

- **Customizing the Debugger**
  Describes how to modify the User Configurable buttons of the main window, and discusses output panel and disassembly buffer lengths.

- **Preparing Your Program for Debugging**
  Discusses compile time options to take full advantage of the Fx Debugger.

- **Starting an Fx Debugging Session**
  Provides details on loading a program and changing the initialization file.

- **Restarting a Debugging Session**
  Introduces the Fx command for restarting a debugging session without leaving the debugger.

- **Debugging with Core Files**
  Describes core files and introduces ways to use core files to pinpoint and correct errors in source code.

- **Examining Source Code**
  Presents methods to examine multiple source files within a program, view procedures, and list procedure calls.

- **Recreating a Debugging Session**
  Shows how to use the log feature of Fx to record and playback commands initiated during a debugging session.
Customizing the Debugger

In order to fully utilize the debugger, it is important to be able to adjust the debugger to take full advantage of each personal approach to debugging. The debugger offers this capability in the means of the User Configurable push buttons. These buttons allow you to execute up to twenty-one frequently used debugger commands that you select, directly from the main window.

Setting the User Configurable Buttons

The Button Configuration Preference pane provides control of the User Configurable buttons. Use the Preferences... menu item and the Preferences dialog popup list to invoke this panel.

Adding a User Configurable Button

To add a button to the main window simply click on the corresponding toggle button in the Preference dialog and the button will immediately become active and appear in the main window. If there is not enough room available the window will be resized and a new row of buttons will be added above the previous row.

Removing a User Configurable Button

To remove a particular button from the main window click upon the appropriate toggle button, and the button will disappear. If the button is the last one in a row, or leaves a gap behind, this space will be reclaimed, either by resizing the window or by filling in the empty location with the left most button of the top row.

Fortran User Guide
Re-Organizing the User Configurable Buttons

To re-configure the ordering of several buttons to meet a specific arrangement it may be best to remove all the buttons and re-select them one-by-one in the order you wish them to appear. Click the Default button to restore the default button configuration. The button configuration is saved automatically upon termination of a debugging session in the file ~/Library/Preferences/com.absoft.software.fx.plist. This allows each user on a machine to have his own configuration.

Setting the Debugger Buffer Lengths

The fields in this pane allow you to conserve memory by limiting the amount of information that is stored in various debugger panes.

Buffer Length Preferences pane

Command Line Output Text Field

This text field determines how many lines are saved in the Command Output pane of the main debugger window.

Memory Window Output Text Field

Use this text field to specify how many lines of information are saved in the Memory Window Output pane of the Memory window.

Symbols Window Text Field

This text field regulates the number of lines of information stored in the Symbols Window Output pane.
Assembly Displayed Text Field
When debugging assembly code, the debugger uses the value of this text field to
determine the number assembly instruction disassembled at one time.

About ~/Library/Preferences/com.absoft.software.fx.plist
Besides user preferences, interface specific information such as the size and location of
debugger windows are also stored in the
~/Library/Preferences/com.absoft.software.fx.plist file each time the debugger is
terminated. This information is used to preserve the debugger windows screen placement
from debugging session to session. Use this information to assist you in organizing the
application's windows in a fashion that suits your needs. To restore the default
preferences and window locations, remove this file from your home directory.

~/Library/Preferences/com.absoft.software.fx.plist versus .fxinit
~/Library/Preferences/com.absoft.software.fx.plist was designed to take advantage of the
capabilities of the Macintosh and the OS X interface. Some functionality of the .fxinit file
may be overridden by the ~/Library/Preferences/com.absoft.software.fx.plist file for the
Macintosh interface, but the functionality provided is essentially the same and should not
cause any problems. ~/Library/Preferences/com.absoft.software.fx.plist only effects the
$case, $explang, $rnstep and $tabsize control variables and cannot be used to change
the values of other control variables.

Preparing Your Program For Debugging
In order to debug at the source level, the -g option must be specified when compiling
source code. By specifying the -g option, compilers will include additional information in
the executable program to describe the source files, procedures, and variables that are
part of the program.

The -g Option
Use the -g option of the standard UNIX compilers to take full advantage of Fx features.
Be aware that the -g option requires compilers, assemblers, and linkers to perform
additional work, thus the process of compiling your code will be slower. In addition, the
information that allows you to debug at the source level increases the amount of disk
space required for your program.

For trivial programs this is not a consideration, but if the program consists of sixty or
seventy source files you may want to note that Fx does not require that all of your source
files be compiled with the -g option. When debugging large programs, you may wish to
select a subset of your source files to compile with the -g option. For example, if you are
attempting to add a new feature to a program, you might re-compile only the files that
implement the new feature.
When preparing your program for debugging, you should keep in mind that many compilers will have options which adversely affect source level debugging. In general, compiler options which cause optimization, such as the \( -O \) option, should not be used while you are still debugging a program. Although it is not impossible to debug optimized code at the source level, optimizations can create additional problems.

**Starting an Fx Debugging Session**

This section gives information on loading programs for debugging with Fx.

**Launching the Debugger**

There are several ways to activate the debugger in the Macintosh OS X environment. You will have to decide which method is the most practical for the program you want to debug. The most important consideration is how you will interact with your program.

If your program requires standard input you should launch the debugger from a Terminal window, so that you may communicate with your program. When you are running the Fx debugger from the Terminal window, your program continues to run in the Terminal window. Since there is no conflict between this window and the windows used by the interface, Fx will not intercept the Terminal input and output of your program, however, you can still redirect the input and output to files using the run command.

If your program runs completely within its own interface windows, or does not require standard input, you may find it easier to launch the debugger from its freestanding or docked icon. Standard output for the debugger warnings and your program will be sent to the Console application log window for you.

**Loading a Program for Debugging**

Start Fx by double clicking its freestanding or docked icon, or launch it from a Terminal window. From the Terminal, you can begin debugging a simple program by entering "open –a Fx.app" and pressing the Return key.

In either case, Fx will display the **Initialize Fx** dialog that is used to specify the executable file, core file, source path(s), and the working directory.
Executable File Text Field
The executable file is the program to be debugged. Since the standard compilers usually create an executable program named `a.out`, Fx will attempt to supply this default name in the Executable File field for the current working directory. If you need to specify the file to be used, enter the name of the file in the Executable File text field, or use the file selection browser button to select the file from the standard open file dialog.

Core File Text Field
Fx allows you to use the core file of a program to determine the location of errors in the code. Fx will automatically use the core file if it exists in the same directory as the program being debugged. If there is no core file in the directory, the Initialize Fx dialog will show “none” in the Core File field. If the core file is in another location, you must specify it by typing its pathname into the Core File text field. Core files are also discussed in the section entitled, Debugging with Core Files later in this section.

Source Path(s) Text Field
Source path(s) field is the directory or directories that contain the source files for the program.

Working Directory Text Field
This text field specifies the directory that will become the program's current directory when running under Fx.

Using the File Selection Dialog
At the right-hand side of each text field in the Initialize Fx dialog there is a buttons that will invoke a file selection dialog. These buttons allow you to choose the executable or core file, along with the corresponding directories. After selecting a file from the standard open dialog, click Open to return to the debugger.

Once the necessary files have been specified in the corresponding text fields, click the OK button in the Initialize Fx dialog to start the debugger.

Executing Fx Commands During Initialization
You can also have Fx automatically execute commands after it has loaded your program. This section discusses the Fx initialization file which makes this possible.

About .fxinit
Each time you start a debugging session, Fx looks for a file named `.fxinit` in the current working directory and in the directory specified by the environment variable HOME. If this file is found, Fx will execute any commands it contains. If `.fxinit` exists in both directories, the file in the HOME directory will be read first.
The primary purpose of this file is to automatically redefine the internal variables within Fx, allowing you to customize Fx's defaults for command behavior. These internal variables, referred to as control variables, are listed in the appendices. An initialization file can also be used to execute any Fx command. For example, if you are repeatedly debugging the same procedures in a program, you might create an initialization file that will automatically set breakpoints on these procedures.

A sample .fxinit file

# Lines which start with the character # and blank lines
# are ignored, allowing comments to be added if desired
# Set the format used for displaying single precision values
change $ffmt = "(E12.4E3)"
# Set the maximum number of array elements displayed when
# unsubscripted arrays are used with the print command
change $acount = 10
# Set the default expression language to FORTRAN
change $deflang = "FORTRAN"
# Set the default tab spacing
change $tabsize = 5

Restarting A Debugging Session

When you start a debugging session, you may realize that you failed to specify a directory containing files needed to debug your program. The Load menu item is used to restart Fx and specify the new directories without terminating the debugging session.

The Load Menu Item

The **Load** menu item in the File menu allows you to change the command line arguments entered when Fx was first launched. The menu item invokes a dialog called the **Reinitialize Fx** dialog, to allow you to change any of input parameters including the executable, core file, the list of source directories, or the working directory.
For example, if you started Fx and found that you forgot to specify the source path /home/spam/sysdep/, you can restart the debugging session by doing the following:

**Example:**

- Choose **Load** from the File menu.
- Enter /home/spam/sysdep/ in the **source path(s)** text field.
- Press the **OK** button

Since the program name and the core file were not changed, Fx will continue to use the original arguments specified: a.out and none.

You can also add additional directories for use with the original program, or specify a completely different directory list. When you specify a new source path list, Fx will replace the current list of source paths.

The **Load** menu item can also be used to begin a debugging session on a completely different program. If you have finished debugging a.out, you can begin debugging another program using a completely different set of parameters.

**Debugging with Core Files**

When running a program under development, a message similar to

```plaintext
Segmentation violation
```

may appear. When this happens the operating system may create a file that contains an image of your program's memory at the time the error occurred. This file, a core file, can be used in Fx to track the errors in the program. Since core files are often enormous in size, they by default may not created on OS X. In order to allow core files to be created you must raise the core file limit from the 0 kbytes default value. To view the current limits, from a shell type, `limit`, and a listing of all the current settings will be displayed. Use the limit command to reset the `coredumpsize`.

**Example:**

To reset the `coredumpsize` to a ten megabyte limit

- Type `limit coredumpsize 10megabyte` from the shell

**Using Core Files**

At the beginning of a debugging session, Fx by default looks in the current working directory for a core file. If a file exists, Fx uses it to determine the location of errors and display the offending source line. Since your program is not executing when a core file is used, the commands for executing programs will be dimmed. If you would like to use a core file that is located elsewhere you may use the **Initialize Fx** dialog to set the pathname of the desired core file.
Often, a core dump will be produced by an error in one of the libraries linked with your program. This can occur because of incorrect arguments passed to a library routine or because there is a bug in the library itself. In either case, you are unlikely to have source code for the library and Fx will display a screen full of assembly language instructions. What you need to know is the point in your program where the library routine was called.

Using the Stack... menu item you can display a list of all the procedures in your program, starting with the main entry point, and ending with the procedure which caused the core dump. By examining this list, you can determine where the error occurred in your program.

When you have determined the location, one course of action is to set a breakpoint and start program execution with the Run menu item. When the breakpoint is encountered, you can use the Print menu item to verify that you are passing the correct arguments to the routine that caused the error.

When a debugging session is initiated without a core file, Fx will load the program and execute it until its main procedure is entered. Further execution of the program under the debugger interface is controlled using the buttons in the main window and the menus.

Examining Source Code

Source code can be viewed within the current source code file or within other source files.

Viewing Multiple Source Files

The Source window simplifies examining source code. With the Source window you can easily find the name of a source file, function or procedure and instantaneously view the source code. This is particularly useful when debugging code written by another programmer when you know the name of a procedure, but do not know its source file. The Source window is accessed through the Source menu item in the Program menu.
The left side of the window displays the source files for a program. Click on the name of a source file to view all of its functions. To display the source code for a function or a source file in the Source Code pane, double-click on the desired item, or highlight it and select the View button. If you want to set a breakpoint without actually viewing the code for a function, use the Break In button to set a breakpoint at the first executable statement of the selected function. To view a source file in a separate debugger window, that supports the Selection menu items and text searching, select the desired source file and click the List File button.

Ordering the Functions

If you prefer to view the functions of a source file in alphabetical order, click the Alphabetical radio button. If you want to view the functions in the order as they appear in a source file, click the Numerical radio button, and the functions will be sorted by line number.

When you want the Source Code pane to return to the next executable source line, use the Show Current menu item in the Edit menu.

Viewing Other Text Files

The Source window can only display files that comprise the source code of a program. To examine other files, such as an include file, to determine the value of a predefined constant or FORTRAN parameter, you can use the List File menu item under the File menu. This menu item allows you to examine the contents of any text file without leaving the debugger. When the List File item is invoked, the standard File Selection dialog will appear. Use this dialog to select and open the include file or other text file. The selected file is opened in a debugger window that supports the Selection menu items and text searching.
Using the Selection Menu to Work with Functions

Alternatives to the Source window are the View Code and Break In menu items of the Selection menu. These menu items in combination with a selected function name, chosen from any one of the many debugger windows, can be used to display code in the Source Code pane. To view the code for a function, select the function name within a statement and choose the View Code item. To set a breakpoint in a function, select the procedure name and choose the Break In menu item.

Viewing Execution Status

The Stack menu item in the Fx Debugger window is used to display the chain of procedure calls which produced the current program state. This item will invoke the Stack window.

In the window, each procedure in the current chain is listed along with the calling procedure that called it and the calling procedure's file and line number if available. Using the View Code command from the selection menu, a procedure may be selected and viewed in the Source Code pane.

Recreating a Debugging Session

When debugging large programs, it often takes a considerable amount of work to get the program to the point where it fails. This process may involve entering many different Fx commands, or changing the values of program variables at different points in the program. Sometimes you will get the program to the point of failure and then accidentally execute the wrong command.

Fx provides the Log menu item to record all the commands you have entered in a separate file and then play them back. This allows you to recreate the debugging session without having to reenter the commands.
Using the Log Menu Item

The Log feature of the debugger allows you to create a record of a debugging session in a separate file. Use the Log Commands button from the Log dialog or the Toggle Log On/Toggle Log Off menu item to turn logging on.

![Log dialog](image)

After you have specified the name of a log file, Fx will store a copy of all subsequent commands used into the file. Fx will, only log those commands that execute successfully. In addition, you can selectively turn logging on or off by toggling the Toggle Log On/Toggle Log Off menu item. You may wish to do this to avoid logging commands such as the Print command which do not affect program execution.

Using the Read Item

After you have recorded a debugging session, you can use the Read Log File menu item in the File menu to replay it. In addition to recreating entire debugging sessions, you can also use the Read Log File command to execute any file of Fx commands. Fx will then execute all of the commands in the file as if you had entered them using the menu items or used the command line box.

When recreating a debugging session, you must make sure that your program is in the same state that is was when you first turned on the Log command. For this reason, you may want to make the Log command the first selection you enter after starting a debugging session.
EXECUTING PROGRAMS IN FX

The previous section covered the basics of launching programs in Fx. This section covers methods you will use to execute programs within the debugger.

- **Executing Single Statements**
  Discusses the use of the **Step Over** and the **Step Into** commands to execute single statements.

- **Using Breakpoints**
  Describes what a breakpoint is, how to install and remove breakpoints, and how to automatically execute a list of commands when a breakpoint is encountered.

- **Executing Programs**
  Provides a guide to restarting or resuming program execution during a debugging session.

- **Animating Program Execution**
  Details how to animate program execution by repeating commands.

- **Calling Program Procedures**
  Describes how to test procedures with different arguments and execute special debugging routines.

**Executing Single Statements**

Instead of executing a program from beginning to end, you may need to execute your program one statement at a time. Fx provides the **Step Over** and **Step Into** menu items to execute single statements. Both items are located in the Debug menu.

**The Step Into Command**

The **Step Into** command will execute the next source line of your program. If the statement that is executed with the **Step Into** command is a call to a subroutine or function, execution will progress through the subroutine or function call and stop at the first executable source line in the function or subroutine.

**The Step Over Command**

This command also executes single statements. However, it does not follow subroutine or function references, like the **Step Into** command, but rather stops on the next source line of the current procedure.
Returning from Subroutines

When executing your program one statement at a time, you may accidentally execute the Step Into command when you really meant to execute the Step Over command. Suddenly, your program is stopped in a procedure in which you have no interest. The Return command will resume execution until the current procedure returns to its calling point, or a breakpoint is encountered.

Using Breakpoints

Although it is possible to use the Step Over and Step Into commands to execute your program until you determine where a problem is, this process is inefficient for most programs. Your program may require a complex series of events to occur, or it may need to run for a considerable amount of time before a problem shows up. Using breakpoints, allow you to execute your program at full speed until a specific procedure or source line is encountered.

Breakpoints are implemented by replacing the machine language instruction at the specified location with an instruction that will cause your program to stop execution. When this instruction is executed, control returns to Fx which then determines that a breakpoint has been encountered and restores the original instruction. When execution of the program is resumed Fx installs the breakpoint again after executing the replaced instruction. You can set breakpoints with the Toggle Break or Break In commands from the Selection menu, from the Breakpoints window, or from the Source window.

Using the Selection Menu to Set Breakpoints

The Selection menu allows you to set breakpoints at a line of source code, or within a subroutine or function in your program.

Setting a Breakpoint on a Line of Source Code

The easiest way to install a breakpoint for a line of source code is to use the Toggle Break command from the Selection menu.
Example:
✓ Triple-click on a source code line in the Source Code pane.
✓ Choose Toggle Break from the Selection menu.

The installed breakpoint symbol, “B”, will appear on the left side of the source code line, indicating that the line contains a breakpoint.

Setting a Breakpoint in a Procedure
To set a breakpoint within a function or procedure, use the Break In menu item in the Selection menu.

Example:
✓ Select the function name.
✓ Choose Break In from the Selection menu.

The installed breakpoint symbol, “B”, will appear on the left side of the source in the Source Code pane.

Setting a Breakpoint with the Breakpoints window
The Breakpoints window allows you to specify a breakpoint location, a skip count, and debugger commands to execute when the breakpoint is reached.
Location Text Field
The location of the breakpoint is entered into this field. A breakpoint location can be specified in one of three ways:

1. For source lines enter:
   “filename”: line number
   or just enter
   :line number
   to use the current source file. The line number must be an integer constant. If the line number is not specified, the breakpoint will be set at the current location of the program counter.

2. For procedure names, enter:
   procedure name: line number
   If the number 1 is used, the breakpoint will be set on the first line of the procedure. If a line number is not specified with the procedure, the breakpoint will be set on the first instruction of the specified procedure. However, if you stop execution on the first executable instruction, you will not be able to examine or modify the values of local variables and procedure arguments until the procedure's preamble code has been executed.

3. For assembly language addresses enter:
   address
   Useful address expressions for specifying breakpoints include the name of an entry point, the name of an entry point plus an integer offset, or an absolute address specified as an integer constant

Associate Text Field
The Associate text field names Fx commands to be invoked when a breakpoint is encountered. You might use this text field to enter the name of a variable whose value is to be printed each time a breakpoint is reached.

Multiple commands may be specified by separating each command with a semi-colon. By doing so, you may be able to temporarily fix a problem without having to edit and compile your source code. Since you can associate any list of Fx commands with a breakpoint, it is possible to stop program execution, change the value of a variable, and then resume program execution without having to enter the commands each time that the breakpoint is encountered.
Example:

Consider the following FORTRAN function:

```
REAL FUNCTION sumarray(array,size)
REAL array(size), result
INTEGER i, array_size, size
array_size = size
DO 10 i=1, array_size
   result = result+array(i)
10 CONTINUE
sumarray = result
RETURN
END
```

Since the local variable `result` is not initialized to zero, this function will return unpredictable results. This problem can be temporarily fixed by entering

```
sumarray:1
```

into the Location text field and the following into the Associate text field:

```
change result=0.0;continue
```

Skip Counts

The Skip Count text field allows you to specify the number of times the breakpoint is to be ignored before program execution is halted at the breakpoint. During the course of a debugging session, you may find that you have isolated a bug in a specific loop in your program. You can set a breakpoint on the start of the loop and repeatedly press the Continue button until you find the problem. However, the problem may only occur on the 100th, 1000th, or 10000th iteration of the loop.

Since pressing the Continue button 100 times can be time consuming, Fx provides the Skip Count text field as a way to automate this procedure. If you are debugging a loop where an error occurs on the 100th iteration, you can set a breakpoint to be invoked at this error by entering a value of 100 in the text field.

