MICROSOFT®

The High Performance Software™

Microsoft® FORTRAN Compiler

for the Apple® Macintosh™

Microsoft Corporation

If you have comments about the documentation that accompanies the software, complete the Document Report at the back of this manual and return it to Microsoft Corporation.

If you have comments about the software, complete the Software Problem Report at the back of this manual and return it to Microsoft Corporation.

Document Number 690500003-210-R00-0685
Part Number 005-096-007

The **Microsoft FORTRAN** documentation is divided into six major sections, separated by tabs. The first four tabs pertain directly to FORTRAN. The final two tabs cover utilities and the Macintosh interface.

Brief introductory material appears before the first tab. This includes a table of contents covering the first four tabs, and a section titled "**Getting Started**." Getting Started examines the contents of the distribution disks, explains how to organize the disks for use on your Macintosh configuration, and steps you through the use of Microsoft FORTRAN to write a short sample program. Getting Started attempts to answer many of the questions that new users have.

Tab 1: The **User's Guide** contains an introduction to FORTRAN, a chapter on the Microsoft FORTRAN compiler, and a chapter on the interactive debugger.

Tab 2: The **Reference Manual** contains six chapters that cover the syntax and semantics of the FORTRAN language.

Tab 3: The **Appendices** discuss overlays, error messages, assembly language subroutines, IEEE floating point, RAD50 character representation, "C" language subroutines, file preconnections, and restrictions on this implementation of FORTRAN 77.

Tab 4: The **Index** covers material in the User's Guide, Reference Manual, and Appendices.

Tab 5: The **Utilities** cover the Linker and Library Manager, Subroutines (spool, time, and date), Edit, and the Resource Compiler (RMaker).

Tab 6: The **Toolbox** includes chapters on the Toolbox Interface, the Event Manager, and the Desk Manager.

After the Toolbox section, you will find a Document Report reply card and a Software Problem Report reply card.

In addition to the written documentation, there may be one or more README files on the distribution disks. When present, these contain additional information developed after the written documentation went to press.

Contents

Reference

Welcome to the World of

Microsoft FORTRAN

for the Apple Macintosh!

The Microsoft FORTRAN Compiler for the Apple Macintosh is a full implementation of FORTRAN 77. It has all the language features of mainframe FORTRAN, plus a powerful interface to the Apple Macintosh's operating system and graphics capabilities.

This introduction helps you get started with this powerful programming language. We go step by step through writing your first short Microsoft FORTRAN program.

This manual presumes familiarity with the FORTRAN language itself, and with the general operation of the Macintosh.

Getting Started

Microsoft FORTRAN comes on two disks - a **compiler disk** and a **utilities disk**. We recommend that you immediately slide open the copy-protect windows in the corners of the disks, if you have not already done so. This prevents you from accidentally modifying your original disks. Make a copy of each of these master disks. Store the original disks in a safe place. Later on, if anything should happen to your working copies, you can make a new copy from the original disks.

Your Microsoft FORTRAN disks contain many files used to create and manage Macintosh applications. These take up almost all of the space on the disks. You may want to create disks with fewer files for everyday development work. This section describes the function of each file so that you can decide which ones you need for a given application.

Do not be alarmed if your disk does not appear exactly as shown in the figures below, or if disk space available or other items differ slightly. This is because of improvements made after the documentation went to press.

The MS FORTRAN Compiler Disk

Figure 1, "Microsoft FORTRAN Compiler Disk," shows the contents of the first Microsoft FORTRAN disk. This disk contains MS FORTRAN, debug, f77 overlays, Edit, and a System folder.

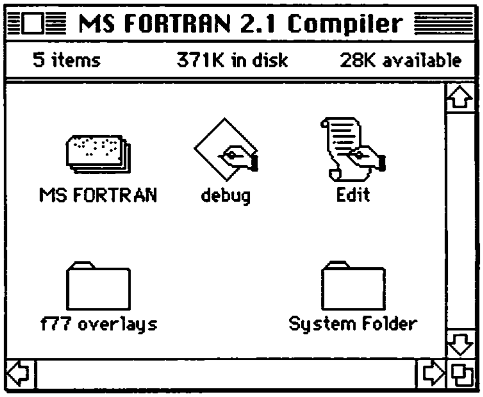


Figure 1. Microsoft FORTRAN Compiler Disk

The **MS FORTRAN Compiler** disk carries the main items that you need to get started - a text editor and the Microsoft FORTRAN Compiler. If you have more than one disk drive, or if you are working with small source files, the organization of this disk may be just right for your FORTRAN program development activities.

Below, we recommend a disk organization for FORTRAN development on a single-drive system, if your programs get too big to fit within the disk space available.

Let's look at the items on the **MS FORTRAN Compiler** disk.

MS FORTRAN, represented by a card deck icon, is the Microsoft FORTRAN Compiler. This application is used to compile a FORTRAN program or subroutine. It is not adequate in itself to compile a program. It requires several support files which are described below. The required support files can be found in the **f77 overlays** folder.

The Microsoft FORTRAN source code debugger is invoked with the application named **debug**, represented by the standard Macintosh icon for application files (a hand writing on a piece of paper). As with Microsoft FORTRAN, this application cannot run by itself. It requires **debug.fc**, a support file described below.

F77 overlays (FORTRAN 77 overlays) contains object code files used to support various portions of the Microsoft FORTRAN system. It includes five compiler overlays, the debugger overlay, and the runtime library. When one of these files is required, it must be available either on the same disk as the application needing it or on the internal drive.

The compiler overlays are **f77.fc**, **f77001.fc**, **f77002.fc**, **f77003.fc**, and **f77004.fc**. These files must be present whenever you compile a program or subroutine.

The debugger overlay is **debug.fc**. It is read in as needed by the debug application. It must be available whenever you debug a program.

The runtime library is named **f77.rl**. This file must be present whenever you run a program that you have compiled with Microsoft FORTRAN. It may also be linked into your application using the Microsoft FORTRAN linker. See the discussion of the **link** program for more information.

Edit, represented by a hand writing on a scroll, is a text editor. It is used to create and edit FORTRAN source code (text) files. It is supplied by Apple, and has no special relationship to Microsoft FORTRAN.

The **System Folder** contains **System** and **Finder**, and possibly other Macintosh system files. They are supplied by Apple, and have no special relationship to Microsoft FORTRAN.

Single Drive Operation

Working on a system with only a single disk drive, the initial organization leaves only enough space for a few small source files. If you need more disk space, you can organize your disks as shown below. By dividing the program edit function from the compile and debug functions, you can work with much larger programs. However, this organization requires you to copy your program file each time you go through the edit-compile cycle.

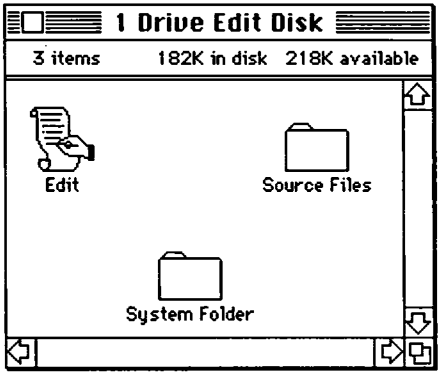


Figure 2. Edit Disk for 1 Drive System

Figure 2, "**Edit Disk for 1 Drive System,**" shows our recommended layout for your edit disk. This allows much more disk space for program source code. Keep the master copy of your program source on this disk. Edit your programs here. When you are ready to compile and test your program, copy the relevant source files onto the compile disk, shown in Figure 3. This helps avoid running out of disk space during the edit and compile steps of program development. As a side benefit, during program testing your master copy of the source is safely outside of the computer.

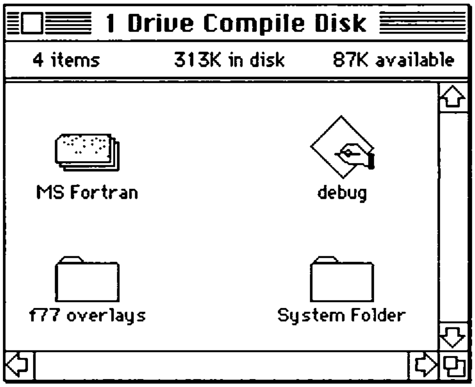


Figure 3. Compile Disk for 1 Drive System

Figure 3, "**Compile Disk for 1 Drive System,**" is the MS FORTRAN Compiler disk with the Edit application removed. This gives more disk space in which to put the files related to your current program.

The Microsoft FORTRAN Utilities Disk

The **MS FORTRAN Utilities Disk** is the second of the two distribution disks. It contains **link**, **lib**, **RMaker**, and folders of **subroutines**, **include files**, and **source files**.

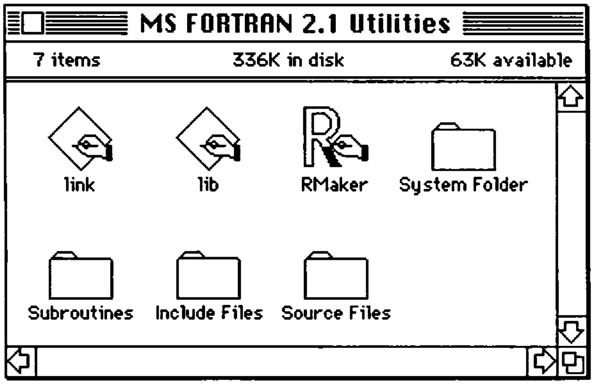


Figure 4. Microsoft FORTRAN Utilities Disk

Link allows you to combine program modules into fewer entities. You may not need to learn its use right away, because Microsoft FORTRAN dynamically links subroutines as needed. Static linking improves execution speed and simplifies installation of your application by your users. **Link** is written in FORTRAN, and requires the run time library **f77.rl** to execute.

Lib works with **link** by organizing your subroutines into libraries. When you link your program, you can specify one or more program libraries. These are searched by the linker to find subroutines used by your application. Only those subroutines needed will be combined into your finished product. **Lib** is written in FORTRAN, and requires the run time library **f77.rl** to execute.

The application **RMaker** (resource maker), represented by a hand painting a large "R", is used to add resources to your Microsoft FORTRAN application file after it has been compiled and linked. It helps you to create custom icons for your applications, and to take advantage of other Macintosh features. Resources and the use of RMaker in creating them are described in the **RMaker** manual. **RMaker** is supplied by Apple. To get the most out of **RMaker**, you may want to read Apple's document Inside Macintosh.

Subroutines

The **subroutines** folder contains compiled external utility subroutines (and in most cases their source code) to support FORTRAN programs you write. These files are not needed to run the Microsoft FORTRAN system unless you write a program which calls them. Only those subroutine files actually used need to be available, and they must be available on the same disk drive as the application calling them. They can also be linked into your application with the Microsoft FORTRAN linker (see the section on LINK). The function of each subroutine is described elsewhere. Three of the subroutines deserve special mention, however: **errmsg.sub**, **spool.sub**, and **toolbx.sub**.

You cannot call **errmsg.sub** directly. It is called by the runtime system whenever your Microsoft FORTRAN application encounters an error during execution. If this subroutine is not present, an error number will be reported; if it is present, a descriptive message will be reported instead. This subroutine does not handle the reporting of compile time errors, and its availability does not affect the appearance of messages output by the compiler.

You can call **spool.sub** directly to print a text file on an Imagewriter printer. However, it is also called by the runtime system whenever a program which writes to unit 6 (preconnected to the printer) is run. The run time system writes such output to a temporary file. When the program ends, **spool.sub** is called to print and then delete the file. For this reason, you cannot print while a program is executing. Also, if **spool.sub** is not available when the program ends, no printing occurs and the temporary file is left on the disk. **The source code for spool.sub can be modified to allow direct printing without spooling**, and contains a great many comments discussing the hardware interface with the printer port. Serious programmers desiring to utilize the full capabilities of the Macintosh will be well repaid by careful examination of this and the other sample FORTRAN and assembly language code provided.

Toolbx.sub can only be called directly. It is your interface to the Macintosh Toolbox ROM routines. It must be present if you are going to generate graphics or use any of the Macintosh interface features, such as multiple windows, menus, or dialogs. The current implementation of this interface subroutine can only call Toolbox routines. There is another type of routine in the Macintosh ROM: the Operating System traps. These include low level file and device I/O routines, some Event Manager routines, Memory Manager routines, and various utilities. If you need these functions, you must write your own assembly language interfaces for them (see Appendix E on calling assembly language from Microsoft FORTRAN).

Include Files

The **include files** folder contains items to shorten your coding time and improve the reliability of your finished product. Some of these are used by the sample source programs provided in the **source files** folder. Many are subsets of **toolbx.par**, split out to improve compile speed by reducing the number of symbols included in your program.

Toolbx.par is a file of constant definitions which simplifies the use of **toolbx.sub**, the interface to the Macintosh Toolbox ROM routines. **Toolbx.sub** takes as its first argument an integer which specifies the ROM routine that you want to call. These integer values are rather arbitrary and difficult to remember, so a set of constant definitions, in the form of FORTRAN parameter statements, has been provided in **toolbx.par**. It can be included in your program (after the program statement) or subprogram (after the subroutine or function statement) with the Microsoft FORTRAN include statement:

```
      program  myprog
      include  toolbx.par
```

or

```
      subroutine  mysub(param)
      include  toolbx.par
```

or

```
      function  myfunc(param)
      include  toolbx.par
```

Thereafter, the first argument to **toolbx.sub** can be a symbolic name. Some of these names are given in the Toolbox documentation in the Microsoft FORTRAN literature; the rest can be found in the Apple document Inside Macintosh. **Toolbx.par** must be included in every program unit (program or subprogram) that uses these symbolic names.

It should be noted that, although the use of **toolbx.par** does not cost either space or time in the compiled application, it does take up memory during compilation. This means that some large programs may not compile with **toolbx.par** included which would compile without it. There are several remedies to this situation.

One method is to get the values of the required Toolbox names and use them directly in calls to **toolbx.sub**. This allows you to leave out **toolbx.par**, and takes the least compile time memory of all. For example, the call

```
    call  toolbx(GETNEXTEVENT,  eventmask,  myevent)
```

can be written as

```
    call toolbx(357, eventmask, myevent)     ! GETNEXTEVENT
```

Another method is exemplified by the program **demo.for** (included in source on your release disk). Those parts of **toolbx.par** which were required for this program were abstracted into the file **demo.inc**, greatly reducing the number of definitions in the program unit and therefore reducing the amount of memory needed during compilation.

Source Files

The **source files** folder contains sample programs, benchmarks, and demos, to highlight the capabilities of Microsoft FORTRAN and allow you to learn by example. Also included is **init.asm**, the main program startup code for programs generated by Microsoft FORTRAN. This is included both for the benefit of those programmers who require this specialized and detailed knowledge, and for use by the compiler: it is automatically **include**'d into the applications you write using Microsoft FORTRAN.

The FORTRAN programs demonstrate some of the capabilities of Microsoft FORTRAN. The subroutine source files are for documentation purposes only. All except **rad50.for** are assembly language files. They were not assembled on a Macintosh, and are not guaranteed to be source compatible with Apple's 68000 assembler for the Macintosh. They are included for those Microsoft FORTRAN users who need to understand their function in detail.

The FORTRAN programs included demonstrate various features of Microsoft FORTRAN. You may find it useful to try compiling one of these programs before writing any of your own to familiarize yourself with the operation of the compiler. A brief description of each of these programs follows.

bases.for - This program does base conversions using the Microsoft FORTRAN decimal, binary, octal, and hexadecimal format descriptors.

byte.for - This is a Sieve of Eratosthenes prime number benchmark taken from Byte Magazine.

demo.for - This program demonstrates the use of some of the Macintosh interface features using **toolbx.sub**. It uses the include file **demo.inc**, an abstract of definitions from **toolbx.par**. This file must be present to compile **demo.for**. Some of the interface features demonstrated are multiple windows, the use of FORTRAN I/O in Macintosh windows, menus, and graphics.

err.for - This program demonstrates program control of floating point error conditions as described in the FORTRAN reference manual.

pi.for - This program calculates the value of pi using the binomial theorem.

type.for - This program types a text file out to the screen, one page at a time.

Additional sample programs may be included on your disk. These are developed as examples in response to technical questions asked by other programmers like yourself. They are provided for your benefit, in case your current or future needs require the special information that they demonstrate. Look for in-line comments within the source files. These will explain the purpose and operation of the sample code.

Writing a Sample Program

This discussion assumes you have only one disk drive. You will edit, compile, and run a simple program.

First, make a copy of the compiler disk. Turn off your Macintosh. Turn it back on. Slide OPEN the disk write protection window of the compiler disk you received. This protects you against accidentally damaging the files on the disk, and ensures that you can always start over. Insert the **MS FORTRAN Compiler** disk into the internal drive of the Macintosh. Your Macintosh should "smile" at you, and eventually present you with a Macintosh desktop.

Eject the compiler disk now. You can do this by holding down the command key (the "cloverleaf" key beside the space bar) and pressing the "E" key. (This is called pressing **"command-E"**.) The disk should be ejected.

Insert a disk on which you will place a copy of the compiler disk. If the new disk has never been used before, you will be asked whether to initialize it. If so, respond to initialize. If the new disk has been used before, and has not already been erased, erase it now.

Next, select the MS FORTRAN Compiler disk by moving the mouse until it is centered on the disk icon. Hold down the mouse button and drag the outline of the compiler disk on top of the target disk. When the target disk becomes highlighted, release the mouse button.

You may be presented with a Macintosh dialog box, asking whether you wish to completely replace the contents of the target disk with the contents of "MS FORTRAN Compiler."

Respond OK by clicking the mouse on that selection within the dialog box.

On a 128K Macintosh, it takes about three minutes (and about twenty disk swaps) to complete the copy.

Now pull down the **Special** menu and select the **Shut Down** option. It may ask you to reinsert your copy disk before it completes the shut down process. Put your original compiler disk in a safe place. Use it only to make new copies.

After the shut down, the Macintosh looks as though you just turned it on. Insert your newly created copy of the compiler disk into the internal drive. It should initialize the Macintosh just as your original compiler disk did.

Open the disk icon. You should see the desk items just as they appear in Figure 1. Start the editor. You can do this by double-clicking the mouse on the Edit icon.

Pull down the **File** menu and select **New**. The editor opens an edit window, ready for you to enter your program.

Press the **Tab** key, then type the following program:

```
    write(9) "Hello"
    pause
    end
```

Now pull down the **File** menu and select the **Save As ...** option. When it asks you for a document name, key in **prog1**. Click on the **save** button. You should hear the disk whir as the file is saved. Then the dialog box disappears and the edit window shows the name **prog1** across the top.

Pull down the **File** menu and select **Quit**. You are returned to the Macintosh desktop. Two new items appear next to the six that were initially there. One is the **DeskScrap**. The other is **prog1**, your source program file. These take up about 3K bytes.

Double-click on the MS FORTRAN icon. A new menu bar appears, with an apple and the words **File**, **Compile**, and **Transfer**. Press **command-f**. This has the same effect as pulling down the **File** menu and selecting **Select File**. A file selection window opens. **Prog1** is the only item displayed. Double-click on **prog1**, or click once on the **open** button. **Prog1** is selected, and the dialog box disappears.

Pull down the **Compile** menu and select **Compile**, or press **command-c**. A compiler window opens, and the following is displayed.

```
1: Symbol table complete
     Memory usage:
       Labels        480 bytes
       Symbols      2040 bytes
       Total       50646 bytes
       Excess      22820 bytes
       Source          3 lines
2: Object file complete
3: Program file complete: 332 bytes
   Elapsed time: 0:24 =   7 lines/minute
```

Your numbers may differ from those shown above.

Pull down the **Transfer** menu and select **Select Application**. A window opens, presenting four applications for you to choose from. They are **debug**, **Edit**, **MS FORTRAN**, and **progl apl**. Click on **progl apl** and then on **open**, or simply double-click on **progl apl**.

After a moment, a window opens and the word **Hello** appears. Your program has executed its write statement, and is now waiting for you to give it input to satisfy its pause statement. Press **Return**. Your program continues from its pause, and then ends. You are returned to the Macintosh desktop.

By dragging the disk directory window farther open, you see one new file on the disk: **progl apl**. It takes about 1K of disk space.

Additional Notes

The following notes address issues that seem to deserve additional attention.

Sample Code Illustrates Features

The source code and include files present on the utilities disk, **MS FORTRAN Utilities**, illustrate features of Microsoft FORTRAN 77. These file are provided to give you examples to work from as you develop your own applications. You can find in them many notations explaining their operation.

A Note to Serious Programmers. It is impossible to emphasize this point too much, and so it is repeated here: Serious programmers will find the answers to many or all of their technical questions as they read through the commented source files provided. This material is not presented as part of the manual because of the degree of detail and complexity involved.

For "vanilla" FORTRAN, this information is not needed. But for full utilization of the Macintosh, this information encapsules the answers to many challenging interface questions.

Final Pause

If the Macintosh desktop appears too quickly after your program ends, consider placing a pause statement in your program just before the final exit. This lets you view the screen at your leisure, but requires you to enter a keystroke before the program can terminate.

The * Unit

The default for the * unit is unit 5 for input and unit 6 for output. On the Macintosh, it is likely that most input and output will be directly through the keyboard and the screen. If this is your situation, use the **compiler options** to redirect the * unit to be unit 9.

Units 5 and 7

Microsoft FORTRAN 77 is designed to support a card reader and magnetic tape. Normally, unit 5 is pre-opened to the card reader, or simulated card reader, and unit 7 is pre-opened to the tape drive. However, these peripherals are not supported by the Macintosh. You may open these unit numbers and use them for normal file access. However, if you use these unit numbers without opening them, you will get unpredictable results. This is in contrast to most other unit numbers, which are not pre-opened to any particular device, and which report an error if access is attempted without first opening them.

Downloading Source Files To The Macintosh

Source files can be downloaded to the Macintosh, but beware of this problem. Macintosh uses a carriage return to separate lines of input. Many computers use a carriage return followed by a line feed to separate lines of input. The compiler accepts either form. However, the debugger does not handle line feeds properly when they appear in the source file.

Debugging Applications That Use Windowing

The debugger uses windowing in the same way that application programs might. Without multitasking capabilities, extra caution should be used during debugging to ensure that window operations apply to the application's windows, and not to the windows opened by the debugger. For example, a windowing application might routinely close the output window provided by FORTRAN before opening other windows. Under the debugger, the window close operation could accidentally terminate the debugger itself, and result in a system error.

End-Of-Record Control

ANSI standard FORTRAN specifies that the WRITE statement generate an end-of-record indicator for each write statement. In some cases, this causes a great burden on programmers, and a number of language extensions are in use by various FORTRAN language vendors to suppress end-of-record characters in program output. The method used by Microsoft FORTRAN for the Apple Macintosh is to have two separate statement verbs: **WRITE** and **TYPE**. The WRITE verb will always add end-of-record control. The TYPE verb will never add end-of-record control. With the TYPE verb, what you send is exactly what goes out - no more, and no less. You are totally responsible for adding your own control characters, including line feeds, form feeds, carriage returns, etc.

README Files

If present on your distribution disks, README files highlight information that was not available in time to be included in the printed manual. Be sure to check for them.

User's Guide

User's Guide

Microsoft® FORTRAN Compiler
for the Apple® Macintosh™

User's Guide

CHAPTER 1

INTRODUCTION

1.1 FORTRAN 77

The FORTRAN programming language, a contraction of the words FORmula TRANslation, is a computational problem solving language. Because it resembles familiar arithmetical language, it greatly simplifies the preparation of problems for machine computation. Data and instructions are organized in a sequence of FORTRAN statements. This sequence of statements is referred to as the source program. A program written in the FORTRAN language can be processed on any machine which has a FORTRAN compiler with little or no modifications to the source program. In this sense, the FORTRAN language is said to be machine independent.

1.1.1 History of the Language

The FORTRAN language, first implemented in 1956, was designed for the solution of mathematical problems. It was released for customer use by IBM Corporation in 1957. In the years between 1957 and 1966, the language underwent several changes and was standardized in 1966 by the American National Standards Institute (ANSI). This 1966 standard, technically known as FORTRAN 66, was the 4th version of FORTRAN, hence the name FORTRAN IV. In the years leading up to 1977 the language evolved and extensions were added to compilers at various computer installations. In 1977 the language again underwent the standardization procedure and on April 3, 1978 the American National Standard Programming Language FORTRAN, known as FORTRAN 77, was established. As of the writing of this document, this is the latest published standard for the FORTRAN language and is the basis of the implementation provided by Microsoft Corporation.

The FORTRAN language, as originally implemented, was designed for use with punched cards. The punched card in the mid to late 1950's was the standard method of loading programs into computers. Later versions of the language have retained the original formatting rules even though the use of punched cards for source program preparation has all but vanished.

1.1.2 MICROSOFT FORTRAN 77

Microsoft Corporation's FORTRAN 77 is a complete implementation of the 1977 ANSI version of the FORTRAN language. In addition, many useful extensions have been added including many of those which will be incorporated in the next standard published by ANSI. The compiler operates in three passes, with each pass consisting of a separate overlay, allowing it to operate in a minimum amount of memory.

Microsoft FORTRAN provides all of the file I/O facilities required by the ANSI standard resulting in the capability of a FORTRAN program to execute with little or no knowledge of the file I/O conventions of a particular operating system.

In addition, standard implicit input and output units are provided allowing a FORTRAN 77 program to input from a data file and output to a line printer without opening, closing, or naming a file. Even the actual unit number that the file is internally connected to never needs to be referenced. The following program will copy its input to its output on standard conforming implementations of FORTRAN:

```
      PROGRAM COPY
      CHARACTER*80 TEXT
1     READ (*,10,END=2) TEXT
      PRINT *,TEXT
      GOTO 1
2     STOP
10    FORMAT (A80)
      END
```

using the structured facilities of Microsoft FORTRAN 77, the same program could be written as:

```
      program copy

      character*80 text
      integer eof

        do
          read (*,'(a)',iostat=eof) text
          if (eof) exit
          print *,text
        repeat

      end
```

1.2 USING THIS DOCUMENT

This document was designed as a reference manual for the use of the Microsoft implementation of FORTRAN 77. As such it is not a tutorial on FORTRAN. The bibliography in Appendix C of this manual lists several books which can be helpful in learning to program with FORTRAN.

The second chapter describes the use of the compiler and the many options that may be selected. A careful reading of this chapter is strongly advised as most questions we are asked can be answered by referring to this chapter. Chapter 3 describes one of the more powerful features of Microsoft FORTRAN 77, DEBUG, the symbolic debugger. The remaining chapters describe the syntax and structure of FORTRAN 77. Contained in the appendices is information on writing assembler procedures for FORTRAN, using the instrinsic function library, and restrictions on the implementation.

1.2.1 CONVENTIONS USED IN THIS DOCUMENT

The terms "computer", "monitor", and "environment" refer in general to the operating system under which the compiler is implemented. Certain restrictions, enchancements, and features may or may not apply due to a particular operating system. Refer to the implementation notes supplied with the compiler for further information.

The term "processor" refers in context both to the compiler and to the execution environment. In particular, the compiler will choose certain instruction sequences and execution options which are appropriate to the hardware machine environment.

1. Unless otherwise indicated, all numbers are in decimal form.

2. [] square brackets indicate that a syntactical item is optional

3. ^c a circumflex followed by a character indicates that an ASCII control character is required. This non-printing character is usually generated by pressing a special key on the terminal simultaneously with the required character. The special key is generally labeled CONTROL, CTRL, or ALT.

4. ... indicates a repetition of a syntactic element

5. RETURN specifies generating the ASCII value 13 from the terminal. This is normally accomplished by pressing the key labeled RETURN, RET, or ENTER.

6. ESC indicates the ASCII escape character (27) which is usually labeled on a terminal keyboard as ESC, ESCAPE, or LEADIN.

7. RAD50 refers to a method of packing three upper case ASCII letters or digits into two bytes of storage, providing an efficient method of compression. Appendix G describes the RAD50 algorithm.

8. The terms relative address and location when used in referencing executable program files and object modules refer to a memory location whose address is specified relative to the base of the program.

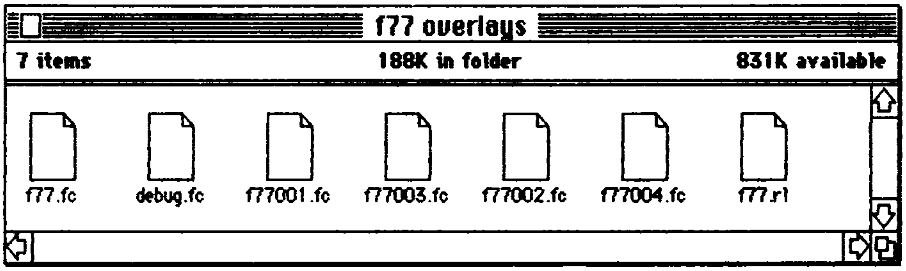
CHAPTER 2

THE MICROSOFT FORTRAN COMPILER

The Microsoft FORTRAN compiler is designed to provide mainframe FORTRAN facilities to the mini and micro computer programmer. A large number of options and debugging tools are available, providing a great deal of flexibility to the programmer who is designing and installing applications and systems software. The compiler can generate completely or partially linked executable object modules or assembler source code. All code produced by the compiler and all library routines are relocatable, position independent, and reentrant. The compiler itself is position independent, but not reentrant nor serially reusable.

2.1 COMPILER INVOCATION

Before attempting to use the Microsoft FORTRAN Compiler system, we suggest making a working copy of all items provided on the distribution diskettes. This includes the Microsoft FORTRAN Compiler itself (represented by a card deck icon), the interactive debugger, Debug, and the F77 Overlays folder, whose contents are represented by ordinary document icons.

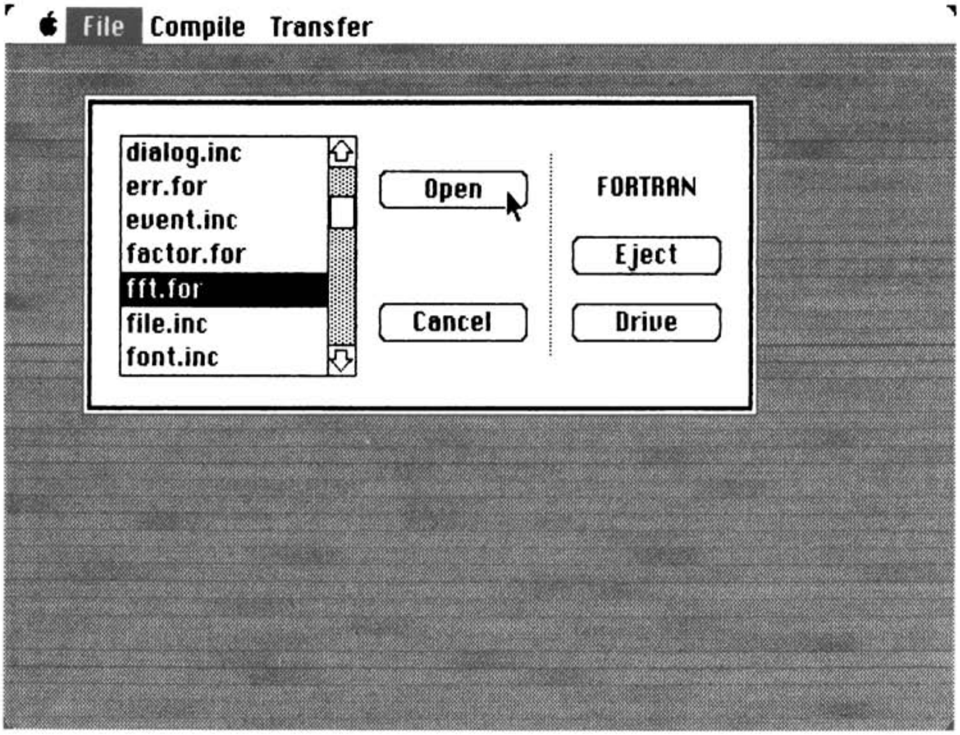


To use the Microsoft FORTRAN Compiler, a source file is first prepared using Edit or some other text editor. If MacWrite is used, care must be taken to save the file as text only before attempting to compile it. MacWrite adds text formatting information which the compiler does not understand.

Source files moved to the Macintosh from other computers using one of the many download utilities can also be compiled by the Microsoft FORTRAN Compiler. It is a good idea to make sure that there are no line feed characters (ASCII 10) in the file. Macintosh text files use only the carriage return character (ASCII 13) to terminate lines. The Microsoft FORTRAN Compiler will compile files with line feeds in them, treating them as blank lines. The Microsoft FORTRAN Compiler debugger also uses the source file, however, and it cannot tolerate line feeds. Such a file cannot be debugged.

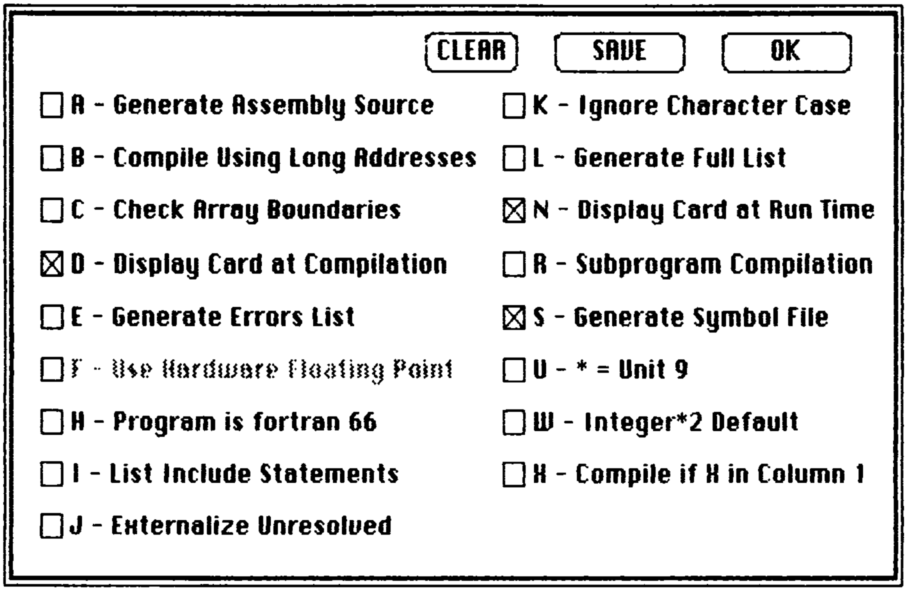
The compiler itself is invoked by opening the card deck icon provided on the Microsoft FORTRAN Compiler diskette. After the icon has been opened, a blank desktop with five menus in the menu bar will be displayed.

To specify the file to be compiled, choose the "Select File" option under the "File" menu. This will display the standard file selection window which you may be familiar with from MacWrite or MacPaint.



Only files with the "TEXT" attribute, represented in the Finder by a document displaying uneven text, will be displayed in the standard file dialog's list. Click on one of these with the mouse, and then click the "OPEN" button. The dialog window will disappear.

At this point, several selections under the menus which were previously displayed in grey and therefore unavailable become black, including all of the selections under the "Compile" menu. There are several options affecting the behavior of the compiler which are specified by choosing "Options" under the "Compile" menu. A dialog window with check boxes for each option will be displayed.

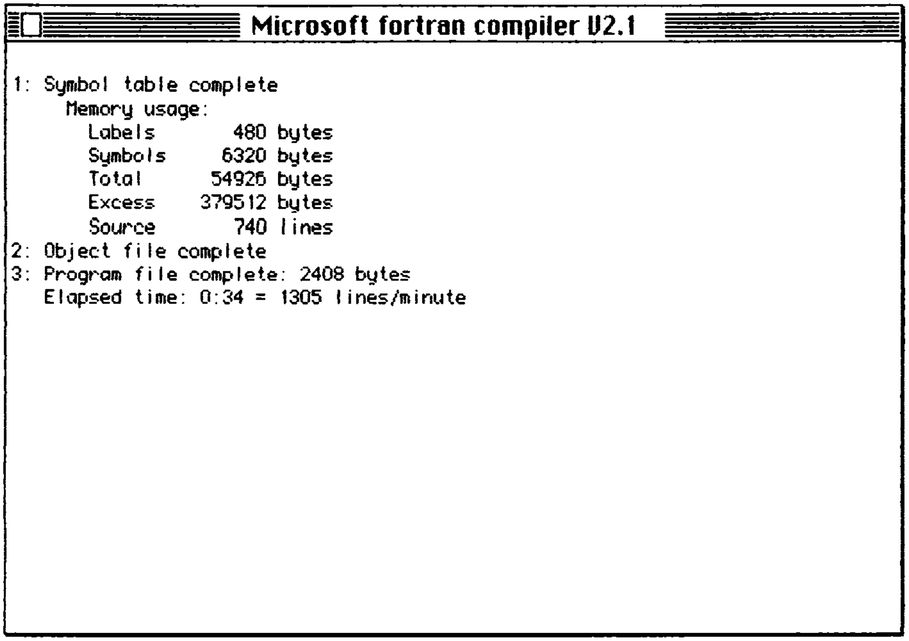


☐ A - Generate Assembly Source	☐ K - Ignore Character Case
☐ B - Compile Using Long Addresses	☐ L - Generate Full List
☐ C - Check Array Boundaries	☒ N - Display Card at Run Time
☒ D - Display Card at Compilation	☐ R - Subprogram Compilation
☐ E - Generate Errors List	☒ S - Generate Symbol File
☐ F - Use Hardware Floating Point	☐ U - * = Unit 9
☐ H - Program is fortran 66	☐ W - Integer*2 Default
☐ I - List Include Statements	☐ X - Compile if X in Column 1
☐ J - Externalize Unresolved	

Each option can be turned on or off by clicking in the corresponding check box or on the option title itself, or by entering the corresponding letter at the keyboard. An "X" in the box next to an option indicates that it is on. Options remain in effect until you leave the Microsoft FORTRAN Compiler system. The effect of each option is described in Section 2.2 of this chapter (COMPILE TIME OPTIONS).

After the options have been set, the file can be compiled by choosing either "Compile", "Compile and Execute", or "Compile and Debug" under the "Compile" menu. The only difference between these three options is the action taken after a successful compile. "Compile" returns you to the Microsoft FORTRAN Compiler environment, allowing you to compile other files. "Compile and Execute" launches the application produced by the compiler, just as if you had opened it from the Finder. When the application is complete, it will return to the Finder desktop. "Compile and Debug" invokes the interactive debugger on the new application.

After the compile option is chosen, the compiler window will appear. This window is essentially a simulated teletype and cannot be sized or moved. It does not update itself after being covered by other windows except by erasing its contents and placing the cursor at its upper left hand corner. Through this window, the compiler reports information to you regarding its progress.



```
1: Symbol table complete
     Memory usage:
       Labels       480 bytes
       Symbols     6320 bytes
       Total      54926 bytes
       Excess    379512 bytes
       Source       740 lines
2: Object file complete
3: Program file complete: 2408 bytes
   Elapsed time: 0:34 = 1305 lines/minute
```

This information can take varying formats depending on what options are selected. The following discussion summarizes the operation of the compiler without any options selected. The variations are described with the discussion of the options in the next section (2.1.1).

2.1.1 Compiler Passes

After performing its initialization, the compiler begins its first pass which involves syntactical analysis, developing symbol tables, and generating an internal intermediate code. At the beginning of this pass, the compiler displays:

```
        1:
```

At the end of this pass the compiler display will appear as follows:

```
        1: Symbol table complete
                Memory usage:
                  Labels   nnnnnnnn bytes
                  Symbols  nnnnnnnn bytes
                  Total    nnnnnnnn bytes
                  Excess   nnnnnnnn bytes
                  Source   nnnnnnnn lines
        2:
```

where nnnnnnnn is a decimal value. "Total" represents the actual amount of memory required by the compiler including its own size, tables, and work stacks.

The compiler then proceeds with the second pass during which run time memory allocation requirements are established, machine object code is generated, and most backward address references are resolved. At the end of this pass, the display will appear as:

```
        1: Symbol table complete
                Memory usage:
                  Labels   nnnnnnnn bytes
                  Symbols  nnnnnnnn bytes
                  Total    nnnnnnnn bytes
                  Excess   nnnnnnnn bytes
                  Source   nnnnnnnn lines
        2: Object file complete
        3:
```

During the final pass, the compiler resolves all remaining address references and installs certain numeric constants which were not established in line. The final compiler display will be:

```
1: Symbol table complete
        Memory usage:
          Labels  nnnnnnnn bytes
          Symbols nnnnnnnn bytes
          Total   nnnnnnnn bytes
          Excess  nnnnnnnn bytes
          Source  nnnnnnnn lines
2: Object file complete
3: Program file complete: nnnnnn bytes
   Elapsed time: m:ss = nnnn lines/minute
```

where nnnnnn is the size of the executable object module.

Because the compiler is disk based, there is a certain constant amount of overhead in file management. The compilation of small programs will reflect this overhead in a relatively low lines per minute statistic. Larger programs will tend to mask this overhead and produce a far greater statistic.

2.1.2 Compiler Error Reports

If syntactical, grammatical, or structural errors were encountered during the first pass, the display will appear as follows:

```
        1: Symbol table complete - 3 errors detected
                Memory usage:
                  Labels  nnnnnnnn bytes
                  Symbols nnnnnnnn bytes
                  Total   nnnnnnnn bytes
                  Excess  nnnnnnnn bytes
                  Source  nnnnnnnn lines
        2: Bypassing
        3: Error report:
```

If control, context, or labeling errors were encountered during the second pass, the compiler will display:

```
        1: Symbol table complete
                Memory usage:
                  Labels  nnnnnnnn bytes
                  Symbols nnnnnnnn bytes
                  Total   nnnnnnnn bytes
                  Excess  nnnnnnnn bytes
                  Source  nnnnnnnn lines
        2: Object file complete - 2 errors detected
        3: Error report:
```

The "Error report" lists the first statement line of the statement in which the error occurred, the line number of the statement, the six characters preceding the offending code, and a message describing the type of error. Every practical effort has been made to avoid meaningless "Syntax error" messages and provide the user with informative descriptions. Consult Appendix D of this manual for a complete list of compile time error messages.

If there are enough error messages, the compiler window will begin to scroll. Error messages may be scrolled out of the window before you can read them. To temporarily stop the printing of error messages, enter Command-S from the keyboard (the Command key and the "S" key simultaneously); to continue, enter Command-Q.

2.1.3 File Names

As mentioned above, a file must have the "TEXT" attribute to be compiled by the Microsoft FORTRAN Compiler. The name of this file can be any legal Macintosh name except that it is limited to 32 characters (not 64 as for the Finder).

To maintain compatibility with other systems, file name extensions on the source file name will be processed if they exist. An extension immediately follows the file name proper and consists of a period followed by one to three characters. The compiler output files have the same name as the input source file with different extensions. If no extension was provided with the source file, the output name extensions are simply added to the input name. If the resulting names would be greater than 32 characters, the input name is shortened before adding the output extension. If an input extension is provided (".for" by convention), it is removed before the output extension is appended. The output extensions are:

1. " apl" - an application created by the compiler. This extension starts with a space, not a period.

2. ".sym" - a Debug symbol file.

3. ".lst" - a compiler list file.

4. ".sub" - an unlinked subroutine file (loaded from the disk when called at run time).

2.1.4 Compiler Output

The principal output of the compiler is a Macintosh application which appears on the Finder desktop as a hand writing on a blank page. This is the default icon for applications. Opening this application will start your program, just as with any other application. As mentioned below, the compiler will optionally produce listing and debug symbol files. These both appear as document files with no attributes, appearing on the Finder desktop as blank page icons. The compiler will also produce subroutine and function files which can be loaded at run time. These appear on the Finder desktop as document files also; they cannot be run by themselves.

Occasionally, other blank page icons with the name of your source code file appended with .fl, .f2, or similar extensions will appear if the compiler failed to run to completion. These are intermediate files used only by the compiler. They can be put in the trash, or the compiler will remove them the next time you compile the same file.

2.1.5 The Transfer Menu

This menu is provided as a convenience for moving easily and quickly from the Microsoft FORTRAN Compiler to other Macintosh applications. The most common application you will want to transfer to is Edit (when the compiler reports an error), so it is listed on the menu for direct selection. You may, however, transfer to any application by clicking the "Select Application" menu item which will display a standard file dialog box.

2.2 COMPILE TIME OPTIONS

2.2.1 The INCLUDE statement

This statement is a compiler directive and is provided as a convenience for copying standard declaration statements, subroutine libraries, and documentation sections directly into a source file at compile time. The syntax of this statement is:

```
        INCLUDE filespec[/L]
```

where: *filespec* is a standard system file specification

/L indicates that the included file is to be listed with the compiler listing

The listing of INCLUDE file statements is normally suppressed, however they may be selectively included in the compiler listing by appending /L to the file specification or using the *I* option switch (see below) to force the listing of all included files.

An included file may not contain an INCLUDE statement. That is, INCLUDE statements may not be nested.