Listing Breakpoints

A dynamic list of all active breakpoints in your program is displayed at the top of the Breakpoints window. This list shows each breakpoint location, skip count, the number of times the breakpoint has been skipped, and any associated commands.

Removing Breakpoints

You may find that you have set a breakpoint at an inappropriate location, or a breakpoint may have served its purpose and is no longer needed. You can remove a breakpoint by selecting it from the list and clicking the Delete button.

To immediately remove all breakpoints from the program, use the Delete All button, or menu item
As a convenience, it is possible to view the code for a particular breakpoint from the Breakpoints window by selecting the desired breakpoint from the list and clicking the View button. The section of source code containing the breakpoint will be shown in the Source Code pane.

Executing Programs

Often you may need to specify arguments or resume program execution after debugging with breakpoints. The Run and Continue commands allow you to execute programs according to these needs.

Using the Run Command

The Run item restarts program execution, and allows you to specify arguments to the program. When you select the Run command, the Run Program dialog appears.

Example:

To restart execution of the current program with the arguments one two three:

- Choose Run from the Debug menu.
- In the Program Arguments text field, type in one two three.
- Press the OK button in the Run Program dialog.

Once you have specified arguments using the Run command, Fx will continue to use those arguments until you specify different ones. To change arguments simply enter the new arguments in the Run Program dialog.

Using the Continue Command

The Continue command resumes execution of a program that has been halted. When invoked, the program is executed until an error occurs, a breakpoint is encountered, or the program runs to completion.

Animating Program Execution

Commands do not have to be issued one at a time, instead, they can be sent to the debugger to be executed repeatedly. The Walk menu item in the Debug menu provides this ability.
Walk Dialog

When the **Walk** menu item is selected from the Debug menu, the **Walk** dialog appears. Specify desired debugger commands in the **Walk Command(s)** field, separating multiple commands with semi-colons.

![Walk dialog](image)

**Example:**

To repeatedly single step through a program while stepping into functions and printing the value of the variable `T`:

- Type `step; print T` into the **Walk Command(s)** field.
- Click **Start**.

**Example:**

To repeatedly single step through code while stepping over functions, and printing the `pc` register value:

- Type `Step; print $pc` into the **Walk Command(s)** text field.
- Click the **Start** button.

Controlling the Animated Execution

After you have initiated a walk command, the debugger enables the **Stop** button of the **Walk** dialog. As the command is repeatedly issued by the **Walk** command, click and hold this button until the debugger responds to halt animated execution at any time.

You may want to speed up or slow down the speed at which **Walk** commands are executed. To change the speed of execution, drag the **Speed** slider of the **Walk** dialog left or right as desired.

Calling Program Procedures

Any time your program is stopped at a breakpoint, Fx allows you to call any of its functions or subroutines as if they were the next statement to be executed. You can use this feature to test a particular routine with a variety of arguments or to call debugging procedures to format data or test specific conditions.
Using the External Procedure Menu Item

The **External Procedure** menu item allows you to execute any routine in your program out of sequence. You can pass arguments to the procedure, or set a breakpoint in the procedure and execute Fx commands while this procedure is active.

When a function is called with this menu item, Fx will automatically display the function's result when it returns.

![Execute Procedure dialog]

**Example:**

If you are debugging a FORTRAN program which contains the following function:

```fortran
REAL FUNCTION SUM3REALS(X,Y,Z)
REAL X,Y,Z
SUM3REALS = X+Y+Z
RETURN
END
```

You can execute this function by entering the following into the **Procedure Name** text field:

`SUM3REALS`

and **Procedure Arguments** text field:

`1.0, 2.0, 3.0`

In the preceding example, the arguments to the function `SUM3REALS` were constants, but they can just as easily be variables or expressions. You are not limited to passing constants to routines called with the **External Procedure** menu item. Keep in mind that Fx does not assure that the number and type of arguments you are passing agrees with what the routine is expecting.

You can also use the **Procedure Arguments** text field to format arbitrary portions of program memory as if it represented a particular data structure in your program. In order to do this, you need only create a small debugging function that takes a pointer to the data structure as an argument and returns a pointer to the arguments as a result.
Setting Breakpoints in External Procedure

Breakpoints can also be set in a procedure before calling it with the External Procedure menu item. Execution of the procedure will stop just as if it had been called normally. Afterwards, other Fx commands can be used to execute the procedure or to display the procedure's arguments and variables.
WORKING WITH PROGRAM VARIABLES

An advantage of source level debugging is the ability to display the values of program variables without having to insert special debugging statements into your program. This section focuses on ways to work with variables during a debugging session.

- **Displaying Variables**
  Presents the menu items for displaying the names, types and memory addresses of your program's variables, arrays and structures.

- **Modifying Variables**
  Provides a guide to changing the values of your program's variables.

- **Finding Overwritten Variables**
  Describes the use of monitors to stop program execution when a variable changes value.

Displaying Program Variables

Viewing program variables can be instrumental in tracking down problems in your code. Fx allows you to examine the contents of your program's variables and data structures whenever your program has stopped execution.

**The Print and Print* Commands**

Showing the value of simple variables is easy; use the Print menu item located in the Selection menu. With this item, the contents of your program's variables are displayed in the Command Output pane of the main debugger window.

For example, if a FORTRAN program defined the following variables:

```fortran
INTEGER I
REAL X
COMPLEX Z
```

You can display their values by highlighting each variable in the Source Code pane and choosing Print from the Selection menu. The values of $I$, $X$, $Z$ will be displayed in the Command Output pane, and will be in the format appropriate for their respective types: $I$ as a decimal integer, $X$ as a single precision floating point number, and $Z$ as a single precision floating point pair. Fx will use the symbol information output by compilers to determine the type and size of a variable.

The Print* item is automatically available for source code written in C and dereferences a pointer variable, sending the value to the Command Output pane.
Displaying Arrays, Structures and Unions

The contents of arrays, structures and unions are displayed in the Command Output pane with the Print menu selection. Individual elements of an array, structure, or union are displayed by highlighting the element in the Command Output pane and choosing the Print menu selection. Pointers to structures and unions can be dereferenced using the Print* menu item.

If the name of an array is selected, Fx will display every element using the indexing conventions of the language in which the element was declared. Fx will also use a format appropriate for its type, or an explicit format can be specified with the Format dialog.

Setting Formats

The Format dialog allows you to select how variables will be printed in the various panes of the debugger. By clicking on the Type and Size buttons, a selection can be made from among the available choices in the pop-up lists. An item in the pop-up list may be dimmed if it is invalid in the current context of the debugger.

<table>
<thead>
<tr>
<th>Address</th>
<th>prints the address of the variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>prints the contents of the variable as a binary integer</td>
</tr>
<tr>
<td>Character</td>
<td>the variable is printed as an ACSH character</td>
</tr>
<tr>
<td>Decimal</td>
<td>prints the variable contents as a decimal integer</td>
</tr>
<tr>
<td>Double</td>
<td>the value is displayed as a double precision floating point number</td>
</tr>
<tr>
<td>Hex</td>
<td>prints the contents of the variable as a hexadecimal integer</td>
</tr>
<tr>
<td>Implicit</td>
<td>selects format based on the type of the variable</td>
</tr>
<tr>
<td>Octal</td>
<td>prints the contents of the variable as an octal integer</td>
</tr>
<tr>
<td>Single</td>
<td>the value is displayed as a single precision floating point number</td>
</tr>
<tr>
<td>String</td>
<td>the value is displayed as a null terminated string</td>
</tr>
<tr>
<td>Unsigned</td>
<td>prints the contents of the variable as an unsigned decimal integer</td>
</tr>
</tbody>
</table>

Implicit: the actual size of the variable

Byte: one byte
Short: two bytes
Long: four bytes
Evaluating Expressions

In addition to displaying the values of variables, the Print command will also evaluate expressions that involve variables, numeric constants, and source language operators. These expressions can be as simple as adding 10 to the contents of a variable or can include multiple variables and the complete set of FORTRAN intrinsic functions. To evaluate an expression, highlight it and choose the Print command from the Selection menu.

Watch and Watch* Commands

At some point during a debugging session, you may want to observe the value of an expression as your program executes. The Watch and Watch* commands will invoke the Watch Variables window which shows the value of selected variables as the program executes. The next example shows how to display the value of the array element \( a(i,j) \) when program execution stops.

Example:

- Highlight the array element
- Choose the Watch command from the Selection menu to invoke the Watch Variables window
- Use the Step Over command to execute the program. The value of the array element changes in the Watch Variables window during execution

Formatting for the Watch and Watch* commands is set using the Format dialog.

The Symbols Window

The Symbols window is used to print and watch the contents of variables, expressions and structures within your programs. The Symbols window has additional features not available with the Print and Watch items described above.
The Symbols window contains four main sections: the Symbol Browser pane, the Type pane, the Symbols Window Output pane, and the Expression pane.

The Symbol Browser pane allows you to view the different types of variables available:

- **Global Symbols**: global variables that are defined in the current source file.
- **Local Symbols**: arguments and local variables for the current procedure.
- **Static Symbols**: file scope static variables for the current source file.

To choose a type of variable to view single-click the symbol type in the first column of the Symbol Browser pane. When you single-click upon a particular variable in the second column of the Symbol Browser pane, the type of the variable is displayed in the Type pane. A double-click on a variable in the Symbol Browser pane causes the value to be displayed in the Symbols Window Output pane. The Symbols Window Output pane also displays the values of expressions typed into the Expression text field. Use the grabber icon of the splitter bars to resize the panes of the Symbols window.

The Expression pane has buttons similar to the menu items in the Selection menu, but unlike the Selection menu items these buttons only pertain to operations on the Expression text field. Any source language expressions, such as array indexing and pointer dereferencing, can be evaluated by entering them in the Expression text field. When the Return key is pressed in the Expression text field, a click on the Print button is simulated, and the result is sent to the Symbols Window Output pane. Select the
buttons located at the bottom of the **Symbols** window to perform other operations on the **Expression** text field.

**Setting Formats and Sizes**

Variables can be printed in additional formats and sizes using the **Format** button to invoke the **Format** dialog, or enter the format directly in command line form at the end of the **Expression** text field.

**Modifying Variables**

Fx allows you to modify the values of program variables when execution of your program is stopped. The **Change...** item is used to assign new values to program variables and Fx control variables.

**The Change ... Item**

The **Change...** item invokes the **Change** dialog. The **Change** dialog allows you to enter the name of the variable and the desired value.

![Change dialog](image)

When assigning new values to variables with the **Change** dialog, Fx will perform the appropriate type conversions when possible and inform you when you have specified a value that is inappropriate for the variable you are modifying. New values can be specified as constants or expressions.

**Finding Overwritten Variables**

One difficult bug to track down is when a program variable is overwritten by a statement in the program that doesn't appear to reference it. Array boundary errors and invalid pointers are typical causes of this behavior.

Sometimes, you will have no idea where in your program the error occurs. When faced with this type of problem you have little choice but to execute portions of your code and check to see if the variable in question has been modified. This type of debugging is both frustrating and time consuming. Fortunately, Fx provides monitors, a feature that will perform much of this task for you.
Using Variable Monitors

When you suspect that the contents of a variable is being overwritten, you can use the Monitors window to set a monitor on a variable that will stop program execution when the value of that variable changes. You can also install monitors that will halt execution of the program when the value of a variable meets a specific condition.

The Monitors Window

The Monitors window includes the Monitor Browser pane which allow you to view, and select active monitors, the Variable and Value text fields for setting and modifying monitors, the Condition pop-up list, and various action buttons.

![Monitors window](image)

Setting a Monitor Variable

Choose the Set button to create a monitor using the current values of the Variable and Value text fields, and the Condition pop-up list. The condition can be set at equal to, not equal, greater than, or less than the value. Be sure to enter the name of the variable, choose the condition for the monitor value, and enter the value at which to stop program execution, before selecting the Set button.

**Example:**

To stop program execution when a variable changes its value:

- Enter the name of the variable into the Variable text field
- Using the Condition pop-up list, choose “not equal”
- Click the Set button
- Execute the program with the Step Over command
Halting Execution

Since the value is not specified, it is implied that the value is equal to the current value. When the value of the variable is changed, an alert dialog will appear and show information on the variable, including its monitor id and current value. While this alert dialog is active the monitor variable that caused execution to stop will be highlighted in the Monitors window. When the dialog is closed, the monitor will be removed.

![Monitor Activated Alert dialog](image)

When the Continue command is used with a variable monitor installed, Fx will execute one source statement of your program and then check the value of the monitored variable. If it has changed, execution of the item will be terminated; otherwise the next source statement will be executed. This process is repeated until the monitored variable changes value or the command finishes execution. After you have installed a variable monitor, when you press the Continue command, you will notice that execution is much slower.

Removing Monitors

Variable monitors are automatically removed when they stop program execution. However, you may wish to remove one before this occurs. The Remove button in the Monitors window allows you to delete the monitor currently selected in the Monitor Browser pane. To remove all active monitors the Remove All button and menu items are provided.

Listing Monitors

All the monitors you have installed, are displayed in the Monitor Browser pane of the Monitors window. This list displays the monitor id numbers, all variables being monitored, and the value that the variables had when the monitor was installed. The id of the monitor is useful if you issue the Unmonitor command from the Command text field.
ASSEMBLY LANGUAGE DEBUGGING

The previous sections described methods and commands for source level debugging. This section describes the features of Fx which support debugging programs at the assembly or machine language level. If most of your programming is done at the source level, you can skip this chapter. However, chances are that eventually you will need to descend to this level of debugging, so you may want to become acquainted with the facilities that are available. This section discusses the following topics:

- **Using Fx Commands at the Assembly Level**
  Presents details on using the commands described in the previous chapters when debugging at the assembly language level.

- **Displaying Program Memory and Registers**
  Presents how to display program memory, and how to display and modify the value of program registers.

**Using Fx Commands at the Assembly Level**

For the most part, Fx commands that were presented in the previous sections can be used at the assembly level. This section provides details on using these commands to debug programs in assembly language.

**Examining Assembly Language Code**

When source level debugging information is not available for a procedure, Fx will automatically switch to assembly language debugging. If you are debugging a procedure that has source level information, you can switch to assembly language debugging by using the **Toggle Source** command.

The **Toggle Source** command displays the disassembled code in the **Source Code** pane. Using this command, the display of the source code can be switched back and forth as necessary to view the code in assembly language or in a high level language. The **Source Code** pane will function in the same way in assembly language: text can be viewed using the scroll bar and the commands in the Selection menu are available.

When Assembly Language is displayed in the **Source Code** pane, the **Step Over** and the **Step Into** commands execute single instructions.

**Resuming Program Execution**

You can use the **Continue** command to resume execution of your program until a particular entry point or instruction is reached. As long as your program has not had its symbol table removed, you can also use the **Return** command to resume execution of your program until the current procedure returns to its calling point.
Setting Breakpoints at the Assembly Level

When debugging at the assembly language level, you can use the **Breakpoints** window to install breakpoints on the entry points to procedures and on particular instructions within procedures. You can also specify skip counts for these breakpoints, and use the associate text field to automatically execute commands when a breakpoint is encountered.

Displaying and Changing Registers

Any time your program has stopped execution, you can display and modify the current values of machine registers. The **Register**... menu item in the Program menu will display the contents of all machine registers.

Using the Register... Menu Item

To display the registers, choose the **Register**... menu item to invoke the Register window.

![Register window](image)

All registers will be shown, along with their values. If a particular machine provides dedicated floating point registers, the values of these registers will be displayed as floating point numbers. All other registers will be display as hexadecimal integers.

Individual registers can be displayed in a variety of formats using the **print** command in the **Command** text field located in the Main Window. When referring to individual registers, the register name must be preceded by a “$” to distinguish it from symbolic names of variables. For example, to print the value of register D0, you would enter `$D0` into the **Command** text field.

Displaying Program Memory

Fx provides a menu item that displays arbitrary locations in program memory. The **Memory**... menu item in the Program menu allows you to output the contents of a single memory location or a range of memory locations to the **Memory Window Output** pane of the **Memory** window.

Fortran User Guide
Using the Memory... Menu Item

Using the Memory... item, you can display memory contents in a variety of formats and sizes, and specify the location to display in a number of different ways. The menu item invokes the Memory window.

![Memory window](image)

The Address text field is used to specify the memory address. The Format... button invokes the Format dialog in order to allow you to choose the display format of the address, and to select the size of the address. To specify the number of memory locations, use the Count text field.

When the Display button is clicked, the memory locations will be shown in the Memory Output pane.

Monitoring Registers and Memory Locations

You can use the Set button in the Monitors window to halt execution of the program when the value of a register or memory location changes at the assembly level.

When working with monitors at the assembly level, the Continue command will execute the program. When the Instruction Step Into item is selected and the Continue command is pressed, the program will execute, including any procedures and function references, and check the value of the monitor. When the Instruction Step Over item is selected, the program will execute the instructions and check the value of the monitor, but does not follow any references to functions or procedures.
COMMAND ARGUMENTS

In order get first time users started with Fx as quickly as possible, the previous chapters have glossed over the details of specifying arguments to Fx commands. This chapter provides more detail on the items that can be specified as arguments to Fx commands. The chapter covers the following topics:

- **Identifier Scoping**
  Describes the scoping conventions used for Fx command arguments.

- **Specifying Constants**
  Describes the syntax for entering constants as command arguments.

- **Specifying Registers**
  Describes using machine registers as command arguments.

- **Expression Interpretation**
  Discusses the interpretation of variables, entry points and constants when used in expressions.

**Identifier Scoping**

Identifier scoping refers to the identifiers that are accessible at the current state of the program being debugged. Some arguments are not dependent upon the program and are always available. Program constants, as well as Fx Control Variables, internal variables within the debugger, would be examples of these type of arguments. Control Variables are listed in the appendices.

Other arguments, such as local variables in the program, are only accessible when the procedure in which they were declared is active. Fx will implicitly determine the appropriate scope, or an identifier's scope can be explicitly stated when necessary.

**Implicit Scoping**

When identifiers are program specific items, Fx determines the appropriate scope using two sets of scoping information: the *actual scope* and the *current scope*. A program's actual scope is the source line, procedure name and source file which contain the next assembly language instruction to be executed, or the last assembly language instruction executed if a core file is being examined. A program's current scope is the filename, procedure name and source line that appear in the Fx status display.

By default, the current scope is identical to the actual scope. However, the current scope may be changed with the scope, View, and view commands listed in **Command Reference** section. Fx will use the procedure name defined by the current scope for searching the program's symbol table for local variables, and the file defined by the
current scope when searching the program's symbol table for static variables and static functions.

**Explicit Scoping**

When necessary, it is possible to explicitly specify the scope for local variables. This allows for multiple activations of the same procedure. The syntax for explicitly scoping local variables is as follows:

```
procedure name {(level)} => variable name
```

where procedure name is the name of the procedure where the local variable is declared.

level is an integer constant and specifies a particular instance of the specified procedure in the current chain of program execution. Level zero refers to the most recent instance of the procedure, level one refers to the second most recent instance, and so on. If not specified, level defaults to zero.

variable name is the name of the local variable.

For example, if the subroutine sub has been called recursively three times resulting in three instances of sub in the current chain of execution, the following commands will display the value of the variable index for each instance of sub.

```
print sub=>(2)index
print sub=>(1)index
print sub=>(0)index
```

**Specifying Symbols**

This section discusses the interpretation of variable and procedure names.

**Symbol Names**

The first character of a symbol name must be an upper or lower case letter or an underscore. The remaining characters can be upper or lower case letters, digits, underscores, or dollar signs. A symbol name is terminated by the first occurrence of a character which is not one of the above. Symbol names are significant to 31 characters.

The Fx control variable $case controls case sensitivity during symbol table searches. This variable is initially set to “both” causing symbol table searches to be case sensitive. However, it can be set to “lower” or “upper” by entering the change command. When $case is set to “lower”, the symbol name extracted from the command line will be folded to lower case before searching the program's symbol table. When $case is set to “upper”, the symbol name will be folded to upper case before the search is performed.

On systems where convention dictates the prepending of a special character to symbol names, the Fx control variable Sleading can be used to eliminate the need to enter this
character every time a symbol name is used. For example, if convention dictates that symbol names must have a leading underscore, setting the value of $leading to "_" will cause Fx to strip a single leading underscore from any name in a program's symbol table, as shown below.

    change $leading = "_"

FORTRAN Symbols

This section describes the FORTRAN data types and symbols understood by Fx and discusses the scoping conventions for each symbol type, the indexing of FORTRAN arrays, and the syntax for specifying character substrings.