2.2.2 The PAGE statement

The PAGE statement is a compiler directive to force the compiler to insert a form feed in the list file. The PAGE statement itself is removed and replaced by the form feed character.

2.2.3 Generating Assembler Source

The compiler normally generates executable machine code and contains branch shortening logic. Unresolved procedure references will be treated as overlays unless linked with the Microsoft linker. There may be situations where it is desirable to have more control over the object code (hand optimization, additional assembler code, etc.). In these instances, the compiler can generate an assembler source file by selecting the A option.

The assembler source code will have the first line of each FORTRAN statement interleaved as commentary. The compiler assumes that the local assembler has no facilities whatsoever. There are no macro definitions nor symbolic assignments; branches are not optimized and every opcode is explicitly stated.

When this option is selected, the first pass proceeds normally, however a special second pass is called which is also the last pass:

```
1: Symbol table complete
        Memory usage:
          Labels  nnnnnnnn bytes
          Symbols nnnnnnnn bytes
          Total   nnnnnnnn bytes
          Excess  nnnnnnnn bytes
          Source  nnnnnnnn lines
2: Source file complete
3: Elapsed time: m:ss = nnnn lines/minute
```

2.2.4 Long Addressing Mode

In order to generate position independent code, the compiler selects program counter relative and base register addressing modes during code generation. Using these modes, certain instructions are limited to a signed 32K byte displacement and cannot reach the entire address space. Some FORTRAN statements can generate machine instructions that exceed this limit: GOTO, DO, ASSIGN and label references. When this occurs, a diagnostic is issued during the last phase of compilation and the program must be recompiled using the B option. Your program will be larger, but will still be position independent.

2.2.5 Array Boundary Error Checking Code

The C option causes the compiler to generate code for validating an array index. The code determines if an individual array element falls within the lower and upper boundaries of the array; individual dimension subscripts are not tested.

Only arrays whose dimension specifications were established with actual declarators have code generated for boundary checking. Adjustable arrays and arrays whose lower and upper bounds are equal to one do not have boundary checking code generated.

2.2.6 Card Number Display at Compile Time

Selecting the D option causes the compiler to display the card number of the statement it is processing during passes 1 and 2. During pass 3 it will display the symbolic name of the program unit it is linking.

The D option is primarily a compiler debugging tool: it indicates the card number of the statement where the compiler choked. However, many users swear that the compiler runs faster with card numbers flashing by.

2.2.7 Generating Inline Hardware Floating Point Code

Hardware support for floating point operations using peripheral devices such as the SKY FFP board or the National Semiconductor N32081 is provided on a threaded code basis in a special runtime library. There is a small amount of overhead involved in calling the routine and returning from it. Under certain circumstances this overhead can be significant and it may become desirable to eliminate the call and return instructions from the execution path. The F compiler option causes inline expansion of the hardware library procedures for single precision add, subtract, multiply, divide, and type conversion operations.

The inline expansion may be selectively controlled through the use of the OPTION statement (see below).

Note: There are currently no hardware floating point devices available for the Macintosh.

2.2.8 FORTRAN 66 compatibility

To a great extent, FORTRAN 77 is compatible with FORTRAN 66. There are, however, some areas where problems may arise in attempting to compile and run a FORTRAN 66 program without modification. These include FORTRAN 66 features no longer supported, resolution of ambiguities, and the inclusion of newer features in FORTRAN 77 that conflict with FORTRAN 66. The major issues are the Hollerith data type, DO loops, and dynamic storage allocation.

This implementation of FORTRAN 77 has been extended to support Hollerith constants in DATA statements, in the argument lists of CALL statements, and in the argument lists of external function references.

The following FORTRAN 66 features are invoked by specifying the _H_ compiler option:

1. Under certain conditions, FORTRAN 66 permitted branching into the range of a DO loop. This was known as "extended range of a DO" and is prohibited in FORTRAN 77. The compiler normally attempts to use machine registers for DO loop indexing: branching out of and then back into a DO loop may cause the index register to be altered yielding unpredictable results. The _H_ option will cause the compiler to use memory rather than registers for DO loop indexing providing for extended range DO loops.

2. 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 _H_ option will cause all DO loops to be executed at least once, regardless of the initial value of the iteration count.

3. 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 can become undefined with the execution of a RETURN statement. The SAVE statement is normally used to prevent this, but the _H_ compiler option will force all program storage to be treated as static. Chapter 4 fully discusses definition status and events that can cause entities to become undefined.

2.2.9 Generating External Symbol References

When generating an assembler source file, the compiler treats unresolved procedure references as overlays. The J option allows you to specify that unresolved references are to be treated as external references for your linker to resolve and causes the compiler to generate appropriate source code.

The J option implicitly sets the A option and can be selectively controlled with the OPTION statement (see below).

2.2.10 Compiler Listings

Three of the compiler options concern listings: L, I, and E. The listings generated by the compiler are useful for debugging and documentation purposes.

A compiler listing takes the following general form:

1. Page headers including source file name and directory, date, and page number.
2. Program unit name and the relative address (in octal) of its entry point.
3. Fifty-five lines of source text per page with each source line numbered, including comment lines.
4. Symbol and label tables.

The symbol table generated by the compiler reports the name, type, and size of variables, arrays, symbolic constants, and function procedures in each program unit. If the symbol is associated with a common block, the name of that common block is listed. In addition, a storage location relative to the base of the data area is given. The location takes the form of a relative address (in octal).

Undeclared symbols, symbolic names which did not appear in a specification statement, are noted with a plus sign (+) following the listing of their name. This is useful for locating possible spelling errors and checking on implicit typing.

The label table section of the listing includes the label and its relative address (in octal).

2.2.10.1 The L Compiler Listing Option

The L option invokes the full listing phase of the compiler. This is actually a separate pass. If any errors were encountered during compilation the error diagnostic will appear in the list file at the offending statement.

2.2.10.2 The I Compiler Listing Option

INCLUDE files normally are not incorporated into a compiler listing. Individual INCLUDE files can be added to the listing by appending "/L" to the file specification in the INCLUDE statement. For documentation purposes it may be useful to have a listing which reflects every source line encountered by the compiler. To force all INCLUDE files to be listed, select the I option.

The I option is recognized only when used with the L option.

2.2.10.3 The E Compiler Listing Option

The E compiler option causes error reporting to be directed to a file rather than the terminal. A file is not created if there were no errors encountered during compilation.

2.2.10.4 List Option Compiler Display

The compiler display, when a list option is specified, will appear as:

```
        1: Symbol table complete
                Memory usage:
                  Labels   nnnnnnnn bytes
                  Symbols  nnnnnnnn bytes
                  Total    nnnnnnnn bytes
                  Excess   nnnnnnnn bytes
                  Source   nnnnnnnn lines
        2: Object file complete
        3: List file complete
        4: Program file complete: nnnnnn bytes
           Elapsed time: m:ss = nnnn lines/minute
```

2.2.11 Folding Symbolic Names to Upper Case

Normally, the compiler considers upper and lower case characters to be unique. If you do not require case sensitivity for your compilations or specifically require that the compiler not distinguish between case, including the K option on the compiler invocation command line will force all symbolic names to be folded to upper case.

2.2.12 Card Numbers with Run Time Errors

It is possible to have the line number reported during error recovery at run time. This option is invoked by adding the N option switch to the file specification at compile time. The N option is intended as a debugging tool and its use is not recommended after the program is completely operational in that it will slow execution speed and add six bytes to the object file for every executable statement.

2.2.13 Creating Unlinked External Subprograms

It is not necessary for all external procedure references to be resolved either at compile time or run time. A completely linked executable object module may be created by either the compiler or the linker. In addition, the FORTRAN runtime system is capable of providing dynamic linkage to external procedures. The unlinked procedure may reside in a library directory on external media or in a sharable area of system memory. The runtime system will attempt to locate (and load) any external procedure whose reference is unresolved at compile time or at link time. If the module was loaded from disk storage, the memory space allocated for its execution and data areas is recovered when it returns to the calling procedure. This is in essence a powerful overlay facility. In addition, commonly used subroutines, such as DATE and TIME may be placed in a sharable area of system memory reducing the runtime memory requirements of all programs requiring them.

Normally, if a source file does not contain a main program, the error diagnostic: "Missing main program" is generated. The R option is used to force the compiler to generate code which can be linked by either of the two methods described above.

As a consequence of dynamic linking, misspelled references to subprograms may cause the compiler to treat them as external, resulting in the runtime error message: "subprogram not found".

2.2.14 Generating Symbol Tables for DEBUG

DEBUG is the symbolic debugger provided with the compiler. In order to use the debugger on a program, a symbol table containing symbolic names, data types and sizes, and relative addresses must be available. The S compiler option is used to inform the compiler that this symbol file is to be created. The symbol file itself is partially generated during pass two and completed during a special pass immediately preceeding the final linking pass. The compiler display when the S option is selected will appear as:

```
        1: Symbol table complete
                Memory usage:
                  Labels  nnnnnnnn bytes
                  Symbols nnnnnnnn bytes
                  Total   nnnnnnnn bytes
                  Excess  nnnnnnnn bytes
                  Source  nnnnnnnn lines
        2: Object file complete
        3: DEBUG symbol file complete
        4: Program file complete: nnnnnn bytes
           Elapsed time: m:ss = nnnn lines/minute
```

When using "Compile and Debug" under the "Compile" menu, a symbol file will always be generated regardless of the setting of the S option.

2.2.15 Redirecting File Preconnections

File unit identifiers which are specified with an asterisk (*) are normally connected to the system input and output devices. In the Microsoft implementation of FORTRAN 77 these are the simulated card reader and line printer spooler. The U option allows you to redirect these connections to unit 9, the terminal.

Units 5 and 6 cannot be redirected in this manner, only units specified with an asterisk.

2.2.16 Changing implicit INTEGER and LOGICAL sizes

Without an explicit length declaration, INTEGER and LOGICAL data types default to thirty-two bits (two machine words). The W option is used to change this default length to sixteen bits (one machine word).

2.2.17 Conditional Compilation

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 considered to be the same as a blank character.

2.2.18 The OPTION statement

The OPTION statement provides a facility for selectively turning certain options on or off during the compilation process. The form of this statement is:

OPTION option list

where option list consists of any combination of B, C, F, J, N, or X. Each of these six options may be modified by a plus (+) or minus (-) sign which turns the option on or off respectively. The plus sign may be omitted. For example, in the code segment:

```
        OPTION CN

        A(IX,IY) = DSQRT(VAL1-PI)
        A(IX+1,IY+1) = A(IX,IX)*2D0

        OPTION -C-N
```

the first OPTION statement turns on array boundary checking and card number reporting at run time. The two assignment statements are compiled with these debugging features enabled. The second OPTION statement turns off these features until another OPTION statement reenables them.

2.3 CONVERTING EXISTING PROGRAMS TO MICROSOFT FORTRAN 77

First and foremost, spend the time and learn how to use the debugger. It can save you hours of time and completely eliminate the need to insert WRITE statements throughout your code.

If your program is written in FORTRAN 66 you may have to decide if you want to convert it to FORTRAN 77 or simply use the H compiler option to establish FORTRAN 66 compatibility. The H option was documented earlier in this chapter in the section on COMPILE TIME OPTIONS and points out areas of potential problems you will have to examine. If you do convert your program to FORTRAN 77 it will run faster.

If your program is written in FORTRAN 77 there will be very little, if anything, you will have to do.

1. Some FORTRAN 77 compilers treat all storage as static but recommend that you include SAVE statements in your subprograms for portability if you depend on retaining the definition status of local variables and named COMMON blocks through successive call and return sequences. If you did not do this originally, you will have to now.

2. You may have to examine explicit OPEN statements with named files if your program came from a different operating system environment. If you relied on file preconnections established through some JCL, the PROGRAM statement (Chapter 9) will allow you to change the default unit numbers from 5 and 6. The Microsoft FORTRAN Compiler for the Macintosh does not have the ability to establish a file preconnection to unit 5 (the simulated card reader).

The Macintosh has I/O capabilities not considered in the design of FORTRAN 77. These are available to you using the tool box subroutine (toolbx.sub) provided with the Microsoft FORTRAN Compiler. To avoid having to convert your programs to use these routines, a simulated teletype environment is provided automatically by applications created by the compiler. This is a non-movable, non-sizable window with no goaway box, which displays 80 columns by 24 lines of text in the Monaco 9 font. If you use standard FORTRAN output statements such as WRITE directed to the console (unit 9), the output will appear in this window. READ statements directed to the console (also unit 9) will read keyboard input, which is echoed on this window. Using toolbx.sub, different fonts and graphics can be displayed in the standard window, or it can be disposed of to make way for more exotic interfaces.

You should make note of the fact that, when your FORTRAN program terminates, the Macintosh will return to the Finder. This erases all program results directed to the console. This is not a problem if your program is interactive, but programs which just display the answer on the console and quit will give you very little time to read it. Such programs should contain at least one PAUSE statement or a READ statement directed to the console so that the program will not exit until a carriage return is entered.

If your program uses extensions to FORTRAN provided by your previous compiler, chances are we have included them for portability. In addition to the various compile time options, below is a list of some of the extensions we provide:

1. You may assign character data types to numeric variables:

   ```
   IVAL = 'TEST'
   ```

2. You may use logical operators on integer variables for boolean operations:

   ```
   IVAL = IVAL .AND. 255
   ```

3. INTEGER*1, INTEGER*2, LOGICAL*1, and LOGICAL*2 data types are available.

4. SHIFT is provided as an integer intrinsic function; you will not have to write your own again.

5. DATE and TIME are provided as subroutines for extracting the system date and time.

6. A means is provided for specifying binary, octal, and hexadecimal constants and I/O format lists.

7. SELECT CASE, CASE, and END SELECT statements are provided for structuring multiple condition execution blocks.

8. DO WHILE, WHILE, END DO, DO, REPEAT, CYCLE, and EXIT statements are provided for looping structures.

9. TYPE and ACCEPT statements are available for performing list-directed I/O to your terminal.

10. An EXECUTE statement is provided for chaining programs together.

11. The compiler is case sensitive and symbolic names may be up to thirty-one characters long.

12. The characters <, >, and = may be used for forming relational operators.

13. Several methods of handling floating point exceptions are provided (see Chapter 9).

CHAPTER 3

DEBUG - THE SYMBOLIC DEBUGGER

DEBUG is a window oriented symbolic debugging tool for FORTRAN programs and external procedures written in FORTRAN. It provides for executing single statements, setting breakpoints, executing blocks of statements, and examining and modifying the contents of program variables.

Throughout this chapter, the term "card" is used to refer to the standard FORTRAN source record layout of seventy-two significant columns. In that FORTRAN statements may be continued over as many as nineteen records, the term card is used to distinguish a single record from a complete statement. Only the first card of any statement is listed in the source code window; continuation cards are not displayed. When listing source in the source code window, DEBUG will display any executable statements contained in INCLUDE files in place of the INCLUDE statement. This is necessary to insure proper card/object code synchronization.

DEBUG is position independent, but is not reentrant nor reusable.

When DEBUG requires a text response, line editing is accomplished using standard Macintosh text editing commands. The mouse may be used to position an insertion point within the text, which will be displayed as a vertical line between two characters. Text entered at the keyboard will be inserted at this point, and the draw point will move to the right of the new character. The backspace key deletes the character immediately to the left of the cursor, and moves the cursor left one character. Dragging the mouse through text will select that text. Backspace will delete any selected text, and any character entered at the keyboard will replace the entire selection.

When DEBUG requires a file name, it will display the Macintosh Standard File Window. This window displays all files of the proper type(s) on the current disk volume in a scrollable window, along with several buttons that can be selected with the mouse. Choose the desired file by clicking on the file name and then selecting OPEN. Selecting CANCEL will abort the operation. The DRIVE button displays files on the alternate disk drive if you have a two drive system, and the EJECT button will eject the disk in the current drive, allowing you to insert another.

3.1 INITIALIZATION

To use DEBUG on a FORTRAN program or subprogram, the source file must be compiled using the S compiler option. This will cause the compiler to generate a symbol file. The name of this symbol file will be the same as the source file with an extension of .SYM. The object file generated will be identical to the standard object file with one exception: subprograms which have a RETURN statement directly preceding the END statement will be longer. The compiler does not normally generate code for an END statement if it is directly preceded by a RETURN statement since they produce exactly the same code, however DEBUG requires that this code be present for synchronization purposes.

The debugger can be invoked in one of two ways. One is by opening the DEBUG file from the Macintosh Finder. This can be done by single clicking the DEBUG icon and then choosing OPEN from the Finder FILE menu, or by double clicking the DEBUG icon. When invoked using this method, DEBUG will display a Standard File window, as described above, allowing you to select any text file on your system.

DEBUG can also be invoked from the Microsoft FORTRAN Compiler. Selecting DEBUG from Microsoft FORTRAN's COMPILE menu will debug the current file, which is specified with the SELECT FILE option under the compiler's FILE menu. The COMPILE AND DEBUG option will compile a file (generating a symbol table regardless of the setting of the compiler options) and then invoke DEBUG on it.

DEBUG will locate the object file and symbol file and load them automatically. The source file will also be located for listing the statement lines in the source code window. The source file is read as a file, and is not loaded into memory.

After all the required files have been located and loaded, if the object file is a main program, execution will be initialized, returning to DEBUG at the first executable statement. If the object file is an external procedure, DEBUG will prompt for the main program with the same file selection box it uses to get the source file name. After all the required files have been located and loaded, execution will proceed until the first statement in the external procedure is encountered. The menu bar will display a single menu named Debug, and two windows will appear. One is a smaller version of the default window used as a console by all Microsoft FORTRAN Compiler applications; the other displays the first page of the source code being debugged. These are described below.

3.2 SOURCE CODE WINDOW

DEBUG keeps a page of twenty-one lines of executable source statements in memory. The Source Code window can be used to view a portion of this text, up to the total 21 lines.

The file name of the source code being debugged is displayed in the window's title bar. To the left of the file name is a goaway box. Clicking on this with the mouse will make the window invisible and inactive. The menu command SOURCE CODE (see THE DEBUG MENU, below) will bring it back. It can also be moved anywhere on the Macintosh screen by clicking the mouse in its title bar, dragging the mouse to the new location, and letting up on the mouse button.

The size of the Source Code window can be changed to view long source lines or to accommodate other windows on the screen. Clicking the mouse in the grow icon at the bottom right will allow an outline of the bottom and right sides of the window to be dragged to a new location. Letting up on the mouse button changes the size of the window to fit the outline.

A smaller version of the default run time window is displayed at the bottom of the screen. This window is not sizable or movable, but can be modified or even replaced through the use of the TOOLBX subroutine in the application being debugged (see "Microsoft FORTRAN Compiler Tool Box Interface", the notes on using the toolbox from the Microsoft FORTRAN Compiler).

Two card numbers are significant to DEBUG: the card number of the current statement to be executed and the card number at the cursor. When single stepping, these values will be the same and the cursor will be on the statement to be executed next. When executing blocks of statements, the cursor will be on the statement where a soft breakpoint is set. The card number of the current card is displayed in the source code window enclosed in a hollow rectangle. A hollow arrow appears to the right of the card number at the cursor.

Two types of breakpoints are used with DEBUG: hard and soft. Hard breakpoints are those which are set explicitly with the mouse. Soft breakpoints are implicitly set by moving the cursor from the card which is to be executed next.

The locations of both the current card and the cursor are affected by executing program statements. The cursor location can also be changed by the SEARCH and FIND LABEL menu options (above) or by the source code window scroll bar. Note that this scroll bar, unlike most scroll bars in Macintosh applications, does not necessarily affect the data displayed in the window. It only moves the cursor; the source code displayed in the window is adjusted to ensure that the cursor is always visible.

The down arrow scroll bar button will cause the cursor to advance to the next card and set a soft breakpoint at that card. If the cursor leaves that portion of the source code currently in memory (21 lines), the next page will be displayed with the cursor at the top.

The up arrow scroll bar button will cause the cursor to go to the previous statement and set a soft breakpoint at the cursor. If the cursor leaves that portion of the source code currently in memory (21 lines), the display will page backwards eleven lines.

The page up and page down scroll bar regions (the gray areas above and below the scroll box or thumb) will cause the cursor to go back or forward 21 lines, respectively. A soft breakpoint is set at the cursor.

Moving the scroll box (or thumb) within the gray area of the scroll bar will move the cursor to the same proportional point in the source file and set a soft breakpoint there. For example, moving the thumb to the top of the scroll bar will put the cursor on the first executable card; moving it to the bottom will put it on the last executable card.

It is not possible to execute FORTRAN statements out of sequence. All memory addressing modes are either PC relative or base register relative. The contents of certain address pointers is dependent on the previous execution sequence. All execution is therefore forced to be sequential.

To conserve memory, the source file is read as a file to locate cards to be displayed. An attempt is made to preserve the current location in this file so that paging forward in the source proceeds as quickly as possible. When jumping around or moving backwards in extremely large source files, a noticeable delay may occur in setting up a Source Code window. Also, a pause may be noted when an INCLUDE statement is encountered.

Only the first card of an executable FORTRAN statement or a statement which generates in line data is displayed. Comments and non-executable statements, except CONTINUE, ELSE, END IF, CASE, and END SELECT, are not displayed. External procedure references are executed as though they were a single statement. If they are not already in memory, they will be fetched, executed, deleted, and control will return at the statement directly following the reference.

To use DEBUG on an external procedure written in FORTRAN, it is necessary to either include it entirely within the main source file, or execute DEBUG in the external procedure mode.

The external procedure mode is indicated by selecting the name of the procedure when DEBUG is invoked. DEBUG will then display a menu of application file names which can be selected with the mouse. Select the name of the FORTRAN program through which the procedure will be referenced. Execution will proceed until the external procedure is referenced at which point the Source Code window will appear displaying it.

3.3 THE DEBUG MENU

Most operations under DEBUG are selected from the DEBUG menu. This is a standard Macintosh menu, and commands can be entered by pressing the mouse button while in the word 'Debug' in the menu bar, dragging to the desired operation, and releasing the button. A summary of these commands follows.

3.3.1 SOURCE CODE

Command Key Equivalent: Command-C.

This option makes the source code window visible. This window is described above in the section 'Source Code Window'.

3.3.2 VARIABLES

Command Key Equivalent: Command-V.

This option makes the Variable and Variable Entry windows active and visible. The Variable window is used to examine and modify the contents of variables within a program unit. The name of the current program unit appears in the title bar of the variable window, followed by the word 'Memory' and the amount of local or dynamic memory required for the program unit. Like the Source Code window (described above), the Variable window can be moved, its size can be changed, and it can be made invisible by clicking its goaway box.

The size of the Variable window can be changed to view variable types with longer displays (such as complex) or to accommodate other windows on the screen. Clicking the mouse in the grow icon at the bottom right will allow an outline of the bottom and right sides of the window to be dragged to a new location. Letting up on the mouse button changes the size of the window to fit the outline.

To examine variables, select the Variable Entry window by clicking on it with the mouse, enter the name of the variable in the text edit box, and click the OK button (or press the Return key). Once entered, a variable will remain selected until it is deleted, so long as the program unit in which it is defined remains active. Only variables in the currently active program unit can be displayed.

If the variable is an array element, its subscript(s) must also be given enclosed in parentheses. A range of array elements may be examined by typing a colon and second subscript enclosed in parentheses after the name and subscript of the array element:

```
VAR(i):(j)
```

When examining variables or array elements, the variable's symbolic name, type, size, and value are displayed.

More variables can be selected for display than can actually be displayed in the Variable window. These are accessed with the scroll bar along the right hand side of the window. Clicking the arrow buttons at the top and bottom of the scroll bar will move the text in the window up or down a line at a time. Clicking the grey areas above or below the scroll box will move a page (one window's worth) up or down. Holding the mouse button down in any of these controls causes the command to be executed repeatedly. Clicking and dragging the scroll box to a new location in the scroll bar cases the top line of the window to be positioned at the same proportionate location within the currently selected variables.

When a variable is selected, it is displayed immediately in the Variable window, provided its position is on the current page. If the variable does not appear, it is above or below the current page, and the scroll bars must be used to center the window over it. The variables selected for display will be listed in the Variable window in the order of declaration.

To delete a variable from the Variable window, click on its name with the mouse. The name will be highlighted. When the mouse button is released, the variable will be deleted.

To modify the contents of a variable or array element, click the value displayed in the window with the mouse. An edit window will appear. This window contains a text edit box, an OK button, and a CANCEL button. The new value is entered using standard Macintosh editing commands. After entering the value, click on the OK button (or press the Return key). To exit without changing the value of the variable, click the CANCEL button.

All floating point values are displayed in scientific notation.

3.3.3 SEARCH

Command Key Equivalent: Command-P.

The SEARCH command is used to locate a specific card or program unit. When this option is selected, an edit window will appear. This window contains a text edit box, an OK button, and a CANCEL button. The card number or program unit is entered using standard Macintosh editing commands. After entering the value, click on the OK button (or press the Return key). To exit, click the CANCEL button.

To locate a particular card, enter its number into the edit window. Cards containing non-executable statements and continuation cards cannot be located as only those cards containing executable FORTRAN statements generate object code and therefore entries in the symbol table. If the card is found, it will be displayed at the top of the source code window.

To locate a program unit, enter its name into the text edit window. The first card of the program unit will be displayed in the source code window.

3.3.4 FIND LABEL

Command Key Equivalent: Command-L.

The LABEL command is used to search for a label within the source file. When this option is selected, an edit window will appear. This window contains a text edit box, an OK button, and a CANCEL button. The last label sought will be selected in the text edit box. A new label will overwrite the old one, it can be edited using the standard Macintosh editing commands, or it can be reused to find the next occurrence of this label. After entering the label, click on the OK button (or press the Return key). The search will begin after the cursor position. To exit, click the CANCEL button.

The label argument can be any five digit integer in the range of 1 to 99999. Embedded blanks are ignored. If the label is found in the source file, DEBUG displays the statement containing the label on the first line of the source code window.

3.3.5 FILE STATUS

Command Key Equivalent: Command-I.

This command reports on units which are connected to files. Information is provided regarding the unit number, file name, access method (direct or sequential), iostat, and status (keep or scratch). For direct access connections, record length and next record are also listed. If an error condition exists in the file connection, a FORTRAN run time error code (see Appendix D) is reported in iostat.

The file status information is displayed in a modal window with two buttons (MORE and CANCEL). This window covers most of the screen, and no other operation can take place while it is visible. Clicking on either button (or pressing the Return key) removes it.

3.3.6 TYPE FILE

Command Key Equivalent: Command-T.

The TYPE FILE option is used to display the contents of a file on the terminal. The text is displayed in a modal window with two buttons (MORE and CANCEL), a page at a time. This window covers most of the screen, and no other operation can take place while it is visible. Clicking on the MORE button (or pressing the Return key) displays the next page (if any), and CANCEL aborts the command and removes the window.

3.3.7 FINISH

Command Key Equivalent: Command-F.

This command causes execution to continue unconditionally. The program will complete its normal execution including implicitly closing all files at termination.

3.3.8 QUIT

Command Key Equivalent: Command-Q.

The QUIT command ceases execution of DEBUG and returns control to the operating system. Execution is first passed through the FORTRAN service module exit routine and all files which do not contain an error condition will be closed.

3.3.9 TRANSFER

Command Key Equivalent: Command-E.

This command allows another application to be opened without going through the Finder. It can be used to enter DEBUG again. A Standard File window is used to allow the selection of any application file on the system. This file will be started as if it had been opened from the Finder.

3.3.10 BREAK POINTS

Command Key Equivalent: Command-B.

This menu option displays all current breakpoints (maximum 8). The breakpoints are displayed in a modal window with two buttons (MORE and CANCEL). This window covers most of the screen, and no other operation can take place while it is visible. Clicking on either button (or pressing the Return key) removes it.

3.3.11 SINGLE STEP

Command Key Equivalent: Command-S (also Return).

When the current card number and the cursor card number are the same (the arrow and box are on the same line) only the statement that the cursor is positioned on will be executed. This is equivalent to pressing the Return key.

3.3.11.1 BLOCK EXECUTE

Pressing the RETURN key or selecting SINGLE STEP when the current card number and the cursor card number values are different will cause all of the statements up to and including the statement that the cursor is positioned on to be executed.

3.3.12 PROCEED TO CURSOR

Command Key Equivalent: Command-G.

This option will execute all statements up to but not including the card the cursor is positioned on.

3.3.13 HOME CURSOR

Command Key Equivalent: Command-H.

This option returns the cursor to the next card to be executed.

3.3.14 SKIP SUBROUTINES

Command Key Equivalent: Command-\.

This option will toggle the subroutine skip flag. When subroutine skip mode is on, a check mark will appear to the left of the 'SUBROUTINE SKIP' option name on the DEBUG menu.

When subroutine skip mode is on, a call to an internal subprogram is executed as if it were a single statement. This is equivalent to moving the cursor to the statement directly following the subroutine call and executing the proceed to cursor command (Command-G).

3.3.15 BREAKPOINTS

To set a breakpoint, click on a card in the source code with the mouse. The card number will be redisplayed inverted (white on black) indicating that a hard breakpoint has been set.

To clear a breakpoint, again click on the card where the breakpoint is set. The card number will be redisplayed in normal text indicating that the breakpoint has been cleared.

3.3.16 PROCEED TO BREAKPOINT

Command Key Equivalent: Command-X.

This option causes execution to proceed to the next breakpoint. When the breakpoint is reached, the line containing the breakpoint is shown in the source code window. If no breakpoints are set or a breakpoint is not encountered, execution proceeds as though the FINISH command were entered.

Reference

Reference

CHAPTER 4

THE FORTRAN 77 PROGRAM

FORTRAN 77 source programs consist of one program unit called the main program and any number of program units called subprograms. A program or program unit is constructed as an ordered set of statements that describes procedures for execution and information to be used by the FORTRAN 77 compiler during the compilation of a source program. Every program unit is written using the FORTRAN 77 character set and follows a prescribed statement line format. A program unit may be one of the following:

1. Main program

2. Subroutine subprogram

3. Function subprogram

4. Block Data subprogram

This chapter describes the format of FORTRAN programs, and the data objects that may be manipulated by them.

4.1 CHARACTER SET

There are 80 characters which have meaning to the compiler. These characters include the 52 upper and lower case letters of the alphabet, the 10 decimal digits from 0 to 9, and the special characters:

Character	Definition
+	plus
-	minus
*	asterisk
/	slash
=	equals
.	decimal point
,	comma
	blank
(	opening parenthesis
)	closing parenthesis
'	apostrophe
:	colon
;	semicolon
"	quotation mark
_	underscore
!	exclamation mark
<	less than
>	greater than

Any of these characters, as well as the remaining printable ASCII characters, may appear in character and Hollerith constants (see below).

4.2 SYMBOLIC NAMES

A symbolic name is used to identify a FORTRAN 77 entity, such as a variable, array, program unit, or labeled common block. The compiler accepts symbolic names of up to thirty-one upper and lower case letters and digits. The first character of a symbolic name must be a letter. The underscore character and the blank character are not significant in a symbolic name and may be used as separators. Upper and lower case letters are distinct unless the compiler case fold option, K, has been selected (see Chapter 2). Symbolic names of greater than thirty-one characters are acceptable, but only the first thirty-one characters are significant to the compiler.

Only the first six characters of the symbolic name of a main program; a subroutine, function, or block data subprogram; a common block; or a virtual array are significant. Lower case letters are folded to upper case. The file naming conventions of operating systems vary wildly, however most provide for at least six upper case letters or digits. Names in this class are stored internally in packed RAD50 form (see Appendix G for a discussion of the RAD50 algorithm). Consequently, only four bytes are required for storing a name or making an external reference.

Global symbolic names are known to every program unit within an executable program and therefore must be unique. The names of main programs; subroutine, function, and block data subprograms; common blocks, and virtual arrays are global symbolic names.

Local symbolic names are known only within the program unit in which they occur. The names of variables, arrays, symbolic constants, statement functions, and dummy procedures are local symbolic names.

4.3 KEYWORDS

A keyword is a sequence of characters that has a predefined meaning to the compiler. A keyword is used to identify a statement or serve as a separator in a statement. Some typical statement identifiers are READ, FORMAT, and REAL. Two separators are TO and THEN.

There are no reserved words in FORTRAN 77, therefore a symbolic name may assume the exact sequence of characters as a keyword. The compiler determines the meaning of a sequence of characters through the context in which the characters are used. A surprising example of a keyword/symbolic name exchange is:

Statement	Meaning
DO10I=1,7	Control statement
DO 10 I = 1. 7	Assignment statement

Note that the embedded blanks are not significant nor are they required as separators for the compiler to determine that the first statement is the initial statement of a DO loop. The absence of a comma in the second statement informs the compiler that an assignment is to be made to the variable whose symbolic name is DO10I.

In some instances it may be impossible for the compiler to determine from the context the meaning the programmer intended. For example:

```
DIMENSION SIN(15)
A = SIN(B)
```

Such ambiguous contexts should obviously be avoided.

4.4 LABELS

A statement label may be placed anywhere in columns 1 through 5 of a FORTRAN 77 statement initial line. A statement label is used for reference in other statements. The following considerations govern the use of the statement label:

1. The label is an unsigned integer in the range of 1 to 99999.
2. Leading zeros and blanks are not significant to the compiler.
3. A label must be unique within a program unit.
4. A label is not allowed on a continuation line.
5. Labels may appear in any numeric order.

The following examples all yield the same label:

```
1101
1 101
11 01
110 1
```

The use of labels has no effect on either the ultimate size of the compiled program and/or its execution speed. However, their inclusion in the source program does increase the memory required for compilation. Labels are used in FORTRAN 77 as their name implies: to label statement lines for reference purposes. Excessive unnecessary labels slow compilation and may even prevent compilation and should therefore be avoided.

4.5 STATEMENTS

Individual statements deal with specific aspects of a procedure described in a program unit and are classified as either executable or nonexecutable. The proper usage and construction of the various types of statements are described in the following chapters.

4.5.1 Executable Statements

Executable statements specify actions and cause the FORTRAN 77 compiler to generate object program instructions. There are 3 types of executable statements:

1. Assignment statements
2. Control statements
3. Input/Output statements

4.5.2 Nonexecutable Statements

Nonexecutable statements are used as directives to the compiler: start a new program unit, allocate variable storage, insert a form feed in the listing, initialize data, set the options, etc. There are 7 types of nonexecutable statements:

1. Specification statements
2. Data initialization statements
3. FORMAT statements
4. Function defining statements
5. Subprogram statements
6. Main program statements
7. Compiler directives

4.5.3 Statement Format

A FORTRAN statement consists of one or more source records referred to as a statement line. Historically a record is equivalent to a card. In current source file formats, a record is one line of text terminated by an end of record character (generally a carriage return, line feed, or carriage return-line feed pair). A statement line consists of 80 character positions or columns, numbered 1 through 80 which are divided into 4 fields:

Field	Columns
Statement label	1-5
Continuation	6
Statement	7-72
Identification	73-80

The Identification field is available for any purpose the programmer may desire and is ignored by the FORTRAN 77 compiler. Historically this field has been used for sequence numbers and commentary. The statement line itself may exceed 80 characters; the compiler ignores all characters beyond column 72.

Statements are placed in columns 1 through 72, formatted according to line types. Their are four line types in FORTRAN 77:

1. Comment Line - used for source program annotation and formatting. A comment line may be placed anywhere in the source program and assumes one of the forms:

   a. Column 1 contains the character C, an asterisk, or an exclamation point. Columns 2 through 72 may contain any sequence of characters or may be blank.

   b. The line is completely blank.

   c. An exclamation point not contained within a character constant designates all characters including the exclamation point through the end of the line to be commentary.

   Comment lines have no effect on the object program and are ignored by the FORTRAN 77 compiler except for display in the listing of the program.

2. End Line - the last line of a program unit.

   a. The word END must appear within columns 7 through 72.

   b. Each FORTRAN 77 program unit must have an END line as its last line to inform the compiler that it is at the physical end of the program unit.

   c. An END line may follow any other type of line.

3. Initial Line - the first and possibly only line of each statement.

   a. Columns 1 through 5 may contain a statement label to identify the statement.

   b. Column 6 must contain a zero or a blank.

   c. Columns 7 through 72 contain all or part of the statement.

4. Continuation Line - used when additional characters are required to complete a statement originating on an initial line.

   a. Columns 1 through 5 must be blank

   b. Column 6 must contain a character other than zero or blank.

   c. Columns 7 through 72 contain the continuation of the statement.

   d. There may be only 19 continuation lines per statement, for a total of 20 lines per statement.

4.5.4 Multiple Statement Lines

Multiple statements may be placed on the same line by separating them with a semicolon (;). Only executable statements may be placed on the same statement line in this manner.

```
        I=10; J=10; N(I,J)=0
```

4.6 DATA ITEMS

The symbolic name used to represent a constant, variable, array, substring, statement function, or external function identifies its data type, and once established, it does not change within a particular program unit. The data type of an array element name is always the same as the type associated with the array.

Special FORTRAN statements, called type statements, may be used to specify a data type as character, logical, integer, real, double precision, or complex. When a type statement is not used to explicitly establish a type, the first letter of the name is used to determine the type. If the first letter is I, J, K, L, M, N, i, j, k, l, m, or n, the type is integer; any other letter yields an implied type of real. The IMPLICIT statement, described later, may be used to change the default implied types. The IMPLICIT NONE statement, also described later, causes the compiler to require declaration of all variables.

An intrinsic function, LOG, EXP, SQRT, INT, etc., may be used with either a specific name or generic name. The data types of the specific intrinsic function names are listed in Table 9-1. A generic function assumes the data type of its arguments as discussed later in Chapter 9.

A main program, subroutine, common block, and block data subprogram are all identified with symbolic names, but have no data type.

4.6.1 Constants

FORTRAN 77 constants are identified explicitly by stating their actual value; they do not change in value during program execution. The plus (+) character is not required for positive constants. The value of zero is neither positive nor negative; a zero with a sign is just zero.

The data type of a constant is determined by the specific sequence of characters used to form the constant. A constant may be given a symbolic name with the PARAMETER statement, but a constant itself may not be constructed using the symbolic name of another constant.

Except within character and Hollerith constants, blanks are not significant and may be used to increase legibility. For example, the following forms are equivalent:

```
3.14159265358979        3.1415 92653 58979
2.71828182845904        2.7182 81828 45904
```

4.6.1.1 Character Constant

A character constant is formed with a string of any of the characters from the ASCII character set. The string is delimited by either apostrophes (') or quotation marks ("). The character used to delimit the string may be part of the string itself by representing it with two successive delimiting characters. The number of characters in the string determines the length of the character constant. A character constant requires a character storage unit (one byte) for each character in the string.

```
        'TEST'                     "TEST"
        'EVERY GOOD BOY'           "EVERY GOOD BOY"
        'Luck is everything'       "Luck is everything"
        'didn''t'                  "didn't"
```

FORTRAN 77 has no facility for specifying or representing a character constant consisting of the null string. If the character constant '' or "" is encountered by the compiler it is interpreted as a single blank character.

4.6.1.2 Logical Constant

Logical constants are formed with the strings of characters, .TRUE. and .FALSE., representing the boolean values true and false respectively. A false value is represented by a field of thirty-two zero bits and a true value is represented by a field of thirty-two one bits. A logical constant requires one numeric storage unit (four bytes).

4.6.1.3 Integer Constant

An integer constant is an exact binary representation of an integer value in the range of -2147483648 to +2147483647 with negative integers maintained in two's complement form. An integer constant is a string of decimal digits which may contain a leading sign. An integer constant requires one numeric storage unit (four bytes).

```
        15
        101
        -72
        1126
        123 456 789
```

4.6.1.3.1 Alternate Integer Bases

The compiler normally expects all numeric constants to be in base ten, however, three alternate unsigned integer bases are available when explicitly specified. These optional bases are binary, octal, and hexadecimal and are designated by preceding the constant with the characters B, O, and Z respectively and delimiting the constant itself with apostrophes. The following examples all result in the assignment of the decimal value 3994575:

```
I = B'1111001111001111001111'
J = O'17171717'
K = Z'3CF3CF'
```

As with all numeric constants, spaces may be used freely to enhance legibility. The following examples produce identical assignment statements:

```
I = B'0011 1100 1111 0011 1100 1111'
J = O'017 171 717'
K = Z'3C F3 CF'
```

4.6.1.4 Real Constant

A real constant consists of an optional sign and a string of digits which contains a decimal point. The decimal point separates the integer part of the constant from the fractional part and may be placed before or after the string indicating that either the integer or fractional part is zero. A real constant may have an exponent which specifies a power of ten applied to the constant. An exponent is appended to a real constant with the letter E and an integer constant in the range of a -37 to +39. If an exponent is given and the fractional part is zero, the decimal point may be omitted. A real constant requires one numeric storage unit (four bytes).

Constant		Value
1E2	=	100.0
-12.76	=	-12.76
1.07E-1	=	.107
0.4237E3	=	423.7

Real values are maintained in IEEE single precision floating point representation. The most significant bit is interpreted as the sign, the next eight bits provide a binary exponent biased by 127, and the remaining twenty-three bits form the binary mantissa with a twenty-fourth bit implied. This representation supplies seven digits of precision and a range of ±0.3402823E+39 to ±0.1175494E-37. (See Appendix F for further information.)

4.6.1.5 Double Precision Constant

A double precision constant is formed in the same manner as a real constant except that the exponent is designated with the letter D and must always be given, even if its value is zero. The exponent range of a double precision constant is -307 to +309. A double precision constant requires two numeric storage units (eight bytes).

Constant	Value
1D2	= 100.0
-12.76D0	= -12.76
1.07D-1	= .107
0.4237D3	= 423.7

Double precision values are maintained in IEEE double precision floating point representation. The most significant bit is interpreted as the sign, the next eleven bits provide a binary exponent biased by 1023, and the remaining fifty-two bits form the binary mantissa with a fifty-third bit implied. This representation supplies sixteen digits of precision and a range of ±0.1797693134862320D+309 to ±0.2225073858507202D-307. (See Appendix F for further information.)

4.6.1.6 Complex Constant

A complex constant is stated using a left parenthesis, a pair of real or integer constants separated by a comma, and a right parenthesis. The first constant is the real portion (in the mathematical sense) and the second is the imaginary portion. A complex constant requires two numeric storage units (eight bytes).

```
(2.76,-3.81)       =  2.76 -3.81i
(-12,15)           = -12.0 +15.0i
(0.62E2,-0.22E-1)  =  62.0 -.022i
```

Microsoft® FORTRAN Compiler
for the Apple® Macintosh™

Reference Manual

4.6.1.7 Hollerith Constant

The Hollerith data type is an older method of representing characters in FORTRAN. While it is not included in the current standard, this implementation of FORTRAN includes the Hollerith data type to provide compatibility for older programs. Like character constants, a Hollerith constant is formed with a string of any of the characters from the ASCII character set. Logical, integer, real, double precision, and complex variables can be defined with a Hollerith value through DATA statements and READ statements.

A Hollerith constant is stated with a nonzero, unsigned integer constant, the letter H, and a string of characters whose length must be the same as the integer constant.

```
4HTEST
14HEVERY GOOD BOY
```

When a Hollerith constant is assigned to a variable it is left justified and space padded if the length of the constant is less than the length of the variable.

4.6.2 Variables

A variable is the used to maintain a FORTRAN 77 quantity and is associated with a single storage location through a symbolic name. Simple variables are often called scalar variables to distinguish them from arrays and array elements (see below). Unlike a constant, the value of a variable can be changed during the execution of a program with assignment statements and input and output statements.

4.6.3 Arrays

An array is a sequence of data elements all of the same type and referenced by one symbolic name. When an array name is used alone it refers to the entire sequence starting with the first element. When an array name is qualified by a subscript it refers to an individual element of the sequence.

4.6.3.1 Array Declarator

An _array declarator_ is used to assign a symbolic name to an array, define its data type (either implicitly or explicitly), and declare its dimension information:

```
a(d [,d]...)
```

where _a_ is the symbolic name that will be used to reference the array and the elements of the array, and _d_ is called a _dimension declarator_. An array declarator must contain at least one and no more than seven dimension declarators. A dimension declarator is given with either one or two arguments:

```
[d1:] d2
```

where _d1_ and _d2_ are called the lower and upper _dimension bounds_ respectively. The lower and upper dimension bounds must be expressions containing only constants or integer variables. Integer variables are used only to define _adjustable arrays_ (described below) in subroutine and function subprograms. If the lower dimension bound is not specified, it has a default value of one.

An array declarator specifies the size and shape of an array: the number of dimensions, the upper and lower bounds of each dimension, and the number of array elements. The number of dimensions is determined by the number of dimension declarators. Dimension bounds specify the size or extent of an individual dimension. While the value of a dimension bound may be positive, negative, or even zero, the value of the lower dimension bound must always be less than than or equal to the value of the upper dimension bound. The extent of each dimension is defined as _d2_-_d1_+1. The number of elements in an array is equal to the product of all of its dimension extents.

Array declarators are called constant, adjustable, or assumed size depending on the form of the dimension bounds. A constant array declarator must have integer constant expressions for all dimension bounds. An adjustable array declarator contains one or more integer variables in the expressions used for its bounds. An array declarator in which the upper bound of the last dimension is an asterisk (*) is an assumed size array declarator. Adjustable and assumed size array declarators may appear only in subroutine and function subprograms.

All array declarators are permitted in DIMENSION and type statements, however only constant array declarators are allowed in COMMON statements. Adjustable and assumed size array declarators do not supply sufficient information to map the COMMON block at compile time.

An array can be either an actual array or a dummy array. An actual array uses constant array declarators and has storage established for it in the program unit in which it is declared. A dummy array may use constant, adjustable, or assumed size array declarators and declares an array that is associated through a subroutine or function subprogram dummy argument list with an actual array.

The number of dimensions and the dimension extents of arrays associated with one another either through common blocks, equivalences, or dummy argument lists need not match.

4.6.3.2 Array Subscript

The individual elements of an array are referenced by qualifying the array name with a subscript:

```
a(s [,s]...)
```

where each s in the subscript is called a subscript expression and a is the symbolic name of the array.

The subscript expressions are numeric expressions whose values fall between the lower and upper bounds of the corresponding dimension. If the value of the expression is not an integer, the compiler supplies the appropriate conversion. There must be a subscript expression for each declared dimension.

Some FORTRAN constructs accept array names unqualified by a subscript. This means that every element in the array is selected. The elements are processed in _column major_ order. The first element is specified with subscript expressions all equal to their lower dimension bounds. The next element will have the leftmost subscript expression increased by one. After an array subscript expression has been increased through its entire extent it is returned to the lower bound and the next subscript expression to the right is increased by one.

Subscript expressions may contain array element and function references. The evaluation of a subscript expression must not affect the value of any other expression in the subscript. This means that functions should not have side effects altering the values of the other subscript expressions.

The order of an array element within the column major storage sequence of the array in memory is called the _subscript value_. This is calculated according to the following table:

TABLE 5-1

Subscript Value

Number of Dimensions	Dimension Declarator	Subscript	Subscript Value
1	(j1:k1)	(s1)	1+(s1-j1)
2	(j1:k1,j2:k2)	(s1,s2)	1+(s1-j1)+(s2-j2)*d1
3	(j1:k1,j2:k2,j3:k3)	(s1,s2,s3)	1+(s1-j1)+(s2-j2)*d1
.	.	.	.
.	.	.	.
.	.	.	.
n	(j1:k1,...,jn:kn)	(s1,...,sn)	1+(s1-j1)+(s2-j2)*d1 +(s3-j3)*d2*d1+... +(sn-jn)*dn-1*dn-2 *...*d1

di = ki-ji+1

Note that subscript values always range from 1 to the size of the array:

```
        DIMENSION X(-4:4),Y(5,5)

        X(3) = Y(2,4)
```

For the array element name X(3), the <u>subscript</u> is (3), the <u>subscript expression</u> is 3 with a value of three, and the <u>subscript value</u> is eight. For the array element name Y(2,4), the <u>subscript</u> is (2,4), the <u>subscript expressions</u> are 2 and 4 with values two and four, respectively, and the <u>subscript value</u> is seventeen. The effect of the assignment statement is to replace the eighth element of X with the seventeenth element of Y.

4.6.3.3 Array Name

When an array name is used unqualified by a subscript, it implies that every element in the array is to be selected as described above. Array names may be used in this manner in COMMON statements for data alignment and sharing purposes, in actual and dummy argument lists to pass entire arrays to other procedures, in EQUIVALENCE statements where it implies the first element of the array, and in DATA statements for giving every element an initial value. Array names may also be used in the input and output statements to specify internal files, format specifications and elements of input and output lists.

4.6.4 Substrings

A substring is a contiguous segment of a character entity and is itself a character data type. It can be used as the destination of an assignment statement or as an operand in an expression. Either a character variable or character array element can be qualified with a substring name:

v([*e*1] : [*e*2])

a(*s* [,*s*]...)([*e*1] : [*e*2])

*e*1 and *e*2 are called *substring expressions* and must have integer values, *v* is the symbolic name of a character variable, and *a*(*s* [,*s*]...) is the name of a character array element.

The values *e*1 and *e*2 specify the leftmost and rightmost positions of the substring. The substring consists of all of the characters between these two positions, inclusive. For example, if A is a character variable with a value of 'ABCDEF', then A(3:5) would have a value of 'CDE'.

The value of the substring expression *e*1 must be greater than or equal to one, and if omitted implies a value of one. The value of the substring expression *e*2 must be greater than or equal to *e*1 and less than or equal to the length of the character entity, and if omitted implies the length of the character entity.

As with arrays, substring expressions may contain array or function references. The evaluation of a function within a substring expression must not alter the value of other entities also occurring within the substring expression. If a substring expression is not integer, automatic conversion to integer is supplied by the compiler.

4.7 STORAGE

Storage refers to the physical computer memory where variables and arrays are stored. Variables and arrays can be made to share the same storage locations through equivalences, common block declarations, and subprogram argument lists. Data items which share storage in this manner are said to be associated.

The contents of variables and arrays are either defined or undefined. All variables and arrays not initially defined through DATA statements are undefined.

A storage unit refers to the amount of storage needed to record a particular class of data. A storage unit can be a numeric storage unit or a character storage unit.

4.7.1 Numeric Storage Unit

A numeric storage unit can be used to hold or store an integer, real, or logical datum. One numeric storage unit consists of four bytes. The amount of storage for numeric data is as follows:

Data Type	Storage
Integer	1 storage unit
Real	1 storage unit
Double precision	2 storage units
Complex	2 storage units
Logical	1 storage unit

4.7.2 Character Storage Unit

A character datum is a string of characters. The string may consist of any sequence of ASCII characters. The length of a character datum is the number of characters in the string. A character storage unit differs from numeric storage units in that one character storage unit is equal to one byte and holds or stores one character.

4.7.3 Storage Sequence

The storage sequence refers to the sequence of storage units, whether they are held in memory or stored on external media such as a disk or a tape.

4.7.4 Storage Association

The storage locations of variables and arrays become associated in the following ways:

1. The EQUIVALENCE statement (described in Chapter 6) causes the storage units of the variables and array elements listed within the enclosing parentheses to be shared. Note that the data types of the associated entities need not be the same.

2. The variable and array names appearing in the COMMON statements (described in Chapter 6) of two different program units are associated.

3. The dummy arguments of subroutine and function subprograms are associated with the actual arguments in the referencing program unit.

4. An ENTRY statement (described in Chapter 9) in a function subprogram causes its corresponding name to be associated with the name appearing in the FUNCTION statement.

4.7.5 Storage Definition

Storage becomes defined through DATA statements, assignment statements, and I/O statements. READ statements cause the items in their associated I/O lists to become defined. Any I/O statement can cause items in its parameter list to become defined (the IOSTAT variable for instance). A DO variable becomes defined as part of the loop initialization process.

The fact that storage can become undefined at all should be carefully noted. Some events that cause storage to become undefined are obvious: starting execution of a program that does not initially define all of its variables (through DATA statements), attempting to READ past the end of a file, and executing an INQUIRE statement on a file that does not exist. When two variables of different types are either partially or totally associated, defining one causes the other to become undefined.

Because FORTRAN 77 provides for both dynamic as well as static storage allocation, certain events can cause dynamically allocated storage to become undefined. In particular, returning from subroutine and function subprograms causes all of their variables to become undefined except for those:

1. in blank common
2. specified in SAVE statements
3. in named common blocks established by higher level referencing procedures

The H compiler option has the effect of an implicit SAVE for every program unit encountered during the current compilation (see Chapter 2).

CHAPTER 5

EXPRESSIONS AND ASSIGNMENT STATEMENTS

Being primarily a computational language, a large number of FORTRAN statements employ expressions. The evaluation of an expression results in a single value which may be used to define a variable, take part in a logical decision, be written to a file, etc. The simplest form of an expression is a scalar value: a constant or single variable. More complicated expressions can be formed by specifying operations to be performed on one or more operands.

There are four types of expressions available in FORTRAN 77: arithmetic, character, relational, and logical. This chapter describes the rules for the formation and evaluation of these expressions.

Assignment statements, together with expressions, are the fundamental working tools of FORTRAN. Assignment statements are used to establish a value for variables and array elements. Assignment statements can also be used to modify the contents of absolute memory locations. Assignment statements _assign_ a value to a storage location.

5.1 ARITHMETIC EXPRESSIONS

An arithmetic expression produces a numeric result and is formed with integer, real, double precision, and complex operands and arithmetic operators. An arithmetic operand may be one of the following:

1. an arithmetic scalar value
2. an arithmetic array element
3. an arithmetic expression enclosed in parentheses
4. the result of an arithmetic function

The arithmetic operators are:

Operator	Purpose
**	exponentiation
*	multiplication
/	division
+	addition or identity
-	subtraction or negation

The operators **, *, and / operate only on pairs of operands, while + and - may operate on either pairs of operands or on single operands. Pairs of operators in succession are not allowed: A+-B must be stated as A+(-B). In addition, there is precedence among the arithmetic operators which establishes the order of evaluation:

Operator	Precedence
**	highest
* and /	intermediate
+ and -	lowest

Except for the exponentiation operator, when two or more operators of equal precedence occur consecutively within an arithmetic expression they may be evaluated in any order if the result of the expression is mathematically equivalent to the stated form. However, exponentiation is always evaluated from right to left:

Expression	Evaluation
A+B-C	(A+B)-C or A+(B-C)
A**B**C	A**(B**C)
A+B/C	A+(B/C)

However, the result of an arithmetic expression involving integer operands and the division operator is the quotient; the remainder is discarded: 10/3 produces an integer result of 3. Consequently, expressions such as I*J/K may have different values depending on the order of evaluation:

```
(4*5)/2 = 10, but 4*(5/2) = 8
```

5.1.1 Data Type of Arithmetic Expressions

When all of the operands of an arithmetic expression are of the same data type, the data type of the result is the same as that of the operands. When expressions involving operands of different types are evaluated, automatic conversions between types occur. These conversions are always performed in the direction of the highest ordered data type presented and the data type of the result is that of the highest ordered operand encountered. Integer is the lowest ordered data type and complex is the highest:

<u>Data Type Conversion Order</u>

integer
real
double precision
complex

Consider the expression I/R*D+C, where I is integer, R is real, D is double precision, and C is complex. The evaluation proceeds as follows:

1. the value of I is converted to real and then divided by the value of R

2. the result of the division is converted to double precision and multiplied by the value of D

3. the result of the multiplication is converted to complex and the value of C is added in

4. the data type of the result of the expression is complex

Parentheses are used to force a specific order of evaluation that the compiler may not override.

When exponentiation of real, double precision, and complex operands involves integer powers, the integer power is not converted to the data type of the other operand. Exponentiation by an integer power is a special operation which allows expressions such as -2.1**3 to be evaluated correctly.

Conversion from real to double precision does not increase the accuracy of the converted value. For example, converting the result of the real expression 1.0/3.0 to double precision yields:

```
        0.333333343267441D+00
```

not:

```
        0.333333300000000D+00 or 0.333333333333333D+00
```

5.1.2 Arithmetic Constant Expression

Arithmetic expressions in which all of the operands are constants or the symbolic names of constants are called arithmetic constant expressions. Integer, real, double precision, and complex constant expressions may be used in PARAMETER statements. Integer constant expressions may also be used in DATA statements and in specification and declaration statements (see Chapter 6).

5.2 CHARACTER EXPRESSIONS

A character expression produces a character result and is formed using character operands and character operators. A character operand may be one of the following:

1. a character scalar value
2. a character array element
3. a character substring
4. a character expression enclosed in parentheses
5. the result of a character function

The only character operator is //, meaning concatenation. Although parentheses are allowed in character expressions, they do not alter the value of the result. The following character expressions all produce the value 'CHARACTER':

```
        'CHA'//'RAC'//'TER'
        ('CHA'//'RAC')//'TER'
        'CHA'//('RAC'//'TER')
```

5.3 RELATIONAL EXPRESSIONS

A relational expression produces a logical result (true or false) and is formed using arithmetic expressions or character expressions and relational operators. The relational operators perform comparisons; they are:

Operator	Comparison
.LT. or <	less than
.LE. or <=	less than or equal to
.EQ. or =	equal to
.NE. or <>	not equal to
.GT. or >	greater than
.GE. or >=	greater than or equal to

Only the .EQ. and .NE. relational operators can be applied to complex operands.

All of the relational operators have the same precedence which is lower than the arithmetic operators and the character operator.

If the data types of two arithmetic operands are different, the operand with the lowest order is converted to the type of the other operand before the relational comparison is performed. However, because conversion from double precision to complex forces a conversion to real for the real portion and the creation of an imaginary portion of zero, the corresponding loss in precision prevents the comparison of a double precision operand with a complex operand.

Character comparison proceeds on a character by character basis using the ASCII collating sequence to establish comparison relationships. Since the letter 'A' precedes the letter 'B' in the ASCII code, 'A' is less than 'B'. Also, all of the upper case characters have lower "values" than the lower case characters. A complete chart of the ASCII character set is provided in the appendices.

When the length of one of the character operands used in a relational expression is shorter than the other operand, the comparison proceeds as though the shorter operand were extended with blank characters to the length of the longer operand.

When an integer variable is compared with a character expression, the integer is treated as though it were a character expression having a length equal to the number of bytes it occupies in storage. This is useful if the integer has been defined with a Hollerith data type.

5.4 LOGICAL EXPRESSIONS

A logical expression is formed with logical or integer operands and logical operators. A logical operand may be one of the following:

1. a logical or integer scalar value
2. a logical or integer array element
3. a logical or integer expression enclosed in parentheses
4. a relational expression
5. the result of a logical or integer function

A logical expression involving logical operands and relational expressions produces a logical result (true or false). When applied to logical operands the logical operators, their meanings, and order of precedence are:

Operator	Purpose	Precedence
.NOT.	negation	highest
.AND.	conjunction	
.OR.	inclusive disjunction	
.EQV.	equivalence	lowest
.NEQV. and .XOR.	nonequivalence	same as .EQV.

A logical expression involving integer operands produces an integer result. The operation is performed on a bit-wise basis. When applied to integer operands the logical operators have the following meanings:

Operator	Purpose
.NOT.	one's complement
.AND.	boolean and
.OR.	boolean or
.EQV.	integer compare
.NEQV. and .XOR.	boolean exclusive or

The integer intrinsic function SHIFT is available to perform left and right logical shifts (see Table 16-1).

5.5 OPERATOR PRECEDENCE

As described above, a precedence exists among the operators used with the various types of expressions. Because more than one type of operator may be used in an expression, a precedence also exists among the operators taken as a whole: arithmetic is the highest, followed by character, then relational, and finally logical which is the lowest.

```
        A+B .GT. C .AND. D-E .LE. F
```

is evaluated as:

```
        ((A+B) .GT. C) .AND. ((D-E) .LE. F)
```

5.6 ARITHMETIC ASSIGNMENT STATEMENT

Arithmetic assignment statements are used to store a value in arithmetic variables. Arithmetic assignment statements take the following form:

v = *e*

where: *v* is the symbolic name of an integer, real, double precision, or complex variable or array element whose contents are to be replaced by *e*

e is a character or arithmetic expression

If the data type of *e* is arithmetic and different than the type of *v*, then the value of *e* is converted to the type of *v* before storage occurs. This may cause truncation.

If the data type of *e* is character, the number of characters taken from the expression *e* is the number of bytes used for the storage *v*. The characters are taken from left to right. If the length of *e* is less than the number of bytes required, the expression *e* is treated as though it were extended to the right with blank characters until it is the same length as *v*.

5.7 LOGICAL ASSIGNMENT STATEMENT

Logical assignment statements are used to store a value in logical variables. Logical assignment statements are formed exactly like arithmetic assignment statements:

v = e

where: v is the symbolic name of a logical variable or logical array element

e is a logical or arithmetic expression

If the data type of e is not logical, the value assigned to v is the logical value false if the value of the expression e is zero. For non-zero values of e, the value assigned to v is the logical value true. This rule for the conversion of an arithmetic expression to a logical value applies wherever a logical expression is expected (i.e. an IF statement).

5.8 CHARACTER ASSIGNMENT STATEMENT

Character assignment statements are used to store a value in character variables:

v = e

where: v is the symbolic name of a character variable, character array element, or character substring

e is an expression whose type is character.

If the length of e is greater than the length of v, the leftmost characters of e are used.

If the length of e is less than the length of v, blank characters are added to the right of e until it is the same length as v.

5.9 ASSIGN STATEMENT

The ASSIGN statement is used to store the address of a labeled statement in an integer variable. Once defined with a statement label, the integer variable may be used as the destination of an assigned GOTO statement (Chapter 7) or as a format descriptor in an I/O statement (Chapter 8). The ASSIGN statement is given in the following manner:

ASSIGN s TO i

where: s is the label of a statement appearing in the same program unit that the ASSIGN statement does.

i is an INTEGER*4 variable name

Caution: No protection is provided against attempting to use a variable that does not contain a valid address as established with the ASSIGN statement.

5.10 MEMORY ASSIGNMENT STATEMENT

The form of a memory assignment statement is:

```
        ma = e
```

where: ma is an absolute memory address

e is any arithmetic, logical, or character expression

A memory address is formed as follows:

BYTE(e)	byte (8 bit) reference
WORD(e)	word (16 bit) reference
LONG(e)	long (32 bit) reference

where: e is an integer expression

For example:

```
        BYTE(Z'FFFFE0') = 10
```

will store the decimal value 10 at the hexadecimal memory byte address FFFFE0.

The BYTE, WORD, and LONG keywords also represent intrinsic functions (see Chapter 9) allowing indirect addressing:

```
        WORD(WORD(O'4000')) = Z'FFFF'
```

results in the storage of the sixteen bit hexadecimal value FFFF at the absolute memory location whose address is the address contained at the octal address 4000.

Note: the incorporation of these features in the compiler removes them from the set of available array names.

CHAPTER 6

SPECIFICATION AND DATA STATEMENTS

Specification statements are used to define the properties of the symbolic entities, variables, arrays, symbolic constants, etc. that are used within a program unit. For this reason, specification statements are also called declaration statements and are grouped together in the declaration section of a program unit: before any statement function statements, DATA statements, and executable statements. Specification statements themselves are classified as nonexecutable.

DATA statements are used to establish initial values for the variables and arrays used within a FORTRAN 77 program. Variables _not_ appearing in DATA statements may contain random values when a program starts executing. The use of undefined variables can cause problems that are difficult to detect when transporting a program from one environment to another, because the previous environment may have set all storage to zeros while the new environment performs no such housekeeping.

6.1 TYPE STATEMENTS

The most common of the specification statements are the type statements. They are used to give data types to symbol names and declare array names. Once a data type has been associated with a symbol name it remains the same for all occurrences of that name throughout a program unit.

6.1.1 Arithmetic and Logical Type Statements

The form of the type statement for the arithmetic and logical data types is:

```
        type [*len] v [,v]...
```

where:

type	can be LOGICAL, INTEGER, REAL, DOUBLE PRECISION, or COMPLEX
v	is the symbolic name of a variable, an array, a constant, a function, a dummy procedure, or an array declarator.
len	is an unsigned integer constant that specifies the length, in bytes, of a variable, an array element, a symbolic constant, or a function.

The following _len_ specifiers are available:

1. LOGICAL*4 is the default for LOGICAL and occupies one numeric storage unit. The default may be changed to LOGICAL*2 with the "W" compiler option (see Chapter 2).

2. LOGICAL*2 data is a representation of the logical values of true and false. This type of logical data occupies one half of one numeric storage unit. A false value is represented by a field of sixteen zero bits and a true value is represented by a field of sixteen one bits.

3. LOGICAL*1 data is a representation of the logical values of true and false. This type of logical data occupies one byte. A false value is represented by a field of eight zero bits and a true value is represented by a field of eight one bits.

4. INTEGER*4 is the default for INTEGER and occupies one numeric storage unit. The default may be changed to INTEGER*2 with the "W" compiler option (see Chapter 2).

5. INTEGER*2 data is an exact binary representation of an integer in the range of -32768 to +32767 with negative integers carried in two's complement form. This type of integer is maintained in one half of one numeric storage unit.

6. INTEGER*1 data is an exact binary representation of an integer in the range of -128 to +127 with negative integers carried in two's complement form. This type of integer is maintained in one byte of storage.

7. REAL*4 is the default for REAL and occupies one numeric storage unit.

8. REAL*8 data is identical to DOUBLE PRECISION and occupies two numeric storage units.

9. COMPLEX*8 data is identical to COMPLEX and occupies two numeric storage units.

6.1.2 Character Type Statement

The form of the type statement for the character data type is:

```
        CHARACTER [*len [,]] v[*len] [,v[*len]]...
```

where:

- _v_ is a variable name, an array name, an array declarator, the symbolic name of a constant, a function name, or a dummy procedure name

- _len_ is either an unsigned integer constant, an integer constant expression within parentheses, or an asterisk within parentheses and specifies the length, in bytes, of a variable, an array element, a symbolic constant, or a function.

If _len_ directly follows the word CHARACTER, the length specification applies to all symbols not qualified by their own length specifications. When _len_ is not specified directly after the word CHARACTER, all symbols not qualified by their own length specifications default to one byte.

The length of symbolic character constants, dummy arguments of subroutine and function subprograms, and character functions may be given as an asterisk enclosed in parentheses: (*). The length of a symbolic constant declared in this manner is the number of characters appearing in the associated PARAMETER statement. Dummy arguments and functions assume the length of the actual argument declared by the referencing program unit.

```
        CHARACTER TITLE*(*)
        PARAMETER (TITLE = 'FORTRAN 77')
```

produces a ten byte symbolic character constant.

6.2 DIMENSION STATEMENT

The DIMENSION statement declares the names and supplies the dimension information for arrays to be used within a program unit.

```
        DIMENSION a(d) [,a(d)]...
```

where a(d) is an array declarator as described in Chapter 4.

Arrays may be declared with either DIMENSION statements, COMMON statements, or type statements, but multiple declarations are not allowed. That is, once a symbolic name has been declared to be an array it may not appear in any other declaration statement with an array declarator in the same program unit. The following three statements declare the same array:

1. `DIMENSION A(5,5,5)`
2. `REAL A(5,5,5)`
3. `COMMON A(5,5,5)`

6.3 COMMON STATEMENT

The COMMON statement is used to declare the storage order of variables and arrays in a consistent and predictable manner. This is done through a FORTRAN data structure called a common block, which is a contiguous block of storage. A common block may be identified by a symbolic name but does not have a data type. Once the order of storage in a common block has been established, any program unit that declares the same common block can reference the data stored there without having to pass symbol names through argument lists. Common blocks are specified in the following manner:

COMMON [/[cb]/] nlist [[,]/[cb]/ nlist]...

where:

cb	is the symbolic name of the common block. If cb is omitted, the first pair of slashes may also be omitted.
nlist	contains the symbolic names of variables, arrays, and array declarators.

When the common block name is omitted, the common block is called blank common. The symbolic name "BLANK" is reserved by the compiler for blank common and if used explicitly as a common block name will result in all entities in the nlist being placed in blank common.

Any common block name or an omitted name (blank common) can occur more than once in the COMMON statements in a program unit. The list of variables and arrays following each successive appearance of the same common block name is treated as a continuation of the list for that common block name.

A common block name can be the same as that of a variable, array, or program unit.

6.3.1 Named and Blank Common Differences

1. If a named common block was declared for the first time in a subroutine or function subprogram, the execution of a RETURN or END statement may cause the variables and arrays in it to become undefined. The SAVE statement (described later in this chapter) can be used to prevent this from occurring. Blank common is associated with the main program unit (whether or not it was actually declared there) and its variables and arrays can never become undefined.

2. Common blocks appearing in subprogram units which are separately compiled and either loaded as overlays or linked must be the same size in all program units where they are declared.

3. DATA statements in block data subprograms may be used to initially define entities in named common blocks, but not blank common.

6.4 EQUIVALENCE STATEMENT

The EQUIVALENCE statement provides a means for one or more variables to share the same storage location. Variables which share storage in this manner are said to be associated. The association may be total if both variables are the same size, or partial if they are of different sizes. The EQUIVALENCE statement is used in the following manner:

```
        EQUIVALENCE (nlist) [,(nlist)]...
```

The symbolic names of at least two variables, arrays, array elements, or character substrings must be specified in each nlist. Only integer constant expressions may be used in subscript and substring expressions. An array name unqualified by a subscript implies the first element of the array.

An EQUIVALENCE statement causes storage for all items in an individual nlist to be allocated at the same starting location:

```
        REAL A,B
        INTEGER I,J
        EQUIVALENCE (A,B), (I,J)
```

The variables A and B share the same storage location and are totally associated. The variables I and J share the same storage location and are totally associated.

Items which are equivalenced can be of different data types and have different lengths. When a storage association is established in this manner several elements of one data type may occupy the same storage as one element of a different data type:

```
        DOUBLE PRECISION D
        INTEGER I(2)
        EQUIVALENCE (D,I)
```

The array element I(1) shares the same storage location as the upper (most significant) thirty-two bits of D, and the array element I(2) shares the same storage location as the lower (least significant) thirty-two bits of D. Because only a portion of D is stored in the same location as I(1), these entities are only partially associated.

The EQUIVALENCE may not specify that an item occupy more than one storage location or that a gap occur between consecutive array elements.

6.4.1 Equivalence of Arrays

The EQUIVALENCE statement can be used to cause the storage locations of arrays to become either partially or totally associated.

```
        REAL A(8),B(8)
        INTEGER I(5),J(7)
        EQUIVALENCE (A(3),B(1)), (A(1),I(1)), (I(4),J(1))
```

Storage would be allocated as follows:

```
        | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10|
        |---------------A---------------|
                |---------------B---------------|
        |---------I---------|
                    |-------------J-------------|
```

6.4.2 Equivalence of Substrings

The EQUIVALENCE statement can be used to cause the storage locations of substrings to become either partially or totally associated.

```
        CHARACTER A(2)*5
        CHARACTER B*8
        EQUIVALENCE (A(2)(2:4),B(4:7))
```

Byte storage would be allocated as follows:

```
        | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10| 11|
        |-------------------A-------------------|
                    |---------------B---------------|
```

Notice that the lengths of the equivalenced substrings need not be the same, as in the above example.

6.4.3 Common and Equivalence Restrictions

The EQUIVALENCE statement can be used to increase the size of a common block by adding storage to the end, but it cannot increase the size by adding storage units prior to the first item in a common block.

The EQUIVALENCE statement must not cause two different common blocks to have their storage associated.

6.5 IMPLICIT STATEMENT

The IMPLICIT statement is used to establish implicit data typing that differs from the default integer and real typing described in Chapter 4. The IMPLICIT can also be used to remove implied typing altogether. The IMPLICIT statement takes the following form:

```
        IMPLICIT type [*len] (a [,a]...) [,typ [*len] (a [,a]...)]
```

where:

type	is a type chosen from the set CHARACTER, LOGICAL, INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or NONE.
len	is the length specifier applied to character, logical, integer, or real variables and is an unsigned, nonzero, integer constant.
a	is an alphabetic specifier which is either a single letter or a range of letters. A range of letters is specified with a character representing the lower bound of the range, a minus, and a character representing the upper bound of the range. The range A-Z specifies all the letters of the alphabet.

If len is not specified, the defaults are:

character	1 byte
logical	4 bytes
integer	4 bytes
real	4 bytes

The IMPLICIT statement must appear before all other declaration statements except PARAMETER statements and specifies the data type of all symbolic names that can take a data type that are not given one explicitly with a type statement. The data type will be the data type that corresponds to the first character of their names.

When NONE appears in place of a type specifier, all variables used within the program unit must appear in an explicit type statement. This transforms FORTRAN into a strongly typed language.

6.6 PARAMETER STATEMENT

The PARAMETER statement allows a constant to be given a symbolic name in the following manner:

```
        PARAMETER (p=c [,p=c]...)
```

where p is the symbolic name that is used to reference the constant and c is a constant expression.

If the data type and length attributes of the symbolic name are to be other than the implied default for that name, then the type (and size) must be previously defined in an explicit type statement or through the typing supplied by an IMPLICIT statement. A character parameter may have its length declared as an asterisk in parentheses, in which case the actual size will be the number of characters in the expression.

The type of the constant expression must match the type of the symbolic name.

```
        INTEGER EOF
        CHARACTER TITLE*(*)
        PARAMETER (PI=3.1415926, THIRD=1.0/3.0)
        PARAMETER (EOF=-1)
        PARAMETER (TITLE='FORTRAN 77')
```

6.6.1 Special use of the PARAMETER statement

As a means of defining character symbolic names with non-printing ASCII characters, a character symbolic name may be defined with an integer constant in the range of 0-255:

```
        CHARACTER EOL,EOF
        PARAMETER (EOL=10,EOF=O'377')
```

6.7 EXTERNAL STATEMENT

The EXTERNAL statement allows symbolic names to be used as arguments in CALL statements and function references without the compiler automatically creating a variable at the point of reference. Symbolic names so declared may or may not have an associated data type. The EXTERNAL statement is given with a list of external or dummy procedure names or instrinsic function names:

```
        EXTERNAL proc [,proc]...
```

where each proc is the symbolic name of a procedure or instrinsic function.

An intrinsic function name appearing in an EXTERNAL statement specifies that the particular instrinsic function has been replaced by a user supplied routine.

6.8 INTRINSIC STATEMENT

The INTRINSIC statement designates symbolic names as intrinsic functions (see Chapter 9). Similar to the EXTERNAL statement, this allows instrinsic functions to be used as arguments in CALL statements and function references without the compiler automatically creating a variable at the point of reference. The intrinsic function name specified in an INTRINSIC statement retains its associated data type. The INTRINSIC statement is given in the following manner:

INTRINSIC func [,func]...

where func is the name of an intrinsic function.

The following intrinsic functions are expanded in line and may not appear in an INTRINSIC statement: INT, IFIX, IDINT, FLOAT, SNGL, REAL, DBLE, CMPLX, ICHAR, CHAR, LGE, LGT, LLE, LLT, MAX, MAX0, AMAX1, DMAX1, AMAX0, MAX1, MIN, MIN0, AMIN1, DMAX1, AMIN0, MIN1, SHIFT, ADJUST, ADJUSTR, BYTE, WORD, and LONG.

The INSTRINSIC statement is used to pass instrinsic functions to external procedures:

```
        INTRINSIC SIN,COS
        DIMENSION A(100)
        CALL TRIG(SIN,A)
        CALL TRIG(COS,B)
        END

        SUBROUTINE TRIG(FUNC,ARRAY)
        DIMENSION ARRAY(100)
        DO 10 I=1,100
10      ARRAY(I) = FUNC(FLOAT(I))
        END
```

6.9 SAVE STATEMENT

FORTRAN 77 permits dynamic as well as static storage allocation. Variables, arrays, and common blocks declared in a main program are allocated statically and always retain their definition status. Variables, arrays, and common blocks which are declared only in subprograms are allocated dynamically when the subprogram is invoked. When the subprogram executes a RETURN or END statement, these items are deallocated and lose their definition status. The SAVE statement is used to retain the definition status of these items by specifying them in the following manner:

```
        SAVE [a [,a]...]
```

where:

a	is either a common block name delimited with slashes, a variable name, or an array name.

If a is not specified, the effect is as though all items in the program unit were presented in the list.

The variables and arrays in a named common block may become undefined if the common block was also declared in a higher level subprogram which subsequently executes a RETURN or END statement and does not contain a SAVE statement specifying the common block. This may occur even if the lower level routine contains a SAVE statement specifying the named common block.

6.10 VIRTUAL STATEMENT

This implementation of FORTRAN 77 supports the additional array declaration statement, VIRTUAL. This statement is used in exactly the same manner as the DIMENSION statement and specifies the symbolic names and dimension specifications of arrays to be located in external storage rather than internal storage. This allows for arrays of a far greater extent than would normally be available. The form of the VIRTUAL statement is:

```
        VIRTUAL a(d) [,a(d)]...
```

where each a(d) is an array declarator (see Chapter 4).

Each symbolic name appearing in a VIRTUAL statement declares it to be an array in that program unit. Unlike arrays declared for internal storage, a virtual array cannot be declared with either a type statement or a COMMON statement, however, a symbolic name appearing without an array declarator in a type statement may also be declared in a succeeding VIRTUAL statement. Obviously, a virtual array may not be associated by means of the COMMON statement or the EQUIVALENCE statement to any other variable.

The storage unit structure for virtual array elements is identical to that for internally stored data.

A virtual array is maintained in a direct access file on the device where execution of the program was initiated. The name of the file is the same as the symbolic name of the array with an extension of VRT. If the specified array does not exist as a file on the device it will be created. If the file already exists, but is of insufficient size to maintain the array, it will be deleted and a new file will be created.

6.10.1 Passing Virtual Arrays to External Procedures

A virtual array may be passed to a subroutine or function subprogram by declaring the array in a VIRTUAL statement with exactly the same name used to declare the array in the calling procedure.

The array name may appear in the actual argument list of the referencing procedure and in the subprogram dummy argument list. The appearance is not strictly necessary, but recommended to simplify conversions from a virtual array to a physical array or to use another compiler; changing the specification statement from VIRTUAL to DIMENSION is all that would be required.

6.10.2 Restrictions on Virtual Arrays

Some operating systems have an upper limit on the number of files that may be open at once. Since a virtual array is accessed as a file, such limitations should be considered in the design of a program which accesses a large combination of files and virtual arrays.

6.11 DATA STATEMENT

Variables, substrings, arrays, and array elements are given initial values with DATA statements. DATA statements may appear only after the declaration statements in the program unit in which they appear. DATA statements take the following form:

DATA _vlist_/_clist_/ [[,] _vlist_/_clist_/]...

where:

vlist contains the symbolic names of variables, arrays, array elements, substrings, and implied DO lists

clist contains the constants which will be used to provide the initial values for the items in _vlist_

A constant may be specified in _clist_ with an optional repeat specifier: a positive integer constant (or symbolic name of a constant) followed by an asterisk. The repeat specifier is used to indicate one or more occurrences of the same constant:

```
        DATA A,B,C,D,E/1.0,1.0,1.0,1.0,1.0/
```

can be written as:

```
        DATA A,B,C,D,E/5*1.0/
```

An array name unqualified by a subscript implies every element in the array:

```
        INTEGER M(5)
        DATA M/5*0/
```

means:

```
        INTEGER M(5)
        DATA M(1),M(2),M(3),M(4),M(5)/0,0,0,0,0/
```

Type conversion is automatically performed for arithmetic constants (integer, real, double precision, and complex) when the data type of the corresponding item in _vlist_ is different. Character constants are either truncated or padded with spaces when the length of the corresponding character item in _vlist_ is either shorter or longer than the constant respectively.

The items specified in _vlist_ may not be dummy arguments, functions, or items in blank common. Items in a named common block can be initialized only within a block data subprogram (see Chapter 9).

6.11.1 Implied DO List In A DATA Statement

An implied DO list is used to initialize array elements as though the assignments were within a DO loop. The implied DO list is of the form:

(_dlist_, _i_ = _m_1, _m_2 [,_m_3])

where:

dlist contains array elements and implied DO lists

_i_ is the DO variable and must be an integer

_m_1, _m_2, and _m_3 are integer constant expressions which establish the initial value, limit value, and increment value respectively (see Chapter 7)

```
INTEGER M(10,10),N(10),L(4)
CHARACTER*3 S(5)

DATA (N(I),I=1,10),((M(I,J),J=3,8),I=3,8)/5*1,5*2,36*99/
DATA (L(I),I=1,4)/'ABCD','EFGH','IJKL','MNOP'/
DATA (S(I),I=1,5)/'ABC','DEF','GHI','JKL','MNO'/
```

CHAPTER 7

CONTROL STATEMENTS

Control statements direct the flow of execution in a FORTRAN 77 program. Included in the control statements are constructs for looping, conditional and unconditional branching, making multiple choice decisions, and halting program execution.

7.1 GOTO STATEMENTS

7.1.1 Unconditional GOTO

The unconditional GOTO statement causes immediate transfer of control to a labeled statement:

```
        GOTO s
```

The statement label s must be in the same program unit as the GOTO statement.

7.1.2 Computed GOTO

The computed GOTO statement provides a means for transferring control to one of several different destinations depending on a specific condition:

```
        GOTO (s [,s]...) [,] e
```

e is an expression which is converted as necessary to integer and is used to select a destination from one of the statements in the list of labels (s [,s]...). The selection is made such that if the value of e is one, the first label is used, if the value of e is two, the second label is used, and so on. The same label may appear more than once in the label list. If the value of e is less than 1 or greater than the number of labels in the list no transfer is made. All of the statement labels in the list must be in the same program unit as the computed GOTO statement.

7.1.3 Assigned GOTO

The assigned GOTO statement is used with an integer variable which contains the address of a labeled statement as established with an ASSIGN statement:

```
        GOTO i [[,] (s [,s]...)]
```

The address of the labeled statement contained in the integer variable i is used as the destination. If the optional list of statement labels, (s [,s]...), appears then i must be defined with the address of one of them or no transfer is made.

7.2 IF STATEMENTS

7.2.1 Arithmetic IF

The arithmetic IF statement is used to transfer control based on the sign of the value of an expression:

```
        IF (e) s1 , s2 , s3
```

_e_ can be an integer, real, or double precision expression which if negative, transfers control to the statement labeled _s_1; if zero, transfers control to the statement labeled _s_2; and if positive, transfers control to the statement labeled _s_3. The statements labeled _s_1, _s_2, and _s_3 must be in the same program unit as the arithmetic IF statement.

7.2.2 Logical IF

The logical IF statement is used to execute another statement based on the value of a logical expression:

```
        IF (e) st
```

The statement _st_ is executed only if the value of the logical expression _e_ is true. The statement _st_ cannot be a DO, END DO, REPEAT, IF, ELSE IF, ELSE, END IF, SELECT CASE, CASE, END SELECT, END, or logical IF statement.

7.2.3 Block IF

A block IF consists of IF (e) THEN, ELSE, and END IF statements. Each IF (e) THEN statement must be balanced by an END IF statement. A block IF provides for the selective execution of a particular block of statements depending on the result of the logical expression e.

```
IF (e) THEN
  block of statements
ELSE
  block of statements
END IF
```

The ELSE statement and the second block of statements are optional. If the value of the logical expression e is true, the first block of statements is executed and then control of execution is transferred to the statement immediately following the END IF statement. If e has a false value, then, if a second block of statements exists (constructed by ELSE or ELSE IF statements) it is executed, and control of execution is transferred to the statement immediately following the END IF statement.

Each block of statements may contain more block IF constructs. Since each block IF must be terminated by an END IF statement there is no ambiguity in the execution path.

A more complicated block IF can be constructed using the alternate form of the ELSE statement: the ELSE IF (e) THEN statement. Multiple ELSE IF (e) THEN statements can appear within a block IF, each one being evaluated if the previous logical expression e has a false value:

```
IF (I.GT.0 .AND. I.LE.10) THEN
  block of statements
ELSE IF (I.GT.10 .AND. I.LE.100) THEN
  block of statements
ELSE IF (I.GT.100 .AND. I.LE.1000) THEN
  block of statements
ELSE
  block of statements
END IF
```

7.3 LOOP STATEMENTS

The DO statements provide the fundamental structure for constructing loops in FORTRAN 77. The basic DO loop, along with extensions available with this compiler; DO and DO WHILE, are discussed in this section.

7.3.1 Basic DO loop

The basic DO statement takes the following form:

```
        DO s [,] i = e1, e2 [,e3]
```

where:

- s is the label of the statement that defines the range of the DO loop and must follow the DO statement in the same program unit
- i is called the DO variable and must be either an integer, real, or double precision scalar variable
- e1, e2, and e3 may be integer, real, or double precision expressions whose values are called the initial value, the limit value, and the increment value, respectively

The loop termination statement, labeled s, must not be a DO, arithmetic IF, block IF, ELSE, END IF, unconditional GOTO, assigned GOTO, SELECT CASE, CASE, END SELECT, RETURN, STOP, or END statement.

DO loops may be nested to any level, but each nested loop must be entirely contained within the range of the outer loop. The termination statements of nested DO loops may be the same.

DO loops may appear within IF blocks and IF blocks may appear within DO loops, but each structure must be entirely contained within the enclosing structure.

7.3.1.1 DO Loop Execution

The following steps govern the execution of a DO loop:

1. The expression _e_1, the initial value, is evaluated and assigned to the DO variable _i_, with appropriate type conversion as necessary.

2. The expressions _e_2 and _e_3, the limit value and increment value respectively, are evaluated. If _e_3 is omitted, it is given the default value of one.

3. The _iteration count_ is calculated from the following expression:

   MAX(INT((_e_2 - _e_1 + _e_3)/_e_3), 0)

   and determines how many times the statements within the loop will be executed.

4. The iteration count is tested, and if it is zero, control of execution is transferred to the statement immediately following the loop termination statement.

5. The statements within the range of the loop are executed.

6. The DO variable is increased by the increment value, the iteration count is decreased by one, and control branches to step four.

Variables that appear in the expressions _e_1, _e_2, and _e_3 may be modified within the loop, without affecting the number of times the loop is iterated.

```
        K = 0
        L = 10
        DO 10 I=1, L
        DO 10 J=1, I
        L = J
10      K = K+1
```

When the execution of both the inner and outer loops is finished, the values of both I and J are 11, the value of K is 55, and the value of L is 10.

7.3.1.2 Transfer into Range of DO Loop

Under certain conditions, FORTRAN 66 permitted transfer of control into the range of a DO loop from outside the range. This was known as the "extended range of a DO". Such a transfer is considered highly unstructured and is prohibited in ANSI FORTRAN 77. However, in this implementation of FORTRAN 77, all DO loops may be considered extended range if the program is compiled with the _H_ option (see Chapter 2) selected.

7.3.2 DO WHILE

The DO WHILE statement is an extension to standard FORTRAN 77 and provides a method of looping not necessarily governed by an iteration count. The form of the DO WHILE statement is:

```
[DO [s[,]]] WHILE (e)
```

where:

_s_ is the statement label of an executable statement that defines the range of the loop. The statement identified by _s_ must follow the DO statement in the sequence of statements within the same program unit as the DO statement. If the label _s_ is omitted, the loop must be terminated with a REPEAT or END DO statement.

_e_ is a logical expression.

The DO WHILE statement tests the logical expression at the top of the loop. If the expression evaluates to a true value, the statements within the body of the loop are executed. If the expression evaluates to a false value, execution proceeds with the statement following the loop:

```
integer status,eof; parameter (eof=-1)

data a,b,c /3*0.0/

status = 0
  while (status<>eof)
    c = c + a*b
    read (*,*,iostat=status) a,b
  repeat
```

7.3.3 Block DO

The block DO extension to standard FORTRAN 77 provides three additional methods for constructing a loop. They are as follows:

```
1.      DO
          block
        REPEAT

2.      DO (i=e1, e2 [,e3])
          block
        REPEAT

3.      DO (e4 TIMES)
          block
        REPEAT
```

All three forms of block DO require a REPEAT or END DO statement to terminate the loop. An EXIT statement (described below) may be used to abnormally exit from the loop and a CYCLE statement (also described below) may be used to force iteration of the loop.

The first case is essentially a DO forever construct for use in situations where the number of loop iterations is unknown and must be determined from some external condition (i.e. processing text files).

The second case is identical to the standard DO loop without a terminating statement label. The value *i* is the DO variable, *e*1 is its initial value, *e*2 is its terminating value and *e*3, if present, is the increment value.

The value *e*4, in the third case, is the iteration count and may be an integer, real, or double precision expression. Where the value *e*4 is not an integer, it is first converted to an integer and the truncated value becomes the iteration count. At least one blank character must be present between the iteration count expression and the keyword TIMES.

7.3.4 END DO and REPEAT

The END DO and REPEAT statements are extensions to standard FORTRAN 77 and are used to terminate DO WHILE loops and block DO structures. Each block DO must have a matching END DO or REPEAT statement. After execution of an END DO or REPEAT statement, the next statement executed depends on the result of the DO loop incrementation processing. The form of the END DO and REPEAT statements is:

```
        END DO (or REPEAT)
```

7.3.5 EXIT

The EXIT statement is also an extension to standard FORTRAN 77 and provides a convenient means for abnormal termination of a DO loop. The EXIT statement causes control of execution to be transferred to the statement following the terminal statement of a DO loop or block DO.

```
          do
            read (*,*,iostat=ios) v1,v2; if (ios=-1) exit
            call process(v1,v2)
          repeat
```

7.3.6 CYCLE

The CYCLE statement is an extension to FORTRAN 77 and causes immediate loop index and iteration count processing to be performed for the DO loop or block DO structure to which the CYCLE statement belongs.

```
        read (*,*) n
        z = 0.0
          do (n times)
            read (*,*) x,y; if (y=0.0) cycle
            z = z + x/y
          repeat
```

7.4 CONTINUE STATEMENT

The CONTINUE statement is used to provide a reference point. It is usually used as the terminating statement of a basic DO loop, but it can appear anywhere in the executable section of a program unit. Executing the CONTINUE statement itself has no effect. The form of the CONTINUE statement is:

```
        CONTINUE
```

7.5 BLOCK CASE

The block CASE structure is an extension to the FORTRAN standard for constructing blocks which are executed based on comparison and range selection. The SELECT CASE statement is used with an END SELECT statement, at least one CASE statement and, optionally, a CASE DEFAULT statement to control the execution sequence. The SELECT CASE statement is used to form a block CASE.