FORTRAN Data Types

Fx supports the following FORTRAN data types:

    INTEGER*1
    INTEGER*2
    INTEGER
    LOGICAL*1
    LOGICAL*2
    LOGICAL
    REAL
    DOUBLE PRECISION
    COMPLEX
    DOUBLE PRECISION COMPLEX
    CHARACTER
    RECORD

The INTEGER*1, INTEGER*2, LOGICAL*1, and LOGICAL*2 data types only apply to variables. There is no way to specify an INTEGER*2 constant. If a constant is assigned to an INTEGER*2 variable using the change command, the constant will be converted before the assignment is performed.

FORTRAN Subroutines and Functions

FORTRAN subroutine and function names are global to the entire program and are accessible at any time during a debugging session. FORTRAN statement functions are invisible to Fx and cannot be specified as command arguments.

FORTRAN Common Blocks

The names of FORTRAN common blocks are global to the entire program and are accessible any time there is a process or core file active. When used as arguments to commands, the contents of common blocks are assumed to be integers.
FORTRAN Local Variables and Procedure Arguments

The names of FORTRAN local variables and procedure arguments are always local to the procedure or function in which they were declared regardless of their actual location in program memory. Local variables and arguments are accessible when the procedure in which they are declared is defined by the current scope. They may also be explicitly scoped.

FORTRAN Array Indexing

FORTRAN arrays are indexed using the conventions of the FORTRAN language. Indexing is performed in column major order and array indices are specified using standard FORTRAN syntax. Individual array-indices may be specified as constants or as expressions involving variables, constants, and operators.

Unsubscripted array names may be specified as arguments to the print command causing every element of the array to be displayed. Note that assumed size arrays cannot be displayed in this manner because the size of the last dimension is unknown. The following examples illustrate FORTRAN array indexing.

```fortran
array (1)
array (i, 2)
array (i+4, k+3, m)
```

FORTRAN Character Substrings

Substrings of character variables and character array elements may be specified using standard FORTRAN syntax. The substring expressions can be simple integer constants or more complicated expressions involving variables, constants and operators. The following examples illustrate character substring syntax.

```fortran
charvar(1:6)
charvar U: j)
chararray(1,2)(i+1:7)
```

C Symbols

This section describes the C data types and symbols understood by Fx, and discusses the scoping conventions for each type, C array indexing, dereferencing pointer variables, and referencing members of structures and unions.
C Data Types

Fx supports the following C data types:

```c
char
unsigned char
short int
unsigned short int
int
unsigned int
long
unsigned long
float
double
```

Note that many C compilers will not make a distinction between `int` and `long` when producing program symbol information.

C Functions

C function names are global to the entire program unless explicitly declared with the reserved word `static`. Non-static functions are accessible at any time during a debugging session. Static functions are only accessible when the file in which they were declared is defined by the current scope.

C Extern Variables

C variables declared with the reserved word `extern` are accessible any time there is a process or core file. If no type information is available for external variables, the type `int` will be assumed.

C Static Variables

The scoping of variables declared with the reserved word `static` follows the conventions of the C language. If a variable is declared outside of a function, it is only accessible when the file in which it was declared is defined by the current scope. If it is declared inside a function, it is only accessible when that function is defined by the current scope. Static variables declared inside of functions may also be explicitly scoped.

C Automatic Variables

Automatic variables are only accessible while the function in which they were declared is defined by the current scope. Automatic variables may also be explicitly scoped. Note that Fx does not distinguish between automatic variables declared at the beginning of a function and those declared within a block of the function's statements.

C Array Indexing and Pointer Dereferencing

Array indexing is performed using the conventions of the C language. Indexing is performed in row major order and indices are specified using standard C syntax.
Individual indices can be specified as integer constants or as expressions involving variables, constants, and operators.

Unsubscripted array names can be specified as arguments to the print command causing every element of the array to be displayed. The following examples illustrate C array indexing:

\[
\begin{align*}
\text{array}[1] \\
\text{array}[i][1] \\
\text{array}[i+1][j+1]
\end{align*}
\]

Pointer variables may be dereferenced using the "*" operator, or they may be indexed as if they had been declared as one dimensional arrays. Consider the following C program fragment:

```c
int array[101];
int *aptr;
aptr = array;
```

The following sets of commands will produce equivalent output.

```c
print *aptr
print aptr[0]
print *(aptr+8)
print aptr[2]
```

Note that Fx does not multiply the constant 8 by the size of an integer before performing the addition.

**C Structure and Union Members**

Structure and union members may be specified as command arguments by using the and "." operators. The names of entire structures and unions may be specified as arguments to the print command causing every member of the structure or union to be displayed.

**Specifying Constants**

Constant arguments may be specified in one of the following forms: integer, floating point, complex, or character. The following sections provide details on each of these constant types.

**Integer Constants**

Integer constants can be entered in decimal, binary, octal, or hexadecimal form.
Decimal Constants

Decimal constants consist of an optional leading sign followed by a string of decimal digits [0-9]. Note that if a sign is not specified and the first digit is a zero, the constant will be interpreted as an octal integer as described below.

The following are valid decimal constants:

\[
10 \\
-22 \\
+100
\]

Binary Constants

Binary constants consist of the letter b or the letter B followed immediately by a string of binary digits [0-1] delimited by single quotation marks or apostrophes. If desired, spaces may be included in the string of digits.

The following are valid binary constants:

\[
b'101' \\
B'111 111'
\]

Octal Constants

Octal constants consist of the letter o or the letter 0 followed immediately by a string of octal digits [0-7] delimited by single quotation marks or apostrophes. If desired, spaces may be included in the string of digits.

Octal constants can also be specified using the form familiar to C programmers, where an octal constant consists of a leading digit zero followed by a string of octal digits [0-7].

The following are valid octal constants:

\[
o'555' \\
o555
\]

Hexadecimal Constants

Hexadecimal constants consist of the letter z or the letter Z followed immediately by a string of hexadecimal digits [0-9, A-F, or a-f] delimited by single quotation marks or apostrophes. If desired, spaces may be included in the string of digits.

Hexadecimal constants can also be specified using the form familiar to C programmers, where an hexadecimal constant consists of the leading digit zero followed the letter x or the letter X and a string of hexadecimal digits [0-9, A-F, or a-f].
The following are valid hexadecimal constants:

\[ \text{z'3F'} \]
\[ 0x3f \]

**Floating Point Constants**

A floating point constant consists of an optional sign and string of decimal digits which contains a decimal point. A floating point constant may have an exponent. An exponent is specified by the letter 'E' or the letter 'D' followed by an optional sign and a string of decimal digits. If an exponent character is specified and the fractional portion of the constant is zero, the decimal point may be omitted.

A floating point constant is initially converted to IEEE 96 bit extended precision regardless of the exponent character. After conversion to extended precision, the constant will then be converted to double or single precision depending upon the specified exponent character. Floating point constants specified with a V exponent character will be converted to double precision. Floating point constants specified with an 'E' exponent character, or without an exponent character, will be converted to single precision.

The following are valid floating point constants:

\[ 12.0 \]
\[ -12.999 \]
\[ 12.999E12 \]
\[ 12.9999D-12 \]

**Complex Constants**

A complex constant consists of a left parenthesis, followed by a pair of floating point constants separated by a comma, followed by a right parenthesis. Double precision complex constants are specified including a 'D' exponent character in one or both of the floating point constants.

The following are valid complex constants:

\[ (12.0, 12.0) \]
\[ (12.9999E-12, -12.9999E10) \]
\[ (100.0D0, 200.0D0) \]

**Character String and C Character Constants**

Character string constants are strings of ASCII characters delimited by either apostrophes or quotation marks. The delimiting character may be included in the string itself by representing it with two successive delimiting characters.

The following are examples of valid character string constants:

"hello world"    'hello world'
"America's finest"    'America''s finest'
If desired, special escape sequences may be embedded in character string constants by using the backslash followed immediately by one of the characters in the following table.

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>Newline</td>
</tr>
<tr>
<td>\t</td>
<td>Tab</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return</td>
</tr>
<tr>
<td>\f</td>
<td>Form feed</td>
</tr>
<tr>
<td>\b</td>
<td>Backspace</td>
</tr>
<tr>
<td>&quot;</td>
<td>Backslash</td>
</tr>
<tr>
<td>\nnn</td>
<td>octal value specified by nnn</td>
</tr>
</tbody>
</table>

The Fx control variable `$escchar` controls whether or not escape sequences are interpreted as described above. When set to zero, which is the default, escape sequences will not be interpreted. `$escchar` must be set to a non-zero value using the `change` command before escape sequences will be interpreted.

C character constants are specified as one-byte character strings. The Fx control variable `$escchar` must be set to a non-zero value in order to use escaped characters in C character constants.

**Specifying Registers**

Registers are entered using the names accepted by the system assembler. In order to distinguish them from symbol names, they must be prefixed with the character “$”. When used in expressions, the data type of registers is assumed to be integer. However, if dedicated floating point registers are available they will be typed appropriately. The contents of registers are always retrieved from the actual scope and are available whenever a process or core file is active.

**Expression Interpretation**

Many Fx commands accept expressions as arguments. Expressions can be simple scalar values, such as a numeric constant or single variable name, or can consist of multiple operands combined with the supported operators for the current expression language.

**Current Expression Language**

The current expression language is determined by the contents of the Fx control variable `$explang`. By default, this variable is set to “automatic” causing the current expression language to be determined by examining the extension of the file name defined by the current scope. When this file name ends in the characters “.c”, the current expression language is C. When the extension is “.f” or “.for”, the current expression language is FORTRAN. If desired, the value of this variable may be explicitly set to “C” or “FORTRAN” with the `change` command, allowing expression evaluation in either of these languages regardless of the current scope.
**Default Expression Language**

When the value of $\text{sexplang}$ is set to “automatic” and it is impossible to determine the appropriate language from the current scope, expressions will be evaluated in the language defined by the Fx control variable $\text{deflang}$. By default, this variable is set to “C” however it may be set to “FORTRAN” using the `change` command.

**Supported Language Operators**

When specifying expressions as arguments to debugger commands, operands may be combined using the operators of the current expression language. Type conversion between operands and operator precedence follow the conventions of the expression language. Note that parentheses may be used to force a specific order of evaluation regardless of the current expression language.

**FORTRAN Operators**

The following table lists the supported FORTRAN operators:

<table>
<thead>
<tr>
<th>FORTRAN operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
</tr>
<tr>
<td>.NEQV., .EQV., .OR., .AND., .GT.¹, .GE.¹, .NE.¹,</td>
</tr>
<tr>
<td>.EQ.¹, .LE.¹, .LT.¹, -,-,*,/,**,</td>
</tr>
<tr>
<td>Unary</td>
</tr>
<tr>
<td>+,-, .NOT.,@²</td>
</tr>
</tbody>
</table>

1. The operators .GT., .GE., .NE., .EQ., .LE., and .LT. may also be specified by >, >=, <>, ==, <=, and < respectively.
2. The `@` operator is used to specify the “contents of” and can be used to provide an additional level of dereferencing for FORTRAN symbols.

**C Operators**

The following table lists the supported C operators:

<table>
<thead>
<tr>
<th>C operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
</tr>
<tr>
<td>&amp;,,</td>
</tr>
<tr>
<td>Unary</td>
</tr>
<tr>
<td>~,!,+-,&amp;,*,+</td>
</tr>
</tbody>
</table>

**FORTRAN Intrinsic Functions**

In addition to the language operators listed above, Fx also provides the numeric FORTRAN intrinsic functions for use in expressions. These include the type conversion functions such as `FLOAT` and `INT`, the trigonometric functions such as `SIN` and `TAN`, and the bit manipulation functions defined by the DOD military standard MIL-STD-1753 such as `IOR`. The intrinsic functions are available regardless of the current expression language. References to intrinsic functions are distinguished from program entry points with the same name by the presence of an argument list.
Value Expressions

Value expressions evaluate to a single numeric value or character string that can be printed, passed as an argument to an intrinsic function or specified as the value to assign to a variable using the change command. Character string expressions are limited to single character string constants, character variables, character array elements, or character substrings. No operators are supported for combining character operands.

When all operands in a value expression are of the same data type, the type of the expression is the same as the type of the operands. When an expression involves operands with different data types, automatic conversion between data types occurs. The data type of the expression result is the data type of the highest operand as defined by the current expression language.

Address Expressions

Address expressions are a subset of possible value expressions and are used to refer to locations in a program's memory space. Since computers are not capable of addressing memory with floating point numbers or character strings, address expressions should only involve integer operands. Although it is possible to specify other types of operands, an error will be reported if the type of an address expression is not integer.

Operand Interpretation

Expression operands are interpreted differently depending upon whether they are used in value expressions or address expressions. The distinction between operand interpretation is generally transparent when debugging programs. Fx is designed to interpret an operand in the manner that makes the most sense for a particular command. For example, when the name of an entry point is used as an argument to the break command, Fx will use the address of the specified procedure as the address of the breakpoint.

The interpretation of any argument can be overridden by using source language operators and the FORTRAN intrinsic functions VAL and LOC. Consider the following print commands.

```
pindent main
print LOC(main)
```

The first print command will display the contents of the first memory location for the entry point `main`. The second will show the address of the entry point `main`.

Fortran User Guide
The following table lists the basic operands and the ways in which they will be interpreted in value expressions and address expressions.

<table>
<thead>
<tr>
<th>Operand</th>
<th>In value expressions</th>
<th>In address expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>numeric value</td>
<td>numeric value</td>
</tr>
<tr>
<td>register name</td>
<td>register contents</td>
<td>register contents$^1$</td>
</tr>
<tr>
<td>variable name</td>
<td>variable contents</td>
<td>variable contents$^2$</td>
</tr>
<tr>
<td>procedure name</td>
<td>contents procedure's first location when specified alone, procedure address when combined with operators</td>
<td>procedure address</td>
</tr>
<tr>
<td>control variable</td>
<td>variable contents</td>
<td>variable address$^3$</td>
</tr>
</tbody>
</table>

1. When a register name is used with the **monitor** command, the contents of the specified register name will be monitored for change. This is the only time a register is considered to have an address.

2. **The dump and monitor** commands will use the variable address when specified alone and the variable contents when used in expressions.

3. Fx control variables can only appear in address expressions when used with the **change** command.
### COMMAND REFERENCE

This section describes each Fx command. In order to assist in finding a particular command, the commands are presented in alphabetical order and the name of each command is followed by a short description of its purpose.

---

**!**

*Executing system commands*

**Description:**

The ! command suspends a debugging session and executes another command or program. After the command or program finishes, the debugging session is resumed at the point where it was suspended.

**Usage:**

```
! command line
```

where *command line* is the name of the command or program followed by its arguments as they would be typed at the shell prompt. All characters following the ! are passed directly to the shell.

**Example:**

The following command suspends the current debugging session and runs the text editor vi(l) on the file source.f:

```
vi source.f
```

**Notes:**

When entering multiple Fx commands on the same command line, it is permissible to precede the ! command by other debugger commands.

In order to use the ! command to recompile the program you are currently debugging, you must use the **kill** command before recompiling. After your program has been compiled, you must use the **reinit** command to make Fx aware of any changes that you made to your program.

**Related Commands:**

- kill, reinit
associate  

Description:

The **associate** command allows debugger commands to be executed when a breakpoint stops program execution. When execution stops at the specified breakpoint, the commands will be executed as if they had been entered from the command line. After the commands have been executed, further commands may be entered.

Usage:

```plaintext
associate address expression "command"
associate {"filename"} {line number} "command"
associate {procedure name} {line number} "command"
```

where *address expression* evaluates to the address of an instruction in the program. Useful address expressions for specifying breakpoints include the name of an entry point, the name of an entry point plus an integer offset, or an absolute address specified as an integer constant.

*command* is any valid debugger command. Multiple commands may be specified by separating them with semi-colons.

*filename* is the name of a source file which is part of the program being debugged. If *filename* is not specified, the file containing the current source line will be used.

*line number* is an integer constant. If the specified *line number* does not correspond to an executable source line but is within a procedure in the specified file, the breakpoint will be set on the next executable source line greater than *line number*. If the specified *line number* is between procedures, the breakpoint will be set on the nearest executable source line. If *line number* is not specified, the first executable source line in the specified file will be used.

*procedure name* is the name of a procedure in the program being debugged. If *line number* is specified, the breakpoint will be set on the first instruction of the specified procedure. If the specified *line number* is one, the breakpoint will be set on the first executable source line of the specified procedure. If a line number other than one is specified, the command is equivalent to “break *filename* *line number*”, where *filename* is the name of the file which contains the specified procedure.

Abbreviation:

*a*
Example:

The following command associates a command list with a breakpoint set on the first executable source line of the procedure subone:

```
associate subone:1 "Print 'At subone1'; registers"
```

Notes:

If a breakpoint is currently set at the specified location, the command list will replace any command list currently associated with it. Otherwise, the breakpoint will be set as if it had been specified with the `break` command.

Adding the `continue` command at the end of the list of commands associated with a breakpoint will cause execution to automatically resume after the other commands have been executed.

If an error occurs during execution of a command associated with a breakpoint, any other commands associated that breakpoint will be ignored.

Related Commands:

`break, delete, Delete, list breakpoints`
break

Setting breakpoints

Description:

The `break` command is used to place a breakpoint at a given location in the program being debugged. The location can be any valid address expression. The location may also be specified as a file relative source line number or as the first executable source line in a procedure.

Usage:

```
break {{address expression} {,skip count}}
break {{"filename"}::{line number} {,skip count}}
break {{procedure name}::{line number} {,skip count}}
```

where `address expression` evaluates to the address of an instruction in the program. Useful address expressions for specifying breakpoints include the name of an entry point, the name of an entry point plus an integer offset, or an absolute address specified as an integer constant.

`skip count` is an integer constant and represents the number of times the breakpoint is to be ignored before program execution is stopped. If `skip count` is not specified, a value of zero will be used causing program execution to stop the first time the breakpoint is encountered.

`filename` is the name of a source file which is part of the program being debugged. If `filename` is not specified, the name of the file containing the current source line will be used.

`line number` is an integer constant. If the specified `line number` does not correspond to an executable source line but is within a procedure in the specified file, the breakpoint will be set on the next executable source line greater than `line number`. If the specified `line number` is between procedures, the breakpoint will be set on the nearest executable source line. If `line number` is not specified, the first executable line in the specified file will be used.

`procedure name` is the name of a procedure in the program being debugged. If no `line number` is specified, the breakpoint will be set on the first instruction of the specified procedure. If the specified `line number` is one, the breakpoint will be set on the first executable source line of the specified procedure. If a `line number` other than one is specified, the command is equivalent to “break `filename line number`”, where `filename` is the name of the file which contains the specified procedure.

Abbreviation:

```
b
```
Example:

The following command sets a breakpoint on the location specified by the address expression \texttt{main+0x50}:

\texttt{break main+0x50}

The following command sets a breakpoint on the seventh line of the file \texttt{source.f}:

\texttt{break "source.f":7}

The following command sets a breakpoint on the first executable line of the procedure \texttt{subone}. A skip count of two is specified, so program execution will not stop until the third time the breakpoint is encountered.

\texttt{break subone:1,2}

Notes:

The number of times a breakpoint has been skipped is reset to zero each time program execution is restarted with the \texttt{run} command.

Related Commands:

\texttt{associate, delete, Delete, list breakpoints}
change

Description:

The change command is used to modify the contents of registers, variables, program memory, and Fx control variables.

Usage:

\[\{change\} \text{ address expression} = \text{value expression}\]

where address expression evaluates to the address of a memory location, a register, or a debugger variable.

value expression specifies the new value to assign to address expression.

Abbreviation:

\[\text{address expression} = \text{value expression}\]

Example:

The following command changes the value of the variable i to 10:

\[
\text{change i=10}
\]

The following command changes the tenth element of the floating point array fp-array to 3.0:

\[
\text{change fp_array(10) = 3.0}
\]

Notes:

Type coercion will be performed where necessary and possible. For example, it is permissible to assign an integer constant to a floating point variable. However, attempts to assign a floating point constant to a character variable, and vice-versa, will result in an error.

It is not possible to assign values to entire arrays, structures, or unions with the change command.

Related Commands:

\[
\text{scope}
\]
continue

Resuming program execution

Description:

The `continue` command resumes execution of the program being debugged. If desired, a temporary breakpoint may be specified. When specified, the location of the temporary breakpoint can be any valid address expression, a file relative source line, or the first executable source line of a procedure. Execution continues until the temporary breakpoint is encountered, a breakpoint is encountered, an error occurs, a monitor stops program execution, or the program runs to completion.

Usage:

```
continue {address expression}
continue {{"filename"} {:line number}}
continue {{procedure name} {:line number}}
```

where `address expression` evaluates to the address of an instruction in the program. Examples of useful address expressions include the name of an entry point, the name of an entry point plus an integer offset, or an absolute address specified as an integer constant.

`filename` is the name of a source file which is part of the program being debugged. If `filename` is not specified, the name of the file containing the current source line will be used.

`line number` is an integer constant. If `line number` is not specified, the first executable source line in the specified file will be used. An error will be reported if the specified `line number` does not correspond to an executable line.

`procedure name` is the name of a procedure in the program being debugged. If no `line number` is specified, execution will continue until the first instruction of the specified procedure. If the specified `line number` is one, execution will continue until the first executable source line of the specified procedure is encountered, otherwise the command is equivalent to "continue `filename :line number"", where `filename` is the name of the file which contains the specified procedure.

If no arguments are specified, execution will continue until a breakpoint is encountered, a monitor evaluates true, an error occurs, or the program runs to completion.

Abbreviation:

```
c
```
Example:

The following command resumes execution of the program being debugged until the location specified by the address expression subone+0x50 is encountered:

```
continue subone+0x50
```

The following command resumes execution of the program being debugged until the first executable source line of procedure suboneis encountered:

```
continue subone:1
```

Related Commands:

- go, Return
Delete  

Description:  

The `Delete` command is used to remove all breakpoints from a program. For the description of a command which deletes specific breakpoints, see the delete command.

Usage:  

```
Delete
```

Abbreviation:  

```
D
```

Related Commands  

`associate`, `break`, `delete`, `list breakpoints`
delete

Removing specific breakpoints

Description:

The delete command is used to remove one or more breakpoints from a program. For a description of a command that removes all breakpoints, see the Delete command.

Usage:

```
delete {breakpoint}
```

where breakpoint is any of the valid breakpoint specifications. See the break command for a description of valid breakpoint specifiers. If breakpoint is not specified, the debugger will display each active breakpoint and ask if it should be deleted. If a “Y” or a “y” is entered, the breakpoint will be deleted. If anything else is entered, the breakpoint will be left in place. Note that you do not have enter a carriage return after responding.

Abbreviation:

d

Example:

The following command deletes a breakpoint that was previously set on the first executable source line of the procedure subone:

```
delete subone:l
```

Related Commands:

associate, break, Delete, list breakpoints
dump

Displaying program memory

Description:

The **dump** command displays program memory starting at a specified address.

Usage:

```
dump address expression {#format {repeat count}}
```

where *address expression* evaluates to an integer address specifying a location in program memory.

*format* is any of the legal display formats. Display formats are described in the section on the **print** command. If *format* is not specified, the contents of the specified location will be displayed as a four-byte hexadecimal number.

*repeat count* is an integer constant specifying the number of values to display. When a *repeat count* greater than one is specified, the specified address will be incremented by the size of the specified format.

Abbreviation:

**du**

Example:

The following command displays the contents of the memory location **0x402790**:

```
dump 0x402790
```

The following command displays the contents of the memory location specified by the contents of the register **$r1**:

```
dump $r1
```

The following command displays the contents of four consecutive memory locations starting at the address of the variable **index1**:

```
dump index1# 4
```

Related Commands:

- **change**, **print**, **scope**, **section**, **watch**
external

Executing procedures out of sequence

Description:

The external command allows program procedures and functions to be executed out of sequence. Arguments may be passed and function results are displayed when a function returns.

Usage:

```
external procedure name {{arg1,...,argN}}
```

where `procedure name` is the name of a procedure or function in the program being debugged.

`arg1,...,argN` are the arguments to the specified procedure. Arguments can be the names of variables accessible from the current scope, constants, or value expressions. The debugger does not insure that the specified arguments agree in type or number with what the procedure expects. The debugger will match the calling conventions of the procedure's source language.

Abbreviation:

```
ex
```

Example:

The following command executes the function `sum3reals` with the arguments `varone,2.0,SIN(1.0)`:

```
external sum3reals(varone,2.0,SIN(1.0))
```

Assuming that the variable `varone` contains the value 1.0, the debugger will display the return value as follows:

```
External procedure sum3reals returns: 3.84147
```
Notes:

Use of the external command requires that the program being debugged has been linked with the library libg.a or that the operating system supports 88open OCS tdesc information.

If breakpoints have been installed in the procedure called with the `external` command, execution will stop as if the procedure had been entered through normal program execution. However, only one `external` command can be active at a given time. An attempt to execute a second external procedure before the first one returns will result in an error.

Passing character string constants and entire structures and unions by value is not supported by the `external` command.

Related Commands:

- `kill`
- `run`
filestatus

Displaying FORTRAN I/O unit information

Description:

The `filestatus` command is used to display information about all connected and preconnected FORTRAN units. For units explicitly connected with a FORTRAN OPEN statement, this command displays the unit number, file name, the state of the `ACCESS=`, `FORM=`, `ACTION=`, `STATUS=` I/O control specifiers used to connect the unit, and the current record number. For preconnected units, this command displays the unit number, and the state of the `ACCESS=`, `FORM=`, `ACTION=` I/O control specifiers.

Usage:

```fortran
filestatus
```

Abbreviation:

`f`

Notes:

This command will only work for programs which use the Absoft FORTRAN runtime library.
go

Resuming program execution

Description:

The `go` command resumes execution of the program being debugged. If desired, a temporary breakpoint may be specified. When specified, the location of the temporary breakpoint can be any valid address expression, a file relative source line, or the first executable source line of a procedure. All breakpoints are ignored during execution of this command. Execution continues until the temporary breakpoint is encountered, an error occurs, a monitor stops program execution, or the program runs to completion.

Usage:

```plaintext
go {address expression}
go {{"filename"} {::line number}}

where address expression evaluates to the address of an instruction in the program. Examples of useful address expressions for the `go` command include the name of an entry point, the name of an entry point plus an integer offset, or an absolute address specified as an integer constant.

filename is the name of a source file which is part of the program being debugged. If filename is not specified, the name of the file containing the current source line will be used.

line number is an integer constant. An error will be reported if the specified line number does not correspond to an executable source line. If not specified, the first executable line for the specified file will be used.

procedure name is the name of a procedure in the program being debugged. If no line number is specified, execution will continue until the first instruction of the specified procedure. If the specified line number is one, execution will resume until the first executable line for the specified procedure is encountered, otherwise the command is equivalent to “go filename line number”, where filename is the name of the file which contains the specified procedure.

If no arguments are specified, execution will continue until an error occurs, a monitor evaluates true, or the program runs to completion.

Abbreviation:

```
g
```
Example:

The following command resumes execution of the program being debugged until the location specified by the address expression `subone+0x50` is encountered:

```
go subone+0x50
```

The following command resumes execution of the program being debugged until the first executable source line of procedure `subone` is encountered:

```
go subone:1
```

Related Commands:

- continue, Return
help

Description:

The help command is used to access the on line help provided by Fx. When specified with no arguments, a brief introduction to the help system will be displayed including the specific commands and keywords for which help is available.

Usage:

help
help command
help keyword

Abbreviation:

h

Example:

The following command displays a list of available Fx commands:

help commands

The following command displays help for the break command:

help break
**Instruction**  

*Stepping over assembly language procedures*

**Description:**

The **Instruction** command executes one or more assembly language instructions, starting with the next instruction to be executed. If one of the instructions to be executed is a call to a procedure, execution of the program will continue until the instruction following the procedure call is encountered or until a breakpoint is encountered in the procedure that is being treated as a single instruction.

**Usage:**

```
Instruction {count}
```

where `count` is an integer expression which specifies the number of instructions to execute. If `count` is not specified, one instruction will be executed.

**Abbreviation:**

I

**Example:**

The following command executes the next five instructions of the current procedure, treating any procedure calls as single instructions:

```
Instruction 5
```

**Notes:**

The **Instruction** command only stops for breakpoints set in a procedure which is being treated as a single instruction. Breakpoints set in the current procedure will be ignored.

**Related Commands:**

- instruction, step, Step
**Instruction**

*Executing single instructions*

**Description:**

The **instruction** command executes one or more assembly language instructions, starting with the next instruction to be executed. If one of the instructions to be executed is a call to a procedure, the procedure will be entered.

**Usage:**

```
instruction {count}
```

where **count** is an integer expression which specifies the number of instructions to execute. If **count** is not specified, one instruction will be executed.

**Abbreviation:**

```
i
```

**Example:**

The following command executes the next five instructions of the current procedure:

```
instruction 5
```

**Notes:**

The **instruction** command ignores all breakpoints.

**Related Commands:**

- Instruction, step, Step
**kill**  

*Terminating the current program*

**Description:**

The *kill* command kills the current process being debugged without exiting the debugger.

**Usage:**

```fortran
kill
```

**Abbreviation:**

ki

**Notes:**

The *kill* command is provided so that a program may be recompiled using the ! command without leaving the debugger. It is not necessary to use the *kill* command before terminating a debugging session with the *quit* command or when restarting program execution with the *run* command as both of these commands implicitly kill the current process.

**Related Commands:**

!, reinit
list ascii

Description:

The `list ascii` command displays a table of printable ASCII characters and their hexadecimal numeric representations.

Usage:

```fortran
list ascii
```

Abbreviation:

`l a`
**list breakpoints**

*Displaying current breakpoints*

**Description:**

The **list breakpoints** command is used to display all currently active breakpoints, any commands associated with them, their skip counts, and the number of times they have been encountered without halting program execution.

**Usage:**

```plaintext
list breakpoints
```

**Abbreviation:**

```
l b
```

**Related Commands:**

- associate
- break
- delete
- Delete
list control

Description:

The list control command displays a list of all currently defined Fx control variables and their current values.

Usage:

list control

Abbreviation:

l c

Notes:

The contents of individual Fx control variables can also be displayed with the print command.

Related commands:

change, print
list entries

Displaying entry point information

Description:

The list entries command displays the names of selected program entry points, the type of result they return, and their source file and line number. If the source file and line number is not available, the address of the first instruction for an entry point will be displayed.

Usage:

list entries {"regular expression"}

where regular expression specifies a pattern to match against all the entry point names in the program being debugged. If not specified, all entry point names will be displayed. A complete discussion of regular expressions can be found in ed(1). Several useful regular expressions are demonstrated in the examples below.

Abbreviation:

l e

Example:

The following command will list all entry points with names containing the Characters fun:

list entries "fun"

The following command will list all entry points with names beginning with the characters fun:

list entries "^fun"

The following command will list all entry points with names ending with the characters fun:

list entries "fun$"
Notes:

The Fx control variable $elist determines whether the list entries command displays information about all program entry points or only the entry points with full symbol information. By default, this variable is set to zero causing the list entries command to skip any entry points without full symbol information. Setting $elist to a non-zero value with the change command will cause all program entry points to be listed.

The Fx control variable $lsort determines whether or not the list of entry points will be sorted. By default, this variable is set to one causing the list to be sorted. For large programs, sorting the list of entry points may take considerable time. Setting $lsort to zero with the change command will speed execution of this command.

Related Commands:

list globals, list locals, list statics
list file

Description:

The list file command allows any file to be examined without leaving the debugger.

Usage:

list file “filename”

where filename is the name of the file to display.

Abbreviation:

l f

Example:

The following command displays the contents of the include file source.h:

list file "source.h"

Notes:

The list file window will remain on the screen until the close command is used to remove it.

Related Commands:

close, next , previous, search, size, shift
list globals

Displaying global symbol information

Description:

The list globals command displays the names of selected global variables for the program being debugged along with their types and locations.

Usage:

list globals "{"regular expression"}"

where regular expression specifies a pattern to match against the global variable names in the program being debugged. If not specified, all global variables will be displayed. A complete discussion of regular expressions can be found in ed(l). Several useful regular expressions are demonstrated in the examples below.

Abbreviation:

lg

Example:

The following command will list all global variables with names containing the characters glob:

list globals "glob"

The following command will list all global variables with names beginning with the characters glob:

list globals "^glob"

The following command will list all global variables with names ending with the characters glob:

list globals "glob$"
Notes:

The Fx control variable $glist determines whether the list globals command displays information about all global variables or just the global variables whose data type is known. By default, this variable is set to zero and the list globals command will only display global variables with defined data type. To display all global variables, set the value of $glist to one with the change command.

The Fx control variable $lsort determines whether or not the list of global variables will be sorted before being displayed. By default, this variable is set to one causing the list to be sorted. For large programs, sorting the list may take a considerable amount of time. Setting $lsort to zero with the change command will speed execution of this command.

Related Commands:

list entries, list locals, list statics
list locals

Displaying local variable information

Description:

The list locals command displays the names of selected local variables in the current procedure along with their types and locations.

Usage:

list locals {"regular expression"}

where regular expression specifies a pattern to match against the local variable names for the current procedure. If not specified, all local variables will be displayed. A complete discussion of regular expressions can be found in ed(1). Several useful regular expressions are demonstrated in the examples below.

Abbreviation:

ll

Example:

The following command will list all local variables with names containing the characters loc:

list locals "loc"

The following command will list all local variables with names beginning with the characters loc:

list locals "^loc"

The following command will list all local variables with names ending with the characters loc:

list locals "loc$"

Notes:

The Fx control variable $lsort determines whether or not the list of local variables will be sorted before being displayed. By default, this variable is set to one causing the list to be sorted. For procedures declaring a large number of local variables, sorting the list may take a considerable amount of time. Setting $lsort to zero with the change command will speed execution of this command.

Related Commands:

list entries, list globals, list statics
list monitors

Description:

The list monitors command displays all currently active monitors with their id numbers and monitor expressions.

Usage:

list monitors

Abbreviation:

l m

Related Commands

monitor, unmonitor
list signal

Description:

The list signal command displays the action that will be taken when signals are presented to your program.

Usage:

list signal

Abbreviation:

l si

Related Commands

signal
**list statics**

*Displaying static symbol information*

**Description:**

The `list statics` command displays the names of selected static variables defined in the current source file.

**Usage:**

```
list statics {"regular expression"}
```

where `regular expression` specifies a pattern to match against the static variable names for the current file. If not specified, all static variables will be displayed. A complete discussion of regular expressions can be found in `ed(1)`. Several useful regular expressions are demonstrated in the examples below.

**Abbreviation:**

`ls`

**Example:**

The following command will list all static variables with names containing the characters `stat`:

```
list statics "stat"
```

The following command will list all static variables with names beginning with the characters `stat`:

```
list statics "^stat"
```

The following command will list all static variables with names ending with the characters `stat`:

```
list statics "stats$"
```
Notes:

The Fx control variable $slist determines whether the list statics command displays information about all static variables or just the static variables whose data type is known. By default, this variable is set to zero causing the list static command to display only the static variables with defined data types. Setting $slist to a non-zero value with the change command will cause all static variables to be listed.

The Fx control variable $lsort determines whether or not the list of static variables will be sorted before being displayed. By default, this variable is set to one causing the list to be sorted. For files which declare a large number of static variables, sorting the list may take a considerable amount of time. Setting $lsort to zero with the change command will speed execution of this command.

Related Commands:

list entries, list globals, list locals
log

Description:

The log command is used to control logging of debugger commands to a file. Files created by the log command can be played back by using the read command.

Usage:

log “filename”
log on
log off

where filename is the name of the file to store debugger commands in. This form of the log command implicitly turns on command logging.

on specifies that logging is to resume. If this command is issued without a preceding log “filename” command, commands will be logged to a file whose name will be constructed using the name of the executable program being debugged with the characters ".fx" appended.

off turns off command logging until the next log “filename” or log on command is issued.

Example:

The following command initiates logging of commands to a file named fxlog:

log "fxlog"

Assuming that command logging has not been previously enabled and that the name of the executable program is a.out, the following command initiates logging of commands to a file named a.out.fx:

log on

Notes:

The debugger will only log commands that execute successfully. Commands that contain spelling mistakes or reference symbols incorrectly will not be saved in the log file.

Related Commands:

read
**monitor**

*Stopping execution when a variable changes*

**Description:**

The **monitor** command is used to install a special form of breakpoint. Unlike traditional breakpoints, which halt execution of a program when a specific instruction is executed, monitors halt execution of a program when a relational expression evaluates true. Monitors can be used to stop execution of a program when a variable changes value, when it equals a specific value, when it is greater than a specific value, or when it is less than a specific value.

**Usage:**

```
monitor address expression {:operator value expression}
```

where address expression evaluates to the address of a location in the program being debugged or a register name. Examples of useful address expressions include the names of local and global variables and subscripted array elements.

operator is one of the supported source language operators for the current expression language.

value expression evaluates to a value to compare against the contents of address expression.

If value expression and operator are not specified, the current contents of address expression and the “not equal” operator for the current expression language will be used.

**Abbreviation:**

`m`

**Example:**

The following commands install a monitor on the variable `chk_flag` and resume execution. Since an operator and value expression are not specified, the monitor will stop execution of the program when the value of `chk_flag` changes:

```
monitor chk_flag
continue
```

As mentioned above, the preceding monitor command is equivalent to

```
monitor chk_flag : .NE. chk_flag
```

when FORTRAN is the current expression language, and

```
monitor chk_flag : != chk_flag
```

when C is the current expression language.

**Fortran User Guide**
The following command installs a monitor on the variable \texttt{index1}. However, an operator and value expression are specified, so this monitor will stop program execution when the value of \texttt{index1} is greater than 100:

\begin{verbatim}
monitor index1 : GT. 100
\end{verbatim}

Notes:

A monitor is automatically removed when it stops execution of the program.

Both \textit{address expression} and \textit{value expression} are evaluated when the monitor is installed. This means that a monitor command such as:

\begin{verbatim}
monitor array(i,j) : .NE. array(i+1,j+1)
\end{verbatim}

will calculate the address of element \texttt{array(i,j)} and retrieve the value of element \texttt{array(i+1,j+1)} when the monitor is installed. Subsequent changes to the variables \texttt{i,j} and \texttt{array(i+1,j+1)} will not affect the monitor.

The monitor command does not resume execution of the program. After monitors have been installed, program execution must be resumed with the \texttt{continue}, \texttt{go}, \texttt{step}, or \texttt{Step} commands. The \texttt{continue} and \texttt{go} commands behave differently when monitors are active. Instead of allowing program execution to resume, they repeatedly execute the command defined by the Fx control variable \texttt{$mstep$}. By default, this variable is set to \texttt{step}, causing monitors to be checked after each source statement. This variable can be changed to \texttt{Step}, \texttt{instruction}, or \texttt{Instruction}.

The Fx control variable \texttt{$mgrain$} determines how many times the command contained in \texttt{$mstep$} is executed before monitors are checked. The default value for this variable is 1, but it can be set to any positive integer.

The following commands illustrate the use of the Fx control variables \texttt{$mstep$} and \texttt{$mgrain$}. The result of these commands is that the value of \texttt{index1} will only be checked after every four source statements and any procedure calls will be treated as a single statements:

\begin{verbatim}
monitor index1
change $mstep = "Step"
change $mgrain = 4
continue
\end{verbatim}

Related Commands:

- \texttt{list monitor, unmonitor}
print

Displaying program variables

Description:

The print command displays the contents of program variables, registers, Fx control variables, and can also be used to evaluate expressions containing these items as well as constants, source language operators and FORTRAN intrinsic functions. Entire arrays, structures and unions can also be displayed. The section command is also available for displaying the contents of arrays.

Usage:

\{print\} value expression \{\#format\}

where value expression specifies the value to be printed. Useful value expressions include variable names, subscripted and unsubscripted array names, structure and union names, and references to structure and union members.

format is one of the valid format specifiers listed in the table below. If format is not specified, a format which is most appropriate for the type of value expression will be used.

Display Formats:

A single character indicating the desired format specifies the display format. Several of the display formats may be followed by a second character that indicates the number of bytes to display in the specified format. The characters used to represent sizes are: b (one byte), s (two bytes), and l (four bytes). The following table lists the display formats that may be specified.

<table>
<thead>
<tr>
<th>Character</th>
<th>Default Size</th>
<th>Other Sizes</th>
<th>Value displayed as</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4 bytes</td>
<td>ignored</td>
<td>address of value</td>
</tr>
<tr>
<td>b</td>
<td>4 bytes</td>
<td>b,s,l</td>
<td>binary integer</td>
</tr>
<tr>
<td>c</td>
<td>1 byte</td>
<td>ignored</td>
<td>ASCH character</td>
</tr>
<tr>
<td>d</td>
<td>4 bytes</td>
<td>b,s,l</td>
<td>decimal integer</td>
</tr>
<tr>
<td>e</td>
<td>8 bytes</td>
<td>ignored</td>
<td>double precision floating point</td>
</tr>
<tr>
<td>f</td>
<td>4 bytes</td>
<td>ignored</td>
<td>single precision floating point</td>
</tr>
<tr>
<td>o</td>
<td>4 bytes</td>
<td>b,s,l</td>
<td>octal integer</td>
</tr>
<tr>
<td>s</td>
<td>$s$len</td>
<td>ignored</td>
<td>null terminated string</td>
</tr>
<tr>
<td>u</td>
<td>4 bytes</td>
<td>b,s,l</td>
<td>unsigned decimal integer</td>
</tr>
<tr>
<td>x</td>
<td>4 bytes</td>
<td>b,s,l</td>
<td>hexadecimal integer</td>
</tr>
</tbody>
</table>

Abbreviation:

value expression
Example:

The following command prints the contents of the variable `varone` in the format most appropriate for its type:

```plaintext
print varone
```

The following command prints a single precision representation of PI:

```plaintext
print 4.0*ATAN(1.0)
```

The following command prints the variable `intvar` as a hexadecimal integer:

```plaintext
print intvar#x
```

Notes:

The only time the `print` command must be explicitly specified is when printing the contents of a variable whose name matches one of the debugger commands. For example, a variable named `continue`.

The Fx control variable `$acount` controls the maximum number of array elements to display when an unsubscripted array name is specified. The default value for this variable is 100. However, it can be changed to any positive integer value using the `change` command.

The Fx control variable `$union` determines whether or not all members of a union are displayed. By default, this variable is set to one so all members are displayed. Setting `$union` to zero with the `change` command will cause only the first member of a union to be displayed.

The Fx control variable `$slen` controls the maximum number of characters printed when using the `s` format. The default value for this variable is 80 but it can be changed to any positive integer value using the `change` command.

The Fx control variable `$ffmt` controls the display format for single precision values. The default value for this variable is “(1PG15.6E2)”. However, it can be changed to any legal FORTRAN format string suitable for writing a single precision value with the `change` command.

The Fx control variable `$efmt` controls the display format for double precision values. The default value for this variable is “(1PG24.15E3)”. However, it can be changed to any legal FORTRAN format string suitable for writing a double precision value with the `change` command.

The Fx control variable `$cmpfmt` controls the display format for FORTRAN `COMPLEX` values. The default for this variable is “('(',1PG15.E2,',',1PG15.6E2,')')”. However, it can be changed to any legal FORTRAN format string suitable for writing a `COMPLEX` value with the `change` command.
The Fx control variable $dcmpfmt controls the display format for FORTRAN DOUBLE PRECISION COMPLEX values. The default value for this variable is “("(('1PG24.E3,'','1PG24.6E3,''))”). However, it can be changed to any legal FORTRAN format string suitable for writing a DOUBLE PRECISION COMPLEX value with the change command.

Related Commands:

dump, scope, section, watch
quit  

Description:

The quit command terminates the current debugging session.

Usage:

quit

Abbreviation:

q

Related Commands:

!, kill, sleep
read  

Description:

The `read` command allows debugger commands to be read from a file. The file may have been created as the result of using the log commands, or can be created by hand using a text editor.

Usage:

```
read {"filename"}
```

where `filename` is the name of a file which contains valid Fx commands. If `filename` is not specified, a file name consisting of the executable object file with the characters “.fx” appended will be used.

If an error occurs while commands are being read from the specified file, the remaining commands in the file will be ignored.

Abbreviation:

```
rea
```

Example:

The following command will cause the debugger to execute the commands in the file `fxlog`:

```
read "fxlog"
```

Notes:

It is permissible to nest `read` commands. That is, a file specified with the `read` command may contain other `read` commands. It should be noted that use of this feature might lead to infinite execution if the file used in a nested `read` command contains a `read` command specifying the original file.

The Fx control variable `$read` can be used to suppress execution of display commands, such as `print` and `registers`, which do not affect the state of the program being debugged. By default, the value of `$read` is set to a non-zero value causing all commands read from a file to be executed. To suppress execution of display commands, change the value of `$read` to zero.

Related Commands:

```
log
```
registers

Description:
The registers command is used to display the current contents of all machine registers.

Usage:

    registers

Abbreviation:

    r

Notes:
The contents of individual registers can be displayed in a variety of formats using the print command.
reinit

Switching programs to debug

Description:

The reinit command is used to begin debugging a new program without terminating and restarting Fx.

Usage:

```
reinit {executable file} {-c corefile} {-P pathlist} {-p pathlist}
```

where `executable file` is the name of a program to begin debugging. If not specified, the name of the last debugged program will be used.

`-c corefile` specifies the name of a core file to use. If not specified and a file named “core” exists in the current directory, this file will be used. To suppress the automatic use of a file name “core”, specify `-c none`. To use a core file named “none”, include a path specification such as `-c ./none`.

`-P pathlist` and `-p pathlist` specify the directories containing the source files for the new program. The form `-P pathlist` replaces the current set of source paths with the specified `pathlist`. The form `-p pathlist` adds the specified path list to the current set of source paths.

Example:

The following command will begin a new debugging session on the program `a.out`, suppressing any core file, and using the source directories `source1` and `source1`:

```
reinit a.out -c none -P source1:source2
```

Abbreviation:

`rei`

Related Commands:

`, kill`
Return

Returning from the current subroutine

Description:

The Return command resumes execution of the program being debugged until the current procedure returns to its calling procedure or a breakpoint is encountered. If the current procedure never returns to its calling procedure, execution will continue until a breakpoint is encountered, an error occurs, or the program runs to completion.

Usage:

Return

Abbreviation:

R

Notes:

If monitors are set during execution of the Return command, they will not be checked until the current procedure returns to its calling procedure or a breakpoint is encountered.

Related Commands:

continue, go
**.run**  

*Starting and restarting program execution*

**Description:**

The **run** command is used to start or restart execution of a program. It can be used to pass arguments to the program and to cause redirection of the program's standard input and output. After the run command is issued, the program will execute until a breakpoint is encountered, an error occurs, or the program runs to completion.

**Usage:**

```
run {arg1 arg2 ... argN}
```

where `arg1 arg2 ... argN` are the arguments to pass to the program. If an argument begins with the character “<” the remaining characters of that argument, or the following argument if no characters follow the “<”, will be used as file name for redirection of standard input. Likewise, if an argument begins with the character “>” the remaining characters, or the following argument if no characters follow the “>”, will be used as a file name for redirection of standard output. Redirection of other file descriptors is not supported.

If arguments are not specified, the arguments from the last execution of this command, if any, will be used.

**Abbreviation:**

`ru`

**Example:**

The following command restarts the program being debugged and passes it the character strings `one`, `two`, and `three` as arguments:

```
ru one two three
```

The following command restarts the program being debugged, redirecting standard output to the file `outfile`:

```
ru >outfile
```

**Notes:**

Each time program arguments are specified with the **run** command, the debugger saves a copy of them in the Fx control variable `Sargs`. If it is desirable to run the program without any arguments, the value of `Sargs` can be set to a string containing only a single space before issuing the **run** command. The following command will set `Sargs` to the appropriate value:

```
change $args=" "
```
Related Commands:

   external, kill
scope

**Description:**

The `scope` command changes the procedure that defines the current scope. Normally, the current procedure is the procedure whose source or assembly language code appears in the standard source window. However, there may be multiple occurrences of the same procedure in the current chain of execution. The scope command provides a method for specifying a particular instance of a procedure.

**Usage:**

```
scope procedure name \{level\}
```

where `procedure name` is the name of a procedure present in the current chain of execution. The `trace` command can be used to display a list of all procedure in the current chain of execution.

`level` is an integer constant and specifies a particular instance of the specified procedure in the current chain of program execution. Level zero refers to the first instance of the procedure, level one refers to the second, and so on. If not specified, the current scope will be changed to the first occurrence of the specified procedure.

**Abbreviation:**

`sc`

**Example:**

The following command changes the current scope to the procedure `main`:

```
scope main
```

The following command changes the current scope to third the occurrence of the procedure `subone`:

```
scope subone(2)
```

**Notes:**

The scope command has no effect when registers are displayed.

When debugging C programs, the debugger may not be able to accurately recreate the values of register variables and parameters that passed in registers from different scopes. The Absoft FORTRAN compiler will automatically store parameters passed in registers in the stack frame when compiling programs for debugging, so the scope command will function correctly.
Related Commands:

change, list locals, print, section, watch
search  

Finding strings in source windows

Description:

The search command searches forward from the current line in the file displayed in the standard source window, or the list file window if it is visible, for a line containing a string matching the specified regular expression. A complete description of regular expressions can be found in ed(l). If the end of the file is reached without finding a match, the search will resume at the first line of the file and continue until a match is found or the current line is encountered.

Usage:

search {"regular expression"}

where regular expression specifies the string to search for. If not specified, the last regular expression will be used.

Abbreviation:

se

Example:

The following command searches the current source file for a line containing the suing subroutine:

search "subroutine"

The following command searches the current source file for a line beginning with the string 1000:

search "^1000"

Notes:

The search command cannot be used to search disassembled code.

When using the Fx character interface with the list file window open, the search will be performed on the file being display in it. When using the Fx graphical interface, the search is always performed on the standard source window.

Related Commands:

list file, view
section

Displaying arrays

Description:

The section command is used to display specific elements of arrays.

Usage:

section (...(array name (index1,...indexN),index1=
    lb1,ub1{,incr1})...,indexN=lbN,ubN{,incrN}) {#format}

section array name [lb1;ub1{;incr1}]...[lbN;ubN{;incrN}] {#format}

where array name is the name of an array accessible within the current scope.

index1,...,indexN are either symbol names or integer constants used to specify subscript(s) for examining the array. If a given subscript is a symbol name it must have a matching range specification of the form indexN=lbN,ubN{,incrN}. If a given subscript is a constant, it must not have a matching range specification.

lbN is an integer expression specifying the starting value for indexN.

ubN is an integer expression specifying the ending value for indexN.

incrN is an integer expression specifying the increment value for a given index. If not specified, a value of one will be used.

format is any of the legal display formats. Display formats are described in the section on the print command. If format is not specified, a format which is most appropriate for the type of the array will be used.

Abbreviation:

Se

Example:

The following commands display every other element of a single dimension FORTRAN array with 20 elements:

section (a(i),i=1,20,2)
section a[1;20;2]

The following commands display every other element of a single dimension C array with 20 elements:

section (a(i),i=0,19,2)
section a[0;19;2]

The following commands display elements 1,1 though 20,1 of a two dimensional FORTRAN array:
section (a(i,l),i=1,20)
section a[1;20][1;1]

The following commands display elements 0,1 though 19,1 of a two dimensional C array:

section (a(i,l),i=0,19)
section a[0;19][1;1]

Notes:

Symbol names used to define indexes are unique to a section command and do not correspond to symbols in the program being debugged. Also, a symbol name used to define an index cannot be used in the integer expressions for lbN, ubN, or incrN.

The section command cannot be used to display arrays which are members of C structures and unions.

Related Commands:

dump, print, scope, watch
signal

Controlling signal actions

Description:

The signal command allows you to control the actions taken when a signal is presented to your program during a debugging session. You can also use this command to send a signal to your program at anytime.

Usage:

```
signal signal number
signal signal number PASS | NOPASS
signal signal number STOP | NOSTOP
```

where signal number is the positive integer which represents the signal you wish to control. Specifying only a signal number will cause that signal to be presented to your program the next time execution is resumed. You can remove any pending signal by specifying zero instead of a signal number.

PASS | NOPASS indicates whether or not your program should be allowed to see a particular signal. Specify PASS if you want the signal to be passed on to your program or NOPASS if you wish Fx to prevent your program from receiving the signal.

STOP | NOSTOP indicates whether or not a signal should stop execution of your program. Specify STOP if you want Fx to stop execution of your program when the signal occurs and NOSTOP if your program should be allowed to continue executing.

Abbreviation:

sig

Examples:

The following command will cause your program to receive a floating point exception signal the next time execution is resumed:

```
.signal 8
```

The following command will prevent your program from seeing future occurrences of the floating point exception signal:

```
signal 8 NOPASS
```

The following command will prevent Fx from stopping execution of your program when a floating point exception signal occurs:

```
signal 8 NOSTOP
```
Notes:

The list signal command displays the current signal settings.

Related Commands:

list signal
sleep

Description:

The `sleep` command suspends execution of the debugger for a specified number of seconds.

Usage:

```
sleep {number of seconds}
```

where `number of seconds` is a positive integer constant. If not specified or zero is specified, a value of one will be used.

Abbreviation:

```
sl
```

Notes:

The `sleep` command exists primarily for creating self-running demonstrations of the debugger. However, there may be occasions to use it for other purposes when using the Fx character interface. Consider the following `associate` command:

```
associate main:1 "print i;print j;continue"
```

Since there are two print commands, the value of `i` may be replaced by the value of `j` before there is time to examine it. The following `associate` command will insert a delay between the two print commands so that both values may be examined:

```
associate main:1 "print i;sleep 5;print j;continue"
```
Step \hspace{1cm} \textit{Stepping over procedure calls}

Description:

The \textbf{Step} command executes one or more source statements, starting with the next statement to be executed. If one of the statements to be executed is a call to a procedure, execution of the program will continue until the statement following the procedure call is encountered or a breakpoint is encountered.

Usage:

\texttt{Step \{count\}}

where \textit{count} is an integer expression which specifies the number of statements to execute. If count is not specified, one statement will be executed.

Abbreviation:

\texttt{S}

Example:

The following command executes the next five statements of the current procedure, treating any procedure calls as single statements:

\texttt{Step 5}

Notes:

The \textbf{Step} command will only stop for breakpoints set in a procedure which is being treated as a single statement. It will ignore any breakpoints set on the instructions which are part of the source statement being executed.

Related Commands:

\texttt{instruction, Instruction, step}
**step**  

*Executing single source statements*

**Description:**

The step command executes one or more source statements, starting with the next statement to be executed. If one of the statements to be executed is a call to a procedure, the procedure will be entered if complete symbol information is available for it. If complete symbol information is not available, execution of the program will continue until the statement following the procedure call is encountered.

**Usage:**

```plaintext
step {count}
```

where `count` is an integer expression which specifies the number of statements to execute. If count is not specified, one statement will be executed.

**Abbreviation:**

`s`

**Example:**

The following command executes the next five statements of the current procedure:

```plaintext
step 5
```

**Notes:**

The **step** command will ignore any breakpoints set on instructions that are part of the source statement being executed.

**Related Commands:**

`instruction`, `Instruction`, `Step`
trace            Displaying a stack trace

Description:

The `trace` command is used to display the chain of procedure calls that produced the current program state. Each procedure in the current chain is listed along with the procedure that called it and the calling procedure's file and line number if available.

Usage:

```
trace
```

Abbreviation:

`t`

Notes:

This command may not function correctly if the program being debugged is stopped during execution of the entry code for a procedure. When the complete entry code of a the procedure has not yet been executed, the words “At Entry” will appear in the status display.
unmonitor

Removing monitors

Description:

The unmonitor command removes a monitor previously installed with the monitor command.

Usage:

unmonitor {monitor id}

where monitor id is the positive integer value associated with the monitor when it was installed by the monitor command. The list monitors command can be used to determine what id is associated with a particular monitor.

If monitor id is not specified, the debugger will display each installed monitor and ask if it should be removed. If a “Y” or a “y” is entered, the monitor will be removed. If anything else is entered, the monitor will remain active. Note that you do not have enter a carriage return after responding.

Abbreviation:

un

Example:

The following command removes the monitor that was assigned id 3 by the monitor command:

unmonitor 3

Related Commands:

list monitors, monitor
unwatch

Removing watch variables

Description:

The **unwatch** command allows you to remove individual watch variables.

Usage:

```
unwatch {watch id}
```

where *watch id* is the positive integer value associated with the watch variable when it was created with the **watch** command. Watch ids are displayed in parentheses before each watch variable.

Abbreviation:

```
unw
```

Example:

The following command removes the watch variable with id 1:

```
unwatch 1
```

Notes:

Removing the last watch variable with the **unwatch** command automatically closes the watch variable window.

Related Commands:

```
watch
```
View

Displaying assembly language code

Description:

The View command is used to examine the assembly language code for the program being debugged.

Usage:

View \{address expression\}
View \{\"filename\"\} \{:line number\}
View \{\{procedure name\}\} \{:line number\}

where address expression evaluates to the address of an instruction in the program. Examples of useful address expressions for the View command include the name of an entry point, the name of an entry point plus an integer offset, or the contents of a pointer to a function.

filename is the name of a source file which is part of the program being debugged. If filename is not specified, the name of the current source file will be used.

line number is an integer constant. If the specified line number exceeds the number of actual lines in the source file, assembly language code for the last executable line of the source file will be displayed.

procedure name is the name of a procedure in the program being debugged. If the specified line number is one, the assembly language code for the first executable source line of the specified procedure will be displayed. If a line number other than one is specified, the command is equivalent to “View filename :line number”, where filename is the name of the file which contains the specified procedure.

If no arguments are specified, the address of the next instruction to be executed will be used.

Abbreviation:

V

Example:

The following command displays assembly language code starting at the address main+0x50:

View main+0x50

The following command displays assembly language code for the procedure subone:

View subone
Notes:

The **View** command implicitly changes the current scope to the file and procedure being displayed. If the procedure is not currently active, the words “Not Active” will appear in the status display.

**Related Commands:**

- list rile, next, previous, view
view

Description:

The view command is used to examine the source code for the program being debugged. For details on a command that allows any file to be examined see the list file command.

Usage:

view \{address expression\}
view \{\{“filename”\} {:line number}\}
view \{\{procedure name\} {:line number}\}

where address expression evaluates to the address of an instruction in the program. Examples of useful address expressions for the view command include the name of an entry point, the name of an entry point plus an integer offset, or the contents of a pointer to a function.

filename is the name of a source file which is part of the program being debugged. If filename is not specified, the name of the file containing the current source line will be used.

line number is an integer constant. If the specified line number is greater than the number of actual lines in the source file, the last line of the source file will be displayed.

procedure name is the name of a procedure in the program being debugged. If the specified line number is one, the first executable source line of the specified procedure will be displayed. If a line number other than one is specified, the command is equivalent to “view filename :line number”, where filename is the name of the file which contains the specified procedure.

If no arguments are specified, the line number of the next statement to be executed and the source file that contains it will be used.

If source line information is not available for a specified procedure or if the specified address expression does not identify a line of source code, assembly language will be displayed instead.

Abbreviation:

\v
Example:

The following command displays the source code for the procedure `subone`:

```
view subone
```

The following command displays the source code for the file `source.c`, starting with line 100:

```
view "source.c":100
```

Notes:

The `view` command implicitly changes the current scope to the file and procedure being displayed. If the procedure is not currently active, the words “Not Active” will appear in the status display.

Related Commands:

`list file, next, previous, search, View`
**walk**  
*Repeatedly executing Fx commands*

**Description:**

The `walk` command allows other debugger commands to be repeatedly issued by the debugger itself. After the `walk` command has been issued, the debugger will continue to execute the commands contained in the debugger variable `$walkcmds` until explicitly stopped, an error occurs, a breakpoint is encountered, a monitor stops execution, or the program runs to completion.

**Usage:**