The form of a block CASE is:

```
SELECT CASE (e)
  CASE (case selector)
    block
 [CASE (case selector)
    block
  ...]
 [CASE DEFAULT]
    block
END SELECT
```

where _e_ is an expression formed from one of the enumerative data types: character, integer, real, or double precision. For the purposes of the block case construct, the value of character expression is its position in the ASCII collating sequence.

A CASE block must contain at least one CASE statement and must be terminated by an END SELECT statement. Control of execution must not be transferred into a block CASE.

CASE blocks are delimited by a CASE statement and the next CASE, CASE DEFAULT, or END SELECT statement. A CASE block may be empty. After execution of a CASE block, control of execution is transferred to the statement following the END SELECT statement with the same CASE level. Block CASE structures may be nested. Since each block CASE must be terminated by an END SELECT statement there is no ambiguity in the execution sequence.

A case selector takes the form of either of the following:

```
        CASE (con[,con,...,con])

        CASE DEFAULT
```

con may be either a value selector or a range selector. A value selector is a constant. A range selector takes one of the following three forms:

```
        con1:con2   where  con1 .LE. e .LE. con2

        con:        where  con  .LE. e

        :con        where     e .LE. con
```

All constants must be of the same type as the expression e in the SELECT CASE statement. A block CASE may have only one CASE DEFAULT statement where control of execution is transferred if no match is found in any other CASE statement. If a CASE DEFAULT statement is not present and no match is found, a run time error is reported.

7.5.1 Execution of a block CASE statement

Execution of block CASE statement causes evaluation of the expression e in the SELECT CASE statement. An attempt is then made to match the value of the expression with the parameters of the case selectors. If a match is made, transfer of control is passed to that case block.

7.5.2 Block CASE Example

```
*
*     routine to count the number and types of characters
*     in a text file
*

      implicit integer(a-z)
      character line*80
      parameter (EOF=-1)

      lines=0; alf=0; num=0; blk=0; trm=0; spl=0

        do
          read (5,'(a)',iostat=ios) line
          if (ios=EOF) exit
          chars = len(trim(line))
          lines = lines+1
            do (i=1, chars)
              select case (line(i:i))
                case ("A":"Z","a":"z")
                  alf = alf+1
                case ("0":"9")
                  num = num+1
                case (" ")
                  blk = blk+1
                case (".","!","?")
                  trm = trm+1
                case default
                  spl = spl+1
              end select
            repeat
        repeat

      end
```

7.6 STOP STATEMENT

The STOP statement terminates execution of a program:

```
        STOP [s]
```

The optional string s may be a character constant or string of five or fewer digits and is output to unit 9 with end of record characters.

7.7 PAUSE STATEMENT

The PAUSE statement suspends execution of a program until the RETURN key is pressed on the terminal connected to unit 9 (see Chapter 8):

```
        PAUSE [s]
```

The optional string s may be a character constant or string of five or fewer digits and is output to unit 9 without end of record characters.

7.8 END STATEMENT

Every program unit must have an END statement which terminates the range of individual program units within a source file. A source file itself may contain more than one program unit; the entry points of the individual program units in the compiled object file are available to the linker. However, only the first program unit can be used as an entry point if the file is loaded as an overlay.

An END statement is executable and if encountered in a main program has the effect of a STOP statement and if encountered in a subroutine or function subprogram has the effect of a RETURN statement. An END statement is given on a statement line by itself with no other characters:

```
        END
```

7.9 EXECUTE STATEMENT

As an extension to FORTRAN 77, a statement is provided for chaining to another program, similar to the CHAIN statement found in other languages. The form of the EXECUTE statement is as follows:

```
        EXECUTE arg
```

where: arg is one of the following:

1. A character expression
2. A character variable name
3. A character substring
4. A character array element

The argument must contain a complete command line acceptable to the operating system. The EXECUTE statement has the effect of a STOP statement except that any open files are not implicitly closed. All files should be closed before an EXECUTE statement is executed to insure that buffers are flushed and scratch files are deleted.

Note: Under UNIX, the argument of the EXECUTE statement is the file name of a script which is passed to the shell.

CHAPTER 8

INPUT/OUTPUT AND FORMAT SPECIFICATION

Input and output statements provide a channel through which FORTRAN 77 programs can communicate with the outside world. Facilities are available for accessing disk and tape files, communicating with terminals and printers, and controlling external devices. FORTRAN 77 input and output statements are designed to allow access to the wide variety of features implemented on various computer systems in the most portable manner possible.

A _format specification_ is used with formatted input and output statements to control the appearance of data on output and provide information regarding the type and size of data on input. Converting the internal binary representation of a floating point number into a string of digits requires a format specification and is called editing. A format specification divides a record into fields, each field representing a value. An explicitly stated format specification designates the exact size and appearance of values within fields.

When an asterisk (*) is used as a format specification it means "list directed" editing. Instead of performing editing based on explicitly stated formatting information, data will be transferred in a manner which is "reasonable" for its type and size.

Throughout the remainder of this chapter, input and output will be referred to in the conventional abbreviated form: I/O.

8.1 RECORDS

All FORTRAN I/O takes place through a data structure called a record. A record can be a single character or sequence of characters or values. A record might be a line of text, the data received from a bar code reader, the coordinates to move a plotter pen, or a punched card. FORTRAN uses three types of records:

* Formatted
* Unformatted
* Endfile

8.1.1 Formatted Record

A formatted record is a sequence of ASCII characters. It may or may not be terminated depending on the operating system. If it is terminated, the usual terminating characters are either a carriage return, a line feed, or both. A single line of text on this page is a formatted record. The minimum length of a formatted record is zero and the maximum length is 1024.

8.1.2 Unformatted Record

An unformatted record is a sequence of values. Its interpretation is dependent on the data type of the value. For example, the binary pattern 01010111 can be interpreted as the integer value 87 or the character value "W" depending on its data type. The minimum length of an unformatted record is zero and the maximum length is 1024 except for unformatted sequential access records containing no record length information (see below) which have unlimited length.

8.1.3 Endfile Record

The endfile record is the last record of a file and has no length. An endfile record may or may not be an actual record depending on the file system of a particular operating system.

8.2 FILES

A file is composed of zero or more records and can be created and accessed by means other than FORTRAN 77 programs. For example, a text processor might be used to create and edit a document file and a FORTRAN 77 program used to manipulate the information in the file.

Files which are usually stored on disks or tapes are called _external_ files. Files can also be maintained in main memory. These are called _internal_ files.

8.2.1 File Name

Most external files are accessed explicitly by their names. While the file naming conventions of operating systems vary greatly, FORTRAN 77 can accommodate most of the differences. The circumstances where a name is not required to access a file are discussed later in this chapter.

8.2.2 File Position

The position within a file refers to the next record that will be read or written. When a file is opened it is usually positioned to just before the first record. The end of the file is just after the last record. Some of the I/O statements allow the current position within a file to be changed.

8.2.3 File Access

The method used to transfer records to and from files is called the access mode. External files may contain either formatted or unformatted records. When the records in a file can be read or written in an arbitrary manner, randomly, the access mode is _direct_. Individual records are accessed through a _record number_, a positive integer. All of the records in a direct access file have the same length and contain only the data actually written to them; there are no record termination characters. Records may be rewritten, but not deleted. Generally, only disk files can use the direct access mode of record transfer.

When the records are transferred in order, one after another, the access mode is _sequential_. The records in a sequential access file may be of different lengths. Some files, like terminals, printers, and tape drives, can only use the sequential access mode.

Formatted sequential access files usually contain textual information and each record has a terminating character(s) as described above.

Unformatted sequential access is generally used for two conflicting, but equally common purposes:

1. For controlling external devices such as plotters, graphics terminals, and machinery as well as processing unencoded binary information such as object files. In this case it is important that the data transferred to and from the external media be a true byte stream containing no record length information.

2. For its data compression and speed of access characteristics. In this case it must be possible to determine the length of a record for partial record reads and backspacing purposes.

This implementation of FORTRAN 77 contains provisions for both of these requirements. The default manner of unformatted processing of a sequential access device is to treat it as a pure byte stream. Partial record reads and backspacing are not possible. The data transmitted is exactly what your WRITE statement specifies or what the external media contains. There is no limit on the length of a record.

When partial record reads and backspacing of unformatted sequential files are required, the runtime system is informed by adding the "BLOCK=-1" specifier to the OPEN statement. The BLOCK specifier is an extension to the ANSI standard normally used to specify the blocking factor applied to magnetic tape. When a file is opened for unformatted sequential access and this specifier is negatively valued, each record written will be preceded and followed by two bytes containing the length of the record. The maximum record length is 1024 bytes.

8.2.4 Internal File

Internal files are comprised of character variables, character array elements, character substrings, or character arrays. An internal file which is a character variable, character array element, or character substring has one record whose length is the length of the character entity. An internal file which is a character array has as many records as there are array elements. The length of an individual record is the length of a character array element. Data may only be transferred through the formatted sequential access mode. Internal files are usually used to convert variables between numeric and character data types.

8.3 I/O SPECIFIERS

FORTRAN 77 I/O statements are formed with lists of specifiers that are used to identify the parameters of the operation and direct the control of execution when exceptions occur.

8.3.1 Unit Specifier

The mechanism through which a channel of communication with a file is established and maintained is called a unit. A unit may be either explicitly or implicitly identified, and may refer to an external or internal file. When the channel is established, the unit is said to be connected to the file. The relationship is symmetric; that is, you can also say that the file is connected to the unit.

A connection to an external file is established and maintained with an <u>external unit identifier</u> which is an integer expression whose value is an arbitrary positive integer. An external unit identifier is global to the program; a file opened in one program unit may be referenced with the same unit number in other program units. There is no relationship between a FORTRAN unit specifier and the numbers used by various operating systems to identify files.

A connection to an internal file is made with an <u>internal file identifier</u> which is the name of the character variable, character array element, character substring, or character array that comprises the file.

Some units are "preconnected" to files:

1. Unit 5 is preconnected to a file specified on the command line and provides a method for simulating the card reader preconnections of older FORTRAN programs.

2. Unit 6 is preconnected to a file which will be spooled to the line printer when the program finishes execution.

3. Unit 7 is preconnected to a magnetic tape drive.

4. Unit 9 is preconnected to the terminal device for both input and output operations.

5. An asterisk as a unit identifier by default refers to unit 5 for input operations and unit 6 for output operations. The default may be changed to unit 9 with the "U" compiler option (see Chapter 2).

A unit specifier is given as:

```
        [UNIT=] u
```

where _u_ is either a positive integer expression representing an external unit identifier, or a character entity representing an internal file identifier.

The characters "UNIT=" may be omitted if the unit identifier occurs first in the list of identifiers.

8.3.2 Format Specifier

The format specifier establishes the method of converting between internal and external representations. It can be given in one of two ways:

```
        [FMT=] f
```

or

```
        [FMT=] *
```

where:

_f_ is the statement label of a FORMAT statement, an integer variable that has been assigned a FORMAT statement label with an ASSIGN statement, a character array name, or any character expression

_*_ indicates "list directed" editing

The characters "FMT=" may be omitted if the format specifier occurs second in the list of identifiers and the first item is the unit specifier with the characters "UNIT=" also omitted. The following are equivalent:

```
        WRITE (UNIT=9, FMT=1000)
        WRITE (9,1000)
```

8.3.3 Record Specifier

The record specifier establishes which direct access record is to be accessed and is given as:

```
        REC = rn
```

where _rn_ is a positive integer expression.

8.3.4 Error Specifier

The error specifier provides a method to transfer control of execution to a different section of the program unit if an error condition occurs during an I/O statement. It takes as an argument the label of the statement where control is to be transferred:

```
        ERR = s
```

where s is the statement label.

8.3.5 End of File Specifier

The end of file specifier provides a method to transfer control of execution to a different section of the program unit if an end of file condition occurs during an I/O statement. It also takes as an argument the label of the statement where control is to be transferred:

```
        END = s
```

where s is the statement label.

8.3.6 I/O Status Specifier

The I/O status specifier is used to monitor error and end of file conditions after the completion of an I/O statement. Its associated integer variable becomes defined with a -1 if end of file has occurred, a positive integer if an error occurred, and zero if there is neither an error nor end of file condition:

```
        IOSTAT = ios
```

where ios is the symbolic name of an INTEGER*4 variable or array element.

8.4 I/O LIST

The I/O list, _iolist_, contains the names of variables, arrays, array elements, and expressions (only in output statements) whose values are to be transferred with an I/O statement. The following items may appear in an _iolist_:

* A variable name

* An array element name

* A character substring name

* An array name which is interpreted as every element in the array

* Any expression (only in an output statement)

8.4.1 Implied DO List In An I/O List

The elements of an _iolist_ in an implied DO list are transferred as though the I/O statement was within a DO loop. An implied DO list is stated in the following manner:

```
( dlist, i = e1, e2 [,e3 ] )
```

where:

_i_ is the DO variable

_e_1, _e_2, and _e_3 establish the initial value, the limit value, and increment value respectively (see Chapter 7).

dlist is an _iolist_ and may consist of other implied DO lists

In a READ statement (see below), the DO variable, _i_, must not occur within _dlist_ except as an element of subscript, but may occur in the _iolist_ prior to the implied DO list.

8.5 READ, ACCEPT, WRITE, PRINT, AND TYPE STATEMENTS

The READ and ACCEPT statements transfer data from files into storage and the WRITE, PRINT, and TYPE statements transfer data from storage to files. They are collectively called data transfer statements and cause one or more records to be transferred. The data transfer statements are:

```
READ (cilist) [iolist]

READ f [,iolist]

ACCEPT [iolist]

WRITE (cilist) [iolist]

PRINT f [,iolist]

TYPE (cilist) [iolist]

TYPE f [,iolist]

TYPE c
```

where:

f	is a format identifier
iolist	is an I/O list
c	is a character constant
cilist	is a parameter control list that may contain:

1. A unit specifier identifying the file connection.
2. An optional format specifier for formatted data transfers.
3. An optional record specifier for direct access connections.
4. An optional error specifier directing the execution path in the event of error occurring during the data transfer operation.
5. An optional end of file specifier directing the execution path in the event of end of file occurring during the data transfer operation.
6. An optional I/O status specifier to monitor the error or end of file status.

The PRINT statement and READ statements that do not contain a _cilist_ implicitly use the asterisk as a unit identifier.

The ACCEPT statement and TYPE statements that do not contain a _cilist_ implicitly use 9 as the unit identifier with "list directed" editing.

8.5.1 Unformatted Data Transfer

Unformatted data transfer is permitted only to external files. Only one unedited record is transferred per data transfer statement.

8.5.2 Formatted Data Transfer

Formatted data transfer requires a format specifier which directs the interpretation applied to items in the _iolist_. Formatted data transfer causes one or more records to be transferred.

8.5.3 Printing

WRITE, PRINT, and TYPE statements specifying unit 6, the line printer spooler file, use the first character of each record to control vertical spacing. This character, called the carriage control character, is not printed and causes the following veritical spacing to be performed before the record is output:

Character	Vertical Spacing
blank	one line
0	two lines
1	top of page
+	no advance (over print)

Any other character appearing in the first position of record or a record containing no characters causes vertical spacing of one line.

8.6 OPEN STATEMENT

The OPEN statement either connects a unit to an existing file or creates a file and connects a unit to it. The OPEN statement has the following form:

```
        OPEN ([UNIT=] u [,olist])
```

where [UNIT=] is the external unit specifier and the optional list, *olist*, consists of zero or more of the following specifiers, each of which must have a variable or constant following the equals sign:

IOSTAT=	an I/O status specifier as described above (see Appendix D for a list of error codes).
ERR=	an error specifier as described above.
FILE=	a character expression which represents the name of the file to be connected to the unit. If this specifier is omitted a file name will be created.
STATUS=	a character expression which must be OLD, NEW, SCRATCH, or UNKNOWN. The file must already exist when OLD is specified. If NEW is specified a file is created and if the file already exists, it will be first deleted. If SCRATCH is specified a file will be created which will exist only during the execution of the program and FILE= must not specified. If UNKNOWN is specified, a file will be created if one does not already exist. The default value is UNKNOWN.
ACCESS=	a character expression which must be SEQUENTIAL or DIRECT and specifies the access mode. The default value is SEQUENTIAL.
FORM=	a character expression which must be FORMATTED or UNFORMATTED specifying the type of records in the file. The default value is UNFORMATTED for direct access files and FORMATTED for sequential access files.
RECL=	a positive integer expression which must be given for direct access file connections and specifies the length of each direct access record. The specifier may be given for connections involving magnetic tape devices in which case its value indicates a fixed length record.

BLANK=	a character expression which must be NULL or ZERO specifying how blank characters in formatted numeric input fields are to be handled. A value of ZERO causes blanks in the input field (leading, embedded, and trailing) to be replaced with zeros. The default value is NULL and causes blanks to be ignored.
SIZE=	a positive integer expression specifying the number of disk blocks to allocate for direct access files opened with a NEW status specifier. This specifier is only recognized on operating systems that distinguish between direct access and sequential access files. The default is 100.
POSITION=	a character expression which must be REWIND, APPEND, or ASIS. If REWIND is specified the file is opened at its beginning position for input or output. If APPEND is specified, the file is opened at its end position for output. The default is ASIS and has the same effect as REWIND for sequential disk files and has no effect for tape devices.
ACTION=	a character expression which must be READ, WRITE, or BOTH. If READ is specified, only READ statements and file positioning statements are allowed to refer to the connection. If WRITE is specified, only WRITE statements and file positioning statements are allowed to refer to the connection. If BOTH is specified, any input/output statement may be used to refer to the connection. The default is BOTH.
BLOCK=	a positive integer expression specifying the number of bytes in a data block (between IRGs) on a magnetic tape file connection.

If a unit is already connected to a file, the connection is first terminated before the new connection is established.

8.7 CLOSE STATEMENT

The CLOSE statement disconnects a file from a unit. The CLOSE statement has the following form:

```
        CLOSE ([UNIT=] u [,clist])
```

where [UNIT=] is the external unit specifier and the optional list, clist, consists of zero or more of the following specifiers, each of which must have a variable or constant following the equals sign:

IOSTAT=	an I/O status specifier as described above (see Appendix D for a list of error codes).
ERR=	an error specifier as described above.
STATUS=	a character expression which must be KEEP or DELETE which determines whether a file will continue to exist after it has been closed. STATUS has no effect if the value of the STATUS specifier in the OPEN statement was SCRATCH. The default value is KEEP.

Normal termination of execution of a FORTRAN 77 program causes all units that are connected to be closed.

8.8 BACKSPACE STATEMENT

The BACKSPACE statement causes the file pointer to be positioned to a point just before the previous record. The forms of the BACKSPACE statement are:

```
        BACKSPACE u
        BACKSPACE ([UNIT=] u [,alist])
```

where [UNIT=] is the external unit specifier and the optional list, alist, consists of zero or more of the following specifiers:

IOSTAT=	an I/O status specifier as described above.
ERR=	an error specifier as described above.

8.9 REWIND STATEMENT

The REWIND statement causes the file pointer to be positioned to a point just before the first record. The forms of the REWIND statement are:

```
        REWIND u
        REWIND ([UNIT=] u [,alist])
```

where [UNIT=] is the external unit specifier and the optional list, alist, consists of zero or more of the following specifiers:

IOSTAT=	an I/O status specifier as described above.
ERR=	an error specifier as described above.

8.10 ENDFILE STATEMENT

The ENDFILE statement writes an endfile record to magnetic tape files and does nothing to disk files. The forms of the ENDFILE statement are:

```
        ENDFILE u
        ENDFILE ([UNIT=] u [,alist])
```

where [UNIT=] is the external unit specifier and the optional list, alist, consists of zero or more of the following specifiers:

IOSTAT=	an I/O status specifier as described above.
ERR=	an error specifier as described above.

8.11 INQUIRE STATEMENT

The INQUIRE statement is used to obtain information regarding the properties of files and units. The forms of the INQUIRE statement are:

```
        INQUIRE (UNIT= u, ilist)
        INQUIRE (FILE= fin, ilist)
```

The first form, inquiry by unit, takes a unit number as the principal argument and is used for making inquiries about specific units. The unit number, u, is a positive integer expression. The second form, inquiry by file, takes a file name as the principal argument and is used for making inquiries about specific named files. The file name, fin, is a character expression. Only one of "UNIT=" or "FILE=" may be specified. One or more of the following ilist specifiers are also used with the INQUIRE statement:

IOSTAT=	an I/O status specifier as described above.
ERR=	an error specifier as described above.
EXIST=	a logical variable or array element which is defined with a true value if the unit or file exists.
OPENED=	a logical variable or array element which is defined with a true value if the unit or file is connected.
NUMBER=	an integer variable or array element which is defined with the number of the unit that is connected to the file.
NAMED=	a logical variable or array element which is defined with a true value if the file has a name.
NAME=	a character variable or array element which is defined with the name of the file.
ACCESS=	a character variable or array element which is defined with either the value SEQUENTIAL or DIRECT depending on the access mode.
SEQUENTIAL=	a character variable or array element which is defined with the value YES or NO indicating whether the file can be connected for sequential access.

DIRECT=	a character variable or array element which is defined with the value YES or NO indicating whether the file can be connected for direct access.
FORM=	a character variable or array element which is defined with either the value FORMATTED or UNFORMATTED depending on whether the file is connected for formatted or unformatted I/O.
FORMATTED=	a character variable or array element which is defined with the value YES or NO indicating whether the file can be connected for formatted I/O.
UNFORMATTED=	a character variable or array element which is defined with the value YES or NO indicating whether the file can be connected for unformatted I/O.
RECL=	an integer variable or array element which is defined with the record length if the file is connected for direct access.
NEXTREC=	an integer variable or array element which is defined with the value of the next record number to be read or written.
BLANK=	a character variable or array element which is defined with either the value NULL or ZERO depending on how blanks are handled.
SIZE=	an integer variable or array element which is defined with the size of the file in bytes.

Some of the specifiers may not be defined if a unit is not connected or a file does not exist. For example:

```
        CHARACTER*20 FN,AM
        LOGICAL OS
        INTEGER RL
        INQUIRE (UNIT=18, OPENED=OS, NAME=FN, ACCESS=AM, RECL=RL)
```

If unit 18 is not connected to a file, OS will be defined with a false value, but FN, AM, and RL will be undefined. If unit 18 is connected for sequential access, OS, FN, and AM will be defined appropriately, but record length is meaningless in this context, and RL will be undefined.

8.12 GIVING A FORMAT SPECIFICATION

An explicit format specification may be given in either a FORMAT statement or in a character array or character expression. A FORMAT statement must be labeled so that it can be referenced by the data transfer statements (READ, WRITE, PRINT, etc.). The form of the FORMAT statement is:

FORMAT *format specification*

When a format specification is given with a character array or character expression (character variables, array elements, and substrings are simple character expressions) it appears as a format specifier in the *cilist* of data transfer statements as described later in this chapter. An array name not qualified by subscripts produces a format specification which is the concatenation of all of the elements of the array. Leading and trailing blanks within the character item are not significant.

A format specification is given with an opening parenthesis, an optional list of *edit descriptors*, and a closing parenthesis. A format specification may be given within a format specification; that is, it may be nested. When a format specification is given in this manner it is called a group specifier and can be given a repeat count, called the group repeat count, which is a positive integer constant immediately preceding the opening parenthesis. The maximum level of nesting is twenty.

The edit descriptors define the fields of a record and are separated by commas except between a P edit descriptor and an F, E, D, or G edit descriptor and before or after slash and colon edit descriptors (see below). The fields defined by edit descriptors have an associated width, called the *field width*.

An edit descriptor is either repeatable or nonrepeatable. Repeatable means that the edit descriptor is to be used more than once before going on to the next edit descriptor in the list. The repeat factor is given immediately before the edit descriptor as a positive integer constant.

The repeatable edit descriptors and their meanings are:

Iw and Iw.m	integer editing
Bw and Bw.m	binary editing
Ow and Ow.m	octal editing
Zw and Zw.m	hexadecimal editing
Fw.d	floating point editing
Ew.d and Ew.dEe	single precision scientific editing
Dw.d	double precision scientific editing
Gw.d and Gw.dEe	general floating point editing
Lw	logical editing
A[w]	character editing

w and e are nonzero, unsigned, integer constants and d and m are unsigned integer constants.

The nonrepeatable edit descriptors and their meanings are:

'h1 h2 ... hn'	character string
"h1 h2 ... hn"	character string
nHh1 h2 ... hn	Hollerith string
nX	skip positions
Tc, TLc, and TRc	tab to column
kP	set scale factor
/	start a new record
:	conditionally terminate I/O
S, SP, and SS	set sign control
BZ and BN	set blank control

h is an ASCII character; n and c are nonzero, unsigned, integer constants; and k is an optionally signed integer constant.

8.13 FORMAT SPECIFICATION AND I/O LIST INTERACTION

During formatted data transfers, the I/O list items and the edit descriptors in the format specification are processed in parallel, from left to right. The I/O list specifies the variables that are transferred between memory and the fields of a record, while the edit descriptors dictate the conversions between internal and external representations.

The repeatable edit descriptors control the transfer and conversion of I/O list items. A repeatable edit descriptor or format specification preceded by a repeat count, *r*, is treated as *r* occurrences of that edit descriptor or format specification. Each repeatable edit descriptor controls the transfer of one item in the I/O list except for complex items which require two F, E, D, or G edit descriptors. A complex I/O list item is considered to be two real items.

The nonrepeatable edit descriptors are used to manipulate the record. They can be used to change the position within the record, skip one or more records, and output literal strings. The processing of I/O list items is suspended while nonrepeatable edit descriptors are processed.

If the end of the format specification is reached before exhausting all of the items in the I/O list, processing starts over at the beginning of the last format specification encountered and the file is positioned to the beginning of the next record. The last format specification encountered may be a group specifier, if one exists, or it may be the entire format specification. If there is a repeat count in front of a group specifier it is also reused.

8.14 INTEGER EDITING

The I, B, O, and Z edit descriptors control the translation of character strings representing integer values to and from the appropriate internal formats.

8.14.1 I Editing

The I_w_ and I_w_._m_ edit descriptors must correspond to an integer I/O list item. The field width in the record consists of _w_ characters.

On input, the I/O list item will be defined with the value of the integer constant in the input field which may have an optional leading sign.

The output field consists of a string of digits representing the integer value which is right justified and may have a leading minus sign if the value is negative. If _m_ is specified, the string will consist of at least _m_ digits with leading zeros as required. The output field will always contain at least one digit unless an _m_ of zero is specified in which case only blank characters will be output. If the specified field width is too small to represent the integer value, the field is completely filled with the asterisk character.

```
        WRITE (9,10) 12, -12, 12
10      FORMAT (2I4,I6.4)

  12 -12  0012
```

8.14.2 B, O, and Z Editing

The B, O, and Z edit descriptors are specified in the same manner as the integer edit descriptor and perform bit editing on binary, octal, and hexadecimal fields respectively. The field width in the record consists of w characters. An input list item can be up to sixty-four bits in length and may have a logical, integer, real, double precision, or complex data type. An output list value can be no longer than thirty-two bits in length and may have a logical, integer, real, or complex data type.

On input, the I/O list item will be defined with the binary representation of the external value.

The output field consists of a string of characters representing the value and is right justified. If m is specified, the string will consist of at least m digits with leading zeros as required. The output field will always contain at least one digit unless an m of zero is specified in which case only blank characters will be output.

```
        WRITE (9,10) 199, 199, 199
10      FORMAT (Z4,O7.6,B9)

  C7 000307 11000111
```

8.15 FLOATING POINT EDITING

The F, E, D, and G edit descriptors control the translation of character strings representing floating point values (real, double precision, and complex) to and from the appropriate internal formats. The edit descriptor must correspond to a floating point I/O list item. On input, the I/O list item will be defined with the value of the floating point constant in the input field.

A complex value consists of a pair of real values and consequently requires two real edit descriptors.

8.15.1 F Editing

The field width of the F_w_._d_ edit descriptor consists of _w_ characters. The fractional portion, if any, consists of _d_ characters. If the specified field width is too small to represent the value, the field is completely filled with the asterisk character.

The input field consists of an optional sign and a string of digits which can contain a decimal point. This may be followed by an exponent which takes the form of either a signed integer constant or the letter E or D followed by an optionally signed integer constant.

The output field consists of a minus sign if the value is negative and a string of digits containing a decimal point with _d_ fractional digits. The value is rounded to _d_ fractional digits and the string is right justified in the field. If the value is less than 1.0, and the field width permits, a leading zero will be incorporated in the string. The position of the decimal point may be modified by the scale factor as described under the _k_P edit descriptor.

```
        WRITE (9,10) 1.23, -1.23, 123.0, -123.0
10      FORMAT (2F6.2,F6.1,F6.0)

  1.23 -1.23 123.0 -123.
```

8.15.2 E and D Editing

The field width of the E*w*.*d*, E*w*.*d*E*e*, and D*w*.*d* edit descriptors consists of *w* characters in scientific notation. *d* specifies the number of significant digits. If *e* is specified, the exponent contains *e* digits, otherwise, the exponent contains two digits for E editing and three digits for D editing.

The input field is identical to that specified for F editing.

The output field consists of a minus sign if the value is negative, a zero, a decimal point, a string of *d* digits, and an exponent whose form is specified in the table below. The value is rounded to *d* fractional digits and the string is right justified in the field. The position of the decimal point may be modified by the scale factor as described under the *k*P edit descriptor.

Edit Descriptor	Absolute value of Exponent	Form of Exponent
E*w*.*d*	$\leq$ 99	E$\pm$nn
E*w*.*d*	100 - 309	$\pm$nnn
E*w*.*d*E*e*	$\leq$ (10***e*)-1	E$\pm$n1n2...n*e*
D*w*.*d*	$\leq$ 99	D$\pm$nn
D*w*.*d*	100 - 309	$\pm$nnn

```
      WRITE (9,10) 1.23, -1.23, -123.0E-6, .123D3
10    FORMAT (2E12.4,E12.3E3,D12.4)

  0.1230E+01 -0.1230E+01 -0.123E-003  0.1230D+03
```

8.15.3 G Editing

The Gw.d and Gw.dEe edit descriptors are similar to the F and E edit descriptors and provide a flexible method of accomplishing output editing.

The input field is identical to that specified for F editing.

The form of the output field depends on the magnitude of the value in the I/O list. F editing will be used unless the value of the item would cause the field width to be exceeded in which case E editing is used. In both cases, the field consists of w right justified characters.

Magnitude of N	Equivalent Conversion
N < 0.1	Ew.d
0.1 ≤ N < 1.0	F(w-4).d, 4X
1.0 ≤ N < 10.0	F(w-4).(d-1), 4X
.	.
.	.
.	.
10**(d-2) ≤ N < 10**(d-1)	F(w-4).1, 4X
10**(d-1) ≤ N < 10**d	F(w-4).0, 4X
N ≥ 10**d	Ew.d[Ee]

```
        WRITE (9,10) 1.0, 10.0, 100.0, 1000.0, 10000.0
10      FORMAT (5G10.4)

 1.000     10.00     100.0     1000.     0.1000E+05
```

8.15.4 P Editing

The _k_P edit descriptor is used to scale floating point values edited with the F, E, D, and G edit descriptors. _k_ is called the _scale factor_ and is given as an integer constant which may negtive, positive, or zero. The scale factor starts at zero for each formatted I/O statement.

If there is an exponent in the input field, the scale factor has no effect, otherwise the external value is equal to the internal value multiplied by 10**_k_.

For output with F editing, the effect of the scale factor is the same as described for input. For E and D editing, the scale factor is used to control decimal normalization of the output value. If _k_ is negative, leading zeros are inserted after the decimal point, the exponent is reduced by _k_, and |_k_| significant digits are lost. If _k_ is positive, the decimal point is moved to the right within the _d_ significant digits, the exponent is reduced by _k_, and no significant digits are lost. The field width remains constant in all cases, meaning that $-d < k < d + 2$.

```
        WRITE (9,10) 1.23, 1.23, 1.23
10      FORMAT (1PF8.4,-1PF8.4,1PE12.4)
```

```
 12.3000   .1230  1.2300E+00
```

8.16 CHARACTER AND LOGICAL EDITING

The A and L edit descriptors control the translation of character strings representing character and logical values to and from the appropriate internal formats.

8.16.1 A Editing

The A[w] edit descriptor is used to copy characters (bytes) to and from I/O list items. If present, w specifies the field width; otherwise the field width is the same as the length of the I/O list item. The only editing performed is to space fill or truncate for input and output respectively.

For input, when w is less than the length of the I/O list item, the characters from the field are left justified and space filled to the length of the item. When w is equal to or greater than the length of the I/O list item, the rightmost characters in the field are used to define the item.

For output, when w is less than or equal to the length of the I/O list item, the field will contain the leftmost w characters of the item. When w is greater than the length of the I/O list item, the item is right justified in the field with leading spaces added as necessary.

```
        WRITE (9,10) 'HELLO, WORLD ', ',', 'WORLD'
10      FORMAT (A5,A,A6)
```

```
HELLO, WORLD
```

8.16.2 L Editing

The Lw edit descriptor must correspond to a logical I/O list item. The field width in the record consists of w characters.

The input field consists of an optional decimal point and either the letter T (true) or F (false). Other characters may follow, but they do not take part in determining the logical value. The field may contain leading spaces.

The output field is right justified and contains either the letter T or F representing the values true and false, respectively.

```
        WRITE (9,10) .TRUE., .FALSE.
10      FORMAT (2L2)
```

```
 T F
```

8.17 SIGN CONTROL EDITING

The S, SP, and SS edit descriptors control the output of optional plus signs. Normally, a leading plus sign is not output for positive numeric values. The SP edit descriptor forces a plus sign to appear in the output field. The S and SS edit descriptors return the processing of plus signs to the default state of not being output.

```
        WRITE (9,10) 123, -123, 123.0, -123.0, 123.0
10      FORMAT (SP,2I5,2F7.1,SS,F7.1)

 +123 -123 +123.0 -123.0  123.0
```

8.18 BLANK CONTROL EDITING

The BN and BZ edit descriptors control the processing of blanks in numeric input fields which can be interpreted either as nulls or zeros. The default for an individual file connection is established with the "BLANK=" specifier. If the specifier does not appear in an OPEN statement blanks are treated as nulls. The BN edit descriptor causes blanks to be treated as nulls and the BZ edit descriptor causes blanks to be treated as zeros.

8.19 POSITIONAL EDITING

The X, T, and / edit descriptors are used to control the position within the record and the position within the file.

8.19.1 X Editing

The nX edit descriptor moves the position within the record n characters forward. On input n characters are bypassed in the record. On output n blanks are output to the record.

```
        WRITE (9,10) -123, -123.0
10      FORMAT (I4,1X,F6.1)

-123 -123.0
```

8.19.2 T, TL, and TR Editing

On output, the entire record is first filled wih spaces. The Tc, TLc, and TRc edit descriptors are also used to move the position within the record, but in a non-destructive manner. This is called tabbing. Position means character position with the first character in the record being at position one. Changing the position within the record does change the length of the record.

The Tc edit descriptor moves to absolute position c within the record. The TLc and TRc edit descriptors move to positions relative to the current position. TRc moves the position c characters to the right and TLc moves the position c characters to the left. c is a positive integer constant.

```
        WRITE (9,10) 89, 567,  23, 1, 4
10      FORMAT (T8,I2,TL5,I3,T2,I2,TL3,I1,TR2,I1)

123456789
```

8.19.3 Slash Editing

The / edit descriptor positions the file at the beginning of the next record. On input it skips the rest of the current record. On output it creates a new record at the end of the file.

The / edit descriptor can be used to skip entire records on input or to write empty records on output. Empty records in internal or direct access files are filled with blanks.

When the / edit descriptor is used with files connected for direct access it causes the record number to be increased and data transfer will be performed with that record.

```
        WRITE (9,10) (A, A=1.0,10.0)
10      FORMAT (5F5.1,/,5F5.1)

  1.0  2.0  3.0  4.0  5.0
  6.0  7.0  8.0  9.0 10.0
```

8.20 COLON EDITING

The : edit descriptor is used to terminate a formatted I/O statement if there are no more data items to process. For example, the : edit descriptor could be used to stop positional editing when there are no more items in the I/O list.

8.21 APOSTROPHE AND HOLLERITH EDITING

Apostrophe and Hollerith edit descriptors are used to copy strings of characters to the output record. These edit descriptors may only be used with the output statements: WRITE, PRINT, and TYPE.

8.21.1 Apostrophe Editing

An apostrophe edit descriptor takes exactly the same form as a character constant as described in Chapter 4. The field width is equal to the length of the string.

```
        WRITE (9,10)
10      FORMAT ('APOSTROPHE',1X,'EDIT FIELDS')

APOSTROPHE EDIT FIELDS
```

8.21.2 H Editing

The nH edit descriptor takes exactly the same form as a Hollerith constant as described in Chapter 4. The field width is equal to the positive integer constant, n, which defines the length of the Hollerith constant.

```
        WRITE (9,10)
10      FORMAT (14HHOLLERITH EDIT,6HFIELDS)

HOLLERITH EDIT FIELDS
```

8.22 LIST DIRECTED EDITING

List directed editing is indicated with an asterisk (*) as a format specifier. List directed editing selects editing for I/O list items appropriate to their data type and value. List directed editing treats one or more records in a file as a sequence of values delimited by value separators. A value separator is one or more blanks, a comma, a slash, or an end of record. Tabs are expanded modulo eight. Blanks can precede and follow the comma and slash separators. Except within a quoted character constant, multiple blanks and end of record characters are treated as a single blank character. An end of record occurring within a quoted character constant is treated as a null.

The values are either constants, nulls, or one of the forms:

r*c

r*

where r is an unsigned, nonzero, integer constant. The first form is equivalent to r occurrences of the constant c, and the second is equivalent to r nulls. Null items are defined by having no characters where a value would be expected, that is, between successive separators or before the first separator in a record.

8.22.1 List Directed Input

A character value is a string of characters between value separators. If the string is quoted embedded blanks are significant and the value can span more than one record. The corresponding I/O list item is defined with the value as though a character assignment statement was performed; left justified and truncated or blank filled as necessary.

Any form suitable for an I edit descriptor can be used for list directed input of an integer item.

Any form suitable for an L edit descriptor can be used for list directed input of a logical item. In particular, .TRUE. and .FALSE. are acceptable.

Real and double precision input is performed with the effect of an F*w*.0 edit descriptor where *w* is the number of characters in the constant. The value can be in any form acceptable to the F edit descriptor.

A complex constant must have an opening parenthesis, a real constant as described above, a comma, another real constant, and a closing parenthesis. Leading and trailing spaces are permitted around the comma. The first constant represents the real portion of the value and the second constant represents the imaginary portion.

Null values have no effect on the corresponding I/O list items; their definition status will not change.

A slash in the input record terminates a list directed input statement. Any unprocessed I/O list items will be left unchanged.

8.22.2 List Directed Output

With the exception of character constants, all output items are separated by a single blank which is generated as part of the string.

Character output is performed using an A edit descriptor. There is no leading blank.

Logical output is performed using an L2 edit descriptor.

Integer output is performed using an I*w* edit descriptor where *w* is one digit greater than the number of digits required to represent the value.

Real and double precision output is performed using a 1PG12.4E2 edit descriptor.

Complex output consists of an opening parenthesis, the real portion of the value, a comma, the imaginary portion of the value, and a closing parenthesis. The numeric portions are formatted with a 1PG12.4E2 edit descriptor.

CHAPTER 9

PROGRAMS, SUBROUTINES, FUNCTIONS, AND BLOCK DATA SUBPROGRAMS

There are six types of procedures available in FORTRAN 77: main programs, subroutines, external functions, statement functions, intrinsic functions, and block data.

The main program is the entry point of a FORTRAN 77 program. The compiler does not require that the main program occur first in the source file, however, every FORTRAN 77 program must have exactly one main program.

Subroutines and external functions are procedures which are defined outside of the program unit that references them. They may be specified either in separate FORTRAN 77 subprograms or by means other than FORTRAN 77 such as assembly language or the C programming language. The appendices contain information on writing external procedures in assembly language and C.

Block data subprograms are nonexecutable procedures which are used to initialize variables and array elements in named common blocks. There may be several block data subprograms in a FORTRAN 77 program.

9.1 PROGRAMS

The PROGRAM statement is given in the following manner:

PROGRAM _pgm_[(_elist_)]

where: _pgm_ is a unique symbolic name which is used as the name of the main program

elist is an optional execution environment parameter list

The PROGRAM statement itself is optional. If it is not supplied, the six most significant characters of the source file name will be used as the name of the main program.

elist is an extension to FORTRAN 77 to provide control over certain runtime execution parameters. The form of an environment parameter specification is:

par = _c_

where: _par_ is the parameter specifier

_c_ is the constant to be applied

The following parameters are available:

1. INPUT accepts an unsigned integer constant which is the unit number of the preconnected sequential input file. This unit is implicitly connected to a file by including the data file name on the command line at run time (e.g. under VERSAdos: MYPROG ;R=DATA). This allows emulating main frame installations where the card reader is a preconnected input unit for program data. The default value for INPUT is 5.

2. OUTPUT accepts an unsigned integer constant which is the unit number of the preconnected sequential output print file. This unit is connected for output to the system line printer at run time. When execution of the program stops, this file is automatically spooled to the system line printer. The default value for OUTPUT is 6.

3. TAPE accepts an unsigned integer constant which is the unit number of the preconnected tape device. This unit is connected for input and output to the default magnetic tape drive at run time. The default value for TAPE is 7.

4. BLOCK accepts an unsigned integer constant which is the block size of data in bytes on a magnetic tape between IRGs. The default value for BLOCK is 512.

5. RECL accepts an unsigned integer constant which indicates that fixed length records are to be used with the connection to the unit designated in the TAPE specifier. For example, RECL=80 indicates card images. The default value for RECL is 0 (variable length records terminated by line-feed).

6. ERR specifies the action to be taken when a floating point error occurs and accepts either an unsigned integer constant or the symbolic name of a subroutine. The default value of ERR is 0 and indicates that floating point errors are to be reported to the operator and program execution is to be terminated. When ERR is set to 1, the IEEE value for infinity is used at the point the error occurred and program execution continues.

   When the ERR specifier is the symbolic name of a subroutine, execution is transferred to the subroutine for program control of error conditions. The subroutine must be in the same source file as the main program. Upon execution of either a RETURN statement or an END statement, control returns to the point of error. The subroutine must declare exactly five dummy arguments:

   ```
   SUBROUTINE FPERR (IERR,ICARD,IVAL,AVAL,DVAL)
   ```

   where IERR contains an error code, ICARD contains the statement number where the error occurred if the program was compiled with the N option (see Chapter 2), and IVAL, AVAL, and DVAL may be used to replace the IEEE value for infinity with a program supplied value. The error codes passed in IVAL are as follows:

	overflow	zero divide	argument
INTEGER		66	
REAL	321	322	323
DOUBLE	577	578	579

   IVAL, AVAL, and DVAL represent INTEGER, REAL, and DOUBLE PRECISION variables respectively and are equivalenced at the point of reference so that definition of IVAL will cause definition of AVAL and DVAL as well. This provides a method of defining floating point variables with hexadecimal and octal integer constants.

7. PRINTER accepts the name of the line printer for spooling the print file. The default value for PRINTER is the default system line printer.

8. SWITCHES accepts an unsigned integer constant specifying a control code for certain parameters relating to the printing of the print file. This code represents the sum of the codes required as listed below:

1	- print banner
2	- do not print banner
4	- delete after printing
8	- do not delete after printing
16	- print page headers
32	- do not print page headers
64	- append a final form feed
128	- do not append a final form feed
256	- wait if print queue is full

   The default value for SWITCHES is 102 = 2+4+32+64

9. COPIES accepts an unsigned integer constant specifying the number of copies of the print file to be spooled. The default value for copies is 1.

10. FORM is the form name to be used for print output. The default value of FORM is NORMAL.

11. LPP is an unsigned integer constant specifying the number of lines per page in the print file. The default value for LPP is the default value for the specified printer.

12. WIDTH is an unsigned integer constant specifying the page width of the print file. The default value for WIDTH is the default value for the specified printer.