```plaintext
walk \{speed\}
```

where `speed` is an integer constant between one and ten which specifies the delay in half-seconds between issuing commands. If `speed` is not specified no delay will occur between commands.

**Abbreviation:**

`w`

**Notes:**

By default, `$walkcmds` contains the character string “step”. This can be changed to any character string using the `change` command. Setting this variable to a nonsense string will result in error and terminate execution of a `walk` command immediately.
**watch**

*Observing program variables*

**Description:**

The **watch** command is used to automatically display the value of an expression after each subsequent debugger command.

**Usage:**

\[
\text{watch } \text{value expression} \begin{cases} \#\text{format} \end{cases}
\]

where *value expression* specifies the value to be printed. Particularly useful value expressions include variable names, subscripted arrays, and references to structure and union members.

*format* is any of the legal display formats. Display formats are described in the section on the **print** command. If *format* is not specified, a format that is most appropriate for the type of value expression will be used.

**Abbreviation:**

**wat**

**Example:**

The following command will cause the contents of the variable *index1* to be displayed after each subsequent command:

```
watch index1
```

**Notes:**

Value expressions involving local variables will only be displayed while the procedure in which they are declared is active. When a procedure that declares a variable that is part of watched expression is inactive, the message “(OUT OF SCOPE)” will be displayed instead of the expression's value.

**Related Commands:**

* dump, print, scope, section
## Appendix A

### Absoft Compiler Option Guide

This appendix summarizes general options for Absoft Pro Fortran compilers and specific options for the Absoft Fortran 90/95 and FORTRAN 77 compilers. Refer to the chapter, *Using the Compilers* for detailed descriptions of the options.

#### ABSOFT PRO FORTRAN COMPILER OPTIONS

<table>
<thead>
<tr>
<th>Option</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-c</code></td>
<td>suppresses creation of an executable file — leaves compiled files in object code format.</td>
</tr>
<tr>
<td><code>-g</code></td>
<td>generates symbol information for Fx™.</td>
</tr>
<tr>
<td><code>-Lpath</code></td>
<td>library file search path specification.</td>
</tr>
<tr>
<td><code>-l name</code></td>
<td>library file specification.</td>
</tr>
<tr>
<td><code>-O</code></td>
<td>enables a group of basic optimizations which will cause most code to run faster without the expense of application size or memory usage.</td>
</tr>
<tr>
<td><code>-o name</code></td>
<td>directs the compiler to produce an executable file called <code>name</code> where <code>name</code> is a Macintosh OS X file name.</td>
</tr>
<tr>
<td><code>-P</code></td>
<td>instrument executable for profiling.</td>
</tr>
<tr>
<td><code>-S</code></td>
<td>generates an assembly language output file.</td>
</tr>
<tr>
<td><code>-s</code></td>
<td>allocate local variables statically.</td>
</tr>
<tr>
<td><code>-u</code></td>
<td>undefine a symbol to the linker.</td>
</tr>
<tr>
<td><code>-v</code></td>
<td>directs the compiler to print status information as the compilation process proceeds.</td>
</tr>
<tr>
<td><code>-w</code></td>
<td>suppresses listing of all compile-time warning messages.</td>
</tr>
<tr>
<td><code>-X option</code></td>
<td>linker option.</td>
</tr>
</tbody>
</table>
FPU CONTROL OPTIONS
- **round=mode**  set the FPU rounding method.
- **trap=exception**  enable FPU exceptions.
- **Q51**  don’t generate FMA instructions.

POWERPC PROCESSOR SPECIFIC OPTIONS
- **N11**  generates long branches.

FORTRAN 90/95 CONTROL OPTIONS
- **B19**  assume pointer aliases exist.
- **B80**  causes the compiler to generate code to write the name of the currently executing procedure to standard out.
- **YEXT_SFX**  append trailing characters to procedure names.
- **B112**  disable stack alignment.
- **dq**  Allow more than 100 error diagnostics.
- **ea**  Causes the f95 compiler to abort the compilation process on the first error that it encounters.
- **en**  Causes the compiler to issue a warning whenever the source code contains an extension to the Fortran 90/95 standard.
- **eR**  Default recursion
- **g**  Generates symbol information for Fx™.
- **Mnn**  Suppresses messages by message number.
- **mnn**  Suppresses messages by message level.
- **P**  Instrument executable for profiling.
- **V**  Causes the f95 compiler to display its version number.
-v  Directs the compiler to print status information as the compilation process proceeds

-w  Suppresses listing of all compile-time warning messages.

FORTRAN 90/95 OPTIMIZATION OPTIONS
-O  enables a group of basic optimizations which will cause most code to run faster without the expense of application size or memory usage.

FORTRAN 90/95 SOURCE FORMAT OPTIONS

-fform  sets the form of the source file to free, fixed, or alt_fixed.

-Wn  sets the line length of source statements accepted by the compiler in Fixed-Form source format.

FORTRAN 90/95 COMPATIBILITY OPTIONS

-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.

-in  set default integer size to n (4 or 8) bytes.

-N113  set default real size to 8 bytes (KIND=8).

-p path  specify module search path

-s  allocate local variables statically

-Rb  generate code to check array boundaries.

-Rc  generate code to validate substring indexes.

-Rp  generate code to check for null pointers.
-Rs  generate code check array conformance.

-tn  this option increases the default temporary string size to $1024 \times 10^n$ bytes.

-xdirective  disable compiler directive in the source file.

-YCFRL  forces the compiler to pass $g77/f2c$ compatible CHARACTER arguments.

-YCOM_NAMES  specify COMMON block names externally in upper or lower case.

-YCOM_PFX  specify COMMON block external name prefix.

-YCOM_SFX  specify COMMON block external name suffix.

-YCSLASH  directs the compiler to transform certain escape sequences marked with a ‘\’ embedded in character constants.

-YEXT_NAMES  Specify procedure names externally in upper, lower, or mixed case.

-YEXT_PFX  Specify procedure external name prefix.

-YEXT_SFX  Specify procedure external name suffix.

-YMS7D  Recognize Microsoft style compiler directives beginning with a ‘$’ in column 1.

-YNDFP  disallow the use of a ‘.’ as a structure field separator.

-YPEI  pointers are Equivalent to Integers allows a Cray-style pointer to be manipulated as an integer.

FORTRAN 77 CONTROL OPTIONS

-A  suppress alignment warnings.

-B19  used when more than one symbolic name is used to reference a variable’s memory location. This can occur when pointers are used, when variables in COMMON are passed as arguments, or when two dummy arguments are the same actual argument.

-B80  causes the compiler to generate code to write the name of the currently executing procedure to standard out.
-N15  append trailing underscores to procedure names.

-B112  disable stack alignment.

-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

-D  used to define conditional compilation variables from the command line (\texttt{-D name[=value])} — if \texttt{value} is not present, the variable is assigned the value of 1

-g  generates symbol information for Fx™.

-I\texttt{path}  specify path to search for INCLUDE files.

-N32  directs the compiler to issue a warning whenever the source code contains an extension to the ANSI FORTRAN 77 standard

-N52  check syntax only.

-N90  forces the compiler to pass \texttt{g77/f2c} compatible CHARACTER arguments.

-N6  directs the compiler to place information in the debugger symbol tables for all structures whether or not they have associated storage

-N116  BLOCK DATA code section.

-P  instrument executable for profiling.

-q  suppress non-diagnostic output.

-Tnn  used to change the number of handles used internally by the compiler.

-tnn  modifies the default temporary string size to \texttt{nn} bytes from the default of 1024 bytes

-v  directs the compiler to print status information as the compilation process proceeds

-w  suppresses listing of all compile-time warning messages

-x  replaces any occurrence of \texttt{X} or \texttt{D} in column one with a blank character: allows a restricted form of conditional compilation
FORTRAN 77 OPTIMIZATION OPTIONS

-\texttt{Hnn} set loop unrolling limit.

-\texttt{hnn} set loop unrolling factor.

-\texttt{N5} treat DATA as constants.

-\texttt{N18} inline function decomposition.

-\texttt{N41} evaluate constant intrinsic functions.

-\texttt{N86} enable address expression optimization.

-\texttt{O} enables a group of basic optimizations which will cause most code to run faster without the expense of application size or memory usage.

FORTRAN 77 SOURCE FORMAT OPTIONS

-\texttt{8} directs the compiler to accept source code written in Fortran 90/95 Free Source Form

-\texttt{N112} directs the compiler to accept source code written in IBM VS Free Form

-\texttt{V} directs the compiler to accept VAX Tab-Format source code

-\texttt{W} directs the compiler to accept statements which extend beyond column 72 up to column 132

FORTRAN 77 COMPATIBILITY OPTIONS

-\texttt{N15} causes the compiler to define \texttt{SUBROUTINE} and \texttt{FUNCTION} names with a trailing underscore

-\texttt{d} causes all \texttt{DO} loops to be executed at least once, regardless of the initial value of the iteration count (FORTRAN 66 convention)

-\texttt{f} folds all symbolic names to lower case

-\texttt{in} changes the default storage length of \texttt{INTEGER} from 4 bytes to \texttt{n} (2 or 8).
-K directs the compiler to transform certain escape sequences marked with a `\` embedded in character constants

-N1 causes all variables initialized with DATA statements to be stored as static variables.

-N2 uses only double precision or double complex transcendental intrinsics

-N3 includes record length information for sequential unformatted files

-N4 suppresses any run-time CASE_DEFAULT error messages

-N7 extends the sign of a value returned from BYTE, and WORD intrinsic functions

-N16 [FMT=] format specifier may be omitted in an I/O statement when [UNIT=] unit specifier is present

-N20 directs the compiler to always evaluate operators of equal precedence from left to right (except for exponential operators)

-N22 don’t mangle COMMON block names with leading “_c”

-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

-N33 causes structure fields to be “packed” — allocated with no space between them

-N34 automatically align COMMON block variables

-N51 if a file is opened as DIRECT access UNFORMATTED, causes the value set with RECL to be interpreted as the number of 32 bit words in a record instead of the number of bytes

-N102 directs the compiler to zero extend INTEGER*1 variables to unsigned entities with a range of 0-255 when loaded from memory

-N109 folds all symbolic names to UPPER CASE

-N113 changes REAL and COMPLEX data types without explicit length declaration to DOUBLE PRECISION and DOUBLE COMPLEX
-N114 issue a warning diagnostic, rather than an error, for undeclared variables in the presence of an IMPLICIT NONE declaration

-N115 Pad source lines to column 72 (or 132 with -W option)

-s forces all program storage to be treated as static: see -N1 also
Appendix B

Terminal Programming

All of the Absoft tools are also designed to be used in a command line environment. This appendix outlines the use of the Macintosh Terminal application which provides a command line interface to the OS X operating system.

Open a terminal through the Finder by navigating to the /Applications/Utilities folder and double clicking on the Terminal application. A window will open and a command prompt will appear. The prompt will look similar to:

[computer_name:~] username%

The ~ symbol is an abbreviation for your home directory (/Users/username/). If you enter a change directory command such as “cd fortran” (assuming you have a folder named fortran in your home directory) the prompt will change to:

[computer_name:~/fortran] username%

and /Users/username/fortran will be the current working directory. To return to your home directory you can enter “cd ..”. This will take you up one directory. You can also enter “cd ~” from any directory to return to your home directory.

The environment variables must be modified so that the system can find the Absoft compilers. Change the current working directory to your home directory (cd ~) and enter the following command to edit a file with a simple text editor:

pico .tcshrc

This launches the text editor pico and specifies that we want to edit the file .tcshrc. The file .tcshrc is a special file which is read every time you open a terminal. Enter the following text on the first line in the editor:

setenv ABSOFT /Applications/Absoft

This creates an environment variable named ABSOFT and sets its value to /Applications/Absoft (the location of the Absoft compilers). The existing environment variable path must be modified to allow the system to find the Absoft compilers. Enter the following text on the second line in the editor:

set path = ( $ABSOFT/bin $path )

A $ before an environment variable indicates that we are referencing the value of that variable. This line modifies the path environment variable, instructing the system to look in the /Applications/Absoft/bin directory for executable programs.
To exit `pico`, press `Ctrl-X` and answer “y” to the “Save modified buffer” question. Because this file is only read when a terminal session is started, it is necessary to close this terminal and then open a new one. The compilers can now be used from the terminal.

Change the current working directory (with the `cd` command) to one containing a Fortran source file. Use the `ls` command to list the files in the current directory.

Compile and link the Fortran file with the command:

```
f77 source.f -o myprogram
```

or

```
f95 source.f95 -o myprogram
```

These commands will automatically compile and link your source file to produce an executable file named `myprogram`. Enter “./myprogram” to run your program. The “./” indicates to look in the current working directory for the file. (Note: omitting “-o name” produces an executable named `a.out`.)

If you just wish to compile your source file into an object file to be linked later, add the `-c` option:

```
f77 source.f -c
```

or

```
f95 source.f95 -c
```

These commands both produce `source.o`. You can link object files together to create executables with the `f77` and `f95` commands as well:

```
f77 source1.o source2.o source3.o -o myprogram
```

or

```
f95 source1.o source2.o source3.o -o myprogram
```

Refer to the chapter **Using the Compilers** for a description of compiler options.
Fx defines a number of control variables, internal variables that modify the operation of commands. The table below lists the names of the Fx control variables, their purpose, and default values.

### Fx Control Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Controls</th>
<th>Legal Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>$acount</td>
<td>maximum number of array elements displayed when unsubscripted arrays displayed with the <code>print</code> command</td>
<td>any positive integer, default is 100</td>
</tr>
<tr>
<td>$args</td>
<td>arguments passed to a program by the <code>run</code> command no arguments are specified</td>
<td>any character string, default is the last arguments specified with a <code>run</code> command</td>
</tr>
<tr>
<td>$case</td>
<td>case folding for symbol table searches</td>
<td>“lower”, “upper”, or &quot;both&quot;, default is “both”</td>
</tr>
<tr>
<td>$cmpfmt</td>
<td>display format for <code>COMPLEX</code> values</td>
<td>any FORTRAN format suitable for displaying <code>COMPLEX</code> values, default is &quot;(('(',1PG15.6E2,',',1PG15.6E2,')')&quot;</td>
</tr>
<tr>
<td>$cwd</td>
<td>controls program's current working directory</td>
<td>a character string which specifies the directory</td>
</tr>
<tr>
<td>$dcmpfmt</td>
<td>display format for <code>DOUBLE COMPLEX</code> values</td>
<td>any FORTRAN format suitable for displaying <code>COMPLEX</code> values, default is &quot;(('(',1PG24.6E3,',',1PG24.6E3,')')&quot;</td>
</tr>
<tr>
<td>$deflang</td>
<td>default expression language</td>
<td>“C” or “FORTRAN”, default is “C”</td>
</tr>
<tr>
<td>$efint</td>
<td>display format for double precision values</td>
<td>any FORTRAN format suitable for displaying double precision values, default is &quot;((1PG24.15E3)&quot;</td>
</tr>
</tbody>
</table>
## Fx Control Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Controls</th>
<th>Legal Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>$elist</td>
<td>whether <strong>list entries</strong> displays all entry points or only those with full symbol information, when non-zero all entry points are displayed</td>
<td>any integer, default is 0</td>
</tr>
<tr>
<td>$explang</td>
<td>current expression language, when set to &quot;automatic&quot;, the expression language is determined by the current source file name</td>
<td>&quot;automatic&quot;, &quot;C&quot;, or &quot;FORTRAN&quot;, default is &quot;automatic&quot;</td>
</tr>
<tr>
<td>$ffmt</td>
<td>display format for single precision values</td>
<td>any FORTRAN format suitable for displaying single precision values, default is &quot;(lPG15.6E2)&quot;</td>
</tr>
<tr>
<td>$glist</td>
<td>whether <strong>list globals</strong> displays all global symbols or only those with defined types, when non-zero all globals are displayed</td>
<td>any integer, default is 0</td>
</tr>
<tr>
<td>$leading</td>
<td>leading characters stripped from names in a program's symbol table</td>
<td>any character string, default is &quot; _ &quot;</td>
</tr>
<tr>
<td>$1sort</td>
<td>whether output from <strong>list entries</strong>, <strong>list globals</strong>, <strong>list locals</strong>, and <strong>list statics</strong> is sorted, when non-zero output is sorted</td>
<td>any integer, default is 1</td>
</tr>
<tr>
<td>$mgrain</td>
<td>the number of times the command in $mstep is executed before checking monitors during execution of a <strong>continue</strong> or <strong>go</strong> command</td>
<td>Any positive integer, default is 1</td>
</tr>
<tr>
<td>$mstep</td>
<td>Fx command executed when a <strong>continue</strong> or <strong>go</strong> command is issued with active monitors</td>
<td>&quot;Step&quot;, &quot;step &quot;, &quot;instruction&quot;, or &quot;Instruction&quot;, default is &quot;step&quot;</td>
</tr>
<tr>
<td>Name</td>
<td>Purpose</td>
<td>Legal Values</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>$$numbers</td>
<td>display of source line numbers when non-zero line number are displayed</td>
<td>any integer, default is 1</td>
</tr>
<tr>
<td>$$prompt</td>
<td>Fx command prompt</td>
<td>any character string, default is &quot;Fx&gt;&quot;</td>
</tr>
<tr>
<td>$$read</td>
<td>execution of commands read from a file, when zero display commands are not executed</td>
<td>any integer, default is 1</td>
</tr>
<tr>
<td>$$sIen.</td>
<td>maximum number of characters displayed with s display format</td>
<td>any positive integer, default is 80</td>
</tr>
<tr>
<td>$$sList</td>
<td>whether the list statics command displays all static symbols or only those with defined types, when non-zero all static symbols are displayed</td>
<td>any integer, default is 0</td>
</tr>
<tr>
<td>$$smooth</td>
<td>smooth scrolling of the source window, when non-zero smooth scrolling is enabled</td>
<td>any integer, default is 0</td>
</tr>
<tr>
<td>$$tabsIze</td>
<td>the number of spaces to expand tabs</td>
<td>any positive integer, default is 8</td>
</tr>
<tr>
<td>$$tnace</td>
<td>maximum number of frames to create when performing a stack trace</td>
<td>any positive integer, default is 50</td>
</tr>
<tr>
<td>$$union</td>
<td>display of union members when unqualified union names are specified with the print command, when non-zero all members of a union are displayed</td>
<td>any integer, default is 1</td>
</tr>
<tr>
<td>$$walkcmds</td>
<td>Fx commands executed by walk command</td>
<td>any list of Fx commands, default is &quot;step&quot;</td>
</tr>
</tbody>
</table>
Appendix D

ASCII Table

ASCII codes 0 through 31 are control codes that may or may not have meaning on Linux. 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. The Dec, Oct, and Hex columns refer to the decimal, octal, and hexadecimal numerical representations.

<table>
<thead>
<tr>
<th>Character</th>
<th>Dec</th>
<th>Oct</th>
<th>Hex</th>
<th>Description</th>
<th>Character</th>
<th>Dec</th>
<th>Oct</th>
<th>Hex</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>0</td>
<td>000</td>
<td>00</td>
<td>null</td>
<td>32</td>
<td>040</td>
<td>20</td>
<td>space</td>
<td></td>
</tr>
<tr>
<td>SOH</td>
<td>1</td>
<td>001</td>
<td>01</td>
<td>start of heading</td>
<td>!</td>
<td>33</td>
<td>041</td>
<td>21</td>
<td>exclamation</td>
</tr>
<tr>
<td>STX</td>
<td>2</td>
<td>002</td>
<td>02</td>
<td>start of text</td>
<td>&quot;</td>
<td>34</td>
<td>042</td>
<td>22</td>
<td>quotation mark</td>
</tr>
<tr>
<td>ETX</td>
<td>3</td>
<td>003</td>
<td>03</td>
<td>end of text</td>
<td>#</td>
<td>35</td>
<td>043</td>
<td>23</td>
<td>number sign</td>
</tr>
<tr>
<td>ECT</td>
<td>4</td>
<td>004</td>
<td>04</td>
<td>end of trans</td>
<td>$</td>
<td>36</td>
<td>044</td>
<td>24</td>
<td>dollar sign</td>
</tr>
<tr>
<td>ENQ</td>
<td>5</td>
<td>005</td>
<td>05</td>
<td>enquiry</td>
<td>%</td>
<td>37</td>
<td>045</td>
<td>25</td>
<td>percent sign</td>
</tr>
<tr>
<td>ACK</td>
<td>6</td>
<td>006</td>
<td>06</td>
<td>acknowledge</td>
<td>&amp;</td>
<td>38</td>
<td>046</td>
<td>26</td>
<td>ampersand</td>
</tr>
<tr>
<td>BEL</td>
<td>7</td>
<td>007</td>
<td>07</td>
<td>bell code</td>
<td>'</td>
<td>39</td>
<td>047</td>
<td>27</td>
<td>apostrophe</td>
</tr>
<tr>
<td>BS</td>
<td>8</td>
<td>010</td>
<td>08</td>
<td>back space</td>
<td>(</td>
<td>40</td>
<td>050</td>
<td>28</td>
<td>opening paren</td>
</tr>
<tr>
<td>HT</td>
<td>9</td>
<td>011</td>
<td>09</td>
<td>horizontal tab</td>
<td>)</td>
<td>41</td>
<td>051</td>
<td>29</td>
<td>closing paren</td>
</tr>
<tr>
<td>LF</td>
<td>10</td>
<td>012</td>
<td>0A</td>
<td>line feed</td>
<td>*</td>
<td>42</td>
<td>052</td>
<td>2A</td>
<td>asterisk</td>
</tr>
<tr>
<td>VT</td>
<td>11</td>
<td>013</td>
<td>0B</td>
<td>vertical tab</td>
<td>+</td>
<td>43</td>
<td>053</td>
<td>2B</td>
<td>plus</td>
</tr>
<tr>
<td>FF</td>
<td>12</td>
<td>014</td>
<td>0C</td>
<td>form feed</td>
<td>,</td>
<td>44</td>
<td>054</td>
<td>2C</td>
<td>comma</td>
</tr>
<tr>
<td>CR</td>
<td>13</td>
<td>015</td>
<td>0D</td>
<td>carriage return</td>
<td>-</td>
<td>45</td>
<td>055</td>
<td>2D</td>
<td>minus</td>
</tr>
<tr>
<td>SO</td>
<td>14</td>
<td>016</td>
<td>0E</td>
<td>shift out</td>
<td>.</td>
<td>46</td>
<td>056</td>
<td>2E</td>
<td>period</td>
</tr>
<tr>
<td>SI</td>
<td>15</td>
<td>017</td>
<td>0F</td>
<td>shift in</td>
<td>/</td>
<td>47</td>
<td>057</td>
<td>2F</td>
<td>slash</td>
</tr>
<tr>
<td>DC1</td>
<td>16</td>
<td>020</td>
<td>10</td>
<td>data link escape</td>
<td>0</td>
<td>48</td>
<td>060</td>
<td>30</td>
<td>zero</td>
</tr>
<tr>
<td>DC2</td>
<td>17</td>
<td>021</td>
<td>11</td>
<td>device control 1</td>
<td>1</td>
<td>49</td>
<td>061</td>
<td>31</td>
<td>one</td>
</tr>
<tr>
<td>DC3</td>
<td>18</td>
<td>022</td>
<td>12</td>
<td>device control 2</td>
<td>2</td>
<td>50</td>
<td>062</td>
<td>32</td>
<td>two</td>
</tr>
<tr>
<td>DC4</td>
<td>19</td>
<td>023</td>
<td>13</td>
<td>device control 3</td>
<td>3</td>
<td>51</td>
<td>063</td>
<td>33</td>
<td>three</td>
</tr>
<tr>
<td>NAK</td>
<td>20</td>
<td>024</td>
<td>14</td>
<td>device control 4</td>
<td>4</td>
<td>52</td>
<td>064</td>
<td>34</td>
<td>four</td>
</tr>
<tr>
<td>SYN</td>
<td>21</td>
<td>025</td>
<td>15</td>
<td>negative ack</td>
<td>5</td>
<td>53</td>
<td>065</td>
<td>35</td>
<td>five</td>
</tr>
<tr>
<td>ETB</td>
<td>22</td>
<td>026</td>
<td>16</td>
<td>synch idle</td>
<td>6</td>
<td>54</td>
<td>066</td>
<td>36</td>
<td>six</td>
</tr>
<tr>
<td>CAN</td>
<td>23</td>
<td>027</td>
<td>17</td>
<td>end of trans blk</td>
<td>7</td>
<td>55</td>
<td>067</td>
<td>37</td>
<td>seven</td>
</tr>
<tr>
<td>EM</td>
<td>24</td>
<td>030</td>
<td>18</td>
<td>cancel</td>
<td>8</td>
<td>56</td>
<td>070</td>
<td>38</td>
<td>eight</td>
</tr>
<tr>
<td>SS</td>
<td>25</td>
<td>031</td>
<td>19</td>
<td>end of medium</td>
<td>9</td>
<td>57</td>
<td>071</td>
<td>39</td>
<td>nine</td>
</tr>
<tr>
<td>ESC</td>
<td>26</td>
<td>032</td>
<td>1A</td>
<td>special sequence</td>
<td>:</td>
<td>58</td>
<td>072</td>
<td>3A</td>
<td>colon</td>
</tr>
<tr>
<td>FS</td>
<td>27</td>
<td>033</td>
<td>1B</td>
<td>escape</td>
<td>;</td>
<td>59</td>
<td>073</td>
<td>3B</td>
<td>semicolon</td>
</tr>
<tr>
<td>GS</td>
<td>28</td>
<td>034</td>
<td>1C</td>
<td>file separator</td>
<td>&lt;</td>
<td>60</td>
<td>074</td>
<td>3C</td>
<td>less than</td>
</tr>
<tr>
<td>RS</td>
<td>29</td>
<td>035</td>
<td>1D</td>
<td>group separator</td>
<td>=</td>
<td>61</td>
<td>075</td>
<td>3D</td>
<td>equal</td>
</tr>
<tr>
<td>US</td>
<td>30</td>
<td>036</td>
<td>1E</td>
<td>record separator</td>
<td>&gt;</td>
<td>62</td>
<td>076</td>
<td>3E</td>
<td>greater than</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>037</td>
<td>1F</td>
<td>unit separator</td>
<td>?</td>
<td>63</td>
<td>077</td>
<td>3F</td>
<td>question mark</td>
</tr>
<tr>
<td>Character</td>
<td>Dec</td>
<td>Oct</td>
<td>Hex</td>
<td>Description</td>
<td>Character</td>
<td>Dec</td>
<td>Oct</td>
<td>Hex</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>----------------------</td>
<td>-----------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>----------------------</td>
</tr>
<tr>
<td>@</td>
<td>64</td>
<td>100</td>
<td>40</td>
<td>commercial at</td>
<td>~</td>
<td>126</td>
<td>176</td>
<td>7E</td>
<td>tilde</td>
</tr>
<tr>
<td>A</td>
<td>65</td>
<td>101</td>
<td>41</td>
<td>upper case letter</td>
<td>127</td>
<td>177</td>
<td>7F</td>
<td>delete</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>66</td>
<td>102</td>
<td>42</td>
<td>upper case letter</td>
<td>128</td>
<td>200</td>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>67</td>
<td>103</td>
<td>43</td>
<td>upper case letter</td>
<td>129</td>
<td>201</td>
<td>81</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>68</td>
<td>104</td>
<td>44</td>
<td>upper case letter</td>
<td>130</td>
<td>202</td>
<td>82</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>69</td>
<td>105</td>
<td>45</td>
<td>upper case letter</td>
<td>131</td>
<td>203</td>
<td>83</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>70</td>
<td>106</td>
<td>46</td>
<td>upper case letter</td>
<td>132</td>
<td>204</td>
<td>84</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>71</td>
<td>107</td>
<td>47</td>
<td>upper case letter</td>
<td>133</td>
<td>205</td>
<td>85</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>72</td>
<td>110</td>
<td>48</td>
<td>upper case letter</td>
<td>134</td>
<td>206</td>
<td>86</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>73</td>
<td>111</td>
<td>49</td>
<td>upper case letter</td>
<td>135</td>
<td>207</td>
<td>87</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>74</td>
<td>112</td>
<td>4A</td>
<td>upper case letter</td>
<td>136</td>
<td>210</td>
<td>88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>75</td>
<td>113</td>
<td>4B</td>
<td>upper case letter</td>
<td>137</td>
<td>211</td>
<td>89</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>76</td>
<td>114</td>
<td>4C</td>
<td>upper case letter</td>
<td>138</td>
<td>212</td>
<td>8A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>77</td>
<td>115</td>
<td>4D</td>
<td>upper case letter</td>
<td>139</td>
<td>213</td>
<td>8B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>78</td>
<td>116</td>
<td>4E</td>
<td>upper case letter</td>
<td>140</td>
<td>214</td>
<td>8C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>79</td>
<td>117</td>
<td>4F</td>
<td>upper case letter</td>
<td>141</td>
<td>215</td>
<td>8D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>80</td>
<td>120</td>
<td>50</td>
<td>upper case letter</td>
<td>142</td>
<td>216</td>
<td>8E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td>81</td>
<td>121</td>
<td>51</td>
<td>upper case letter</td>
<td>143</td>
<td>217</td>
<td>8F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>82</td>
<td>122</td>
<td>52</td>
<td>upper case letter</td>
<td>144</td>
<td>220</td>
<td>90</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>83</td>
<td>123</td>
<td>53</td>
<td>upper case letter</td>
<td>145</td>
<td>221</td>
<td>91</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>84</td>
<td>124</td>
<td>54</td>
<td>upper case letter</td>
<td>146</td>
<td>222</td>
<td>92</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>85</td>
<td>125</td>
<td>55</td>
<td>upper case letter</td>
<td>147</td>
<td>223</td>
<td>93</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>86</td>
<td>126</td>
<td>56</td>
<td>upper case letter</td>
<td>148</td>
<td>224</td>
<td>94</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>87</td>
<td>127</td>
<td>57</td>
<td>upper case letter</td>
<td>149</td>
<td>225</td>
<td>95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>88</td>
<td>128</td>
<td>58</td>
<td>upper case letter</td>
<td>150</td>
<td>226</td>
<td>96</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>89</td>
<td>129</td>
<td>59</td>
<td>upper case letter</td>
<td>151</td>
<td>227</td>
<td>97</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>90</td>
<td>130</td>
<td>5A</td>
<td>opening bracket</td>
<td>152</td>
<td>230</td>
<td>98</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[</td>
<td>91</td>
<td>131</td>
<td>5B</td>
<td>back slash</td>
<td>153</td>
<td>231</td>
<td>99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>]</td>
<td>92</td>
<td>134</td>
<td>5C</td>
<td>closing bracket</td>
<td>154</td>
<td>232</td>
<td>9A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>93</td>
<td>135</td>
<td>5D</td>
<td>underscore</td>
<td>155</td>
<td>233</td>
<td>9B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_</td>
<td>94</td>
<td>136</td>
<td>5E</td>
<td>circumflex</td>
<td>156</td>
<td>234</td>
<td>9C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>`</td>
<td>95</td>
<td>137</td>
<td>5F</td>
<td>underscore</td>
<td>157</td>
<td>235</td>
<td>9D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>96</td>
<td>140</td>
<td>60</td>
<td>grave accent</td>
<td>158</td>
<td>236</td>
<td>9E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>97</td>
<td>141</td>
<td>61</td>
<td>lower case letter</td>
<td>159</td>
<td>237</td>
<td>9F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>98</td>
<td>142</td>
<td>62</td>
<td>lower case letter</td>
<td>160</td>
<td>240</td>
<td>A0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>99</td>
<td>143</td>
<td>63</td>
<td>lower case letter</td>
<td>161</td>
<td>241</td>
<td>A1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>100</td>
<td>144</td>
<td>64</td>
<td>lower case letter</td>
<td>162</td>
<td>242</td>
<td>A2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>101</td>
<td>145</td>
<td>65</td>
<td>lower case letter</td>
<td>163</td>
<td>243</td>
<td>A3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>102</td>
<td>146</td>
<td>66</td>
<td>lower case letter</td>
<td>164</td>
<td>244</td>
<td>A4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>103</td>
<td>147</td>
<td>67</td>
<td>lower case letter</td>
<td>165</td>
<td>245</td>
<td>A5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>104</td>
<td>148</td>
<td>68</td>
<td>lower case letter</td>
<td>166</td>
<td>246</td>
<td>A6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>105</td>
<td>151</td>
<td>69</td>
<td>lower case letter</td>
<td>167</td>
<td>247</td>
<td>A7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td>106</td>
<td>152</td>
<td>6A</td>
<td>lower case letter</td>
<td>168</td>
<td>250</td>
<td>A8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>k</td>
<td>107</td>
<td>153</td>
<td>6B</td>
<td>lower case letter</td>
<td>169</td>
<td>251</td>
<td>A9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>l</td>
<td>108</td>
<td>154</td>
<td>6C</td>
<td>lower case letter</td>
<td>170</td>
<td>252</td>
<td>AA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>m</td>
<td>109</td>
<td>155</td>
<td>6D</td>
<td>lower case letter</td>
<td>171</td>
<td>253</td>
<td>AB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>110</td>
<td>156</td>
<td>6E</td>
<td>lower case letter</td>
<td>172</td>
<td>254</td>
<td>AC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>111</td>
<td>157</td>
<td>6F</td>
<td>lower case letter</td>
<td>173</td>
<td>255</td>
<td>AD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>112</td>
<td>160</td>
<td>70</td>
<td>lower case letter</td>
<td>174</td>
<td>256</td>
<td>AE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q</td>
<td>113</td>
<td>161</td>
<td>71</td>
<td>lower case letter</td>
<td>175</td>
<td>257</td>
<td>AF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>114</td>
<td>162</td>
<td>72</td>
<td>lower case letter</td>
<td>176</td>
<td>260</td>
<td>B0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>115</td>
<td>163</td>
<td>73</td>
<td>lower case letter</td>
<td>177</td>
<td>261</td>
<td>B1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>116</td>
<td>164</td>
<td>74</td>
<td>lower case letter</td>
<td>178</td>
<td>262</td>
<td>B2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>u</td>
<td>117</td>
<td>165</td>
<td>75</td>
<td>lower case letter</td>
<td>179</td>
<td>263</td>
<td>B3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>118</td>
<td>166</td>
<td>76</td>
<td>lower case letter</td>
<td>180</td>
<td>264</td>
<td>B4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>w</td>
<td>119</td>
<td>167</td>
<td>77</td>
<td>lower case letter</td>
<td>181</td>
<td>265</td>
<td>B5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>120</td>
<td>170</td>
<td>78</td>
<td>lower case letter</td>
<td>182</td>
<td>266</td>
<td>B6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>121</td>
<td>171</td>
<td>79</td>
<td>lower case letter</td>
<td>183</td>
<td>267</td>
<td>B7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>122</td>
<td>172</td>
<td>7A</td>
<td>lower case letter</td>
<td>184</td>
<td>270</td>
<td>B8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{</td>
<td>123</td>
<td>173</td>
<td>7B</td>
<td>opening brace</td>
<td>185</td>
<td>271</td>
<td>B9</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>124</td>
<td>174</td>
<td>7C</td>
<td>vertical bar</td>
<td>186</td>
<td>272</td>
<td>BA</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>125</td>
<td>175</td>
<td>7D</td>
<td>closing brace</td>
<td>187</td>
<td>273</td>
<td>BB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Character</td>
<td>Dec</td>
<td>Oct</td>
<td>Hex</td>
<td>Character</td>
<td>Dec</td>
<td>Oct</td>
<td>Hex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td></td>
<td></td>
</tr>
<tr>
<td>¼</td>
<td>188</td>
<td>274</td>
<td>BC</td>
<td>p</td>
<td>222</td>
<td>336</td>
<td>DE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>½</td>
<td>189</td>
<td>275</td>
<td>BD</td>
<td>b</td>
<td>223</td>
<td>337</td>
<td>DF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>¾</td>
<td>190</td>
<td>276</td>
<td>BE</td>
<td>à</td>
<td>224</td>
<td>340</td>
<td>E0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>á</td>
<td>191</td>
<td>277</td>
<td>BF</td>
<td>á</td>
<td>225</td>
<td>341</td>
<td>E1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>À</td>
<td>192</td>
<td>300</td>
<td>C0</td>
<td>a</td>
<td>226</td>
<td>342</td>
<td>E2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Á</td>
<td>193</td>
<td>301</td>
<td>C1</td>
<td>a</td>
<td>227</td>
<td>343</td>
<td>E3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Â</td>
<td>194</td>
<td>302</td>
<td>C2</td>
<td>a</td>
<td>228</td>
<td>344</td>
<td>E4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ã</td>
<td>195</td>
<td>303</td>
<td>C3</td>
<td>a</td>
<td>229</td>
<td>345</td>
<td>E5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Á</td>
<td>196</td>
<td>304</td>
<td>C4</td>
<td>æ</td>
<td>230</td>
<td>346</td>
<td>E6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>À</td>
<td>197</td>
<td>305</td>
<td>C5</td>
<td>c</td>
<td>231</td>
<td>347</td>
<td>E7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Æ</td>
<td>198</td>
<td>306</td>
<td>C6</td>
<td>è</td>
<td>232</td>
<td>350</td>
<td>E8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ç</td>
<td>199</td>
<td>307</td>
<td>C7</td>
<td>é</td>
<td>233</td>
<td>351</td>
<td>E9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ç</td>
<td>200</td>
<td>310</td>
<td>C8</td>
<td>è</td>
<td>234</td>
<td>352</td>
<td>EA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>È</td>
<td>201</td>
<td>311</td>
<td>C9</td>
<td>ë</td>
<td>235</td>
<td>353</td>
<td>EB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>È</td>
<td>202</td>
<td>312</td>
<td>CA</td>
<td>i</td>
<td>236</td>
<td>354</td>
<td>EC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>È</td>
<td>203</td>
<td>313</td>
<td>CB</td>
<td>i</td>
<td>237</td>
<td>355</td>
<td>ED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>È</td>
<td>204</td>
<td>314</td>
<td>CC</td>
<td>i</td>
<td>238</td>
<td>356</td>
<td>EE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>È</td>
<td>205</td>
<td>315</td>
<td>CD</td>
<td>ì</td>
<td>239</td>
<td>357</td>
<td>EF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>È</td>
<td>206</td>
<td>316</td>
<td>CE</td>
<td>ò</td>
<td>240</td>
<td>360</td>
<td>F0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>È</td>
<td>207</td>
<td>317</td>
<td>CF</td>
<td>ù</td>
<td>241</td>
<td>361</td>
<td>F1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>208</td>
<td>320</td>
<td>D0</td>
<td>o</td>
<td>242</td>
<td>362</td>
<td>F2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ñ</td>
<td>209</td>
<td>321</td>
<td>D1</td>
<td>ó</td>
<td>243</td>
<td>363</td>
<td>F3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Õ</td>
<td>210</td>
<td>322</td>
<td>D2</td>
<td>ò</td>
<td>244</td>
<td>364</td>
<td>F4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Õ</td>
<td>211</td>
<td>323</td>
<td>D3</td>
<td>ô</td>
<td>245</td>
<td>365</td>
<td>F5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Õ</td>
<td>212</td>
<td>324</td>
<td>D4</td>
<td>Õ</td>
<td>246</td>
<td>366</td>
<td>F6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Õ</td>
<td>213</td>
<td>325</td>
<td>D5</td>
<td>Õ</td>
<td>247</td>
<td>367</td>
<td>F7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ő</td>
<td>214</td>
<td>326</td>
<td>D6</td>
<td>o</td>
<td>248</td>
<td>370</td>
<td>F8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>×</td>
<td>215</td>
<td>327</td>
<td>D7</td>
<td>û</td>
<td>249</td>
<td>371</td>
<td>F9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ø</td>
<td>216</td>
<td>330</td>
<td>D8</td>
<td>ù</td>
<td>250</td>
<td>372</td>
<td>FA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ê</td>
<td>217</td>
<td>331</td>
<td>D9</td>
<td>û</td>
<td>251</td>
<td>373</td>
<td>FB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ê</td>
<td>218</td>
<td>332</td>
<td>DA</td>
<td>û</td>
<td>252</td>
<td>374</td>
<td>FC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ê</td>
<td>219</td>
<td>333</td>
<td>DB</td>
<td>ý</td>
<td>253</td>
<td>375</td>
<td>FD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ê</td>
<td>220</td>
<td>334</td>
<td>DC</td>
<td>ù</td>
<td>254</td>
<td>376</td>
<td>FE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ê</td>
<td>221</td>
<td>335</td>
<td>DD</td>
<td>ý</td>
<td>255</td>
<td>377</td>
<td>FF</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix E

Bibliography

FORTRAN 90/95
These books and manuals are useful references for the Fortran 90/95 programming language and the floating point math format used by Absoft Pro Fortran on Linux.

Michael Metcalf and John Reid, *FORTRAN 90/95 explained*, Oxford University Press (1996)


FORTRAN 77
These books and manuals are useful references for the FORTRAN language and the floating point math format used by Absoft Pro Fortran on Linux.


Harry Katzban, Jr., *FORTRAN 77*, Van Nostrand Reinhold Company (1978)


Appendix F

Technical Support

The Absoft Technical Support Group will provide technical assistance to all registered users of current products. They will not answer general questions about operating systems, operating system interfaces, graphical user interfaces, or teach programming. 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 language reference manuals to be sure your problem is not covered here. Specifically, refer to the chapter Using the Compilers in this manual. 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) step-by-step example if possible.

Contacting Technical Support:

Address: Absoft Corporation
Attn: Technical Support
2781 Bond Street
Rochester Hills, MI 48309
<table>
<thead>
<tr>
<th></th>
<th>Phone Number</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Telephone</td>
<td>(248) 853-0095</td>
<td>9am - 3pm EST</td>
</tr>
<tr>
<td>FAX</td>
<td>(248) 853-0108</td>
<td>24 Hours</td>
</tr>
<tr>
<td>Email</td>
<td><a href="mailto:support@absoft.com">support@absoft.com</a></td>
<td>24 Hours</td>
</tr>
<tr>
<td>World Wide Web</td>
<td><a href="http://www.absoft.com">http://www.absoft.com</a></td>
<td></td>
</tr>
<tr>
<td>Section</td>
<td>Page(s)</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>---------</td>
<td></td>
</tr>
<tr>
<td>132 column source code</td>
<td>54, 71</td>
<td></td>
</tr>
<tr>
<td>386, porting from</td>
<td>95</td>