Example:

```
     PROGRAM MYPROG(INPUT=1,OUTPUT=2,PRINTER=TI810,COPIES=2)
```

Note: the implementation of the various switches and printer parameters is operating system dependent; all of the features may not be supported on your operating system.

9.2 SUBROUTINES

A subroutine is a separate procedure which is defined external to the program unit which references it and is specified in a subroutine subprogram. A subroutine may be referenced within any other procedure of the executable program. While the ANSI standard prohibits a subroutine from referencing itself, either directly or indirectly, this implementation of FORTRAN 77 allows recursion. The form of a subroutine subprogram declaration is:

```
        SUBROUTINE sub [([arg] [,arg]...)]
```

where <u>sub</u> is a unique symbolic name that is used to reference the subroutine. Only the first six characters of the symbolic name are significant. ([arg] [,arg]...) is an optional list of variable names, array names, dummy procedure names, or asterisks that identifies the dummy arguments that are associated with the actual arguments in the referencing statement.

A subroutine is referenced with a CALL statement which has the form:

```
        CALL sub [([arg] [,arg]...)]
```

where <u>sub</u> is the symbolic name of a subroutine or dummy procedure and ([arg] [,arg]...) is the list of actual arguments which are associated with the arguments in the SUBROUTINE statement.

If the subroutine is undefined when the CALL statement is executed an attempt is made to read the subroutine as an overlay from the disk.

9.2.1 Subroutine Arguments

The argument lists of CALL and SUBROUTINE statements have a one to one correspondence; the first actual argument is associated with the first dummy argument and so on. The actual arguments in a CALL statement are assumed to agree in number and type with the dummy arguments declared in the SUBROUTINE statement. No type checking is performed by the compiler or the run time system to insure that this assumption is followed.

The addresses of labeled statements may be passed to subroutines by specifying the label preceded by an asterisk in the actual argument list and specifying an asterisk only in the corresponding position in the dummy argument list of the SUBROUTINE statement. This allows you to return to a location in the calling procedure other than the statement that immediately follows the CALL statement (see RETURN below).

Dummy procedure names allow you pass the names of procedures to other subprograms. The dummy procedure name can then be referenced as though it were the actual name of an external procedure.

9.3 FUNCTIONS

A function returns a value to the point within an expression that references it. An external function is specified in a separate procedure called a function subprogram. A statement function is defined in a single statement within a program unit and is local to that program unit. Intrinsic functions are library procedures provided with the FORTRAN 77 environment and are available to any program unit in an executable program. A function name may not be used on the left side of an equals sign except for an external function name and then only within the program unit which defines it.

A function reference is made in the form of an operand in an expression. The function name is given with an argument list enclosed in parentheses. The parentheses must be used even if there are no arguments to the function so that the compiler can determine that a function reference is indeed being made and not simply a reference to a variable.

9.3.1 External Functions

An external function may be referenced within any other procedure in an executable program. The recursive use of external functions is allowed as an extension to the FORTRAN 77 standard. Character functions must be declared with integer constant lengths so that the compiler can determine the size of the character value which will be returned. The form of a function subprogram declaration is:

```
        [typ [*len]] FUNCTION fun ([arg] [,arg]...)
```

where fun is a unique symbolic name that is used to reference the function. Only the first six characters of the symbolic name are significant. ([arg] [,arg]...) is an optional list of variable names, array names, or dummy procedure names that identifies the dummy arguments that are associated with the actual arguments in the referencing statement.

As indicated, the function can be given an optional type and length attribute. This can be done either explicitly in the FUNCTION statement or in a subsequent type statement, or implicitly following the data typing rules described in Chapter 4. Note that an IMPLICIT statement may change the data type and size.

When a character function is given a length attribute of *(*) it assumes the size established in the corresponding character declaration in the referencing program unit.

The symbolic name used to define the function must be assigned a value during the execution of the function subprogram. It is the value of this variable which is returned when a RETURN or END statement is executed.

If the function is undefined when it is referenced, an attempt is made to read the function as an overlay from the disk.

9.3.2 Statement Functions

A statement function is specified with a single statement which may appear only after the declaration section and before the executable section of the program unit in which it is to be used. A statement function is defined in the following manner:

```
        fun ([arg[,arg]...]) = e
```

where fun is the name that is used to reference the function, ([arg[,arg]...]) is the dummy argument list, and e is an expression using the arguments from the dummy argument list.

The dummy argument names used in the statement function argument list are local to the statement function and may be used elsewhere in the program unit without conflict.

A statement function statement must not contain a forward reference to another statement function. The compilation of a statement function removes the symbolic name of the function from the list of available names for variables and arrays within the program unit in which it is defined. Any variable or array which is defined in a program unit may not be redefined as a statement function.

Character statement functions may not use the *(*) size specifier.

9.3.3 Intrinsic Functions

Intrinsic functions contained in the math library do not follow the typing rules for user defined functions and cannot be altered with an IMPLICIT statement. The types of these functions and their argument list definitions appear in Table 9-1.

The generic names listed in Table 9-1 are provided to simplify the use of intrinsic functions which take different types of arguments. Except for the type conversion functions, the type of a generic function is the same as the type of its arguments.

9.4 ENTRY

The ENTRY statement may only be used within subroutine and function subprograms and provides for multiple entry points into these procedures. The form of an ENTRY statement is the same as that for SUBROUTINE and FUNCTION statements except that the key word ENTRY is used. An optional type clause may precede an ENTRY statement used in a FUNCTION subprogram. An ENTRY statement may not occur within any block structure (DO, IF, or CASE). The number of dummy arguments appearing in an ENTRY statement must agree with the number of arguments in the corresponding SUBROUTINE or FUNCTION statements.

In a function subprogram, a variable name that is used as the entry name must not appear in any statement that precedes the appearance of the entry name except in a type statement. All function and entry names in a function subprogram share an equivalence association.

Entry names used in character functions must have a character data type and the same size attribute as the name of the function subprogram itself.

9.5 RETURN

The RETURN statement ends execution in the current subroutine or function subprogram and returns control of execution to the referencing program unit. The RETURN statement may only be used in function and subroutine subprograms. Execution of a RETURN statement in a function returns the current value of the function name variable to the referencing program unit. The RETURN statement is given in the following manner:

```
        RETURN [e]
```

where e is an integer expression allowed only in subroutine RETURN statements and causes control to be returned to a labeled statement in the calling procedure associated with an asterisk in the dummy argument list. The first alternate return address corresponds to the first asterisk, the second return address to the second asterisk, etc. If the value of e is less than one or greater than the number of asterisks, control is returned to the statement immediately following the CALL statement.

9.6 PASSING PROCEDURES IN DUMMY ARGUMENTS

When a dummy argument is used to reference an external function, the associated actual argument must be either an external function or an intrinsic function. When a dummy argument is associated with an intrinsic function there is no automatic typing property. If a dummy argument name is also the name of an intrinsic function then the intrinsic function corresponding to the dummy argument name is removed from the list of available intrinsic functions for the subprogram.

If the dummy argument is used in a CALL statement then the name cannot be used as a variable or a function.

9.7 PASSING RETURN ADDRESSES IN DUMMY ARGUMENTS

If a dummy argument is an asterisk, the compiler will assume that the actual argument is an alternate return address passed as a statement label preceded by an asterisk. No check is made by the compiler or by the run time system to insure that the passed parameter is in fact a valid alternate return address.

9.8 COMMON BLOCKS

A common block is used to provide an area of memory whose scoping rules are greater than the current program unit. Because association is by storage offset within a known memory area, rather than by name, the types and names of the data elements do not have to be consistent between different procedures. A reference to a memory location is considered legal if the type of data stored there is the same as the type of the name used to access it. However, the compiler does not check for consistency between different program units and common blocks.

Common blocks are allocated at the point of first use and deallocated at the end of that program unit. When a common block is needed and does not exist, it will be created. This means that common blocks in a main program are global to the entire execution of a program, even if no values within the block are referenced or used by the executable section of the main program. Also, any subroutine or function which creates a common block will delete it when it returns, except for those that are specified by SAVE statements. Blank common is never deleted.

The total amount of memory required by an executable program can be reduced by using common blocks as a sharable storage pool for two or more subprograms. Because references to data items in common blocks is through offsets and because types do not conflict across program units, the same memory may be remapped to contain different variables.

Table 9-1 Intrinsic Functions

Specific Name	Generic Name	Usage	Argument Type	Result Type	Notes
		Type Conversion			
INT	INT	INT(x)	any	integer	1
IFIX	INT	IFIX(r)	real	integer	1
IDINT	INT	IDINT(d)	double	integer	1
REAL	REAL	REAL(x)	any	real	2
FLOAT	REAL	FLOAT(i)	integer	real	2
SNGL	REAL	SNGL(d)	double	real	2
DBLE	DBLE	DBLE(x)	any	double	3
CMPLX	CMPLX	CMPLX(x)	any	complex	4
ICHAR		ICHAR(a)	character	integer	5
CHAR		CHAR(i)	integer	character	5
		Truncation			
AINT	AINT	AINT(r)	real	real	1
DINT	AINT	DINT(d)	double	double	1
		Nearest Whole Number			
ANINT	ANINT	ANINT(r)	real	real	
DNINT	ANINT	DNINT(d)	double	double	
		Nearest Integer			
NINT	NINT	NINT(r)	real	integer	
IDNINT	NINT	NINT(d)	double	integer	
		Absolute Value			
ABS	ABS	ABS(x)	any	any	6
IABS	ABS	IABS(i)	integer	integer	6
DABS	ABS	DABS(d)	double	double	6
CABS	ABS	CABS(c)	complex	real	6
		Remaindering			
MOD	MOD	MOD(x,y)	any	any	
AMOD	MOD	AMOD(r,s)	real	real	
DMOD	MOD	DMOD(d,e)	double	double	
		Transfer of Sign			
ISIGN	SIGN	ISIGN(i,j)	integer	integer	
SIGN	SIGN	SIGN(r,s)	real	real	
DSIGN	SIGN	DSIGN(d,e)	double	double	

Specific Name	Generic Name	Usage	Argument Type	Result Type	Notes
		Positive Difference			
IDIM	DIM	IDIM(i,j)	integer	integer	
DIM	DIM	DIM(r,s)	real	real	
DDIM	DIM	DDIM(d,e)	double	double	
		Double Precision Product			
DPROD		DPROD(r,s)	real	double	
		Choosing the Largest Value			
MAX	MAX	MAX(x,y,...)	any	any	
MAX0	MAX	MAX0(i,j,...)	integer	integer	
AMAX1	MAX	AMAX1(r,s,...)	real	real	
DMAX1	MAX	DMAX1(d,e,...)	double	double	
AMAX0		AMAX0(i,j,...)	integer	real	
MAX1		MAX1(r,s,...)	real	integer	
		Choosing the Smallest Value			
MIN	MIN	MIN(x,y,...)	any	any	
MIN0	MIN	MIN0(i,j,...)	integer	integer	
AMIN1	MIN	AMIN1(r,s,...)	real	real	
DMIN1	MIN	DMIN1(d,e,...)	double	double	
AMIN0		AMIN0(i,j,...)	integer	real	
MIN1		MIN1(r,s,...)	real	integer	
		Imaginary Part of a Complex Argument			
AIMAG		AIMAG(c)	complex	real	6
		Conjugate of a Complex Argument			
CONJG		CONJG(c)	complex	complex	6
		Square Root			
SQRT	SQRT	SQRT(r)	real	real	
DSQRT	SQRT	DSQRT(d)	double	double	
CSQRT	SQRT	CSQRT(c)	complex	complex	
		Exponential			
EXP	EXP	EXP(r)	real	real	
DEXP	EXP	DEXP(d)	double	double	
CEXP	EXP	CEXP(c)	complex	complex	

Specific Name	Generic Name	Usage	Argument Type	Result Type	Notes
		Natural Logarithm			
LOG	LOG	LOG(x)	any	any	
ALOG	LOG	ALOG(r)	real	real	
DLOG	LOG	DLOG(d)	double	double	
CLOG	LOG	CLOG(c)	complex	complex	
		Common Logarithm			
LOG10	LOG	LOG10(x)	any	any	
ALOG10	LOG	ALOG10(r)	real	real	
DLOG10	LOG	DLOG10(d)	double	double	
		Sine			
SIN	SIN	SIN(r)	real	real	7
DSIN	SIN	DSIN(d)	double	double	7
CSIN	SIN	CSIN(c)	complex	complex	7
		Cosine			
COS	COS	COS(r)	real	real	7
DCOS	COS	DCOS(d)	double	double	7
CCOS	COS	CCOS(c)	complex	complex	7
		Tangent			
TAN	TAN	TAN(r)	real	real	7
DTAN	TAN	DTAN(d)	double	double	7
		Arcsine			
ASIN	ASIN	ASIN(r)	real	real	
DASIN	ASIN	DASIN(d)	double	double	
		Arccosine			
ACOS	ACOS	ACOS(r)	real	real	
DACOS	ACOS	DACOS(d)	double	double	
		Arctangent			
ATAN	ATAN	ATAN(r)	real	real	
DATAN	ATAN	DATAN(d)	double	double	
ATAN2	ATAN2	ATAN2(r,s)	real	real	
DATAN2	ATAN2	DATAN2(d,e)	double	double	

Specific Name	Generic Name	Usage	Argument Type	Result Type	Notes
		Hyperbolic Sine			
SINH	SINH	SINH(r)	real	real	
DSINH	SINH	DSINH(d)	double	double	
		Hyperbolic Cosine			
COSH	COSH	COSH(r)	real	real	
DCOSH	COSH	DCOSH(d)	double	double	
		Hyperbolic Tangent			
TANH	TANH	TANH(r)	real	real	
DTANH	TANH	DTANH(d)	double	double	
		Length of a Character Argument			
LEN		LEN(a)	character	integer	11
		Location of a Substring			
INDEX		INDEX(a,b)	character	integer	10
		Trim Trailing Blanks			
TRIM		TRIM(a)	character	character	13
		String Replication and Justification			
REPEAT		REPEAT(a,i)	character	character	14
ADJUSTL		ADJUSTL(a)	character	character	15
ADJUSTR		ADJUSTR(a)	character	character	16
		Lexical Comparisons			
LGE		LGE(a,b)	character	logical	12
LGT		LGT(a,b)	character	logical	12
LLT		LLT(a,b)	character	logical	12
LLE		LLE(a,b)	character	logical	12
		Absolute Memory Addressing			
BYTE		BYTE(i)	integer	integer	
WORD		WORD(i)	integer	integer	
LONG		LONG(i)	integer	integer	
		Logical Shift			
SHIFT		SHIFT(i,j)	integer	integer	

9.9 NOTES ON INTRINSIC FUNCTIONS

Intrinsic functions, sometimes referred to as mathematics library functions, are presented in Table 9-1. This table presents all of the intrinsic functions, their definitions, number of arguments required, types of arguments and function, and the generic and specific names of each function. The following are notes referenced in Table 9-1:

1. If _a_ is real or double precision, there are two cases: if |_a_|<1, then int(_a_)=0; if |_a_|>1, then int(_a_) is an integer which is rounded toward zero and has the same sign as _a_. If _a_ is complex then the real part of the argument is returned.

2. If _a_ is integer or double precision, then the function REAL(_a_) will return as much precision as can be specified in a real variable. If _a_ is complex then the real portion is returned. If the argument is integer then the function FLOAT will return the same result.

3. This function will return a double precision result which contains all the precision of the argument passed. If the argument is of type complex then the real portion is used.

4. CMPLX may have one or two arguments. If there is one and the type is complex then the argument is returned unmodified. If there is one argument of any other type then the value is converted to a real and returned as the real part and the imaginary part is zero. If there are two arguments then they must be the same type and cannot be complex. The first argument is returned as the real part and the second is the imaginary part.

5. ICHAR provides type conversion from character to integer, based on ASCII value of the argument.

6. A complex value is expressed as an ordered pair of reals, (_ar_,_ai_), where the first is the real part and the second is the imaginary part.

7. All arguments are expressed in radians.

8. The result of a function of type complex is the principal value.

9. All arguments in an intrinsic function reference must be of the same type.

10. INDEX(_a_1,_a_2) returns an integer value indicating the starting position of the first occurrence of _a_1 in _a_2. A zero is returned if there is no match or _a_2 is shorter than _a_1.

11. The string passed to the LEN function does not need to be defined before the reference to LEN is executed.

12. LGE, LGT, LLE, and LLT return the same result as the standard relational operators.

13. TRIM(_a_) returns the value of the character expression _a_ with trailing blanks removed.

14. REPEAT(_a_,_n_) replicates the character expression _a_, _n_ times where _n_ is an integer expression.

15. ADJUSTL(_a_) returns a character result which is the same as its argument except leading blanks have been removed and sufficient trailing blanks have been added to make the result the same length as _a_.

16. ADJUSTR(_a_) returns a character result which is the same as its argument except trailing blanks have been removed and sufficient leading blanks have been added to make the result the same length as _a_.

9.9.1 Range of Arguments and Results Restrictions

The second argument of the MOD, AMOD, and DMOD functions must not be zero.

Zero is returned if the value of the first argument of ISIGN, SIGN, or DSIGN is zero.

The argument of the SQRT or DSQRT function must not be negative. CSQRT returns the principal value with the real portion greater than or equal to zero. When the real portion is zero, the imaginary portion is greater than or equal to zero.

The argument of the ALOG, DLOG, ALOG10, and DLOG10 functions must be greater than zero. Both portions cannot be zero for CLOG.

Automatic argument reduction allows the arguments of the SIN, DSIN, COS, DCOS, TAN, and DTAN functions to be greater than 2pi.

The argument of the ASIN or DASIN function must be less than or equal to one. The range of the result is between -pi/2 and pi/2 inclusive.

The argument of the ACOS or DACOS function must be less than or equal to one. The range of the result is between 0 and pi inclusive.

The range of the result for the ATAN and DATAN functions is between -pi/2 and pi/2 inclusive. If the value of the first argument of ATAN2 or DATAN2 is positive, the result is positive. If the value of the first argument is zero, the result is zero if the second argument is positive and pi if the second argument is negative. If the value of the first argument is negative, the result is negative. If the value of the second argument is zero, the absolute value of the result is pi/2. The arguments must not both have the value zero. The range of the result for ATAN2 and DATAN2 is between -pi and pi inclusive.

9.10 BLOCK DATA

A BLOCK DATA statement takes the following form:

```
        BLOCK DATA [sub]
```

Where _sub_ is the unique symbolic name of the block data subprogram.

There may be more than one named block data subprogram in a FORTRAN 77 program, but only one unnamed block data subprogram.

Only COMMON, DATA, DIMENSION, END, EQUIVALENCE, IMPLICIT, PARAMETER, SAVE, and type declaration statements may be used in a block data subprogram.

In this implementation of FORTRAN 77, block data subprograms are compiled as data initialization subroutines which are _implicitly_ called when storage is allocated for a named common block. If a block data subprogram is compiled separately, it must be linked with the linker to the subprogram which first establishes storage for the named common block.

Appendices

Appendices

APPENDIX A

OVERLAYS

Three methods of linking procedures together are available in Microsoft FORTRAN 77. The most straightforward method is to present the compiler with a single source file containing all of the procedures used by your program. The compiler will create a single application file.

The second method involves the traditional use of a linker. Files containing one or more procedures are compiled separately and then linked using the Microsoft linker, LINK, to create a single application module. The linker and library manager manual details the use of the linker.

The third method is to create overlay modules which can be loaded and linked dynamically at run time. This appendix describes the dynamic linking process, which allows you to call an external procedure from a secondary storage device as an overlay.

Creating procedures which execute as overlays is useful during the development stages of a program and for the sharing of commonly used subroutines between several users or tasks. Overlays are also useful on systems where the program is larger than the available memory.

A.1 REFERENCING AN OVERLAY

When the compiler encounters an unresolved external procedure reference, it automatically generates code to load it as an overlay. This code may be later modified by the linker to generate a direct call if the procedure is statically linked.

When a run time reference is made to a subroutine or function that has not been linked to the program, a call is made to the overlay manager which is contained in the run time library, F77.RL. The overlay manager takes one argument which is the external procedure name packed RAD50. A file name is generated using an appropriate path name and extension. The path names are those directories listed in your system implementation notes under "run time search paths." On those systems which distinguish between upper and lower case characters in file names, the procedure name is folded to lower case before being used.

The various directories are searched for the file, which if located, is loaded into memory, stripping any header block in the process. Execution is then passed to the procedure.

If the procedure file cannot be located, the run time library will issue the message "subprogram not found" (error number 75). Before reading the procedure into memory, a check is made to insure that enough memory is available to the FORTRAN program. If the amount is insufficient, a "heap space overflow" message (error number 64) is emitted.

This process may be nested (a procedure loaded from the disk may cause the loading of another) to a level of fifty. Recursive invocations of a procedure do not force new copies to be loaded; control of execution simply passes to the entry point of the procedure.

Multiple entry points are not supported in the dynamic linking process, however, an overlay may contain other procedures which are local to itself. The file name must be the same as the symbolic name of the procedure.

When a RETURN or END statement is encountered in an overlay, the memory space used by the overlay is freed for use by the next routine that is fetched from the disk.

APPENDIX B

THE ASCII CHARACTER SET

CHARACTER	DEC	OCT	HEX	NAME
NULL	0	000	00	null
SOH	1	001	01	start of heading
STX	2	002	02	start of text
ETX	3	003	03	end of text
ECT	4	004	04	end of transmission
ENQ	5	005	05	enquiry
ACK	6	006	06	acknowledge
BEL	7	007	07	bell code
BS	8	010	08	back space
HT	9	011	09	horizontal tab
LF	10	012	0A	line feed
VT	11	013	0B	vertical tab
FF	12	014	0C	form feed
CR	13	015	0D	carriage return
SO	14	016	0E	shift out
SI	15	017	0F	shift in
DLE	16	020	10	data link escape
DC1	17	021	11	device control 1
DC2	18	022	12	device control 2
DC3	19	023	13	device control 3
DC4	20	024	14	device control 4
NAK	21	025	15	negative acknowledge
SYN	22	026	16	synchronous idle
ETB	23	027	17	end of transmission blocks
CAN	24	030	18	cancel
EM	25	031	19	end of medium
SS	26	032	1A	special sequence
ESC	27	033	1B	escape
FS	28	034	1C	file separator
GS	29	035	1D	group separator
RS	30	036	1E	record separator
US	31	037	1F	unit separator

CHARACTER	DEC	OCT	HEX	NAME
SP	32	040	20	space
!	33	041	21	exclamation mark
"	34	042	22	quotation mark
#	35	043	23	number sign
$	36	044	24	dollar sign
%	37	045	25	percent sign
&	38	046	26	ampersand
'	39	047	27	apostrophe
(	40	050	28	opening parenthesis
)	41	051	29	closing parenthesis
*	42	052	2A	asterisk
+	43	053	2B	plus
,	44	054	2C	comma
-	45	055	2D	minus
.	46	056	2E	period
/	47	057	2F	slash
0	48	060	30	zero
1	49	061	31	one
2	50	062	32	two
3	51	063	33	three
4	52	064	34	four
5	53	065	35	five
6	54	066	36	six
7	55	067	37	seven
8	56	070	38	eight
9	57	071	39	nine
:	58	072	3A	colon
;	59	073	3B	semicolon
<	60	074	3C	less than
=	61	075	3D	equal
>	62	076	3E	greater than
?	63	077	3F	question mark
@	64	100	40	commercial at

CHARACTER	DEC	OCT	HEX	NAME
A	65	101	41	upper case letter
B	66	102	42	upper case letter
C	67	103	43	upper case letter
D	68	104	44	upper case letter
E	69	105	45	upper case letter
F	70	106	46	upper case letter
G	71	107	47	upper case letter
H	72	110	48	upper case letter
I	73	111	49	upper case letter
J	74	112	4A	upper case letter
K	75	113	4B	upper case letter
L	76	114	4C	upper case letter
M	77	115	4D	upper case letter
N	78	116	4E	upper case letter
O	79	117	4F	upper case letter
P	80	120	50	upper case letter
Q	81	121	51	upper case letter
R	82	122	52	upper case letter
S	83	123	53	upper case letter
T	84	124	54	upper case letter
U	85	125	55	upper case letter
V	86	126	56	upper case letter
W	87	127	57	upper case letter
X	88	130	58	upper case letter
Y	89	131	59	upper case letter
Z	90	132	5A	upper case letter
[	91	133	5B	opening bracket
\	92	134	5C	back slash
]	93	135	5D	closing bracket
^	94	136	5E	circumflex
_	95	137	5F	underscore
`	96	140	60	grave accent
a	97	141	61	lower case letter
b	98	142	62	lower case letter
c	99	143	63	lower case letter
d	100	144	64	lower case letter
e	101	145	65	lower case letter
f	102	146	66	lower case letter
g	103	147	67	lower case letter
h	104	140	68	lower case letter
i	105	151	69	lower case letter
j	106	152	6A	lower case letter
k	107	153	6B	lower case letter
l	108	154	6C	lower case letter
m	109	155	6D	lower case letter
n	110	156	6E	lower case letter
o	111	157	6F	lower case letter

CHARACTER	DEC	OCT	HEX	NAME
p	112	160	70	lower case letter
q	113	161	71	lower case letter
r	114	162	72	lower case letter
s	115	163	73	lower case letter
t	116	164	74	lower case letter
u	117	165	75	lower case letter
v	118	166	76	lower case letter
w	119	167	77	lower case letter
x	120	170	78	lower case letter
y	121	171	79	lower case letter
x	122	172	7A	lower case letter
{	123	173	7B	opening brace
|	124	174	7C	bar
}	125	175	7D	closing brace
	126	176	7E	tilde
DEL	127	177	7F	delete

APPENDIX C

BIBLIOGRAPHY

Loren P. Meissner and Elliot I. Organick, FORTRAN 77, Addison-Wesley Publishing Company (1980)

Harry Katzan, Jr., FORTRAN 77, Van Nostrand Reinhold Company (1978)

J.N.P. Hume and R.C. Holt, Programming FORTRAN 77, Reston Publishing Company, Inc. (1979)

Harice L. Seeds, FORTRAN IV, John Wiley & Sons (1975)

Jehosua Friedmann, Philip Greenberg, and Alan M. Hoffberg, FORTRAN IV, A Self-Teaching Guide, John Wiley & Sons, Inc. (1975)

V. Thomas Dock, FORTRAN IV Programming, Reston Publishing Company (1972)

Daniel D. McCracken, A Guide To FORTRAN IV Programming, John Wiley & Sons (1965)

Mario V. Farina, FORTRAN IV Self-Taught, Prentice-Hall Inc. (1966)

James S. Coan, Basic FORTRAN, Hayden Book Company (1980)

Brian W. Kernighan and P.J. Plauger, Software Tools, Addison-Wesley Publishing Company (1976)

Brian W. Kernighan and P.J. Plauger, The Elements of Programming Style, McGraw-Hill Book Company (1978)

American National Standard Programming Language FORTRAN, X3.9-1978, ANSI, 1430 Broadway, New York, N.Y. 10018

COMPUTER, A Proposed Standard for Binary Floating-Point Arithmetic, Draft 8.0 of IEEE Task P754, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720 (1981)

M. Abramowitz and I.E. Stegun, Handbook of Mathematical Functions, U.S. Department of Commerce, National Bureau of Standards (1972)

APPENDIX D

ERROR MESSAGES

D.1 COMPILE TIME ERROR MESSAGES

```
missing END statement
symbol must be a PARAMETER
first line of statement is a continuation
label on a continuation line
more than 19 continuation lines
expecting end of statement
illegal character
RETURN statement within a main program
specification statement syntax error
invalid statement label
duplicate statement label
duplicate subprogram name
duplicate variable name
more than one main program
missing main program
missing program unit statement
missing argument
numeric overflow
illegal statement ordering
SYNTAX error
undeclared dimension specifier
undefined statement label
undeclared array
too many right parentheses
too many left parentheses
ASSIGN statement syntax error
missing label on FORMAT statement
FORMAT statement syntax error
FORMAT specifier field width underflow
I/O control list specifier error
illegal statement following logical IF
maximum of seven dimensions exceeded
array subscript error
```

invalid alternate numeric base
main program in a subprogram compilation
illegal EQUIVALENCE of COMMON blocks
previous EQUIVALENCE variable
spelling error?
unterminated DO loop in program unit
unterminated IF block in program unit
ELSE without IF (exp) THEN block
illegal argument
DATA statement syntax error
illegal DATA variable (not in COMMON)
illegal DATA variable (in COMMON)
I/O control list specifier error
illegal dimension specifier
alternate RETURN not allowed in FUNCTION
data type of symbol not established
illegal statement in a BLOCK DATA
variable in BLOCK DATA is not in COMMON
INCLUDE file not found
alpha character expected
illegal INCLUDE statement in INCLUDE file
invalid INTRINSIC function name
statement cannot be reached
invalid program control list specifier
illegal CHARACTER length specification
unterminated CASE block in program unit
CASE statement expected
variable previously declared in COMMON
DATA list variable/constant mismatch
I/O control argument is not CHARACTER
FMT reference is not a FORMAT statement
illegal symbol in a SAVE statement
COMMON block not previously declared
illegal symbol in a DATA statement
label reference is to a FORMAT statement
expecting a symbol
expecting an opening parenthesis
unexpected end of statement
invalid OPTION specifier
REPEAT without DO
ENDIF without IF (exp) THEN block
array boundary error
data type of symbol is undefined
implied DO list syntax error
assumed size specifier in actual array
assumed size specifier must be last
illegal DO variable
expecting a LOGICAL constant

D.2 RUNTIME ERROR CODES

0	-	operator interrupt
1-63	-	reserved for host operating system
64	-	insufficient memory
65	-	numeric overflow
66	-	divide by zero
67	-	argument error
68	-	stack underflow
69	-	stack overflow
70	-	COMMON buffer not found
71	-	illegal record length
72	-	record overflow
73	-	duplicate virtual array
74	-	virtual buffer not found
75	-	subprogram not found
76	-	FORMAT syntax error
77	-	file not open for WRITE
78	-	file not open for READ
79	-	SELECT CASE match error
80	-	argument list mismatch
81	-	FORMAT descriptor error
82	-	array boundary error
83	-	illegal function call
84	-	floating point hardware not found
85	-	illegal substring expression

D.3 MACINTOSH SPECIFIC RUNTIME ERROR CODES

General System Errors

1 - queue element not found during deletion
2 - invalid queue element
3 - core routine number out of range
4 - unimplemented core routine

I/O System Errors

17 - control error
18 - status error
19 - read error
20 - write error
21 - invalid I/O unit
22 - unit empty error
23 - open error
24 - close error
25 - tried to remove an open driver
26 - driver not found
27 - I/O call aborted
28 - driver not opened

File System Errors

33 - directory full
34 - disk full
35 - no such volume
36 - I/O error
37 - bad file name
38 - file not open
39 - end of file
40 - tried to position to before start of file (r/w)
41 - memory full (open) or file won't fit (load)
42 - too many files open
43 - file not found
44 - diskette is write protected
45 - file is locked
46 - volume is locked
47 - File is busy (delete)
48 - duplicate filename (rename)
49 - file already open with write permission
50 - error in user parameter list
51 - invalid file reference number
52 - get file position error
53 - volume not on line error (was Ejected)
54 - permissions error (on file open)
55 - drive volume already on-line at MountVol
56 - no such drive (tried to mount a bad drive num)
57 - not a mac diskette (sig bytes are wrong)
58 - volume in question belongs to an external fs
59 - file system error during rename
60 - bad master directory block
61 - write permissions error

Disk, Serial Ports, Clock Specific Errors

64 - drive not installed
65 - r/w requested for an off-line drive
66 - couldn't find 5 nibbles in 200 tries
67 - couldn't find valid addr mark
68 - read verify compare failed
69 - addr mark checksum didn't check
70 - bad addr mark bit slip nibbles
71 - couldn't find a data mark header
72 - bad data mark checksum
73 - bad data mark bit slip nibbles
74 - write underrun occurred
75 - step handshake failed
76 - track 0 detect doesn't change
77 - unable to initialize IWM
78 - tried to read 2nd side on a 1-sided drive
79 - unable to correctly adjust disk speed
80 - track number wrong on address mark
81 - sector number never found on a track
85 - unable to read same clock value twice
86 - time written did not verify
87 - parameter ram written didn't read-verify
88 - InitUtil found the parameter ram uninitialized
89 - SCC receiver error (framing, parity, OR)
90 - Break received (SCC)

Storage Allocator Errors

108 - not enough room in heap zone
109 - Handle was NIL in HandleZone or other;
111 - WhichZone failed (applied to free block);
112 - trying to purge a locked or non-purgeable block;
110 - address was odd, or out of range;
113 - Address in zone check failed;
114 - Pointer Check failed;
115 - Block Check failed;
116 - Size Check failed;

Resource Manager Errors (other than I/O Errors)

192 - Resource not found
193 - Resource file not found
194 - AddResource failed
195 - AddReference failed
196 - RmveResource failed
197 - RmveReference failed

Scrap Manager Errors

100 - No scrap exists error
102 - No object of that type in scrap

D.4 MACINTOSH SYSTEM ALERT ERROR CODES

These errors are not reported by the Microsoft FORTRAN Compiler, but will instead be reported by the system in an alert box with a bomb icon.

32767 - general system error
1 - bus error
2 - address error
3 - illegal instruction error
4 - zero divide error
5 - check trap error
6 - overflow trap error
7 - privilege violation error
8 - trace mode error
9 - line 1010 trap error
10 - line 1111 trap error
11 - miscellaneous hardware exception error
12 - unimplemented core routine error
13 - uninstalled interrupt error
14 - IO Core Error
15 - Segment Loader Error
16 - Floating point error
17 - package 0 not present
18 - package 1 not present
19 - package 2 not present
20 - package 3 not present
21 - package 4 not present
22 - package 5 not present
23 - package 6 not present
24 - package 7 not present
25 - out of memory
26 - can't launch file
28 - stack has moved into application heap
27 - file system map has been trashed
30 - request user to reinsert off-line volume
31 - not the disk I wanted

Storage Allocator Trouble Codes

32 - Set Logical Size Error.
33 - Adjust Free Error.
34 - Adjust Counters Error.
35 - Make Block Free Error.
36 - Set Size Error.
37 - Initialize Memory Manager Error.

APPENDIX E

STRUCTURE FOR ASSEMBLY LANGUAGE EXTERNAL PROCEDURES

A facility is available for using external procedures written in assembly language with FORTRAN. These external procedures are separately generated using the resident assembler and linker and may be placed in library directories for use by any FORTRAN program requiring them. Five such procedures: ARGS, SPOOL, TIME, DATE, and ERRMSG are provided with the compiler.

The procedure has complete access to the entire run time library and all of the intrinsic functions contained within it. A description of the routines and the method of accessing them is contained in the next appendix.

In order for a procedure written in assembly language to communicate with the calling program, certain program structure specifications must be followed. The remainder of this appendix describes that structure.

Note: Several examples are given on the following pages using assembler opcodes and directives based on Motorola standard mnemonics. Your resident assembler may use different opcodes and directives.

E.1 STACK FRAME

The execution environment is heavily dependent upon a data structure known as a stack frame. The stack frame is used to maintain pointers to arguments passed to a procedure and to establish local dynamic memory. A6 is used as the frame pointer and it is absolutely critical that your routine maintain this register. All FORTRAN procedures use the following code sequence to establish a stack frame and allocate local memory:

```
        LINK    A6,#-100        * allocate 100 bytes of local memo
        MOVEA.L A7,A3           * use A3 as a local base register
        ...     ...
        UNLK    A6              * restore previous frame
        RTS
```

E.2 STACK LAYOUT AND REGISTER ASSIGNMENT

On entry to an external procedure the stack, indexed by A7, contains pointers to the arguments that the calling procedure passed. A map of the stack with its contents follows. This map is laid out such that the first location described is the "top" of the stack and is referenced by A7 (all locations are long word except as noted):

```
   RTS - return address to calling procedure
   an  - address of nth argument in the call list
   ::
   a3  - address of third argument in the call list
   a2  - address of second argument in the call list
   al  - address of first argument in the call list
   sl  - length of first argument (word)
   s2  - length of second argument (word)
   s3  - length of third argument (word)
   ::
   sn  - length of nth argument (word)
```

Only CHARACTER arguments have their lengths placed in sl-sn.

On entry to an external procedure the following registers contain information:

```
   D0  - number of arguments passed
   A0  - communications area pointer
   A4  - library jump table pointer
   A5  - work stack pointer
   A6  - stack frame pointer
```

If your procedure is to be called as an overlay, it must preserve A0.

E.3 ARGUMENT PASSING

Arguments may be freely passed back and forth between program units via dummy argument lists. Actual argument addresses are placed on the stack from left to right. Constants are copied and their addresses are placed on the stack to avoid inadvertently changing their values in the calling procedure. To access a particular argument, load its address as an offset from A7 or, if you established a stack frame, A6. The following example will scale a vector by a constant (see Appendix F for details on using the library routine MULF).

```
C     FORTRAN calling sequence:

      CALL SCALE(VECTOR,LENGTH,FACTOR)


*
* assembly language subroutine:
*

SCALE:  MOVEA.L 12(A7),A2     * load pointer to VECTOR
        MOVEA.L 8(A7),A1      * load pointer to LENGTH
        MOVE.L  (A1),D2       * load LENGTH
        MOVEA.L 4(A7),A1      * load pointer to FACTOR
        MOVE.L  (A1),D1       * load FACTOR

LOOP:   MOVE.L  (A2),D0       * load vector element
        JSR     MULF(A4)      * scale it
        MOVE.L  D0,(A2)+      * store scaled vector element
        SUBQ.L  #1,D2         * done?
        BNE.S   LOOP          *   no

        RTS

        END
```

E.4 STORAGE

E.4.1 Dynamic Storage

The simplest method of allocating dynamic storage for your procedure is to use the LINK and UNLK instructions with A6 as the linkage register or frame pointer. You may also use the A5 work stack for up to 200 bytes of local storage:

```
LEA     -200(A5),A5
```

The A5 stack does not have to be restored when your procedure returns.

E.4.2 Static Storage

Two types of static storage are available: local and common. Allocating and locating static storage is accomplished by passing arguments to a run time library routine. Local storage is allocated or located by passing arguments in registers and common storage is allocated or located by passing a pointer to a list of parameter blocks. Arguments for both storage types must be passed to the allocation routine even if only one type of storage is required. The run time allocation routine returns a pointer to local storage in A3 and a list of pointers to common blocks below the A6 frame pointer.

The arguments for allocating local storage are passed as long word items in D0 and D2. D0 contains the name of the local storage module packed RAD50 and D2 contains the size as an even number of bytes.

Each parameter block for a common storage block contains three long word entries. The first long word contains the name of the common block packed RAD50. The second long word contains the amount of memory to allocate as an even number of bytes. The third long word must be zero. The list of parameter blocks is terminated by long word containing a zero. Al is passed as a pointer to the parameter blocks.

The first example allocates or locates a local block of storage and no common blocks:

```
        MOVE.L  #$4D5B0820,D0     * name = LOCAL
        MOVE.L  #1024,D2          * size = 1k
        LEA     BLOCK(PC),A1      * load pointer to common blocks
        JSR     8(A4)             * allocate or locate
        ...     ...               * a pointer to local storage is
        ...     ...               *   returned in A3
        RTS

BLOCK:  DC.L    0                 * no common storage
```

The second example allocates or locates two common blocks and no local storage:

```
        LINK    A6,#-8            * allocate space for pointers
        MOVEQ   #0,D2             * no local storage
        LEA     BLOCK(PC),A1      * load pointer to common blocks
        JSR     8(A4)             * allocate or locate
        ...     ...               * -4(A6) = pointer to BLK1
        ...     ...               * -8(A6) = pointer to BLK2
        UNLK    A6
        RTS

BLOCK:  DC.L    $0EB6C1C0         * name = BLK1
        DC.L    4564              * size = 4564
        DC.L    0                 * must be zero
        DC.L    $0EB6C800         * name = BLK2
        DC.L    200               * size = 200
        DC.L    0                 * must be zero

        DC.L    0                 * terminate parameter list
```

E.4.3 Global Storage

Sixteen bytes of global storage have been reserved in the run time communications block for end user applications. The run time communications block is indexed by address register A0 and the reserved area begins at 4(A0).

E.5 COMMON BLOCKS

All references to data within a COMMON block should take into account the fact that the actual data is offset by four bytes from the base of the module. The full listing created by the compiler generates relative offsets for variables with these four bytes accounted for.

COMMON blocks use the first word of storage as a reference level counter. Whenever a subprogram declares a COMMON block, this word is incremented. The second word in each COMMON block is reserved and should not be used.

E.6 FUNCTION PROCEDURES

External procedures designed as functions require special handling in that they must always return a value to the calling procedure. For LOGICAL, INTEGER, and REAL functions, the return value is placed in D0. For DOUBLE PRECISION functions, the most significant portion of the result is placed in D0 and the least significant portion is placed in D1. For COMPLEX functions, the real portion of the return value is placed in D0 and the imaginary portion is placed in D1.

CHARACTER functions present particular problems in that they can be designed to return variable length results depending on the length attributes declared in the calling procedure. In order to provide dynamic space for the accumulation of intermediate values as well as the result, the calling procedure allocates space for CHARACTER function results and passes this address to the function on the stack. This address is passed as the nth dummy argument and its size is passed as the nth length attribute. This preallocated storage space does not have to be used, but all CHARACTER functions must return the <u>address</u> of the return value in A1. The following example will fold alpha characters in an arbitrary CHARACTER argument to lower case:

```
C     FORTRAN calling sequence:

      s2 = lcs(sl)

*
* assembly language subroutine:
*

LCS:    MOVEA.L 8(A7),Al       * load pointer to argument
        MOVEA.L 4(A7),A2       * load pointer to result
        MOVE.W  12(A7),Dl      * load length of argument
        MOVE.W  14(A7),D2      * load length of result

L1:     MOVE.B  (Al)+,D0       * get next input character
        CMPI.B  #'A',D0        * less than an A?
        BCS.S   L2             *   yes
        CMPI.B  #'Z',D0        * greater than a Z?
        BHI.S   L2             *   yes
        BSET    #5,D0          * fold to lower case
L2:     MOVE.B  D0,(A2)+
        SUBQ.W  #1,D2          * result exhausted?
        BEQ.S   L4             *   yes
        SUBQ.W  #1,Dl          * argument exhausted?
        BNE.S   Ll             *   no
L3:     MOVE.B  #' ',(A2)+     * space fill result
        SUBQ.W  #1,D2          * result exhausted?
        BNE.S   L3             *   no

L4:     MOVEA.L 4(A7),Al       * load pointer to result
        RTS

        END
```

E.7 EXAMPLES

Several assembler source files have been supplied with your compiler and should be examined carefully for examples of assembly language interfaces to Microsoft FORTRAN 77:

1. ARGS - returns command line arguments.
2. DATE - returns the system date in three integer variables.
3. TIME - returns the system time as seconds since midnight in an integer variable. This routine utilizes the intrinsic function library.
4. SPOOL - sends a print file to the system line printer. This routine is normally used to spool the file created by FORTRAN WRITE statements to unit 6. It is, however, a user callable routine that accepts a variable length argument list.

APPENDIX F

IEEE FLOATING POINT AND THE MATHEMATICS LIBRARY

The following information is provided for those assembly language programmers who wish to access the FORTRAN intrinsic functions directly. These functions are contained in executable form in the run time libraries, F77.RL (software floating point) and HDW.RL (hardware floating point).

All of the FORTRAN intrinsic functions are position independent and reentrant. Some require the use of other functions and routines also contained in both libraries.

F.1 ARGUMENT TYPES

F.1.1 Integer

Integer values are signed thirty-two bit entities with negative integers carried in two's complement form.

F.1.2 Real and Double Precision

Single and double precision floating point values are maintained in the format described in <u>A Proposed Standard for Binary Floating Point Arithmetic</u> by IEEE Task P754 in <u>COMPUTER</u> magazine.

```
------------------------------------
|s|   e    |          f            |
------------------------------------
 0         8                       31

   single precision format
```

```
------------------------------------------------------------------
|s|    e      |                       f
------------------------------------------------------------------
 0           11

   double precision format
```

The largest possible exponent for both formats is reserved to represent the symbolic entity Not a Number (NaN).

The components of a single precision value are a one bit sign, an eight bit exponent biased by 127, and a twenty-three bit fraction with an implied most significant bit. The interpretation of a single precision value <u>v</u> is as follows:

```
(a) if e=255 and f<>0, then v=NaN
(b) if e=255 and f=0, then v=signed infinity
(c) if 0<e<255, then v=2**(e-127)*1.f
(d) if e=0 and f<>0, then v=2**(-126)*0.f
(e) if e=0 and f=0, then v=0
```

The extremes of representation for single precision are:

```
2** 127*1.111...111  -->  0.3402823E+39
2**-126*1.000...000  -->  0.1175494E-37
```

The components of a double precision value are a one bit sign, an eleven bit exponent biased by 1023, and a fifty-two bit fraction with an implied most significant bit.

```
(a) if e=2047 and f<>0, then v=NaN
(b) if e=2047 and f=0, then v=signed infinity
(c) if 0<e<2047, then v=2**(e-1023)*1.f
(d) if e=0 and f<>0, then v=2**(-1022)*0.f
(e) if e=0 and f=0, then v=0
```

The extremes of representation for double precision are:

```
2** 1023*1.111...111  -->  0.179769313486232OD+309
2**-1022*1.000...000  -->  0.2225073858507202D-307
```

Round and guard bits are maintained. Round to nearest, with rounding to even in case of a tie, is the only rounding mode implemented.

F.1.3 Complex

A complex argument is passed to a library routine as two single precision floating point values representing the real and the imaginary components.

F.2 ARGUMENT PASSING

Two methods are employed for passing arguments and receiving results with the library routines:

1. REGISTER: For INTEGER and REAL functions, the principal argument is passed in D0 and the secondary argument (if any) is passed in D1. The result replaces the contents of D0. For DOUBLE PRECISION and COMPLEX functions, the principal argument is passed in the register pair D0/D1 and the secondary argument (if any) is passed in the register pair D1/D2. The result replaces the contents of the register pair D0/D1. All registers except D2, D3, and A1 are preserved.

2. POINTER: Pointers to the elements of the argument list are placed left to right on the A7 stack. Your routine is responsible for restoring the stack after the call. All registers except D2, D3, and A1 are preserved.

F.3 ERROR TRAPPING

Error trapping is provided for overflow, divide by zero, and argument errors. When an error condition occurs, control of execution is passed to the appropriate trap routine in F77.RL. Underflow is not considered an error condition.

F.4 LIBRARY ACCESS

Library routines are accessed through a jump table. The pointer to the jump table is in the first location of the runtime communications block which is indexed by A0.

```
        MOVEA.L (A0),A4          * address of jump table in A4
        MOVE.L  #$3F800000,D0
        MOVE.L  #$40000000,D1
        JSR     ADDF(A4)         * perform addition

        MOVEA.L (A0),A4          * address of jump table in A4
        PEA     ONE(PC)          * set pointer to argument
        JSR     SIN(A4)          * calculate sine
        ADDQ.W  #4,A7            * restore the stack

ONE:    DC.L    $3F800000
```

A summary of the routines and the jump table offsets follows.

F.5 INTRINSIC FUNCTION LIBRARY ROUTINES

Name	Type	Purpose	Arguments	Offset	Notes
MULI	Integ	multiplication	REGISTER	64	1,2
DIVI	Integ	division	REGISTER	68	1,2,3
ADDF	Real	subtraction	REGISTER	72	4
SUBF	Real	subtraction	REGISTER	76	4,5
MULF	Real	multiplication	REGISTER	80	4
DIVF	Real	division	REGISTER	84	3,4
ADDL	Doubl	subtraction	REGISTER	88	4
SUBL	Doubl	subtraction	REGISTER	92	4,5
MULL	Doubl	multiplication	REGISTER	96	4
DIVL	Doubl	division	REGISTER	100	3,4
CMPL	Doubl	comparison	REGISTER	104	6
ADDC	Cmplx	complex addition	REGISTER	108	
SUBC	Cmplx	complex subtraction	REGISTER	112	
MULC	Cmplx	complex multiplication	REGISTER	116	
DIVC	Cmplx	complex division	REGISTER	120	
CVTIF	Real	Integ to Real	REGISTER	124	
CVTIL	Doubl	Integ to Doubl	REGISTER	128	
CVTFI	Integ	Real to Integ	REGISTER	132	
CVTLI	Integ	Doubl to Integ	REGISTER	136	
CVTFL	Doubl	Real to Doubl	REGISTER	140	
CVTLF	Real	Doubl to Real	REGISTER	144	
II	Integ	Integ**Integ	REGISTER	148	
FI	Real	Real**Integ	REGISTER	152	
FF	Real	Real**Real	REGISTER	156	
LI	Doubl	Doubl**Integ	REGISTER	160	
LL	Doubl	Doubl**Doubl	REGISTER	164	
CI	Cmplx	Cmplx**Integ	REGISTER	168	
CC	Cmplx	Cmplx**Cmplx	REGISTER	172	
AINT	Real	truncation	POINTER	176	
DINT	Doubl	truncation	POINTER	180	
ANINT	Real	nearest whole number	POINTER	184	
DNINT	Doubl	nearest whole number	POINTER	188	
NINT	Integ	nearest integer	POINTER	192	
IDNINT	Integ	nearest integer	POINTER	196	
MOD	Integ	remaindering	POINTER	200	
AMOD	Real	remaindering	POINTER	204	3
DMOD	Doubl	remaindering	POINTER	208	3
SQRT	Real	square root	POINTER	212	7
DSQRT	Doubl	square root	POINTER	216	7
CSQRT	Cmplx	complex square root	POINTER	220	
EXP	Real	e**x	POINTER	224	7
DEXP	Doubl	e**x	POINTER	228	7
CEXP	Cmplx	complex e**X	POINTER	232	
LOG	Real	natural logarithm	POINTER	236	7
DLOG	Doubl	natural logarithm	POINTER	240	7
CLOG	Cmplx	complex log	POINTER	244	
LOG10	Real	common logarithm	POINTER	248	7
DLOG10	Doubl	common logarithm	POINTER	252	7

Name	Type	Purpose	Arguments	Offset	Notes
SIN	Real	trigonometric sine	POINTER	256	
DSIN	Doubl	trigonometric sine	POINTER	260	
CSIN	Cmplx	complex sine	POINTER	264	
COS	Real	trigonometric cosine	POINTER	268	
DCOS	Doubl	trigonometric cosine	POINTER	272	
CCOS	Cmplx	complex cosine	POINTER	276	
TAN	Real	trigonometric tangent	POINTER	280	
DTAN	Doubl	trigonometric tanget	POINTER	284	
ATAN	Real	trigonometric arctan	POINTER	288	
DATAN	Doubl	trigonometric arctan	POINTER	292	
ATAN2	Real	trigonometric arctan	POINTER	296	
DATAN2	Doubl	trigonometric arctan	POINTER	300	
ASIN	Real	trigonometric arcsine	POINTER	304	
DASIN	Doubl	trigonometric arcsine	POINTER	308	
ACOS	Real	trigonometric arccosine	POINTER	312	
DACOS	Doubl	trigonometric arccosine	POINTER	316	
SINH	Real	hyperbolic sine	POINTER	320	
DSINH	Doubl	hyperbolic sine	POINTER	324	
COSH	Real	hyperbolic cosine	POINTER	328	
DCOSH	Doubl	hyperbolic cosine	POINTER	332	
TANH	Real	hyperbolic tangent	POINTER	336	
DTANH	Doubl	hyperbolic tangent	POINTER	340	

Notes:

1. overflow ignored
2. secondary argument preserved
3. division by zero trapped
4. overflow trapped
5. sign of secondary argument is complemented
6. signed comparison of secondary argument to principal
7. argument error trapped

F.6 USEFUL CONSTANTS AND COMMON MATHEMATICAL RELATIONSHIPS

```
       pi = 3.141592653589793
     1/pi = 0.318309886183790

   pi/180 = 0.017453292519943 radians  (1 degree)
 pi/10800 = 0.000290888208666 radians  (1 minute)
pi/648000 = 0.000004848136811 radians  (1 second)

   180/pi =     57.2957795131 degrees  (1 radian)
 10800/pi =   3437.7467707849 minutes  (1 radian)
648000/pi = 206264.8062470964 seconds  (1 radian)

    sec x = 1/cos x
  cosec x = 1/sin x
  cotan x = 1/tan x

for complex numbers where the complex number z = x+iy

   sin iy = i sinh y
   cos iy = i cosh y

sinh x+iy = sinh x cos y + i cosh x sin y
cosh x+iy = cosh x cos y + i sinh x sin y


        e = 2.718281828459045

     e**x = y       x = ln y
    10**x = y       x = log y
```

conic sections:

circle

```
         x**2 + y**2 = a**2
```

parabola

```
         y=ax**2
```

ellipse

```
         x**2/a**2 + y**2/b**2 = 1
```

hyperbola

```
         x**2/a**2 - y**2/b**2 = 1
```

quadratic surfaces:

circular cone

```
         x**2/a**2 + y**2/a**2 - z**2 = 0
```

elliptic cone

```
         x**2/a**2 + y**2/b**2 - z**2 = 0
```

ellipsoid

```
         x**2/a**2 + y**2/b**2 + z**2/c**2 = 1
```

hyperboloid

```
        -x**2/a**2 - y**2/b**2 - z**2/c**2 = 1
```

elliptic paraboloid

```
         x**2/a**2 + y**2/b**2 = z
```

hyperbolic paraboloid

```
         x**2/a**2 - y**2/b**2 = z
```

APPENDIX G

RAD50 REPRESENTATION

Many areas of the compiler and the runtime system make use of an efficient method of compressing three ASCII characters into two bytes of storage. This method is known as RAD50 packing. The basis of the method is arithmetic performed under radix 50 octal, hence the name RAD50.

G.1 THE RAD50 CHARACTER SET

Only the uppercase letters, digits, and the blank character can be represented in this fashion. Lower case letters are folded to upper case. The ASCII characters are coded from 0-47 octal:

Character	Octal Code
blank	0
A-Z	1-32
a-z	1-32
0-9	36-47

The values from 33-35 are not used in FORTRAN, although other systems incorporating RAD50 packing commonly assign $, ., and , to these values.

G.2 THE RAD50 ALGORITHM

The RAD50 procedure for packing three ASCII characters into two bytes is as follows:

1. The code of the first character is multiplied by 3100 octal (50 x 50).

2. The code of the second character is multiplied by 50 octal and added to the first.

3. The code of the third character is added the sum of the first and the second.

If less than three characters are to be packed, the string must be space filled.

G.3 FORTRAN SUBROUTINE FOR RAD50 CONVERSION

Generally, six ASCII characters are packed into four bytes of storage. The six characters may represent a file name, procedure, or common block (Chapter 9). The following subroutine is provided for performing RAD50 conversions.

```
*
* FORTRAN subroutine for RAD50 conversions
*
*   string - six byte character variable
*   rad50  - four byte integer variable
*   mode   - four byte integer variable (0=pack, 1=unpack)
*
      subroutine RAD50 (string,rad50,mode)

      character string(6)
      integer*2 rad50(0:1)
      integer i, j, k, l
      integer PACK, UNPACK, RADIX

      parameter (PACK=0, UNPACK=1, RADIX=O'50')


      select case (mode)

*
* "packing" routine
*
        case (PACK)

           rad50(0)=0; rad50(1)=0

             do (i=1,6)
                 select case (string(i))
                   case ("A":"Z")
                     l = 64
                   case ("a":"z")
                     l = 96
                   case ("0":"9")
                     l = 18
                   case (" ")
                     l = 32
                   case default
                     write (9,101) string(i)
                     stop
                 end select
               j = (i-1)/3
               k = RADIX**mod(6-i,3)
               rad50(j) = rad50(j) + (ichar(string(i))-l)*k
             repeat
```

```
*
* "unpacking" routine
*

        case (UNPACK)

          do (i=1,6)
            j = rad50((i-1)/3)
            if (j<0) j = j+65536
            k = mod(i,3)
              if (k=1) then
                k = j/RADIX**2
              else if (k=2) then
                k = mod(j,RADIX**2)/RADIX
              else
                k = mod(j,RADIX)
              end if
              select case (k)
                case (1:26)
                  k = k+64
                case (30:39)
                  k = k+18
                case (0)
                  k = 32
                case default
                  write (9,102)
                  stop
                end select
            string(i) = char(k)
          repeat

*
* any other value for mode is an error
*
        case default

          write (9,100) mode
          stop

      end select

      return

100   format ("illegal value for mode: ",i3)
101   format ("illegal character: ",a1)
102   format ("illegal value in rad50")

      end
```

APPENDIX H

CALLING C FUNCTIONS FROM FORTRAN

Microsoft FORTRAN 77 is designed to interface easily and naturally to most implementations of the C programming language. A C function may be invoked with either a subroutine call or a function reference. The following considerations should be made before attempting to call a C function from FORTRAN:

1. C must maintain a frame pointer in address register A6; all other registers may be considered volatile.

2. FORTRAN arguments are always passed by address; never by value. When passing constants, except for CHARACTER arguments, FORTRAN copies the constant (to prevent inadvertent modification) and passes the address of the copy of the constant. This means that C functions which are called by FORTRAN must declare their arguments to be pointers.

3. FORTRAN pushes argument addresses on the stack from left to right, while C expects arguments on the stack to have been passed from right to left. Consequently, either the FORTRAN referencing statement or the C function declaration statement must have their argument lists reversed. FORTRAN is responsible for taking argument pointers off the stack.

4. C must return INTEGER, LOGICAL and REAL values in data register D0 and DOUBLE PRECISION and COMPLEX values in data registers D0 and D1. All floating point values ("float" and "double") must be IEEE compatible. FORTRAN expects -1 rather than 1 for the logical value true. Some implementations of the C language return floating point values in global variables rather than in registers.

5. FORTRAN returns a pointer to the value of a CHARACTER function in address register A1. You probably will not be able to use a C function which returns characters or structures.

6. C expects all string arguments to be null terminated. FORTRAN always space fills character arguments. To pass a string to C, use the following FORTRAN expression to remove trailing blanks and correctly terminate the string.

   ```
   TRIM(STRING)//CHAR(0)
   ```

In order to link C functions to FORTRAN programs, the compiler must generate an assembly language source file with external references properly declared. This is accomplished by specifying the A and J compiler options. The compiler output is assembled using the resident assembler.

Some C runtime environments need global variables such as "errno" which are usually implicitly provided in the program segment "main". Global declarations for these variables should be added to your C function. The C function must be compiled only to the object file stage. This is usually accomplished with a c option.

When all the required pieces are in object file format, the resident linker is used to put them all together. The local C library will usually have to be specified on the linker command line.

The creation of the final executable file is dependent on the resident assembler and linker. Consequently, your program will probably no longer be reentrant nor position independent. You will not be able to use the FORTRAN linker. All other features of Microsoft FORTRAN 77 are still available, including virtual arrays and overlays.

APPENDIX I

FILE PRECONNECTIONS

The current implementation of the Microsoft FORTRAN Compiler supports the following preconnected units:

Unit	Description
5	card reader (Input)
6	line printer (Output)
7	magnetic tape (Input/Output)
9	user terminal (Input/Output)

For the Macintosh, unit 9 is the Macintosh keyboard for input and the currently active window for output. Text output to unit 9 is printed in the current font, character size, and character attributes. When any FORTRAN application is started, a default window is created and made current with the Monaco-9 font and plain text attributes.

Output to unit 9 can be temporarily stopped by entering Command-S (the Command key and the "S" key held down simultaneously). To continue, enter Command-Q.

The files normally preconnected to units 5 and 7 are not present on the Macintosh and attempting to use these unit numbers as preconnected units will lead to unpredictable results. However, the unit numbers may be used after connecting them to a file with an OPEN statement.

APPENDIX J

RESTRICTIONS ON THE MICROSOFT IMPLEMENTATION OF FORTRAN 77

1. INTEGER*1 and INTEGER*2 are provided as data types for symbols but are not supported as constants. Expression evaluation is always performed using thirty-two bit values to prevent overflow of intermediate results. It is not possible to pass a one or two byte constant through a procedure argument list.

2. The maximum length of a direct access record, a formatted sequential access record, and an unformatted sequential access record with record length information (see Chapter 8) is 1024 bytes.

3. When using list directed input on character data types the constant may be delimited by either apostrophes or quotation marks. If the constant is not delimited, a space or end of record serves as a value separator.

4. The console device (UNIT 9) cannot be rewound nor backspaced.

5. The runtime system attempts to distinguish between block and character structured files. Block files are accessed using internally buffered I/O, while character file buffers are flushed after every WRITE statement. If execution is aborted, the buffers of block structured files may not be flushed.

6. Leading as well as trailing blanks are removed from file names.

7. Symbolic names are significant to thirty-one upper and lower case characters.

Microsoft® FORTRAN Compiler for the Apple® Macintosh™

Appendices

Index

Index

Microsoft® FORTRAN Compiler
for the Apple® Macintosh™

Index

Index

Utilities

Microsoft® FORTRAN Compiler
for the Apple® Macintosh™

Utilities

Microsoft® FORTRAN Compiler
for the Apple® Macintosh™

Linker and Library Manager

Contents

CHAPTER 1

INTRODUCTION TO THE MANUAL

1.1 PREFACE

This manual explains the use of the Microsoft FORTRAN 77 linker (link) and the library manager (lib). The Microsoft FORTRAN Compiler can generate completely or partially linked object modules or assembler source code. The linker and library manager deal with the object modules produced by the compiler.

It is assumed that you are already familiar with the Microsoft FORTRAN Compiler.

1.2 NOTATION AND TERMS

The following is an explanation of the notation and terms we use in the manual:

{}	Curly brackets contain optional elements.
Infile	Input file.
Outfile	Output file.

CHAPTER 2

FORTRAN 77 LINKER (LINK)

2.1 DESCRIPTION

Linking external procedures to a program, in general, involves locating the procedures and making them accessible to the program. A completely linked executable object module may be created by either the Microsoft FORTRAN Compiler, or the linker, link. The compiler and the linker perform static (permanent) linking. In addition, the FORTRAN run time system is capable of providing dynamic (temporary) linking to external procedures.

If the run time system locates an external procedure, either in memory or on the disk, a copy of the procedure is placed in memory as an overlay. Execution of the procedure is done just as if it were part of the original source file. This is quite useful for referencing commonly used FORTRAN subprograms or FORTRAN compatible assembly language routines, such as the date and time subroutines supplied with the compiler.

Dynamic linking allows you to skip the traditional link and load process. Furthermore, having one shareable copy of a commonly used procedure stored on the disk saves on disk space. Note that for dynamic linking, there can only be one entry point per file, and the file name must have the same name as the subprogram.

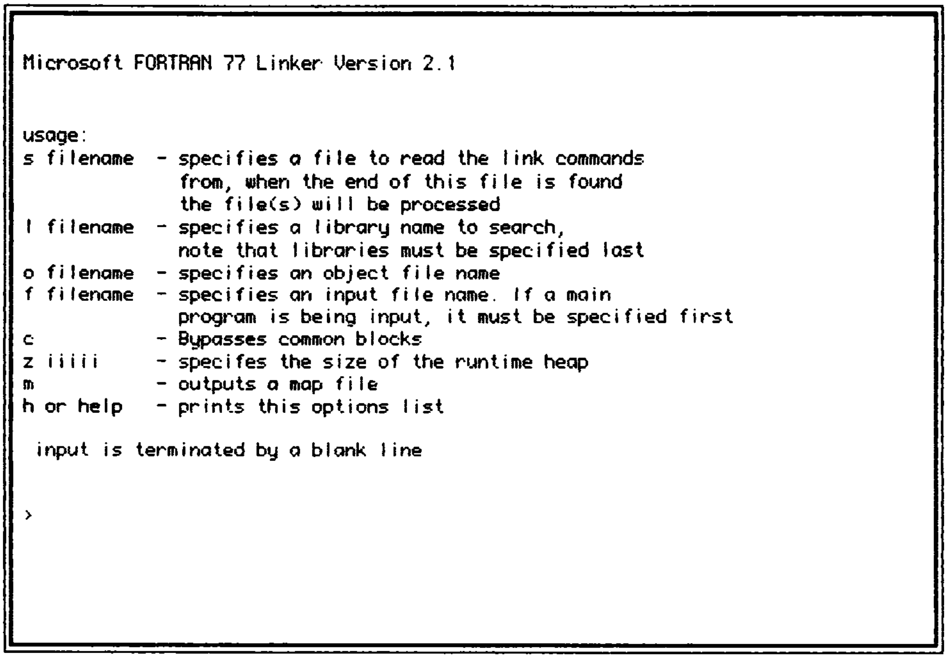
Dynamic linking may not always be desirable; you may want to have a fully linked program file. The primary purpose of link is to bind separate procedure files into a single resolved file. It accepts procedure file names and library file names as input. link will link procedure files specified interactively and/or procedures files contained in libraries to a main program or subprogram.

link is a two pass linker. In the first pass the linker scans all the files and collects information. In the second pass the linker links procedure files, and may resolve references using the information it collected during the first pass.

A statically linked program will run faster because the linked procedures are a part of the program when it is executed; whereas, dynamically linked external procedures are linked during execution of a program by the run time system.

2.2 INVOCATION

From the Macintosh desktop, open the link application by selecting its icon and choosing Open from the File menu or by "double clicking" the link icon. The linker window will appear as:



```
Microsoft FORTRAN 77 Linker Version 2.1


usage:
s filename  - specifies a file to read the link commands
              from, when the end of this file is found
              the file(s) will be processed
l filename  - specifies a library name to search,
              note that libraries must be specified last
o filename  - specifies an object file name
f filename  - specifies an input file name. If a main
              program is being input, it must be specified first
c           - Bypasses common blocks
z iiiii     - specifes the size of the runtime heap
m           - outputs a map file
h or help   - prints this options list

 input is terminated by a blank line


>
```

2.3 COMMANDS

INPUT FILES

The f command is used to specify the files which contain the procedures you want to link together. If you have a main program, it must be in the first file you specify. The linker provides a default extension of ".sub" for all file names you enter after the first one. If you do not have a main program the first file name must be given with the explicit extension of ".sub"; otherwise, the linker will assume it contains a main program.

SCRIPT FILE

The s command allows you to specify linker commands in a script. This is especially useful if, during application development, you are linking a particularly long or complicated list of procedures.

OUTPUT FILE

The o command provides a means to specify an explicit linker output file. If an output file is not specified, the first file name in the file list will be the default output file. The output file is the only file that is modified; however, if the linker exits due to an error, the output file will not be modified.

LIBRARY FILES

Libraries can be searched for procedures by using the l command. Microsoft FORTRAN 77 Libraries are maintained by the library manager, lib. ".fl" is the default extension for libraries.

The run time library, f77.rl can be linked to a main program by specifying it as a library file.

LOAD MAP FILE

The load map option, m, causes a load map file to be created. The name of the file containing the map listing is the same name as the output file with an extension of ".mp". See section 1.7 (MAPS), for a description of the load map.

HEAP SIZE

The heap size option allows you to change the size of the heap of your program. The heap is the memory space where f77.rl communication occurs; where static storage, file buffers, and virtual array buffers are allocated; and where overlays are loaded. The size is specified with the z option and is the number of kilobytes reserved for the heap. See section 1.8 (SPECIAL USES) for more information on manipulating heap sizes.

COMMON BLOCK LINKAGE SUPPRESSION

Normally, link attempts to link common block declarations to BLOCK DATA subprograms that declare the common blocks. If your program has more than 150 common block declarations, the linker's common block table will overflow. If the linker emits the error message:

```
?common block table overflow in file: filename
Use the 'c' option as described in linker manual.
```

you have two alternatives:

If your program does not require linking common blocks to BLOCK DATA subprograms, you can use the c option to link your files together. The c option causes the linker to suppress common block linkage.

If you are linking BLOCK DATA subprograms, use the following procedure:

1. declare those common blocks that need initialization in your main program.
2. link only the necessary BLOCK DATA subprograms to your main program.
3. link all other desired files to the output file which was created by step 2 using the c option.

2.4 OPERATION

2.4.1 Procedure files

Procedure files specified with the f command are linked to a main program or subprogram unconditionally. They may be either FORTRAN procedure files or assembly language procedure files. The linker keeps a list of all the entry points it finds in procedure files. An error message will be printed if a duplicate entry point is encountered and the linker will exit.

If the linker encounters duplicate named common initialization in a BLOCK DATA subprogram while searching a file from the file list, it will display an error message and then exit. This will not occur if the c option is used.

2.4.2 Libraries

Library procedure modules are linked conditionally. They are linked only if they contain a procedure which will satisfy an unresolved reference.

If the linker encounters duplicate entry names or duplicate named common initializations while searching the libraries, it simply counts the number of collisions. It will not give an error message. The first occurrence of an entry name will be used, and the first initialization of a named common will be used.

2.4.3 Linker display

The linker reports information to the user regarding its progress. This display can take varying formats depending on whether the load map option is specified.

2.4.3.1 Linker Passes

<u>link</u> first displays a variety of information about the program before any of the files are linked. This information is explained below. Second, <u>link</u> displays the names of the files and the library procedure modules as they are linked. Finally, <u>link</u> displays the final code size in bytes, the heap size, and the names of any unresolved external references and their relative locations.

The following discussion summarizes the operation of the linker without the m and c options specified.

PASS ONE

The linker begins its first pass which involves counting the number of entry points, unresolved references, uninitialized named commons, and block data commons while it searches files from both the file list and the libraries. Only information from library procedure files that are to be linked is counted. After pass one, link displays this information:

```
1: reading tables - pass one complete
           n entry points
           n unresolved external references
           n uninitialized named common references
           n block data commons
           n entry name collisions
           n block data named common collisions
2:
```

where n is an integer.

1. "entry points" are defined by PROGRAM, SUBROUTINE, FUNCTION, ENTRY, and BLOCK DATA statements. The number of entry points collected by the linker will include even those entry points which are not referenced.

2. "unresolved external references" are those references which are not, as of pass one, resolved. Some of these unresolved references may become resolved during pass two. The names of all unresolved references in the final code will be displayed just before the linker exits.

3. "uninitialized named commons references" are those named commons which are not initialized by BLOCK DATA subprograms.

4. "block data commons" are those named commons which are initialized in BLOCK DATA subprograms.

5. "entry name collisions" are those entry names found more than once while the linker is searching libraries. All entry name occurrences found after the first occurrence are ignored.

6. "named common collisions" are those named commons found initialized in more than one BLOCK DATA subprogram. Only the first initialization of a named common found by the linker is recognized.

   For example, the following would cause one block data named common collision:

   In a program:

```
        Block data ABC
        Common /a/aa,bb,cc
        Common /b/dd,ee,ff
        Data aa,bb,cc/1.0,2.0,3.0/
        Data ee,ff,gg/4.0,5.0,6.0/
        end
```

   In a library procedure file:

```
        Block data XYZ
        Common /a/xx,yy,zz     ! "a" common block collision
        Common /c/cc,dd,ee
        Data xx,yy,zz/12.0,22.0,33.0/
        Data cc,dd,ee/44.0,55.0,66.0/
        end
```

PASS TWO

The linker then proceeds with the second pass during which files from the file list are linked unconditionally and library procedure files are linked conditionally.

After pass two, link displays the names of the procedure files it used:

```
1: reading tables - pass one complete
           n entry points
           n external references
           n uninitialized named common references
           n block data commons
           n entry name collisions
           n block data named common collisions
2: Processing -
        XXXX     at 0xnnnnnnnn
        XXXX     at 0xnnnnnnnn
        XXXX     at 0xnnnnnnnn
```

where XXXX is either the name of a file from the file list or a procedure file contained in a library, and 0xnnnnnnnn (in hexadecimal) is the relative location of XXXX.

Finally, the linker displays the final code size in bytes, the heap size, and the names of any unresolved external references and their relative locations.

```
1: reading tables - pass one complete
           n entry points
           n external references
           n uninitialized named common references
           n block data commons
           n entry name collisions
           n block data named common collisions
2: Processing -
        XXXX     at 0xnnnnnnnn
        XXXX     at 0xnnnnnnnn
        XXXX     at 0xnnnnnnnn
3: Program file complete:    nnn bytes
   Heap size:  nnnnn bytes

   Unresolved references:

XXXX  0xnnnnnnnn  XXXX  0xnnnnnnnn
```

2.5 ASSEMBLY LANGUAGE PROCEDURES FILES

If a procedure was assembled by the resident assembler, it must be position independent and have been linked by the resident linker to create an executable image. The name of the entry point is the first six characters of the file name, and the entry point is the first word of the code in the procedure. The FORTRAN linker can then link the assembly procedure file. Appendix F of the Microsoft FORTRAN Compiler manual describes writing assembly language procedures callable from FORTRAN.

2.6 LIBRARIES

Libraries contain collections of procedures, called modules, and are created and managed by the Microsoft FORTRAN library manager, lib. The linker makes only one pass through each library searching for modules which contain entry points that satisfy unresolved external references. A library module itself can contain unresolved external references. Since the linker only makes one pass through a library, a module which is only referenced by another module, must follow the referencing module.

2.6.1 Ordering within libraries

Procedures can be arbitrarily ordered within a module. However, the modules themselves may need to be ordered. For example, given a main program "main", and two modules "a.sub" and "b.sub", "a.sub" contains a subroutine called by "main", and "b.sub" contains one subroutine which is called only by "a.sub". If the modules containing "a.sub" and "b.sub" are in library A, then "b.sub" must follow "a.sub" in the library. If modules "a.sub" and "b.sub" are in library A and library B respectively, then library B must follow library A in the list of libraries specified with the l option.

Alternatives to ordering modules and libraries are to either list libraries repeatedly with the l option, or repeatedly invoke the linker until all references are resolved. When a library is listed more than once, each occurrence of the library is treated as if it were a separate library file. Since the linker can be used serially, multiple invocation may used as a method of resolving backward library references.

2.7 MAPS

The load map file contains the following information:

```
Linkage Map of: <filename>
Code size:     nnn bytes
Heap size:     nnn bytes


         Entry points:

XXXX      0xnnnnnnnn  XXXX      0xnnnnnnnn  .....



         Unresolved external references:

XXXX      0xnnnnnnnn  XXXX      0xnnnnnnnn  .....
```

where <filename> is the name of the output file, nnn is an integer, XXXX is the name of a procedure, and 0xnnnnnnnn is a relative location.

1. Entry points

   All entry names found in the final code and their relative locations.

2. Unresolved external references

   All unresolved external references found in the final code and their relative locations at which they are referenced.

2.8 SPECIAL USES

The following is a list of special uses that the linker provides:

1. a way to gather information about a single file. The linker will display the number of entry points, unresolved external references, uninitialized named commons, and block data commons found in the file. Also, the size of the code, the heap size, and the names of any unresolved external references and their relative locations are displayed. You get the same information as explained in section 1.4 (OPERATION). You may also specify the load map option.

2. a way to manipulate the heap size of your program. You can make the heap size for your program larger or smaller.

3. a way to create a stand alone module. A program is a stand alone module if all references are resolved and the run time library is linked.

4. a way to bind subprograms together without a main program. Suppose you have a subprogram which you want dynamically linked in order to save on memory space. This subprogram repeatedly calls several external procedures which themselves must be dynamically linked by the run time system. Dynamically linking these procedures takes up more time than if these procedures were statically linked, using link, to the subprogram.

2.9 ERROR MESSAGES

<u>?File not found -</u> filename
The procedure file you specified could not be located with the volume information (or defaults) given.

<u>?Run time library can only be linked once</u>
The run time library already is linked to the first file specified in the file list.

<u>?multiple entry name in file:</u> filename
A duplicate entry name was encountered in the procedure file while the linker was searching it.

<u>?common block name collision in file:</u> filename
A duplicate initialization of a named common was encountered in the procedure file while the linker was searching it.

<u>?cannot be linked without a main program</u>
The run time library, F77L.RL, cannot be linked without specifying a main program.

<u>?illegal output file extension: 'MP'</u>
The output file cannot use the load map file extension.

<u>?cannot link just libraries</u>
You can only link libraries when a procedure file is specified.

<u>?file is not a FORTRAN library:</u> filename
The library file you specified was not created by <u>lib</u>.

<u>?File is illegal type:</u> filename
The procedure file you specified is not an executable file.

<u>?entry table overflow in file:</u> filename
The maximum number of entry points the linker can keep track of is 250.

<u>?external table overflow in file:</u> filename
The maximum number of unresolved external references that the linker can keep track of is 500.

<u>?block data table overflow in file:</u> filename
The maximum number of block data entry points that the linker can keep track of is 50.

<u>?common block table overflow in file:</u> filename
The maximum number of named common blocks that the linker can keep track of is 150.

?error while renaming tempfile _to_ outfile
    link was unable to rename the temporary file to the output file name you specified. The temporary file is in the current directory.

CHAPTER 3

FORTRAN 77 LIBRARY MANAGER (LIB)

3.1 DESCRIPTION

lib provides a means for grouping FORTRAN and assembly language procedure files together in a library which can be used for input to the Microsoft FORTRAN 77 linker, link. Procedure files must be in executable format.

Libraries contain collections of procedures, called modules. The linker makes only one pass through each specified library searching for modules which contain entry points that satisfy unresolved external references. Since the linker only makes one pass through a library, a module which is only referenced by another module, must follow the referencing module.

3.2 FORTRAN PROCEDURE FILES

FORTRAN procedure files may have more than one entry point, and may contain unresolved references. Valid entry points are those defined by SUBROUTINE, FUNCTION, ENTRY, and BLOCK DATA statements.

3.3 ASSEMBLY LANGUAGE FILES

If a procedure was assembled by the resident assembler, it must also have been linked by the resident linker to create an executable image. It must also be position independent, and have one entry point being the first word of code in the file. Appendix F of the Microsoft FORTRAN Compiler manual describes writing assembly language procedures callable from FORTRAN.

3.4 INVOCATION

From the Macintosh desktop, open the _lib_ application by selecting its icon and choosing Open from the File menu or by "double clicking" the _link_ icon. A prompt for the name of the library file will be issued:

```
Library File:
```

Enter the name of an existing or new library after the prompt. The library manager prompt ">" will then be given, indicating that _lib_ is ready for commands.

3.5 COMMANDS

lib accepts the following seven commands:

a files	add a list of files to the library. A file cannot be placed in a library twice.
c files	replace the files in the library with those on the command line.
d files	delete a list of files from the library.
l	list all files in the library.
m files	list a load map of all the entry names and their relative locations which are in the library's procedure files.
h	list the library's commands. An invalid command will also cause the command list to be listed.
q	exit from the library manager. If any additions, changes, or deletions have been requested, all procedure files are shown as they are being processed.

3.6 ERROR MESSAGES

filename _is not a FORTRAN library_
The library file name you entered was not a FORTRAN procedure library file.

library not found
The library file you entered was not found in the current or specified directory.

filename _not found in library_
The procedure file you want to change or delete is not in the library.

filename _not found_
The file you entered on the command line could not be located.

library full
The maximum number of procedure files a library may contain is 210.

filename _is already in the library_
The procedure file you wanted to add is already in the library.

You may also see any of the standard FORTRAN error messages.

APPENDIX A

LINKER TABLE FORMAT

A.1 TABLE FORMAT

The Microsoft FORTRAN Compiler appends a linker table to each output file for use by the linker, link.

In the following, all addresses are relative to the first word of executable code in a file.

The table has two sections: an entry section and an external section.

Each element of the entry section is composed of two long words. The first long word contains the name of the entry packed RAD50. The second long word contains the address of the entry. The entry section is terminated by a long word containing zero.

Following the entry section is the external section. Each element of the external section is one long word. The long word contains an address which is the location of a 'MOVE.L #N,D1' instruction, where 'N' is the name of the external reference packed RAD50 (used by the overlay manager as the root file name). The external section is terminated by a long word containing zero.

The last four bytes of the file contain the address of the linker table.

Microsoft® FORTRAN Compiler
for the Apple® Macintosh™

Subroutines

SPOOL

SPOOL is an external FORTRAN subroutine for spooling a disk file to a line printer. SPOOL is automatically called by the run time system to print files created by writing to unit 6. SPOOL can also by called as a subroutine to print any ASCII text file. To spool a file to a printer, call SPOOL as follows:

```
CALL SPOOL(FILE,SWTCHES,COPIES,LPP,WIDTH)
```

where:

FILE	is a character expression which, with trailing blanks removed, is the name of the file to be spooled. The expression must be terminated by a least one space character.
SWTCHES	is an integer expression specifying a control code for certain parameters relating to the printing of the file. This code represents the sum of the codes required as listed below:

```
 4 - DELETE
 8 - NODELETE
16 - HEADER
32 - NOHEADER
```

COPIES	is an integer expression specifying the number of copies to be printed.
LPP	is an integer expression specifying the number of lines per page.
WIDTH	is an integer expression specifying the page width.

If any of the integer arguments are passed as variable names, they must be declared as four byte integers (INTEGER*4). Arguments may be omitted from right to left.

TIME

TIME is an external FORTRAN subroutine for returning the system time as seconds since midnight. TIME is called as follows:

```
      INTEGER SECONDS
      CALL TIME(SECONDS)
      WRITE (9,100) SECONDS
100   FORMAT (I6)
      END
```

```
 63256
```

DATE

DATE is an external FORTRAN subroutine for returning the system date. DATE is called as follows:

```
      INTEGER MM,DD,YY
      CALL DATE(MM,DD,YY)
      WRITE (9,100) MM,DD,YY
100   FORMAT (I2.2,2('/',I2.2))
      END
```

```
09/18/83
```

Microsoft® FORTRAN Compiler
for the Apple® Macintosh™

Edit Manual

Edit

This manual describes Edit, Apple's general-purpose text editor. In the context of the Microsoft FORTRAN language system, its primary use is to enter and edit FORTRAN source programs. This document has been reprinted with the permission of Apple Computer, Inc.

Files Required

If you wish to move the Editor to another disk, you must move the file named Edit. The transfer facility in the Editor was designed for the Macintosh 68000 Development System and has hard wired names for the applications supplied with that system.

Invoking the Editor

There are several ways to use the Editor:

1. from the Finder, select and open the application named Edit.

2. from the Finder, select and open a text file created by the Editor. You can open up to four files simultaneously by selecting a group of them (by shift-clicking them or dragging across multiple icons) before opening one of them. All files created using the Editor can be selected.

3. select Edit from the Transfer option of the compiler or debugger.

About the Editor

The Editor is a disk-based editor. Thus, it is capable of editing documents much larger than will fit in memory. When a document is open, you can use the scroll bars to move, both vertically and horizontally, through the document. The Editor brings new portions of the document into memory as they're needed.

To create a new document, select New from the File menu.

There are several ways to open existing documents:

1. To open an existing document, select the uppermost Open command from the File menu. This opens a standard file selection box from which you select the file to be opened. All files with type 'TEXT' can be opened from this menu.

2. You can also open files (including non-text files) by selecting the name of the file in an open document, and then choosing the other Open command from the File menu.

3. Finally, you can open a document by typing Command-K followed by the name of the file to be opened (including volume name if needed), and pressing Return. This technique is not listed in a menu, and it gives no visual feedback until the file is opened or not found.

As many as four such documents can be on the desktop at a time. When you quit the Editor or transfer to another application, the Editor gives you a chance to save each document that has been altered.

About the Editor

Editor documents consist of lines of text that are separated by Return characters. The Editor has no tools for manipulating or organizing pages, paragraphs, sentences, or pictures.

When you type long lines of text, characters may be placed past the right edge of the window. To see these characters, use the horizontal scroll bar. It is possible to type a line longer than can be seen using the scroll bar. The text on such lines is not lost, but neither is it visible. To see the whole line, insert a Return into middle of the line, breaking the line into smaller pieces.

The Editor displays an entire document in text of a single size and font. The Monaco font, a monospaced font, is the default. Different documents on the desktop can have different fonts and font sizes.

Editing

Editing involves inserting text at the insertion point and removing, moving, copying, or replacing a selection. Any character or sequence of characters in a document can be selected and edited.

You can replace the selection by typing or pasting. You can remove, move, or copy the selection using commands from the Edit menu or their keyboard equivalents. Cut or copied selections can be pasted into another place in the document, into another window (such as the Find or Change window), or into another document altogether.

You can find and change text using the Find and Change commands in the Search menu. These commands search for a specific string starting at the current insertion point. If the string is found, it's either selected and displayed or replaced. If not, a box is displayed to notify you that the string was not found. When you select Find or Change, the currently selected string is used as the default string to find.

Tabs and Alignment

The Editor has several features that help organize programs visually. Tab stops allow you to align columns of text at regular intervals across the page; the Set Tabs command in the Format menu lets you set the distance between tab stops.

The Auto Indent command in the Format menu lets you turn Auto Indent on and off. If Auto Indent is on, the insertion point is automatically lined up with the leftmost edge of the previous line each time you press Return. To back the cursor up to the left edge of the screen, use the Backspace key. If Auto Indent is off, the insertion point is placed at the left margin.

The Align command in the Edit menu aligns the left margin of all the lines in a selected block of text. The Move Left and Move Right commands, also in the Edit menu, move all the lines in a selected block of text one space left or right. If a proportional font is selected, the width of one space is usually quite small. The easiest way to move a block of text several spaces is to press the keyboard equivalent several times in succession.

Document Format

Text created by the Editor is saved as a document file. A document file is a text-only file that can be used by other applications that use text-only files. For example, the Text Only option of MacWrite (see Save As in the MacWrite manual) creates text-only files that can be used by the Editor.

A text-only file is a stream of ASCII characters. It contains no special formatting information.

Printing Documents

The Print command in the File menu allows you to send a copy of the document to a printer. After selecting this command, you are presented with two dialog boxes. The first lets you specify the size of the paper you are using. The second dialog box lets you choose the print quality (High, Standard, or Draft), which pages to print, how many copies to print, and whether the paper is continuous or separate sheets.

These two boxes are standard printing dialog boxes, and are discussed in some detail in the other manuals (for example, MacWrite).

Microsoft® FORTRAN Compiler
for the Apple® Macintosh™

Resource Compiler (RMaker) Manual

Resource Compiler

This manual describes RMaker, an Apple application that is used to produce resource files and to integrate resources into applications. This document has been reprinted with the permission of Apple Computer, Inc.

The first part of this manual describes RMaker. The next part of the manual describes how to create an RMaker input file using predefined resource types and user-defined resource types. The final part of the manual tells how to use RMaker to create a new resource file from the input the file.

About RMaker

RMaker is the Macintosh 68000 Development System's Resource Compiler. It is very similar to the RMaker program in the Lisa Workshop, but some changes have been made to the syntax. Be careful if you are converting resource files from one system to the other.

RMaker takes a text file as input, and produces a resource file. The text file contains an entry for each resource, as described below. These entries can specify all information necessary to define the resources, or they can cause existing resources to be read from other files.

RMaker Input Files

An RMaker input file is a text file, that may be created using the Editor. By convention, RMaker input files have the extension .R.

RMaker ignores all comment lines and blank lines (except in some cases a blank line may be required). It also ignores leading and embedded spaces (except in lines defined to be strings). Comment lines begin with an asterisk. To put comments at the end of other RMaker lines, precede the comment with two consecutive semicolons (;;).

For example, during program development, you'll typically use separate application and resource files. Once the application is finished, you should combine these files. Simply use the INCLUDE statement to read in the application created by the Microsoft FORTRAN Compiler. It is already stored as resources of type CODE.

Naming the Resource File

The first nonblank and noncomment line of the input file specifies the name of the resource file to be created. If the file is to be an application, it should have no extension. If not, the file will be a resource file and should have the extension .Rsrc. The line following the resource's filename should either specify the file type and creator bytes for the Finder, or be blank. For example, the two lines

```
NewResFile.Rsrc
PNTGMPNT
```

specify the file names NewResFile.Rsrc as the output file, and the bytes 'PNTGMPNT'as the type and creator bytes. These bytes tell the Finder that the file is a painting file, created by MacPaint. (The Finder will try to launch MacPaint if you select and open this file!)

More typically, these two lines will look like this:

```
MyApplication
APPLMYAP
```

This designates the file MyApplication as the output file. The file is an application (type 'APPL') of type 'MYAP'.

If you do not specify a value for these bytes, they are set to 0.

Appending to an Existing Resource File

If you wish to add the resource defined in your input file to those in an existing resource file, simply precede the filename with an exclamation point. For example

```
!OldResFile.Rsrc
```

tells RMaker to add the new resources to the file OldResFile.Rsrc

Note

> If you are adding resources to an application created by the Microsoft FORTRAN Compiler and you also want to use the Microsoft FORTRAN linker (LINK) to link external subprograms to your application, use the linker first. The linker expects the application file to contain only those resources created by the Microsoft FORTRAN compiler (3 CODE type resources). The extra resources added by RMaker will confuse the linker and have unpredictable results.

Adding Resources

The rest of the resource file consists of INCLUDE statements and "Type statements".

INCLUDE statements are used to read in entire resource files. An INCLUDE statement looks like this:

```
        INCLUDE  filename
```

Type statements consist of the word "Type" followed by the resource type and, below that, one or more resource definitions. The resource type must be capitalized to match the predefined resource type.

The following statement creates three resources of type 'STR'.

```
        TYPE STR
          ,1
        This is a string
          ,2
        Gnirts a si siht
          ,3
        Hits is a grints
```

It is not necessary for all resources of a given type to be declared together, however, all resources of a type must have unique resource ID's. If you specify a resource ID that is already in use, the new resource replaces the old one.