<td></td>
</tr>
<tr>
<td>Absoft address</td>
<td>321</td>
<td></td>
</tr>
<tr>
<td>Absoft Editor commands</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Edit menu</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>File menu</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>Help menu</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>Tools menu</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>View menu</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>Window menu</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>ABSOFT_RTL_FLAGS</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>Adding Menus</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>advanced optimizations</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>Alert dialog box</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>amake tool</td>
<td>132</td>
<td></td>
</tr>
<tr>
<td>ANSI C source format</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>Apple Events</td>
<td>115</td>
<td></td>
</tr>
<tr>
<td>class</td>
<td>116</td>
<td></td>
</tr>
<tr>
<td>ID 116</td>
<td></td>
<td></td>
</tr>
<tr>
<td>target</td>
<td>116</td>
<td></td>
</tr>
<tr>
<td>array</td>
<td></td>
<td></td>
</tr>
<tr>
<td>boundary checking</td>
<td>52, 63</td>
<td></td>
</tr>
<tr>
<td>ASCII table</td>
<td>311, 315</td>
<td></td>
</tr>
<tr>
<td>assembly language</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>interfacing with FORTRAN</td>
<td>155</td>
<td></td>
</tr>
<tr>
<td>ATTRIBUTES directive</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>basic optimizations</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>BLOCK=-1 specifier</td>
<td>66</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>function results</td>
<td>151</td>
<td></td>
</tr>
<tr>
<td>interfacing with FORTRAN(see interfacing FORTRAN and C)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C/C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>options</td>
<td>76</td>
<td></td>
</tr>
<tr>
<td>carriage returns</td>
<td>94</td>
<td></td>
</tr>
<tr>
<td>CASE DEFAULT</td>
<td>69</td>
<td></td>
</tr>
<tr>
<td>Check Syntax menu command</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>Clear menu command</td>
<td>107</td>
<td></td>
</tr>
<tr>
<td>Clipboard</td>
<td>106</td>
<td></td>
</tr>
<tr>
<td>Close All menu command</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>Close menu command</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>Command-Enter</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>Command-Period</td>
<td></td>
<td></td>
</tr>
<tr>
<td>termination</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>Command-Period checking</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>Command-Return</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>COMMON blocks from C</td>
<td>154</td>
<td></td>
</tr>
<tr>
<td>COMMON, aligning data</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>Compile menu command</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>compiler directives</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>ATTRIBUTES directive</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>FIXED directive</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td>FIXEDFORMLINESIZE directive</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td>FREE[FORM] directive</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td>NAME directive</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>NOFREEFORM directive</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td>PACK[ON] directive</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>PACKOFF directive</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>STACK directive</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>compiler options</td>
<td>303</td>
<td></td>
</tr>
<tr>
<td>-8, Fortran 90</td>
<td>71</td>
<td></td>
</tr>
<tr>
<td>-A, ANSI C source format</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>-A, suppress alignment warnings</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>-ansi, ANSI C</td>
<td>81</td>
<td></td>
</tr>
<tr>
<td>-B19, Assume Pointer Aliases Exist</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>-C, check boundaries</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>-c, relocatable object</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>-D, define compiler variable</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>-d, define preprocessor variable</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>-d, one trip DO loops</td>
<td>67</td>
<td></td>
</tr>
<tr>
<td>-E, preprocess to file</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>-ea, stop on error</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>-ej, one trip DO loops</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>-en, non-standard usage</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>-ep, demote Double Precision</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>-eq, allow greater than 100 errors</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>-ER, default recursion</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>-et, runtime stack trace</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>-f, case fold</td>
<td>66, 89</td>
<td></td>
</tr>
<tr>
<td>-f, case folding</td>
<td>154</td>
<td></td>
</tr>
<tr>
<td>-f, fixed source form</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>-f, freed source form</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>-fhandle-exceptions, exception handling</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>-fnoinlines, no inlines</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>-g, Fx debugging</td>
<td>89</td>
<td></td>
</tr>
<tr>
<td>-H, include tree to sider</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>-H, max lines to unroll</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>-h, unroll count</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>-i, integer sizes</td>
<td>51, 65</td>
<td></td>
</tr>
<tr>
<td>-K, escape sequences</td>
<td>54, 69</td>
<td></td>
</tr>
<tr>
<td>-K, K &amp; R C source format</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>-L, library path specification</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>-l, library specification</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>-MS7D, Microsoft directives</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>-N1, static storage</td>
<td>66, 69</td>
<td></td>
</tr>
<tr>
<td>-N102, zero extend INTEGER*1</td>
<td>67</td>
<td></td>
</tr>
<tr>
<td>-N109, case fold</td>
<td>67</td>
<td></td>
</tr>
<tr>
<td>-N11, use long branches</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>-N110, no Common mangling</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>-N112, IBM VS Free-Form</td>
<td>71</td>
<td></td>
</tr>
<tr>
<td>-N113, floating point sizes</td>
<td>51, 69</td>
<td></td>
</tr>
<tr>
<td>-N114, warnings for undeclared variables</td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>-N115, pad source lines</td>
<td>71</td>
<td></td>
</tr>
<tr>
<td>-N116, BLOCK DATA code section</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>-N124, procedure trace</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>-N15, append underscore</td>
<td>67</td>
<td></td>
</tr>
<tr>
<td>-N16, UNIT specifier</td>
<td>69</td>
<td></td>
</tr>
<tr>
<td>-N18, function specifier</td>
<td>74</td>
<td></td>
</tr>
<tr>
<td>-N2, double precision</td>
<td>68</td>
<td></td>
</tr>
<tr>
<td>-N20, left-to-right</td>
<td>69</td>
<td></td>
</tr>
<tr>
<td>-N22, set Common</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>-N26, set big-endian</td>
<td>71</td>
<td></td>
</tr>
<tr>
<td>-N26, set little-endian</td>
<td>71</td>
<td></td>
</tr>
<tr>
<td>-N3, record lengths</td>
<td>66</td>
<td></td>
</tr>
<tr>
<td>-N32, non-ANSI</td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>-N33, don’t align structure fields;</td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>-N34, align COMMON</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>-N4, CASE DEFAULT</td>
<td>69</td>
<td></td>
</tr>
<tr>
<td>-N41, evaluate constant functions</td>
<td>74</td>
<td></td>
</tr>
<tr>
<td>-N5, treat DATA as constants</td>
<td>74</td>
<td></td>
</tr>
</tbody>
</table>

Fortran User Guide
Index

-N51, 32 bit RECL ............................................66
-N52, check syntax only ..................................73
-N57, align structure fields to 2-byte boundaries;75
-N58, align structure fields to 4-byte boundaries;76
-N59, align structure fields to 8-byte boundaries;76
-N7, sign extend ..............................................68
-N80, procedure trace ......................................49
-N9, Command-Period ....................................30
-N90, CHARACTER argument parameters ........73
-nostdinc, no standard includes ..................79
-O, advanced optimizations ...........................31
-O1, basic optimizations ................................31
-O2, advanced optimizations ...........................31
-Oa, pointer aliases ......................................62
-p, module files ............................................48
-pedantic, ANSI warnings ..............................77
-pedantic-errors, treat all warnings as errors ....78
-q, quiet .....................................................48, 62
-QS1, no fma instructions ..............................32, 34
-Rb, check array conformance .......................52
-Rb, check boundaries ...................................52
-round, FPU rounding mode ..........................33
-Rp, check pointers ......................................52
-Rs, check substrings ....................................52
-S, language assembly ...................................34
-s, static storage ...........................................51, 66, 89
-SysModFiles, F95 include directory...............48
-T, max internal handle ................................47, 61
-t, temporary strings ....................................47, 61
-U, loop unrolling ........................................75
-u, undefined preprocessor variable ..............79
-u, undefined symbol ....................................83
-v, show progress ........................................48, 62
-v, VAX Tab-Format .......................................71
-v, verbose ..................................................78
-VAX compatibility .......................................66
-W, line length .............................................54
-w, suppress all warnings ...............................77
-w, suppress compiler warnings ...............48, 62
-W, wide format ...........................................71
-Wall, warnings for all common ......................78
-Werror, treat warnings as errors .................78
-Wuninitialized, warnings for uninitialized ....78
-Wunused, warnings for unused .................77
-x, disable compiler directive .......................50
-x, conditional compilation ............................63
-Xlinker, linker options ................................83
-YCFRL=1, CHARACTER argument parameters ...51
-YCOM_NAMES, COMMON block case ...............56
-YCOM_PFX, COMMON block prefix .................56
-YCOM_SFX, COMMON block suffix .................56
-YDEALLOC, cache control .............................56
-YEXT_NAMES, external names ....................54
-YEXT_PFX, external symbol prefix ..............56
-YEXT_SFX, external symbol suffix ..............53
-YNDFP, type elements ................................55
-YPEI, pointers equivalent to integers ............52
-z, suppress messages ...................................47
-Z, suppress warning number .........................48
complex data types ........................................49
equivalent declarations in C ......................147
conditional compilation ......................63
conditional compilation variables ..........64
constant functions ..................................74
constant propagation ................................74
continuation lines .....................................90
conventions used in the manual ...............16, 2
Copy menu command ..................................16, 107
Customize Toolbar menu command ............23
Cut menu command ......................................16, 106
DATA treated as constants .......................74
DATE subroutine .......................................90
documentation conventions .......................2
debugging ..................................................155
delete menu command ................................16
display compiler alignment warning messages ..73
divide by zero exceptions ......................101
dO loops ..................................................67
dO Loops ..................................................51
dO Loops ..................................................51
dO Loops ..................................................51
dO Loops ..................................................51
dO Loops ..................................................51
dO Loops ..................................................51
dO Loops ..................................................51
dO Loops ..................................................51
end-of-file character .................................105
Environment Preferences ............................11
errors ......................................................47
at runtime ..............................................91
event loops .............................................109
exceptions ..............................................101
divide by zero ..........................................101
operand error ..........................................101
overflow ................................................101
extensions ................................ ...............92
key Microsoft FORTRAN ..............................92
key Sun FORTRAN ones ...............................94
key VAX FORTRAN ones .............................89
key VS FORTRAN ones ...............................91
extensions to FORTRAN 77 ......................2
external procedure name .............................98
f77.exe .................................................128
File ......................................................13
Find menu command ....................................17
Find Next menu command .............................17
FIXED directive .........................................58
FIXEDFORMLINESIZE directive .................58
floating point ...........................................102
floating point unit exception handling ..........101
rounding direction ....................................101
FORM=’BINARY’ specifier ..........................93
Format Preferences .....................................9
Color .......................................................9
Language ................................................9
Fortran 77 .................................................1
Fortran 77 extensions ......................2
Fortran 77 Fixed Source Form ..................71
Fortran 90 Fixed Source Form ..................54
Fortran 90 Free Source Form ...................53, 71

Fortran User Guide
### Index

<table>
<thead>
<tr>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>151</td>
<td>intrinsic functions</td>
</tr>
<tr>
<td>150</td>
<td>LOC parameters</td>
</tr>
<tr>
<td>153</td>
<td>math routines</td>
</tr>
<tr>
<td>153</td>
<td>VAL parameters</td>
</tr>
<tr>
<td>75</td>
<td>loop unrolling</td>
</tr>
<tr>
<td>96</td>
<td>MacFortran</td>
</tr>
<tr>
<td>96</td>
<td>MacFortran II</td>
</tr>
<tr>
<td>96</td>
<td>MacFortran/020</td>
</tr>
<tr>
<td>20</td>
<td>math routines</td>
</tr>
<tr>
<td>153</td>
<td>FORTRAN memory management</td>
</tr>
<tr>
<td>97</td>
<td>menu adding</td>
</tr>
<tr>
<td>110</td>
<td>checkmarks deleting</td>
</tr>
<tr>
<td>112</td>
<td>deleting an item</td>
</tr>
<tr>
<td>112</td>
<td>item, enabling and disabling</td>
</tr>
<tr>
<td>93</td>
<td>metacommands, Microsoft FORTRAN</td>
</tr>
<tr>
<td>93</td>
<td>Microsoft FORTRAN metacommands</td>
</tr>
<tr>
<td>92</td>
<td>porting from</td>
</tr>
<tr>
<td>48</td>
<td>module files</td>
</tr>
<tr>
<td>30</td>
<td>MRWE customization</td>
</tr>
<tr>
<td>107</td>
<td>Event Loop programming</td>
</tr>
<tr>
<td>112</td>
<td>mrwe_DoMenu</td>
</tr>
<tr>
<td>94</td>
<td>MS-DOS, porting from</td>
</tr>
<tr>
<td>94</td>
<td>MS-DOS files, translating</td>
</tr>
<tr>
<td>124</td>
<td>Multiple Windows</td>
</tr>
<tr>
<td>105</td>
<td>-N9 output</td>
</tr>
<tr>
<td>57</td>
<td>NAME directive naming conventions</td>
</tr>
<tr>
<td>153</td>
<td>New menu command</td>
</tr>
<tr>
<td>13</td>
<td>New Window menu command</td>
</tr>
<tr>
<td>21</td>
<td>Next Error menu command</td>
</tr>
<tr>
<td>21</td>
<td>NeXT, porting from</td>
</tr>
<tr>
<td>78</td>
<td>no inlines</td>
</tr>
<tr>
<td>58</td>
<td>NOFREEFORM directive non ANSI warnings</td>
</tr>
</tbody>
</table>

Fortran 95 options ............................................. 46

FORTRAN math routines calling from C ........................................ 153
FPU rounding mode ................................................ 33
fr.exe ........................................................................ 128
FREE[FORM] directive ................................................ 58
frequent Command-Period checks .................................. 30
Fspli t utility tool ..................................................... 129
FTNCHEK utility tool .................................................. 129
function call to C from FORTRAN .......................... 151
call to FORTRAN from C ........................................ 152
decomposition .......................................................... 74
Fx 157 assembly .............................................................. 223
Breakpoints ................................................................ 184, 208
Changing Variables ..................................................... 165
Command Text Field .................................................. 194
Continue ................................................................. 212
Core File ..................................................................... 200
Displaying Program Variables ..................................... 216
Executing Single Statements ....................................... 164, 207
Launching the Debugger ............................................. 159
Monitors ...................................................................... 186
Preferences .................................................................. 170
Preparing Your Program For Debugging ...................... 198
Registers ...................................................................... 191
Restart ........................................................................ 179
Setting a Breakpoint .................................................. 162
Stack ............................................................................ 191
User Configurable Buttons .......................................... 196
Variables ...................................................................... 184
Walking ....................................................................... 181
g77 .............................................................................. 148
gcc ............................................................................... 148
Go to Line menu command .......................................... 20
graying of text ............................................................ 2
Help menu ................................................................... 23
ibe.exe ........................................................................ 128
IBM RS/6000, porting from ........................................ 94
IDATE subroutine ..................................................... 90
IEEE floating point math ............................................ 102
IMSL libraries ........................................................... 86, 87
include tree to stderr .................................................. 78
input from keyboard ................................................... 105
Insert Continuation menu command ......................... 19
Intel 386, porting from ........................................... 95
interfacing FORTRAN and C 
calling FORTRAN math routines ................................ 153
compatible type declarations ..................................... 147
function call to C from FORTRAN .......................... 151
function call to FORTRAN from C .......................... 152
function results ......................................................... 151
LOC function ............................................................. 151
passing an array ........................................................ 151
passing pointers ........................................................ 149
passing strings .......................................................... 152
passing values .......................................................... 149
reference parameters ................................................ 149
passing to C .............................................................. 150
passing to FORTRAN ................................................ 149
VAL function .............................................................. 150

Fortran User Guide
Index

one trip DO ...........................................67
open additional windows..........................124
Open menu command ................................14
operand menu command ............................101
optimization
  constant propagation...........................74
  function decomposition .......................74
optimizations
  loop unrolling ..................................75
options ..............................................303
options, manual convention ......................2
other porting issues ................................96
overflow exceptions ................................101
PACK[ON] directive .................................59
PACKOFF directive .................................59
page setup dialog ..................................106
Page Setup menu command .......................15
Paste menu command ................................16, 107
Plug-Ins .............................................84
porting code .........................................89
doesn’t run correctly; ...............................91
from IBM RS/6000 .................................94
from Intel 386 .......................................95
from Microsoft FORTRAN .................92
from MS-DOS .......................................92
from NeXT ..........................................94
from SCO Unix ..................................95
from Sparc ..........................................94
from Sun FORTRAN.................................94
from VAX FORTRAN ...............................89
from VS FORTRAN .................................91
PowerPC options ..................................34
ppcbe.exe ...........................................128
preconnected units ..................................104
Preferences menu command .....................9
Previous Error menu command ..................21
Print menu command ...............................15
procedure naming conventions .................154
profiling ..............................................155
Properties menu command .......................18
qualifiers, VAX FORTRAN ......................90
Quit menu command ................................106
RAN function ......................................90
rc, resource compiler .............................127
Receiving Apple Events ..........................119
RECLS
  for 32-bit words ..................................66
relocatable object ..................................34
removing linefeed character .....................94
Required Apple Events ..........................117
RevEnd menu command ............................15
road maps ...........................................2
RS/6000, porting from ............................94
runtime error messages ...........................91
Save All menu command ..........................15
Save As menu command ...........................14
Save menu command ................................14
SCO Unix, porting from .........................95
scriptable application ...........................123
SECONDS subroutine ............................90
Select All menu command .......................16
Sending Apple Events ............................118
Shift Left menu command .......................48, 62
show compiler progress ..........................48
sign extension ......................................68
source line length ..................................54
Sparc, Absoft compiler for .....................94
Sparc, porting from ..............................94
square brackets, defined .........................2
STACK directive ....................................60
stack trace ...........................................31
static storage .......................................51
Stop menu command ................................21
string length .......................................47
strings
  passing FORTRAN to C .........................152
STRUCTURE fields, aligning ..................70, 75, 76
Sun FORTRAN, porting from .....................94
support ..............................................321
suppress list of compiler warning messages ..48
TABSIZE variable .................................129
target file, updating ..............................133
technical support ...................................321
text selection ....................................... 5
Tile Horizontally menu command ...............22
Tile Vertically menu command .................22
TIME subroutine ..................................90
Tools menu ..........................................20
Tools Preferences ..................................13
tracing ..............................................49, 63
treat all warnings as errors ....................78
undefined symbol ..................................83
underlined text, defined .........................2
Undo menu command ..............................16, 106
UNIT specifier ......................................69
updating target file ..............................133
use long branches ..................................30
using compilers .................................... 6
VALUE statement ..................................150
VAST pre-processor ............................... 85
VAX compatibility ..................................66
VAX FORTRAN
  porting from .....................................89
  qualifiers ...........................................90
  VAX Tab-Format source ....................... 71
  verbose ........................................... 78
  View menu .........................................18
  VS FORTRAN, porting from ...................91
  wide source format ..............................71
  Window menu .......................................21
  Window menu command .........................23
Y2K bug ............................................ 3