A resource looks like this:

```
        [resource name] ,resource ID [(resource attribute byte)]
        type-specific data
```

The square brackets indicate that the resource name and resource attribute byte are optional. Don't place these brackets in your input file. The comma before the resource ID is mandatory. The default attribute byte is 0. Here are some sample resource definitions:

```
        TYPE STR
        NewStr ,4 (32)
        This resource has a name and an attribute byte!!
          ,5
        This one has only a resource ID.
        MyNewStr,6
        This has a name and a resource ID.
```

The type specific data is different for each resource type. As you have probably guessed, the type-specific data for a 'STR' resource is simply a string. The next section describes the type-specific data for the resource types defined by RMaker.

Defined Resource Types

RMaker has 12 defined resource types: 'ALRT', 'BNDL', 'CNTL', 'DITL', 'DLOG', 'FREF', 'GNRL', 'MENU', 'PROC', 'STR', 'STR#', and 'WIND'. The format for type-specific data for each type is shown by example, below. The type 'GNRL' is used to define your own resource types. It is explained later.

Syntax of RMaker Lines

There are a few general rules that apply to lines read by RMaker.

1. Leading and embedded blanks are ignored, except when necessary to separate multiple numbers on a line, or when they are part of a string.

2. Numbers are decimal unless specified otherwise.

3. RMaker is sensitive to line breaks. Thus if a type description, below, shows four values on a single line, you must put four values on a single line.

Two special symbols can be used in resource definitions: the continuation symbol (++) and the entry ASCII symbol (\).

++	goes at the end of a line that is continued on the next line.
/	precedes to hexadecimal digits. The ASCII character is entered into the resource definition.

Look at the description of the 'STR' type for examples of these special symbols.

You will notice that some of the resources are listed as templates, while others are not. A template is a list of parameters to build a Toolbox object; it is not the object itself.

ALRT Alert Template

```
TYPE ALRT
  ,128                  ;; resource ID
50 50 250 250           ;; top left bottom right
1                       ;; resource ID of item list
7FFF                    ;; stages word in hexadecimal
```

BNDL Application Bundle

```
TYPE BNDL
  ,128                  ;; resource ID
MPNT 0                  ;; bundle owner
ICN#                    ;; resource type
0 128 1 129             ;; local ID 0 maps to resource ID 128; 1 to 129
FREF                    ;; resource type
0 128 1 129             ;; local ID 0 maps to resource ID 128; 1 to 129
```

Note: the number of mappings from local ID to resource ID is variable. Simply include multiple mappings on a single line.

CNTL Control Template

```
TYPE CNTL
  ,130                  ;; resource ID
stop                    ;; title
244 40 260 80           ;; top left bottom right
Invisible               ;; see note
0                       ;; ProcID (control definition ID)
0                       ;; RefCon (reference value)
0 1 0                   ;; minimum maximum value  *** order different??
```

Note: Controls can be defined to be visible or Invisible. Only the first character (V or I) is significant.

DITL Dialog or Alert Item List

```
TYPE DITL
  ,129                   ;; resource ID
5                        ;; 5 items in list

staticText               ;; static text dialog item (see note)
20 20 32 100             ;; top left bottom right
Whoopie                  ;; message

editText                 ;; editable text dialog item (see note)
20 120 32 200            ;; top left bottom right
Default message          ;; message

radioButton              ;; radio button dialog item (see note)
40 40 60 150             ;; top left bottom right
Hello                    ;; message

checkBox Disabled        ;; disabled dialog item (see note)
75 40 95 150             ;; top left bottom right
Goodbye                  ;; message

button                   ;; button dialog item (see note)
75 160 95 200            ;; top left bottom right
Hi!                      ;; message
```

Note: Five types of dialog items are defined: Static text, Editable text, Radio Buttons, Check Boxes, and Buttons. These items are assumed to be enabled. Otherwise you may specify Disabled. Only the first character of an item definition word is significant (S,E,R,C,B,D).

DLOG Dialog Template

```
TYPE DLOG
  ,3                     ;; resource ID
This is a dialog box.    ;; message
100 100 190 250          ;; top left bottom right
Visable GoAway           ;; box status (see note)
0                        ;; procID (dialog definition ID)
0                        ;; refCon (reference value)
129                      ;; ID of item list ('DITL', above)
```

Note: A dialog box can be Visible or Invisible. GoAway and NoGoAway determine whether or not the dialog box has a close box. Only the first characters (V,I,G,N) are significant.

FREF File Reference

```
TYPE FREF
  ,128                  ;; resource ID
APPL 0                  ;; file type, local ID of icon

  ,129                  ;; resource ID
TEST 127 myFile         ;; file type, local ID of icon, filename
```

Note: If there is no filename, it can be omitted.

MENU Menu

```
TYPE MENU
  ,3                    ;; resource ID
Transfer                ;; menu title
Edit                    ;; item 1
Asm                     ;; item 2
Link                    ;; item 3
(-                      ;; item 4 (draw a line)
Exec                    ;; item 5
                        ;; MUST be followed by a blank line!!
```

PROC Procedure

```
TYPE PROC
  ,128                  ;; resource ID
MyProcedure             ;; filename
```

This type is used to create resources that contain code. It reads the first code segment from an application file (the 'CODE' resource with ID = 1), strips the first four bytes of it (used by the Segment Loader), and saves it as a resource of type 'PROC'. It is useful for defining code types such as 'DRVR', 'WDEF', and 'PACK'. An example is given below in the section on creating your own resource types.

STR String

```
TYPE STR                    ;; 'STR ' (space required)
  ,1                        ;; resource ID
This is a string            ;; and a string

  ,23                       ;; resource ID
This is a string ++         ;; and a long string
that shows the line ++
continuation characters.

  ,25 (32)                  ;; resource ID, optional attribute byte
I've got attributes!        ;; and a string

  ,27                       ;; resource ID
Testing, \31, \32, \33      ;; 'Testing, 1, 2, 3' the hard way
```

STR# A Number of Strings

```
TYPE STR#
  ,1                        ;; resource ID
4                           ;; number of strings
This is string one          ;; and the strings
And string two
Third string
Bench warmer
```

WIND Window Template

```
TYPE WIND
  ,128
Wonder Window               ;; title
40 80 120 300               ;; top left bottom right
Invisible GoAway            ;; window status (see note)
0                           ;; ProcID (window definition ID)
0                           ;; RefCon (reference value)
```

Note: A window can be Visible or Invisible; GoAway and NoGoAway determine whether or not the window has a close box. Only the first character of each option (V,I,G,N) is significant.

Creating Your Own Types

There are two ways to create your own resource types. The first is to equate a new type to an existing type. For example, you can create a resource of type 'DRVR' like this:

```
TYPE DRVR = PROC          ;; type DRVR is just like PROC
  ,17 (32)                ;; resource ID, attribute byte
MyDriver                  ;; filename
```

The file MyDriver should be a single-segment application, as created by Linker. Recall that the PROC type reads in the resource of type 'CODE' with ID = 1, then strips off the header types.

The other way to create your own type is to equate the new type to 'GNRL', and then to specify the precise format of the resource. A set of element type designators lets you define the type of each element that is to be placed in the resource.

Here are the element type designators:

.P	Pascal string
.S	String without length byte
.I	Decimal integer
.L	Decimal long integer
.H	Hexadecimal
.R	Read resource from file. .R is followed by: filename type ID

For example, to define a resource of type 'GNRL' consisting of the integer 57 followed by the Pascal string 'Finance Charges', you could use the following type assignment:

```
TYPE CHRG = GNRL          ;; define type 'CHRG'
  ,200                    ;; resource ID
.I                        ;; a decimal integer
57
.P                        ;; a Pascal string
Finance Charges
```

A more practical example: An application that has its own icon must define an icon list, and reference it using 'FREF' (described above). Such an icon list can be defined as follows:

```
        TYPE ICN# = GNRL            ;; icon list for an application
          ,128                      ;; resource ID
        .H                          ;; enter 2 icons in hexadecimal
        0001 0002 0003 0004         ;; each is 32 bits by 32 bits
            ....
        007D 007E 007F 0080         ;; for 128 words total
```

The .R type designator is used to include an existing resource as part of a new resource type. For example, to read an existing 'FONT' resource into a new resource of type 'FONT', use the following resource definition:

```
        TYPE FONT = GNRL            ;; define a new type
          ,268                      ;; resource ID
        .R                          ;; read from the System file
        System FONT 268             ;; the 'FONT' resource with ID=26
```

Using RMaker

Once you have created the input file to RMaker, the hard work is done. Simply select and open the application RMaker. The standard file selection window is automatically opened. Select the file you want to compile and off it goes.

By default, the standard file selection window displays all the text files on the disk. If you want to display only the .R files, Cancel the selection window, select .R Filter from the File menu, then select Compiler from the File menu to redisplay the file selection window.

When RMaker is compiling a file, the name of the source file is displayed in the upper left of the window, and the name of the output file is displayed in the upper right. As the file is compiled, the current size of the resource data, the size of the resource map, and the total size are tracked on the right half of the screen. In addition, as each line is compiled, it is displayed on the screen.

If there are no errors in the RMaker input file, a resource file with the specified name is created.

Errors in the Input File

If an error occurs, the line containing the error is the last line on the screen. RMaker then displays a box with an error message in it.

RMaker Error Messages

Here is a list of the error messages that can be displayed by RMaker. A brief description accompanies the messages that are not entirely self-explanatory.

An Input/Ouptut error has occurred
Bad attributes parameter
Bad bundle definition
Bad format number
Bad format resource designator in GNRL type: This is any error in a user-defined resource type.
Bad ID Number
Bad item type
Bad object definition: This can happen if the specified file is of the wrong type.
Bad type or item declaration
Can't add to the file -- disk protected or full?
Can't create the output file
Can't load INCLUDE file
Can't open the output file
Out of memory
Syntax error in source file
Unknown type: The specified resource type is not defined.

Toolbox

Microsoft® FORTRAN Compiler
for the Apple® Macintosh™

Toolbox

Contents

Microsoft® FORTRAN Compiler
for the Apple® Macintosh™

Toolbox Interface

1.0 The Microsoft FORTRAN Compiler Toolbox Interface

Access to the Macintosh ROM routines is provided through a single general FORTRAN procedure called toolbx.sub. This one procedure allows you to make nearly 500 different requests of Macintosh resources including quickdraw, the menu manager, the font manager, and the window manager. A parameter file, toolbx.par, which can be used with the FORTRAN INCLUDE statement, is supplied with the compiler and contains all the definitions you will need to call a tool box procedure. Toolbx.sub does not provide access to the OS Utilities routines with the exception of SysBeep.

CAUTION: The Apple document Inside Macintosh specifies the use of certain initialization calls in your application program before using various features of the Toolbox. The Microsoft FORTRAN Compiler run time system calls four of these for you: **INITGRAF**, **INITWINDOWS**, **INITMENUS**, and **INITFONTS**. These must not be called twice; no FORTRAN program should call these routines.

Toolbx.sub can be referenced as a function when the Mactintosh ROM routine returns results and it can be called as a subroutine when no results are expected or required. It can be linked to your program or reside on the disk and be loaded as an overlay. When toolbx.sub is loaded as an overlay, the first time it is called it will mark itself as permanent in the FORTRAN heap so that successive calls will not have to access the disk.

Note that when **toolbx.sub** is used as a function, it must be declared just as any other FORTRAN function would. For usage with multiple data types (ie, toolbx.sub returns INTEGER*4 in one call and LOGICAL*2 in another), declare toolbx as INTEGER*4. The Microsoft FORTRAN Compiler will handle the conversion to LOGICAL automatically.

The first parameter in a tool box call argument list is always an INTEGER which specifies the name of the Macintosh routine to execute. This name represents a number that is used to refer to the desired operation. When toolbx.sub is called, the operation argument is tested for validity. If it is an illegal value (ie, -1), then control is immediately returned to the calling program. Since there is no illegal value that can be returned by the tool box routines, the Microsoft FORTRAN Compiler has no way of knowing if an error has been detected. Therefore, be warned that there will be no error message in the event of a tool box parameter range error.

There can be as many as ten additional parameters in a toolbox call. This typical sequence of calls to toolbx.sub draws a horizontal line in the current window from (30,30) to (80,30).

```
      include toolbx.par
      call  toolbx(MOVETO,30,30)
      call  toolbx(LINETO,80,30)
```

The following definitions will help you get the most use from the tool box routines.

1.0.1 Arguments

Scalar arguments passed to toolbx.sub are always **INTEGER*4** variables or constants. Structures (Pascal Records) are passed as arrays of integers or characters and may have any size suitable to the Macintosh ROM routine. Since FORTRAN always passes arguments by address, one of the responsibilities of toolbx.sub is to format your argument list into one acceptable to Macintosh.

Unfortunately, the toolbox routines do not use standard FORTRAN character strings. Toolbox string arguments must be preceeded by a length byte (like a Pascal LSTRING).

```
      character*6  longstring
      character*255  longerstring
      longstring  =  char(5)//'Hello'
      longerstring  =  char(12)//'Hello,  world'
```

1.0.2 Pixels and Bits

A pixel is a "picture element." It is the smallest entity that can be written on the screen. Each pixel displayed is represented in Macintosh memory by exactly one bit or "binary digit." There are eight bits in a byte, which is the size of an INTEGER*1 variable. A FORTRAN variable can be used to define a pattern of pixels through bits as well as a mathematical quantity such as length or position.

1.0.3 Bit Images and Bitmaps

Bit images are a string of bits that represent a collection the sequential rows of a rectangular image. A bitmap is the data structure that references a bit image. It contains the pointer to the bit image, the number of bytes in each row, and the bit image's rectangular boundary. Using the Microsoft FORTRAN Compiler, the following example declares a bitmap structure:

```
    integer*2  bitmap(7)
    integer*4  bitptr
    integer*2  rowbytes,bounds(4)
    equivalence  (bitmap(1),bitptr)
    equivalence  (bitmap(3),rowbytes)
    equivalence  (bitmap(4),bounds(1))
```

1.0.4 Coordinates and Points

The origin of the coordinate plane (0,0) is the upper left hand corner of the window. Positive horizontal coordinates increase to the right and positive vertical coordinates increase down the screen. Some of the routines specify absolute coordinates in the window and others specify coordinates which are relative to some current position.

Coordinates are usually specified by a POINT, which can be defined with the Microsoft FORTRAN Compiler as:

```
    integer*2  point(2)
    data  point  /50,100/
```

This specifies a point 50 units below and 100 units to the right of the origin. Note that the vertical component is specified first and the horizontal component last (i.e. (y,x)); this is the reverse of the usual mathematical convention of (x,y). Note also that some toolbox calls take the components directly rather than as points, and that the order these components appear in the toolbox is generally horizontal first (i.e., (x,y)). For example, EQUALPT takes two points:

```
    integer*2  pta(2),  ptb(2)
    logical  pointflag
    pointflag  =  toolbx(EQUALPT,  pta,  ptb)
```

SETPT takes two components in (x,y) order:

```
    integer  x,  y
    integer*2  pt(2)
    data  x,  y  /5,  10/
    call  toolbx  (SETPT,  pt,  x,  y)
```

1.0.5 Patterns

A pattern is a 64 bit image usually defined as an eight by eight pixel square. It is most easily defined in FORTRAN as an eight element INTEGER*1 array. A pattern is used to specify a repeating design. The binary integer constant in the Microsoft FORTRAN Compiler is quite useful for defining patterns:

```
    integer*1  pattern(8)
    pattern(1)  = b'00011000'
    pattern(2)  = b'00100100'
    pattern(3)  = b'01000010'
    pattern(4)  = b'10000001'
    pattern(5)  = b'10000001'
    pattern(6)  = b'01000010'
    pattern(7)  = b'00100100'
    pattern(8)  = b'00011000'
```

1.0.6 Pointers and Handles

The Macintosh toolbox utilities create and access a variety of structures in memory. Some of them require you to supply the memory for these structures from FORTRAN. Others will get contiguous pieces of memory dynamically from a structure called the heap. This is done by the Memory Manager, described in greater detail in the Memory Manager Programmer's Guide in Inside Macintosh. These pieces of memory are called **blocks**. There are two basic kinds of blocks: **relocatable** and **nonrelocatable**. Relocatable blocks can be moved around within the heap to create space for other blocks; nonrelocatable blocks can never be moved.

The memory that the Macintosh gets from the heap cannot be accessed with ordinary FORTRAN variables. There is no way, for example, to declare an array in your program and have its space allocated from the Macintosh heap during execution. Instead, when a toolbox utility allocates a block, it returns either a pointer or a handle.

Nonrelocatable blocks are referred to by pointers. A pointer is an absolute address to a location in Macintosh memory. Since these blocks are always at the same location in memory, the pointers will remain valid for as long as the block exists.

Relocatable blocks pose a greater problem. If necessary to make room for some other block, the Memory Manager will move relocatable blocks at any time to a new location. This would leave any pointers you might have to the block pointing to the wrong place in memory, or "dangling."

To help avoid dangling pointers, the Memory Manager maintains a single master pointer to each relocatable block, allocated from within the same heap as the block itself. The master pointer is created at the same time as the block and set to point to it. What you get back from the toolbox for such a block is a pointer to the master pointer, called a handle. If the Memory Manager later has to move the block, it has only to update the master pointer to point to the block's new location; the master pointer itself is never moved.

The toolbox utilities that use the Memory Manager to allocate blocks of memory for their structures will return either a pointer or a handle to the block. You can store these in FORTRAN INTEGER variables. The main use of pointers and handles under FORTRAN is to refer to the structures built with toolbox utilities in calling other toolbox utilities, so you will seldom need to access a pointer or handle directly.

If you need to access a Macintosh structure through a pointer, you can use the Microsoft FORTRAN Compiler **LONG**, **WORD**, and **BYTE** functions which return the contents of absolute memory locations. For example, if the variable mypointer contains a pointer to an INTEGER value on the heap, the value could be accessed by:

```
    integer mypointer, myint
    myint = LONG(mypointer)
```

To access a structure through a handle, you must first reference the handle, that is, get a copy of its master pointer. This copy can then be used as a pointer, giving access to the structure through LONG, WORD, and BYTE. For example, if the variable myhandle contains a handle to an INTEGER value, the value can be accessed by:

```
    integer myhandle, temp, myint
    temp = LONG(myhandle)              ! Get the master pointer.
    myint = LONG(temp)
```

This copy of the pointer is only guaranteed to be valid until the Memory Manager is next used to allocate a block of memory. To be safe, you should not assume that a copy of a master pointer will be valid after any call to the toolbox.

A handle or pointer will generally refer to some structure considerably more complex than an integer. The most common structure is a record: a contiguous list of values of various types and sizes. If you need to extract a value from a record to which you have a pointer, you will need to know the offset of that value within that type of record. Offsets for some of the most common records are given elsewhere in this manual; others must be calculated from information in Inside Macintosh. To access a value at a given offset, add the offset to the pointer and use LONG, WORD, or BYTE, as appropriate. For example, there is a one byte flag in each window record (see Window Initialization and Allocation) which is 1 if the corresponding window is visible on the screen or 0 if it is not. It has an offset of 110 and can be accessed in the following manner, assuming mywindow contains a window pointer:

```
    integer*1 visible
    integer mywindow
    visible = BYTE(mywindow + 110)
```

1.0.7 TOOLBX(PTR) Function

Many toolbox utilities operate on pointers. Usually these pointers will have been created by other toolbox utilities, so that all you have to do is pass them from one call to another. Occasionally, however, it is necessary to pass a FORTRAN structure (such as an array or string) to a toolbox utility expecting a pointer. Since FORTRAN does not support pointers directly, the Microsoft FORTRAN Compiler offers a PTR function, which returns the absolute address of a variable. PTR uses the same calling conventions as the other toolbx calls. For example, the GETNEWWINDOW utility initializes a window record and adds it to the Window Manager's list (see the section Window Initialization and Allocation). It can get the memory for this record either from the Macintosh heap (described above) or from your program. When you supply the memory, it must be in the form of a pointer:

```
integer*1  windowrecord(154)
integer  windowptr
integer  wstorage
integer  behind
integer  windowid
behind = -1                          ! Put in front.
windowid = 256                       ! Assumes window resource.
wstorage = toolbx(PTR, windowrecord)
windowptr  =  toolbx(GETNEWWINDOW,windowid,wstorage,behind)
```

The PTR function can be used with any of the valid FORTRAN data types.

The following sections describe the various calls that can be made to the interface procedure. More detailed information can be found in the manual, Inside Macintosh, available from Apple Computers. Inside Macintosh is the authoritative source of any information relating to the internal workings of the Macintosh. At the time of printing, the cost was approximately $100. For further details concerning current price and shipping information, call:

(408) 988 - 6009

or write to the following address:

"Inside Macintosh"
Apple Computers
467 Saratoga Avenue
Suite 621
San Jose, California 95121

1.1 GrafPort Routines

The overriding structure behind all QuickDraw commands and functions is the GrafPort. The grafport is where the Macintosh stores and maintains any information that pertains to its window (or port). It is valid to have more than one grafport open at any given time. Control can be transferred from one port to another without losing data.

Device - The Macintosh's output device number. Initially set to 0 (the Macintosh's screen).

Portbits - The bitmap whose pointer addresses the grafport's bit image. Initially uses the entire Macintosh's screen as the the bit image (0, 0, 512, 342).

Portrect - A subset of the grafport's bitmap, it defines the area of the window where output can take place. Initially set to the Macintosh's screen (0, 0, 512, 342).

Visrgn - Defines the visible portion of the grafport's window. Normally never changed at the programmer level, this field is updated by some of the window routines. Initially set to the Macintosh's screen (0, 0, 512, 342).

Cliprgn - Defines an area inside the portrect that limits the area of the screen where drawing can occur. Initially set to the Macintosh's screen (0, 0, 512, 342).

Bkpat - Is the pattern used to fill a graphic figure when it is erased, and to fill in an area left by a scroll operation. Initially set to white.

Fillpat - Is the pattern that displays when a graphic operation is invoked that has the "PAINT" prefix. Initially set to black.

The next 5 grafport fields save attributes of the current pen. For further details, see Pen and Line-Drawing Routines.

Pnloc - Saves the current pen location. Initially set to 0 horizontally and 0 vertically.

Pnsize - Saves the current pen size. With an initial width and height of 1.

Pnmode - Saves the current pen transfer mode. Initially set to patcopy.

Pnpat - Saves the current pen drawing pattern. Initially set to black.

Pnvis - Designates whether any drawing is to take place or not . Initially set to 0 (visible).

The next 5 grafport fields define characteristics of any text that may be output. For further details, see Text Drawing Routines.

Txfont - Defines the font number of the character font to be used. Initially set to 0 (system font).

Txface - Defines the current text style value. Initially set to 0 (normal).

Txmode - Defines the current text transfer mode. This transfer mode is similar to the pen transfer modes. Initially set to 1 (srcor).

Txsize - Stores the current text size in points. Initially set to 0 (font manager decides).

Spextra - Saves the number of pixels to be added to all spaces in a string. Initially set to 0. This is erroneously defined as a word (INTEGER*2) in Inside Macintosh. It is actually a long word (INTEGER*4).

Fgcolor - Designates the current foreground color (when color becomes available). Initially set to black.

Bkcolor - Designates the current background color (when color becomes available). Initially set to white.

Colrbit - Defines the color plane for color imaging software (when color becomes available). Initially set to 0.

Patstretch - Used to determine if a pattern needs to be expanded when output to a printer. This field should not be changed by the programmer. Initially set to 0 (no expansion).

The next three fields are related in that they determine that an object (picture, region, or polygon) is currently being defined. A field is activated when an open operation (OPENPICTURE, OPENRGN, or OPENPOLY) is executed. At this point, the field contains the absolute address of the object's data area. If, during the definition of an object, the associated field is set to 0, then the object definition is disabled until the contents are restored or the object definition is terminated with a close operation (CLOSEPICTURE, CLOSERGN, or CLOSEPOLY).

Picsave - Defines the "open" state of a picture. Initially set to 0 (closed).

Rgnsave - Defines the "open" state of a region. Initially set to 0 (closed).

Polysave - Defines the "open" state of a region. Initially set to 0 (closed).

Grafprocs - Is used in the storage of a special data structure that the application defines to customize QuickDraw procedures. This requires specific knowledge of the way the Macintosh processes graphics. For further information, consult the manual Inside Macintosh.

The structure of the grafport contains arrays, pointers, and integer values. The grafport can be represented in the Microsoft FORTRAN Compiler in the following manner:

```
integer*2  grafport(54)
integer*2  device
integer*2  portbits(7)
integer*2  portrect(4)
integer*4  visrgn
integer*4  cliprgn
integer*1  bkpat(8)
integer*1  fillpat(8)
integer*2  pnloc(2)
integer*2  pnsize(2)
integer*2  pnmode
integer*1  pnpat(8)
integer*2  pnvis
integer*2  txfont
integer*2  txface
integer*2  txmode
integer*2  txsize
integer*4  spextra
integer*4  fgcolor
integer*4  bkcolor
integer*2  colrbit
integer*2  patstretch
integer*4  picsave
integer*4  rgnsave
integer*4  polysave
integer*4  grafprocs
```

```
    equivalence  (grafport(1),device)
    equivalence  (grafport(2),portbits(1))
    equivalence  (grafport(9),portrect(1))
    equivalence  (grafport(13),visrgn)
    equivalence  (grafport(15),cliprgn)
    equivalence  (grafport(17),bkpat(1))
    equivalence  (grafport(21),fillpat(1))
    equivalence  (grafport(25),pnloc(1))
    equivalence  (grafport(27),pnsize(1))
    equivalence  (grafport(29),pnmode)
    equivalence  (grafport(30),pnpat(1))
    equivalence  (grafport(34),pnvis)
    equivalence  (grafport(35),txfont)
    equivalence  (grafport(36),txface)
    equivalence  (grafport(37),txmode)
    equivalence  (grafport(38),txsize)
    equivalence  (grafport(39),spextra)
    equivalence  (grafport(41),fgcolor)
    equivalence  (grafport(43),bkcolor)
    equivalence  (grafport(45),colrbit)
    equivalence  (grafport(46),patstretch)
    equivalence  (grafport(47),picsave)
    equivalence  (grafport(49),rgnsave)
    equivalence  (grafport(51),polysave)
    equivalence  (grafport(53),grafprocs)
```

For all examples in the remainder of this documentation using grafports, this structure will apply.

The array grafport can be passed to the **OPENPORT** toolbox utility (below) to be filled out. However, you will often have only a pointer to a grafport record which has been allocated by the Macintosh Memory Manager (see The Microsoft FORTRAN Compiler Tool Box Interface). If you need to access the fields of such a record, you will have to use the Microsoft FORTRAN Compiler functions LONG, WORD, or BYTE, which take an absolute address and return the value stored there. To get a pointer to a particular field, you add the offset of that field to the pointer. The offsets for the grafport record can be defined in Microsoft FORTRAN by:

```
    integer device;              parameter (device = z'0')
    integer portbits;            parameter (portbits = z'2')
    integer portrect;            parameter (portrect = z'10')
    integer visrgn;              parameter (visrgn = z'18')
    integer cliprgn;             parameter (cliprgn = z'1C')
    integer bkpat;               parameter (bkpat = z'20')
    integer fillpat;             parameter (fillpat = z'28')
    integer pnloc;               parameter (pnloc = z'30')
    integer pnsize;              parameter (pnsize = z'34')
    integer pnmode;              parameter (pnmode = z'38')
    integer pnpat;               parameter (pnpat = z'3A')
    integer pnvis;               parameter (pnvis = z'42')
    integer txfont;              parameter (txfont = z'44')
    integer  txface;                 parameter  (txface  =  z'46')
    integer  txmode;                 parameter  (txmode  =  z'48')
    integer  txsize;                 parameter  (txsize  =  z'4A')
    integer  spextra;                parameter  (spextra  =  z'4C')
    integer  fgcolor;                parameter  (fgcolor  =  z'50')
    integer  bkcolor;                parameter  (bkcolor  =  z'54')
    integer  colrbit;                parameter  (colrbit  =  z'58')
    integer  patstretch;             parameter  (patstretch  =  z'5A')
    integer  picsave;                parameter  (picsave  =  z'5C')
    integer  rgnsave;                parameter  (rgnsave  =  z'60')
    integer  polysave;               parameter  (polysave  =  z'64')
    integer  grafprocs;              parameter  (grafprocs  =  z'68')
```

INITGRAF initializes the QuickDraw global variables. INITGRAF is intended to be called once and only once. The Microsoft FORTRAN Compiler calls this procedure before any program execution begins. Therefore, **do not** call this procedure. It is documented for user information only.

OPENPORT initializes the elements of the given grafport and allocates space for the visible and clipping regions. A port must have been opened before it can be used. When a port is opened, it automatically becomes the current port.

```
    integer*4  grafptr
    grafptr  =  toolbx  (PTR,grafport(1))
    call  toolbx  (OPENPORT,grafptr)
```

INITPORT reinitializes a port that has already been opened and makes it the current port.

```
    integer*4  grafptr
    grafptr  =  toolbx  (PTR,grafport(1))
    call  toolbx  (INITPORT,grafptr)
```

CLOSEPORT deallocates the memory that OPENPORT allocated for the visible and clipping regions.

```
    integer*4  grafptr
    grafptr  =  toolbx  (PTR,grafport(1))
    call  toolbx  (CLOSEPORT,grafptr)
```

SETPORT is used to assign the given grafport to the current port.

```
    integer*4  grafptr
    grafptr  =  toolbx  (PTR,grafport(1))
    call  toolbx  (SETPORT,grafptr)
```

GETPORT determines which grafport is currently active and returns a pointer to that port.

```
integer*4  grafptr
call  toolbx  (GETPORT,grafptr)
```

GRAFDEVICE sets the logical output device for the current grafport to the given value. The initial value is 0 (Macintosh screen).

```
integer*4  device
device = 0
call  toolbx  (GRAFDEVICE,device)
```

SETPORTBITS provides the ability to change the current bitmap to any previously defined bitmap.

```
integer*2  bitmap(7)
call  toolbx  (SETPORTBITS,bitmap)
```

PORTSIZE changes the size of the active area (portrect) of the current window. This operation has no effect on the screen.

```
integer*4  width,height
data  /20,-20/
call  toolbx  (PORTSIZE,width,height)
```

MOVEPORTTO changes the position of the active area (portrect) of the current window. This operation has no effect on the screen.

The leftglobal and topglobal values define the distance between upper left corner of the current bitmap boundaries and that of the current portrect.

```
integer*4  leftglobal,topglobal
data  leftglobal,topglobal  /256,171/
call  toolbx  (MOVEPORTTO,leftglobal,topglobal)
```

SETORIGIN redefines the graphic screen's reference point. This procedure causes the coordinate (0,0) to be moved to the absolute location (h,v). All subsequent absolute moves are made relative to this new location. Any relative movement or drawing will not be affected by this procedure. It does not affect the screen until an attempt is made to draw a figure based on absolute coordinates.

```
     integer*4  h,v
     data  h,v  /110,220/
     call  toolbx  (SETORIGIN,h,v)
```

SETCLIP defines the clipping region of the current grafport to be equivalent to the given region.

```
     integer*4  myregion
     myregion  =  toolbx  (NEWRGN)
     call  toolbx  (SETCLIP,myregion)
```

GETCLIP defines the given region to be equivalent to the clipping region of the current grafport.

```
     integer*4  myregion
     myregion  =  toolbx  (NEWRGN)
     call  toolbx  (GETCLIP,myregion)
```

CLIPRECT redefines the clipping region of the current grafport to be a rectangular equivalent of the given rectangle.

```
     integer*2  rect(4)
     data  rect  /10,10,210,210/
     call  toolbx  (CLIPRECT,rect)
```

BACKPAT assigns the given pattern of the current grafport's background pattern (bkpat).

```
      integer*1  pat(8)
      data  pattern  /b'10101010',
     +               b'01010101',
     +               b'10101010',
     +               b'01010101',
     +               b'10101010',
     +               b'01010101',
     +               b'10101010',
     +               b'01010101'/
      call  toolbx  (BACKPAT,pat)
```

1.2 Cursor-Handling Routines

These routines manipulate the cursor which initially appears as an arrow pointing to the upper left. The cursor tracks the mouse and its position cannot be changed by any means other than mouse movement. The cursor has the following attributes: visible or not visible, pattern, and state of underlying pixels.

INITCURSOR sets the cursor to its default state: a visible, upper left pointing arrow. This call also sets the cursor level to zero.

```
    call  toolbx(INITCURSOR)
```

HIDECURSOR removes the cursor from the screen and decrements the cursor level. Every call to HIDECURSOR should be balanced by a call to SHOWCURSOR.

```
    call  toolbx(HIDECURSOR)
```

SHOWCURSOR increments the cursor level and displays the cursor if the level is zero. The cursor level cannot be incremented past zero, so extra calls to SHOWCURSOR have no effect.

```
    call  toolbx(SHOWCURSOR)
```

OBSCURECURSOR hides the cursor until the next time the mouse is moved. This call does not effect the cursor level.

```
    call  toolbx(OBSCURECURSOR)
```

SETCURSOR is used to change the cursor image. This call is made with a structure containing three parameters. The new cursor image and its attributes are defined from two 16 by 16 bit patterns. The first pattern contains the data which defines the image of the cursor and the second pattern is the mask which specifies its appearance.

data	mask	resulting pixel on the screen
0	1	white
1	1	black
0	0	same as pixel under cursor
1	0	inverse of pixel under cursor

The new cursor's hot spot is specified by vertical and horizontal coordinates. The hot spot aligns a relative point in the image with the mouse position. Typically you will specify a hot spot which coincides with the tip of your cursor.

The routine below demonstrates the SETCURSOR call by changing the cursor to a right pointing hand.

```
integer*2  data(16),mask(16),hotspot(2)
integer*2  cursor(34)

equivalence  (cursor(1),data(1))
equivalence  (cursor(17),mask(1))
equivalence  (cursor(33),hotspot(1))

data(1)    = b'0000000000000000'
data(2)    = b'0000000000000000'
data(3)    = b'0000000000000000'
data(4)    = b'0000000000000000'
data(5)    = b'0000011100000000'
data(6)    = b'0001100100000000'
data(7)    = b'0010001000000000'
data(8)    = b'0100011000000000'
data(9)    = b'1100011111111110'
data(10)  = b'1000110000000001'
data(11)  = b'1001011111111110'
data(12)  = b'1110010000010000'
data(13)  = b'1000011111100000'
data(14)  = b'1000010000100000'
data(15)  = b'1100011111000000'
data(16)  = b'0111111110000000'

   do  (i=1,16)
     mask(i)  = data(i)
   repeat

hotspot(1) = 9          ! vertical
hotspot(2) = 16         ! horizontal

call  toolbx(SETCURSOR,cursor)
```

1.3 Pen and Line-Drawing Routines

These routines all make use of a concept known as a pen. This is not the cursor which usually appears as an arrow, but a drawing tool with independent size, pattern, and location attributes. Initially, the pen is one pixel in size, black, and is located at 12,3 (vertical, horizontal).

HIDEPEN decrements the current grafport's pnvis variable which is initialized to zero. When pnvis is negative, the pen does not draw on the screen.

```
    call  toolbx(HIDEPEN)
```

SHOWPEN increments **pnvis**. When pnvis is equal to or greater than zero, the pen draws on the screen.

```
    call  toolbx(SHOWPEN)
```

GETPEN returns the location of the pen as vertical and horizontal coordinates.

```
    integer*2  penloc(2)
    call  toolbx(GETPEN,penloc)
```

Penloc(1) is the vertical coordinate and penloc(2) is the horizontal coordinate.

GETPENSTATE returns the current pen location, size, pattern, and mode. This call is useful for storing the current pen state in procedures that temporarily change the pen characteristics.

```
    integer*2  penloc(2),pensize(2),penmode
    integer*1  pattern(8)
    common  penloc,pensize,pattern,penmode
    call  toolbx(GETPENSTATE,penloc)
```

SETPENSTATE sets the current pen location, size, pattern, and mode. This call is usually made at the end of a procedure that has made temporary changes to the pen state.

```
    integer*2  penloc(2),pensize(2),penmode
    integer*1  pattern(8)
    common  penloc,pensize,pattern,penmode
    call  toolbx(GETPENSTATE,penloc)
```

PENSIZE allows you to set the width and height of the pen.

```
    integer*4  w,h
    call  toolbx(PENSIZE,w,h)
```

PENMODE defines the transfer mode that will be used to draw all graphics. There are four basic types of transfer modes: **copy, or, xor,** and **bic**. Each of these types has an alternate form that inverts each pixel of the source pattern (white to black and black to white) before performing the designated function, and is given the prefix "**not**". The operations that are performed are:

Transfer Mode	Mode Value	Action on Black Pixels	Action on White Pixels
patcopy	8	force black	force white
pator	9	force black	leave alone
patxor	10	invert	leave alone
patbic	11	force white	leave alone
notpatcopy	12	force white	force black
notpator	13	leave alone	force black
notpatxor	14	leave alone	invert
notpatbic	15	leave alone	force white

Nothing is drawn if an invalid mode is given (ie - not one of the source transfer modes or a negative value). The initial mode is patcopy.

```
    integer*4  patcopy
    data  patcopy  /8/
    call  toolbx  (PENMODE,patcopy)
```

PENPAT is used to change the pen pattern. A discussion of patterns is given in the introduction to the tool box interface.

```
    integer*1  pattern(8)
    call  toolbx(PENPAT,pattern)
```

PENNORMAL resets the pen to its initial state with a size of one bit and a pattern of black.

```
    call  toolbx(PENNORMAL)
```

MOVETO changes the pen location to new horizontal and vertical coordinates.

```
    integer*4  h,v
    call  toolbx(MOVETO,h,v)
```

MOVE changes the pen location relative to the current pen location.

```
integer*4 dh,dv
call toolbx(MOVE,dh,dv)
```

If the current horizontal and vertical pen coordinates are 50 and 30, the MOVE routine with dh and dv values of 5 and -5 will change the pen coordinates to 55 and 25.

LINETO draws a line from the current pen location to the horizontal and vertical coordinates specified in the argument list. After the line is drawn, the pen location is updated to the coordinates of the line end point.

```
integer*4 h,v
call toolbx(LINETO,h,v)
```

LINE draws a line to a point which is specified relative to the current pen location. After the line is drawn, the pen location is updated to the coordinates of the line end point.

```
integer*4 dh,dv
call toolbx(LINE,dh,dv)
```

1.4 Text-Drawing Routines

The Macintosh does not print text like other computers you may be familiar with. It uses calls to QuickDraw which write or draw text to the currently active window. Therefore, text output is a part of the grafport. Each grafport contains the information needed to produce text output.

FORTRAN's write and type statements perform the same function as the toolbox's output statements. They take into account the current font, face, size and spaceextra attributes.

TEXTFONT sets the text font of the current grafport to the value held in font. The possible font values are:

```
systemfont = 0    ! chicago, used by the system for text output.
applfont   = 1    ! application font, geneva by default.
newyork    = 2
geneva     = 3
monaco     = 4
venice     = 5
london     = 6
athens     = 7
sanfran    = 8
toronto    = 9

integer*4  monaco
data  monaco  /4/
call  toolbx  (TEXTFONT,monaco)
```

TEXTFACE defines the text syle for the current grafport. Multiple text styles are defined by adding the individual text styles together.

```
normal    = 0
bold      = 1
italic    = 2
underline = 4
outline   = 8
shadow    = 16
condense  = 32
extend    = 64

integer*4  bold,underline,shadow
integer*4  style
data  bold,underline,shadow  /1,4,16/
style = bold + underline + shadow
call  toolbx  (TEXTFACE,style)
```

TEXTMODE sets the transfer mode for drawing text in the current grafport. The mode should be one of the following:

```
srccopy = 0
srcor   = 1
srcxor  = 2
srcbic  = 3
```

The use of the text transfer modes is the same as the use of the pattern transfer modes in the PENMODE command (see Pen and Line-Drawing Routines).

The initial transfer mode is srcor.

```
integer*4  srccopy
parameter  (srccopy = 0)
call  toolbx  (TEXTMODE,srccopy)
```

TEXTSIZE defines the size (in points) of the text. Although it is legal to specify any size, the text output is better if a font of the chosen size is available from the font manager. The next best thing is to choose an even multiple of a particular font. If a 0 is passed as the size, the output will be the size (or as close to it as possible) of the system font.

```
integer*4  size
data  size  /9/
call  toolbx  (TEXTSIZE,size)
```

SPACEEXTRA sets the spextra field in the current grafport. This value is used to expand the width of each space character in a line of text. Both positive and negative values are legal, but take care when using negative values that the lines do not become unreadable.

```
integer*4  extra
data  extra  /2/
call  toolbx  (SPACEEXTRA,extra)
```

DRAWCHAR outputs the given character to the current grafport. The character is positioned above and to the right of the current pen location. After the character has been drawn, the pen location is updated to the bottom left corner of the next character position. If the given character is not a part of the current font, the font's missing symbol is drawn.

Since all parameters to toolbx are 4 bytes long, it is necessary to pass a 4 character string to DRAWCHAR. Only the last (rightmost) character is drawn; the other three characters are ignored.

```
character*4  ch
data ch /'   A'/
call  toolbx  (MOVETO,3,100)
call  toolbx  (DRAWCHAR,ch)
```

DRAWSTRING calls **DRAWCHAR** once for each character in the string. QuickDraw does no formatting (carriage return, line feed, etc.). The first character of the string must contain the number of characters in the string, consequently the maximum length of the string is 255 characters.

```
character*256  string
string = char(6)//'abcdef'
call  toolbx  (MOVETO,3,100)
call  toolbx  (DRAWSTRING,string)
```

DRAWTEXT performs the same function as DRAWSTRING with the exception that the data is retrieved from a text buffer. The first character to be output is indexed by firstbyte and bytecount defines the number of characters to output. The following example outputs the characters "defghijklm":

```
character*255  textbuf
integer*4  firstbyte
integer*4  bytecount
data  firstbyte,bytecount  /4,10/
textbuf  =  'abcdefghijklmnopqrstuvwxy'
call  toolbx  (MOVETO,3,100)
call  toolbx  (DRAWTEXT,textbuf,firstbyte,bytecount)
```

CHARWIDTH is an INTEGER function that returns the width (in pixels) of the given character. The function takes into account all the current font parameters that determine the actual size of the character. As for DRAWCHAR above, it is necessary to pass a 4 character string to CHARWIDTH.

```
integer*4  chwide
character*1  ch
data ch /'   A'/
chwide = toolbx  (CHARWIDTH,ch)
```

STRINGWIDTH adds the widths of all the characters in a string (as determined by **CHARWIDTH**) and returns it in an INTEGER*4 variable.

```
integer*4  strngwide
character*20  strng
strng  =  'abcdefghijklmnopqrst'
strngwide  =  toolbx  (STRINGWIDTH,strng)
```

TEXTWIDTH performs the same function as **STRINGWIDTH,** but performs the operation on a text buffer as described in DRAWTEXT.

```
character*25 textbuf
integer*4 firstbyte
integer*4 bytecount
integer*4 textwide
data firstbyte /4/
data bytecount /10/
textbuf = 'abcdefghijklmnopqrstuvwxy'
textwide = toolbx (DRAWTEXT,textbuf,firstbyte,bytecount)
```

GETFONTINFO returns information about the current font size and spacing. The information is returned in a structure that contains four parts: **ascent, descent, widmax,** and **leading**. Ascent designates the number of pixels between the baseline (the vertical element of the current pen position) and the top of the tallest possible character of the current size (ascent line). Descent represents the distance between the baseline and the vertical position of the lowest character descender, which is the portion of a character that dips below the baseline; in a "y" or "g". Widmax is the width of the widest character (excluding intercharacter spacing). Leading defines the vertical distance between the descent line (the bottom of the lowest descender) and the ascent line that would be immediately below it (the interline spacing).

```
integer*2 fontinfo(4)
integer*2 ascent
integer*2 descent
integer*2 widmax
integer*2 leading
equivalence (fontinfo(1),ascent)
equivalence (fontinfo(2),descent)
equivalence (fontinfo(3),widmax)
equivalence (fontinfo(4),leading)
call toolbx (GETFONTINFO,fontinfo(1))
```

1.5 Rectangle, Oval, and Arc Routines

These routines perform operations on rectangular regions. A rectangular region is defined with a four element INTEGER*2 array. The first and second array elements specify the y and x coordinates respectively of the upper left hand corner of the rectangular region. The third and fourth elements specify the y and x coordinates of the lower right hand corner of the region.

A rectangle is completely defined by the array which specifies the region. Ovals are drawn inside rectangular regions. An arc is a wedge-shaped section of an oval that fits inside a rectangular region.

None of these routines changes the current pen location.

1.5.1 Calculations on Rectangles

SETRECT assigns the four boundary values into the rectangle array. This procedure performs the same function as four assignment statements and is meant to reduce source code size in Pascal, however, it will always produce more object code in FORTRAN.

```
integer*2  rect(4)
call  toolbx  (SETRECT,rect,10,10,210,210)
```

OFFSETRECT logically moves a rectangle a distance of dh in the horizontal direction, and dv in the vertical direction. If dh and dv are positive, the rectangle is moved to the right and down. Negative values cause movement in the opposite direction. This procedure has no effect on the screen.

```
integer*2  rect(4)
integer*4  dh,dv
data  rect  /10,10,210,210/
data  dh,dv  /50,-5/
call  toolbx  (OFFSETRECT,rect,dh,dv)
```

INSETRECT causes the rectangle to shrink or expand. If dh and dv are positive, the rectangle shrinks toward its center. Negative values cause the rectangle to expand from the center. If the height or width is less than one as a result of the operation, the rectangle is set to empty (0,0,0,0).

```
    integer*2  rect(4)
    integer*4  dh,dv
    data  rect  /10,10,210,210/
    data  dh,dv  /50,-5/
    call  toolbx  (INSETRECT,rect,dh,dv)
```

SECTRECT is a logical function that returns true if the two source rectangles intersect at any point. If an intersection has occurred, the area of intersection is used to define a destination rectangle. If the source rectangles have no common points, the destination rectangle is defined as empty and the function returns false. Further, an intersection of only one point or line returns false since the resulting rectangle would be considered empty. The destination rectangle may be one of the source rectangles.

```
    integer*2  srcrectA(4)
    integer*2  srcrectB(4)
    integer*2  dstrect(4)
    logical*4  rectflag
    data  srcrectA  /10,10,210,210/
    data  srcrectB  /100,100,300,150/
    rectflag  =  toolbx  (SECTRECT,srcrectA,srcrectB,dstrect)
```

UNIONRECT defines a new rectangle based on the smallest rectangle that encloses both source rectangles. The destination rectangle may be one of the source rectangles.

```
    integer*2  srcrectA(4)
    integer*2  srcrectB(4)
    integer*2  dstrect(4)
    data  srcrectA  /10,10,210,210/
    data  srcrectB  /100,100,300,150/
    call  toolbx  (UNIONRECT,srcrectA,srcrectB,dstrect)
```

PTINRECT is a logical function that returns true if the pixel below and to the right of the given point is part of the given rectangle.

```
    integer*2  point(2)
    integer*2  rect(4)
    logical*4  pointflag
    data  point  /125,150/
    data  rect  /10,10,210,210/
    pointflag  =  toolbx  (PTINRECT,point,rect)
```

PT2RECT defines the smallest rectangle that will contain both of the given points.

```
    integer*2  pointA(2),pointB(2)
    integer*2  rect(4)
    data  pointA,pointB  /10,10,20,20/
    call  toolbx  (PT2INRECT,pointA,pointB,rect)
```

PTTOANGLE defines an angle based upon an imaginary line from the center of the given rectangle to some other point. The angle is defined as the number of degrees from the "12 o'clock" position, travelling clockwise around the rectangle's center, to the imaginary line. The angle returned is adjusted for non-square rectangles. For example, if the given point were the upper right corner of the given rectangle, the returned angle would be 45 degrees, regardless of the aspect ratio of the rectangle.

```
    integer*2  rect(4)
    integer*2  point(2)
    integer*4  angle
    data  rect  /10,10,340,200/
    data  point  /340,10/
    call  toolbx  (PTTOANGLE,rect,point,angle)
```

EQUALRECT is a logical function that returns true if the two given rectangles have the same boundary values.

```
    integer*2  rectA(4),rectB(4)
    logical*4  rectflag
    data  rectA  /10,10,210,210/
    data  rectB  /10,10,210,210/
    rectflag = toolbx  (EQUALRECT,rectA,rectB)
```

EMPTYRECT is a logical function that returns true if the given rectangle is empty (left ≥ right or top ≥ bottom).

```
    integer*2  rect(4)
    logical*4  rectflag
    data  rect  /10,10,210,210/
    rectflag = toolbx  (EMPTYRECT,rect)
```

1.5.2 Graphic Operations

FRAMERECT draws the hollow outline of the specified rectangle using the current pen size and pen pattern. The outline is as tall as the pen height and as wide as the pen width.

```
    integer*2  rect(4)
    data  rect  /10,10,210,210/
    call  toolbx(FRAMERECT,rect)
```

PAINTRECT paints the specified rectangle in the current pen pattern.

```
    integer*2  rect(4)
    data  rect  /10,10,210,210/
    call  toolbx(PAINTRECT,rect)
```

ERASERECT paints the specified rectangle in the current background pattern.

```
    integer*2  rect(4)
    data  rect  /10,10,210,210/
    call  toolbx(ERASERECT,rect)
```

INVERTRECT inverts the pixels enclosed by the specified rectangle; white pixels become black and black pixels become white.

```
    integer*2  rect(4)
    data  rect  /10,10,210,210/
    call  toolbx(INVERTRECT,rect)
```

FILLRECT fills the specified rectangle with the given pattern.

```
      integer*2  rect(4)
      integer*1  pattern(8)
      data  rect  /10,10,210,210/
      data  pattern  /b'10101010',
     +                b'01010101',
     +                b'10101010',
     +                b'01010101',
     +                b'10101010',
     +                b'01010101',
     +                b'10101010',
     +                b'01010101'/
      call  toolbx(FILLRECT,rect,pattern)
```

FRAMEOVAL draws the hollow outline of an oval inside the coordinates specified by the rectangle using the current pen size and pen pattern. The outline is as tall as the pen height and as wide as the pen width.

```
    integer*2  oval(4)
    data  oval  /10,10,210,210/
    call  toolbx(FRAMEOVAL,oval)
```

PAINTOVAL paints the specified oval in the current pen pattern.

```
     integer*2  oval(4)
     data  oval  /10,10,210,210/
     call  toolbx(PAINTOVAL,oval)
```

ERASEOVAL paints the specified oval in the current background pattern.

```
     integer*2  oval(4)
     data  oval  /10,10,210,210/
     call  toolbx(ERASEOVAL,oval)
```

INVERTOVAL inverts the pixels enclosed by the specified oval; white pixels become black and black pixels become white.

```
     integer*2  oval(4)
     data  oval  /10,10,210,210/
     call  toolbx(INVERTOVAL,oval)
```

FILLOVAL fills the specified oval with the given pattern.

```
      integer*2  oval(4)
      integer*1  pattern(8)
      data  oval  /10,10,210,210/
      data  pattern  /b'10101010',
     +                b'01010101',
     +                b'10101010',
     +                b'01010101',
     +                b'10101010',
     +                b'01010101',
     +                b'10101010',
     +                b'01010101'/
      call  toolbx(FILLOVAL,oval,pattern)
```

FRAMEROUNDRECT draws the hollow outline of a round cornered rectangle using the current pen size and pen pattern. The shape of the corners is governed by the two oval parameters, width and height. The outline is as tall as the pen height and as wide as the pen width.

```
     integer*2  rect(4)
     integer*4  w,h
     data  rect  /10,10,210,210/
     data  w,h  /20,20/
     call  toolbx(FRAMEROUNDRECT,rect,w,h)
```

PAINTROUNDRECT paints the specified round cornered rectangle with the current pen pattern.

```
    integer*2  rect(4)
    integer*4  w,h
    data  rect  /10,10,210,210/
    data  w,h  /20,20/
    call  toolbx(PAINTROUNDRECT,rect,w,h)
```

ERASEROUNDRECT paints the specified round cornered rectangle with the current background pattern.

```
    integer*2  rect(4)
    integer*4  w,h
    data  rect  /10,10,210,210/
    data  w,h  /20,20/
    call  toolbx(ERASEROUNDRECT,rect,w,h)
```

INVERTROUNDRECT inverts all the pixels in the specified round cornered rectangle. Black pixels become white and white pixels become black.

```
    integer*2  rect(4)
    integer*4  w,h
    data  rect  /10,10,210,210/
    data  w,h  /20,20/
    call  toolbx(INVERTROUNDRECT,rect,w,h)
```

FILLROUNDRECT fills the specified round cornered rectangle with the given pattern.

```
    integer*2  rect(4)
    integer*4  w,h
    integer*1  pattern(8)
    data  rect  /10,10,210,210/
    data  w,h  /20,20/
    data  pattern  /b'10101010',
   +                b'01010101',
   +                b'10101010',
   +                b'01010101',
   +                b'10101010',
   +                b'01010101',
   +                b'10101010',
   +                b'01010101'/
    call  toolbx(FILLROUNDRECT,rect,w,h,pattern)
```

FRAMEARC draws an arc of the oval that fits inside the specified rectangular region using the current pen size and pen pattern. The arc is defined with two angles given in degrees. The first angle indicates where the arc is to begin and the second angle specifies the extent of the arc. Positive angles swing clockwise and negative angles swing counter-clockwise. Zero degrees is at twelve o'clock, ninety degrees at three o'clock, 180 degrees at six o'clock, and 270 degrees is at nine o'clock. All other angles are measure relative to the corner points of the enclosing rectangle; forty-five degrees is at the upper right hand corner of the rectangle, regardless of whether the rectangle is a square or not. The outline of the arc is as tall as the pen height and as wide as the pen width.

```
     integer*2  rect(4)
     integer*4  start,arc
     data  rect  /50,50,150,150/
     data  start,arc  /0,45/
     call  toolbx(FRAMEARC,rect,start,arc)
```

PAINTARC paints the specified arc in the current pen pattern.

```
     integer*2  rect(4)
     integer*4  start,arc
     data  rect  /50,50,150,150/
     data  start,arc  /0,45/
     call  toolbx(PAINTARC,rect,start,arc)
```

ERASEARC paints the specified arc in the current background pattern.

```
     integer*2  rect(4)
     integer*4  start,arc
     data  rect  /50,50,150,150/
     data  start,arc  /0,45/
     call  toolbx(ERASEARC,rect,start,arc)
```

INVERTARC inverts the pixels enclosed by the specified arc; white pixels become black and black pixels become white.

```
     integer*2  rect(4)
     integer*4  start,arc
     data  rect  /50,50,150,150/
     data  start,arc  /0,45/
     call  toolbx(INVERTARC,rect,start,arc)
```

FILLARC fills the specified arc with the given pattern.

```
      integer*2  rect(4)
      integer*4  start,arc
      integer*1  pattern(8)
      data  rect  /10,10,210,210/
      data  start,arc  /0,45/
      data  pattern  /b'10101010',
    +                 b'01010101',
    +                 b'10101010',
    +                 b'01010101',
    +                 b'10101010',
    +                 b'01010101',
    +                 b'10101010',
    +                 b'01010101'/
      call  toolbx(FILLARC,rect,start,arc,pattern)
```

1.6 Region Routines

These routines perform operations on regions. A region is made up of a collection of lines and shapes. The region can be a single figure, a figure with holes in it, several disjoint figures, or some combination of these. Regions are built dynamically, that is they are opened, defined, and closed. Manipulation of a region requires an INTEGER*4 variable to hold the handle (pointer to the absolute address) of the region data.

1.6.1 Region Definition and Termination

NEWRGN is an INTEGER*4 function that allocates data space for a region. This function produces no graphic output. It does, however, return the pointer that will be used to perform calculations and graphic operations on the region.

```
integer*4  myregion
myregion = toolbx  (NEWRGN)
```

DISPOSERGN deallocates the memory space that NEWRGN defined. Use this procedure to delete a region from memory. WARNING - Do not try to perform operations on a region once you have disposed of it.

```
integer*4  myregion
myregion = toolbx  (NEWRGN)
call  toolbx  (DISPOSERGN,myregion)
```

COPYRGN allows the duplication of entire region structures. Once the source region is copied into the destination region, it either can be changed or disposed of without affecting the other. COPYRGN does not create a region; the destination region must be predefined by **NEWRGN**.

```
integer*4  srcregion
integer*4  dstregion
srcregion = toolbx  (NEWRGN)
dstregion = toolbx  (NEWRGN)
call  toolbx  (COPYRGN,srcregion,dstregion)
```

SETEMPTYRGN initializes the given region to an empty state. Once performed, the region may be redefined in any way or disposed.

```
integer*4  myregion
myregion = toolbx  (NEWRGN)
call  toolbx  (SETEMPTYRGN,myregion)
```

SETRECTRGN initializes the structure of the given region (as in SETEMPTYRGN) and defines it to be a rectangle with the given boundaries. Note that if the given rectangle is empty (ie - left ≥ right or top ≤ bottom) the effect of this call is equivalent to SETEMPTYRGN.

```
    integer*4  myregion
    integer*4  left,top,right,bottom
    data  left,top,right,bottom  /100,100,200,200/
    myregion = toolbx  (NEWRGN)
    call  toolbx  (SETRECTRGN,left,top,right,bottom)
```

RECTRGN is synonymous with **SETRECTRGN**. The only exception is the way the rectangular boundaries of the region are defined. Instead of individual boundary values, RECTRGN takes a rectangle array (ie - INTEGER*2 RECT(4)).

```
    integer*4  myregion
    integer*2  rect(4)
    data  rect  /100,100,200,200/
    myregion = toolbx  (NEWRGN)
    call  toolbx  (RECTRGN,rect)
```

OPENRGN initiates the actual definition of a region. While a region is open, all calls to **LINE, LINETO,** and the operations that draw framed shapes (except arc) affect the outline of the region. **OPENRGN** calls **HIDEPEN,** so unless there has been an unbalanced **SHOWPEN,** the pen will not draw (see Pen and Line-Drawing Routines).

A region divides the bitmap into two groups: those that lie inside of the region and those that are not. A region should consist of one or more closed loops (where a closed loop is defined as any framed shape or a set of lines that connect to each other or a framed shape).

WARNING - Do not open a region while one is already open. The results are unpredictable.

The following example under **CLOSERGN** demonstrates the use of **OPENRGN.**

CLOSERGN concludes the definition of a region. The list of graphic objects is converted into a region definition and saved at the given region pointer. Note that CLOSERGN calls SHOWPEN to offset the call to HIDEPEN that OPENRGN uses.

WARNING - Perform only one CLOSERGN for every OPENRGN.

```
    integer*4  myregion
```

```
integer*2 rect(4)
myregion = toolbx (NEWRGN)
call toolbx (OPENRGN)
   call toolbx (SETRECT,rect,100,100,200,150)
   call toolbx (FRAMEOVAL,rect)
   call toolbx (MOVETO,101,125)
   call toolbx (LINETO,150,250)
   call toolbx (MOVETO,199,125)
   call toolbx (LINETO,150,250)
call toolbx (CLOSERGN,myregion)
call toolbx (PAINTRGN,myregion)
call toolbx (DISPOSERGN,myregion)
```

1.6.2 Calculations on Regions

OFFSETRGN logically moves a region a distance of **dh** horizontally and **dv** vertically. The screen is only affected when a graphic operation is performed on the region. If dh is positive, the region is moved to the right. If dv is positive, the region is moved down. The shape of the region does not change as it is moved.

```
integer*4 myregion
integer*4 dh,dv
data dh,dv /50,-50/
myregion = toolbx (NEWRGN)
call toolbx (OPENRGN)
   call toolbx (MOVETO,100,100)
   call toolbx (LINETO,200,100)
   call toolbx (LINETO,200,200)
   call toolbx (LINETO,100,100)
call toolbx (CLOSERGN,myregion)
call toolbx (OFFSETRGN,myregion,dh,dv)
```

INSETRGN changes the size of the region. For positive values of dh and dv, all points of the region are moved inward a distance of dh on the horizontal axis and dv vertically. For negative values, the points are moved outward. The object always remains centered during this operation. This operation does not affect the screen until a graphic operation is performed.

```
integer*4 myregion
integer*4 dh,dv
data dh,dv /20,20/
myregion = toolbx (NEWRGN)
call toolbx (OPENRGN)
   call toolbx (MOVETO,100,100)
   call toolbx (LINETO,200,100)
   call toolbx (LINETO,200,200)
   call toolbx (LINETO,100,100)
call toolbx (CLOSERGN,myregion)
```

```
    call  toolbx  (INSETRGN,myregion,dh,dv)
```

SECTRGN defines a new region based on the intersection of two other regions. Note that this does not create a new region. The destination region can be one of the source regions. If the regions do not intersect, then the destination region is defined as empty.

```
    integer*4  srcregionA
    integer*4  srcregionB
    integer*4  dstregion
    srcregionA = toolbx  (NEWRGN)
    srcregionB = toolbx  (NEWRGN)
    dstregion = toolbx  (NEWRGN)
    call  toolbx  (SECTRGN,srcregionA,srcregionB,dstregion)
```

UNIONRGN defines a new region based on the union of two other regions. Note that this does not create a new region. The destination region can be one of the source regions. If both regions are empty, then the destination region is defined as empty.

```
    integer*4  srcregionA
    integer*4  srcregionB
    integer*4  dstregion
    srcregionA = toolbx  (NEWRGN)
    srcregionB = toolbx  (NEWRGN)
    dstregion = toolbx  (NEWRGN)
    call  toolbx  (UNIONRGN,srcregionA,srcregionB,dstregion)
```

DIFFRGN defines a new region based on the difference of the first and the second source regions. Note that this does not create a new region. The destination region can be one of the source regions. If the first source region is empty, then the destination region is defined as empty.

```
    integer*4  srcregionA
    integer*4  srcregionB
    integer*4  dstregion
    srcregionA = toolbx  (NEWRGN)
    srcregionB = toolbx  (NEWRGN)
    dstregion = toolbx  (NEWRGN)
    call  toolbx  (DIFFRGN,srcregionA,srcregionB,dstregion)
```

XORRGN defines a new region based on the difference between the union and the intersection of the two regions. Note that this does not create a new region. The destination region can be one of the source regions. If the regions are coincident, then the destination region is defined as empty.

```
integer*4 srcregionA
integer*4 srcregionB
integer*4 dstregion
srcregionA = toolbx (NEWRGN)
srcregionB = toolbx (NEWRGN)
dstregion = toolbx (NEWRGN)
call toolbx (XORRGN,srcregionA,srcregionB,dstregion)
```

PTINRGN is a logical function that returns true if the pixel that is directly below and to the right of the given point is a part of the given region. The point is defined as a two element INTEGER*2 array with the first element being the vertical value and the second being the horizontal.

```
integer*4 myregion
integer*2 point(2)
logical*4 pointflag
data point /110,150/
myregion = toolbx (NEWRGN)
call toolbx (OPENRGN)
   call toolbx (MOVETO,100,100)
   call toolbx (LINETO,200,100)
   call toolbx (LINETO,200,200)
   call toolbx (LINETO,100,100)
call toolbx (CLOSERGN,myregion)
pointflag = toolbx (PTINRGN,point,myregion)
```

RECTINRGN is a logical function that returns true if the given rectangle intersects the given region.

```
integer*4 myregion
integer*2 rect(4)
logical*4 rectflag
data rect /125,150,250,175/
myregion = toolbx (NEWRGN)
call toolbx (OPENRGN)
   call toolbx (MOVETO,100,100)
   call toolbx (LINETO,200,100)
   call toolbx (LINETO,200,200)
   call toolbx (LINETO,100,100)
call toolbx (CLOSERGN,myregion)
rectflag = toolbx (RECTINRGN,rect,myregion)
```

EQUALRGN is a logical function that returns true if the two given regions are identical in size, shape, and location. Any two empty regions are always considered equal.

```
integer*4 srcregionA
integer*4 srcregionB
logical*4 equalflag
srcregionA = toolbx (NEWRGN)
```

```
    srcregionB = toolbx (NEWRGN)
    equalflag = toolbx (EQUALRGN,srcregionA,srcregionB)
```

EMPTYRGN is a logical function that returns true if the given region is empty. An empty region is one that has been allocated (through **NEWRGN**), but has no size or shape data associated with it.

```
    integer*4  myregion
    logical*4  emptyflag
    myregion = toolbx (NEWRGN)
    emptyflag = toolbx (EMPTYRGN,myregion)
```

1.6.3 Graphic Operations on Regions

These routines all depend on the coordinate system of the current grafport. If a region is drawn in a different grafport than the one in which it was defined, it may not appear in the proper position inside the window.

FRAMERGN draws an outline just inside of the given region. The border is drawn with respect to the current pen pattern, mode, and size. The pen location is not changed by this procedure.

```
    integer*4  myregion
    myregion = toolbx (NEWRGN)
    call toolbx (OPENRGN)
       call toolbx (MOVETO,100,100)
       call toolbx (LINETO,200,100)
       call toolbx (LINETO,200,200)
       call toolbx (LINETO,100,100)
    call toolbx (CLOSERGN,myregion)
    call toolbx (FRAMERGN,myregion)
```

PAINTRGN fills the given region using the current pen pattern and mode. The pen location is not changed by this procedure.

```
    integer*4  myregion
    myregion = toolbx (NEWRGN)
    call toolbx (OPENRGN)
       call toolbx (MOVETO,100,100)
       call toolbx (LINETO,200,100)
       call toolbx (LINETO,200,200)
       call toolbx (LINETO,100,100)
    call toolbx (CLOSERGN,myregion)
    call toolbx (PAINTRGN,myregion)
```

ERASERGN fills the given region with the current background pattern. The current pen pattern and mode are ignored. The pen location is not changed by this procedure.

```
integer*4 myregion
myregion = toolbx (NEWRGN)
call toolbx (OPENRGN)
   call toolbx (MOVETO,100,100)
   call toolbx (LINETO,200,100)
   call toolbx (LINETO,200,200)
   call toolbx (LINETO,100,100)
call toolbx (CLOSERGN,myregion)
call toolbx (ERASERGN,myregion)
```

INVERTRGN negates all the pixels that are a part of the given region. All white pixels become black, and all black pixels become white. The current pen size, pattern, and mode are all ignored. The pen location is not changed by this procedure.

```
integer*4 myregion
myregion = toolbx (NEWRGN)
call toolbx (OPENRGN)
   call toolbx (MOVETO,100,100)
   call toolbx (LINETO,200,100)
   call toolbx (LINETO,200,200)
   call toolbx (LINETO,100,100)
call toolbx (CLOSERGN,myregion)
call toolbx (ERASERGN,myregion)
```

FILLRGN fills a given region with the given pattern. This procedure is synonymous with the call to PAINTRGN with the exception that the fill pattern is defined as a parameter and the current pen pattern, mode, and background pattern are ignored. The pen location is not changed by this procedure.

```
 integer*4 myregion
 integer*1 pattern(8)
 data pattern /b'10101010',
+             b'01010101',
+             b'10101010',
+             b'01010101',
+             b'10101010',
+             b'01010101',
+             b'10101010',
+             b'01010101'/
 myregion = toolbx (NEWRGN)
 call toolbx (OPENRGN)
    call toolbx (MOVETO,100,100)
    call toolbx (LINETO,200,100)
    call toolbx (LINETO,200,200)
    call toolbx (LINETO,100,100)
 call toolbx (CLOSERGN,myregion)
 call toolbx (FILLRGN,myregion,pattern)
```

1.7 Bit Transfer Operation Routines

These routines provide the ability to perform operations on a large block of pixels with one call. The bit transfer operations manipulate bitmaps, so the blocks of pixels can be moved or copied without regard to the contents of the bit image.

SCROLLRECT moves a block of pixels a distance of dh horizontally and dv vertically. The block of pixels is defined as the intersection of the given rectangle, the visible region (current window), the clipping region (default is the boundaries of the current window), the current port rectangle, and the current port's bounds. The window that is supplied by the Microsoft FORTRAN Compiler will define the scroll rectangle to be any visible portion of the window. This operation only affects the bits in the defined scroll rectangle, all others remain unchanged.

When a rectangle is scrolled, a space is made. This space defines a region and is stored in updatergn. **SCROLLRECT** will not create updatergn.

```
integer*2  rect(4)
integer*4  dh,dv
integer*4  updatergn
data  rect  /10,10,210,210/
data  dh,dv  /20,30/
updatergn = toolbx  (NEWRGN)
call  toolbx  (SCROLLRECT,rect,dh,dv,updatergn)
```

COPYBITS moves a bit image from the bitmap of one graphics port to the bitmap of another. The bitmap data for each port is contained in the graphics port structure (see Graphics Port Operations). When moved, the image found in srcrect is scaled to fit dstrect. Then the image is clipped by a mask (mskrgn). The clipping may be suppressed by passing the number 0 instead of a predefined region pointer. The bit transfer is performed with respect to the specified source transfer mode. The source transfer modes are:

```
srccopy    = 0
srcor      = 1
srcxor     = 2
srcbic     = 3
notsrccopy = 4
notsrcor   = 5
notsrcxor  = 6
notsrcbic  = 7
```

The source transfer modes are analogous to the pattern transfer modes that are outlined in the Pen and Line Drawing section, but the values are different.

```
integer*2  srcbits(5),dstbits(5)   ! Bitmaps from 2 grafports.
integer*2  srcrect(4),dstrect(4)
integer*4  srcor
integer*4  maskrgn
data  srcrect  /10,10,30,30/
data  dstrect  /50,80,100,160/
data  srcor  /1/
maskrgn = toolbx  (NEWRGN)
call  toolbx  (OPENRGN)
   call  toolbx  (MOVETO,100,40)
   call  toolbx  (LINETO,50,70)
   call  toolbx  (LINETO,130,80)
   call  toolbx  (LINETO,100,40)
call  toolbx  (CLOSERGN,maskrgn)
call  toolbx  (COPYBITS,srcbits,dstbits,srcrect,dstrect,
+                srcor,maskrgn)
```

1.8 Color Routines

These are not enabled at present, but will be available when the Macintosh supports multiple colors. The colors are specified by integers. There are eight standard colors, which are defined by the following values:

Color	Value
Black	33
White	30
Red	205
Green	341
Blue	409
Cyan	273
Magenta	137
Yellow	69

FORECOLOR sets the foreground color of the current grafport to the given color. The initial foreground is still going to be black.

```
integer*4 color
data color /33/
call toolbx (FORECOLOR,color)
```

BACKCOLOR sets the background color of the current grafport to the given color. The initial background is still going to be white.

```
integer*4 color
data color /33/
call toolbx (BACKCOLOR,color)
```

COLORBIT is a procedure to be used by printing software and other color imaging software. The colrbit field of the grafport structure (see GrafPort Routines) is set to the value held in whichbit. QuckDraw will use this information to determine which color plane to work with. The Macintosh can support up to 32-bits of color information per pixel (ranging from 0 to 31, inclusive). Colrbit is initialized to 0 by OPENPORT and INITPORT.

```
integer*4 whichbit
data whichbit /5/
call toolbx (COLORBIT,whichbit)
```

1.9 Picture Routines

These routines create, define, and delete the QuickDraw structure known as a picture. A picture is made up of one or more graphic commands that are fully enclosed in a rectangle (called the picture frame). A picture may be drawn in any grafport, onto any bitmap. The flexibility of pictures lies in the **dynamic scaling**. For example, a circle that is framed by a square would become an oval if the shape of the destination frame is not square.

When a picture is created, the state of the current grafport is saved. One of the things saved is the **cliprgn**, which specifies where graphics will be visible. The default cliprgn for a grafport, and the one you will get if you do not explicitly set it, is the "wide-open" region (-32767, -32767, 32767, 32767). When you draw a picture in a larger frame, the cliprgn is also enlarged. A common problem is that the default cliprgn cannot be enlarged without overflowing, and overflow results in an empty region. Since graphics will only be visible inside the cliprgn, and since nothing can be inside an empty region, you will see nothing.

The solution is to set the cliprgn to something smaller before trying to use pictures. The bounds rectangle in the portbits field or the portrect of the current grafport (defined in the grafport documentation) are reasonable choices, though an arbitrary rectangle big enough to hold your graphics will do.

An example of the picture routines follows this section.

OPENPICTURE is an INTEGER*4 function. This function returns an absolute address that is used to let QuickDraw know which picture is being defined. Note that OPENPICTURE calls **HIDEPEN** to disable any drawing during the picture definition. If there has been an unbalanced call to **SHOWPEN**, then drawing will still be enabled. Do not attempt to open another picture while one is already opened.

CLOSEPICTURE terminates the definition of the currently open QuickDraw picture. CLOSEPICTURE calls **SHOWPEN** to reenable drawing. If there has been an unbalanced call to **HIDEPEN**, drawing will be disabled. There should be one and only one call to **CLOSEPICTURE** for every call to **OPENPICTURE**.

DRAWPICTURE scales the given picture to the given destination rectangle size and draws the picture.

KILLPICTURE deletes the picture from the Macintosh's memory. Do not try to draw a picture once it has been killed.

The following is a sample program that defines a QuickDraw picture, draws it, and then kills it. It assumes a current, visible grafport with coordinates from 0,0 to 400,300 visible; smaller windows may clip some of the drawings.

```
program drawcircles
integer*2 rect(4)
integer*4 mypicture
integer*4 toolbx
include toolbx.par

call toolbx (SETRECT,rect,0,0,400,300)
call toolbx (CLIPRECT, rect)
call toolbx (SETRECT,rect,0,0,100,100)
mypicture = toolbx (OPENPICTURE,rect)
   call toolbx (SETRECT,rect,20,20,30,50)
   call toolbx (FRAMEOVAL,rect)
   call toolbx (SETRECT,rect,30,30,80,40)
   call toolbx (FRAMERECT,rect)
   call toolbx (SETRECT,rect,80,20,90,50)
   call toolbx (FRAMEOVAL,rect)
call toolbx (CLOSEPICTURE)
call toolbx (SETRECT,rect,0,0,200,200)
call toolbx (DRAWPICTURE,mypicture,rect)
call toolbx (SETRECT,rect,0,0,100,100)
call toolbx (DRAWPICTURE,mypicture,rect)
call toolbx (SETRECT,rect,100,100,200,200)
call toolbx (DRAWPICTURE,mypicture,rect)
call toolbx (KILLPICTURE,mypicture)
stop
end
```

1.10 Polygon Routines

These routines build irregularly shaped figures known as polygons. Polygons are similar to regions in that they are defined in the same manner; the figure is opened, a list of Quickdraw commands is loaded, and the figure is closed. One important difference between the two is the scaling function for polygons treats it as a **continuous** shape (diagonal lines are optimized to appear **smooth**). A region, on the other hand, is simply a **bit image** and when scaled, the bits are left to fall where they may.

1.10.1 Calculations on Polygons

OPENPOLY initiates the definition of a polygon. Being an INTEGER*4 function, it returns an absolute address that QuickDraw will use to perform operations on the polygon. A polygon is defined by moving to a specific place on the current bitmap and drawing a set of connecting lines until the desired figure is complete. QuickDraw will supply a line from the last endpoint to the polygon's initial position, if the two points are not the same. As with regions and pictures, OPENPOLY calls **HIDEPEN** so that no drawing occurs during the polygon definition. Do not attempt to open another polygon while one is already open.

CLOSEPOLY terminates the definition of a polygon. CLOSEPOLY calls **SHOWPEN** to offset the call to HIDEPEN by OPENPOLY. There should be one and only one call to CLOSEPOLY for every call to OPENPOLY.

```
integer*4 mypoly
mypoly = toolbx (OPENPOLY)
   call toolbx (MOVETO,100,100)
   call toolbx (LINETO,200,100)
   call toolbx (LINETO,150,200)
   call toolbx (LINETO,100,100)
call toolbx (CLOSEPOLY)
```

KILLPOLY deletes the given polygon from memory and returns the space to the Macintosh. Do not attempt to manipulate a polygon once it has been killed.

```
integer*4 mypoly
mypoly = toolbx (OPENPOLY)
call toolbx (CLOSEPOLY)
call toolbx (PAINTPOLY,mypoly)
call toolbx (KILLPOLY,mypoly)
```

OFFSETPOLY is the procedure that is used to logically move a polygon once it has been defined. The screen is not changed by this operation. The polygon is moved a distance of dh in the horizontal direction (to the right for positive values) and dv in the vertical direction (down for positive values). The polygon's size and shape are not affected.

```
integer*4 mypoly
integer*4 dh,dv
data dh,dv /20,-20/
mypoly = toolbx (OPENPOLY)
call toolbx (CLOSEPOLY)
call toolbx (OFFSETPOLY,mypoly,dh,dv)
```

1.10.2 Graphic Operations on Polygons

FRAMEPOLY draws an outlined representation of the given polygon. The current pen pattern, mode, and size are all active and reflected in the drawing of the polygon. Note that the pen hangs below and to the right of the border while drawing occurs. For pen sizes of greater than (1,1), this may cause the framed polygon to appear to be slightly larger than expected. All other operations affect the interior of the polygon, only.

If a polygon is currently open, the call to FRAMEPOLY will affect the outline of the figure just as if the individual line commands had been given.

```
integer*4 mypoly
call toolbx (FRAMEPOLY,mypoly)
```

PAINTPOLY fills the given polygon with the current fill pattern with respect to the current pen transfer mode. The pen location is unaffected by the routine.

```
integer*4 mypoly
call toolbx (PAINTPOLY,mypoly)
```

ERASEPOLY fills the given polygon with the current background pattern with respect to the current pen transfer mode. The pen location is unaffected by the routine.

```
integer*4 mypoly
call toolbx (ERASEPOLY,mypoly)
```

INVERTPOLY inverts the color of every pixel in the given polygon (black changes to white and white changes to black). The pen location is unaffected by the routine.

```
     integer*4  mypoly
     call  toolbx  (INVERTPOLY,mypoly)
```

FILLPOLY fills the given polygon with the given pattern, always using the patcopy transfer mode. The pen location is unaffected by the routine.

```
      integer*4  mypoly
      integer*1  pat(8)
      data  pat  /b'10101010',
     +           b'01010101',
     +           b'10101010',
     +           b'01010101',
     +           b'10101010',
     +           b'01010101',
     +           b'10101010',
     +           b'01010101'/
      call  toolbx  (FILLPOLY,mypoly,pat)
```

1.11 Point Routines

A point is a mathematical representation of a pixel on a coordinate plane. The location of a point is defined as the place where a specified horizontal grid line crosses a specified vertical grid line. The Macintosh provides utilities that test and manipulate points.

ADDPT adds the source point to the destination point and returns the result in the destination point.

```
    integer*2  srcpt(2),dstpt(2)
    data  srcpt  /10,20/
    data  dstpt  /5,5/
    call  toolbx  (ADDPT,srcpt,dstpt)
```

SUBPT subtracts the source point from the destination point and returns the result in the destination point.

```
    integer*2  srcpt(2),dstpt(2)
    data  srcpt  /10,20/
    data  dstpt  /50,50/
    call  toolbx  (SUBPT,srcpt,dstpt)
```

SETPT loads the horizontal and vertical parameters into the destination point.

```
    integer*2  pt(2)
    integer*4  h,v
    data h,v  /5,5/
    call  toolbx  (SETPT,pt,h,v)
```

EQUALPT is a logical function that returns true if the two given points are the same.

```
    integer*2  ptA(2),ptB(2)
    logical*4  pointflag
    data  ptA  /10,20/
    data  ptB  /10,20/
    pointflag  =  toolbx  (EQUALPT,ptA,ptB)
```

LOCALTOGLOBAL converts the given point from a local coordinate in the current grafport to a point in the Macintosh's global coordinate system. The advantage to this is that the point can now be compared to all other global points, and be converted to a point in another coordinate system of another grafport.

```
    integer*2  pt(2)
```

```
     data pt /5,5/
     call toolbx (LOCALTOGLOBAL,pt)
```

GLOBALTOLOCAL converts the given point from the Macintosh's global coordinate system into a local point in the current grafport. This routine performs the opposite function that **LOCALTOGLOBAL** performs.

```
     integer*2 pt(2)
     data pt /5,5/
     call toolbx (GLOBALTOLOCAL,pt)
```

1.12 Miscellaneous QuickDraw Utilities

QuickDraw offers several utility procedures and functions for special operations. These utilities cover a wide range of categories from scaling the coordinates of a point to returning a random number.

RANDOM is an INTEGER*4 function that returns a uniformly distributed psuedo-random integer within the range of -32768 to 32767, inclusive.

```
    integer*4  randomnumber
    randomnumber = toolbx (RANDOM)
```

GETPIXEL is a LOGICAL function that returns true if the given pixel is black and false if it is white.

```
    integer*4  h,v
    logical*4  pixelflag
    data  h,v  /100,200/
    pixelflag = toolbx  (GETPIXEL,h,v)
```

STUFFHEX loads a string of hexadecimal digits into a data stucture whose absolute address is passed in thingptr.

```
    integer*4  stripes,stripesptr
    character*255  hexstring
    data  hexstring  /'0102040810204080'/
    stripesptr = toolbx  (PTR,stripes)
    call  toolbx  (STUFFHEX,stripesptr,hexstring)
```

There is <u>no range checking</u> in STUFFHEX. Data can easily overrun the given structure and destroy the data that follows, unless extreme care is taken.

SCALEPT redefines the point that is passed, based on the relationship between the source and destination rectangles.

```
    integer*2  pt(2)
    integer*2  srcrect(4),dstrect(4)
    data  pt  /10,10/
    data  srcrect  /10,10,20,20/
    data  dstrect  /10,10,30,30/
    call  toolbx  (SCALEPT,pt,srcrect,dstrect)
```

MAPPT associates a pixel in the source rectangle to one that is proportionally in the same position of the destination rectangle, and returns the coordinates of the new pixel in pt. For example, a pixel that lies directly in the center of the rectangle (5,5) - (10,10) has the coordinates of (7,7). When that pixel is mapped into the rectangle (20,20) - (35,35), that pixel value becomes (27,27). The given point isn't required to be part of the source rectangle and it is also legal for the source and destination rectangles to overlap.

```
integer*2  pt(2)
integer*2  srcrect(4),dstrect(4)
data  pt  /10,10/
data  srcrect  /10,10,20,20/
data  dstrect  /10,10,30,30/
call  toolbx  (MAPPT,pt,srcrect,dstrect)
```

MAPRECT performs the same function as **MAPPT** but with a rectangle (which in fact is just a very large and sometimes oddly shaped pixel).

```
integer*2  r  (4)
integer*2  srcrect(4),dstrect(4)
data  r  /10,10,15,15/
data  srcrect  /10,10,20,20/
data  dstrect  /10,10,30,30/
call  toolbx  (MAPRECT,r,srcrect,dstrect)
```

MAPRGN performs the same function as **MAPRECT** but with a region (which is always enclosed with a rectangle).

```
integer*4  myregion
integer*2  srcrect(4),dstrect(4)
data  srcrect  /10,10,20,20/
data  dstrect  /10,10,30,30/
myregion  =  toolbx  (NEWRGN)
call  toolbx  (MAPRGN,myregion,srcrect,dstrect)
```

MAPPOLY performs the same function as **MAPRGN** but with a polygon (which, like a region, is always enclosed with a rectangle).

```
integer*4  mypoly
integer*2  srcrect(4),dstrect(4)
data  srcrect  /10,10,20,20/
data  dstrect  /10,10,30,30/
myregion  =  toolbx  (OPENPOLY)
call  toolbx  (MAPPOLY,mypoly,srcrect,dstrect)
```

1.13 Window Routines

The following routines deal with windows.

1.13.1 Window Initialization and Allocation

These routines are used to define and manipulate one or more windows on the Macintosh desktop. Windows are stored in a structure called a window record. The window record is made up of 17 fields that define the attributes of a window. The fields of a window record are:

Port - The window's grafport.

Windowkind - The window's classification. Values from 1 to 7 are reserved for the system. Any value of 8 or greater may be used by the application.

Visible - A flag that, when true, indicates that the window is visible.

Hilited - Determines whether a window is to be highlighted or not.

Goawayflag - Determines whether a window is to have a "Go Away" box or not.

Spareflag - Available for future implementation.

Strucrgn - A pointer to the absolute address (handle) of the window's structure region.

Contrgn - The handle to the window's content region.

Updatergn - The handle to the window's update region.

Windowdefproc - The handle to the window's definition function. This field is changed and used by the window manager and should not be accessed directly.

Datahandle - The handle to data that pertains to the window definition function. This field may be used to store actual data, rather than a handle, if the data is not more than four bytes long.

Titlehandle - The handle to the window's title (if one exists).

Titlewidth - The width of the window's title. This field is used by the window manager and is normally not a concern to the application.

Controllist - The handle to the window's control list.

Nextwindow - The absolute address of the next window in the window list. This pointer refers to the window that is directly beneath this window. If 0, then this window is the last in the list (lying directly against the desktop).

Windowpic - The handle to the QuickDraw picture that represents this window.

Refcon - An INTEGER*4 value that is used by an application to store and access miscellaneous data.

The window record can be declared in the Microsoft FORTRAN Compiler in the following manner:

```
integer*1  windowrecord(156)
integer*2  grafport(54)
integer*2  windowkind
logical*1  visible
logical*1  hilite
logical*1  goawayflag
logical*1  spareflag
integer*4  strucrgn
integer*4  contrgn
integer*4  updatergn
integer*4  windowdefproc
integer*4  datahandle
integer*4  titlehandle
integer*2  titlewidth
integer*4  controllist
integer*4  nextwindow
integer*4  windowpic
integer*4  refcon
equivalence  (windowrecord(1),grafport)
equivalence  (windowrecord(109),windowkind)
equivalence  (windowrecord(111),visible)
equivalence  (windowrecord(112),hilite)
equivalence  (windowrecord(113),goawayflag)
equivalence  (windowrecord(114),spareflag)
equivalence  (windowrecord(115),strucrgn)
equivalence  (windowrecord(119),contrgn)
equivalence  (windowrecord(123),updatergn)
equivalence  (windowrecord(127),windowdefproc)
equivalence  (windowrecord(131),datahandle)
equivalence  (windowrecord(135),titlehandle)
equivalence  (windowrecord(139),titlewidth)
equivalence  (windowrecord(141),controllist)
equivalence  (windowrecord(145),nextwindow)
equivalence  (windowrecord(149),windowpic)
equivalence  (windowrecord(153),refcon)
```

The array windowrecord can be passed to the **NEWWINDOW** toolbox utility (below) to be filled out and added to the Macintosh window list. You can also have the Macintosh Memory Manager allocate space for the window record (see The Microsoft FORTRAN Compiler Toolbox Interface). In this case you will only have a window pointer, an integer variable containing the absolute memory address of the window record. If you need to access the fields of such a record, you will have to use the Microsoft FORTRAN Compiler functions **LONG**, **WORD**, or **BYTE**, which take an absolute address and return the value stored there. To get a pointer to a particular field, you would add the offset of that field to the window pointer. The offsets for the window record can be defined in Microsoft FORTRAN by:

```
integer windowport;      parameter (windowport = z'0')
integer windowkind;      parameter (windowkind = z'6C')
integer wvisible;        parameter (wvisible = z'6E')
integer whilited;        parameter (whilited = z'6F')
integer wgoaway;         parameter (wgoaway = z'70')
integer wspare;          parameter (wspare = z'71')
integer structrgn;       parameter (structrgn = z'72')
integer contrgn;         parameter (contrgn = z'76')
integer updatergn;       parameter (updatergn = z'7A')
integer windowdef;       parameter (windowdef = z'7E')
integer wdatahandle;     parameter (wdatahandle= z'82')
integer wtitlehandle;    parameter (wtitlehandle = z'86')
integer wtitlewidth;     parameter (wtitlewidth = z'8A')
integer wcontrollist;    parameter (wcontrollist = z'8C')
integer nextwindow;      parameter (nextwindow = z'90')
integer windowpic;       parameter (windowpic = z'94')
integer wrefcon;         parameter (wrefcon = z'98')
integer windowsize;      parameter (windowsize = z'9C')
```

INITWINDOWS initializes the Window System and draws the desktop. The desktop is initially a filled gray rectangle with rounded corners.

INITWINDOWS is intended to be called once and only once. The Microsoft FORTRAN Compiler calls this procedure before any program execution begins. Therefore, do not call this procedure. It is documented for user information only.

GETWMGRPORT determines the absolute address of the Window Manager Port. This value is important to many of the remaining window manipulation routines.

```
integer*4 mywindow
call toolbx (GETWMGRPORT,mywindow)
```

NEWWINDOW is an INTEGER*4 function that returns the absolute address of the new window and adds a new window to the window list. The exact specifications of the window are defined by wstorage, boundsrect, title, visible, procid, behind, goawayflag, and refcon.

Wstorage represents the absolute address of the window record structure. If this value is 0, the window record will be allocated on the heap. The disadvantage to this is that the record is non-relocatable and may fragment the heap space. Therefore, it is not recommended that wstorage be passed a 0.

Boundsrect defines the window size and location in global coordinates. When converted to local coordinates, it becomes the portrect for the window's grafport (see the QuickDraw section for a more detailed description of the structure of the grafport).

Title is the character string that will appear at the top of the window. If the string is too long, it will be truncated on the right to the number of visible characters. As with other strings, the first character must hold the length of the character string.

Visible is a logical that, when true, forces the window to be visible upon initialization.

Procid is an integer value that defines the window type. The possible window types are:

```
documentproc  = 0   ! standard document window
dboxproc      = 1   ! alert box or modal dialog box
plaindbox     = 2   ! plain box
altdboxproc   = 3   ! plain box with shadow
nogrowdocproc = 4   ! document window without size box
rdocproc      = 10  ! rounded-corner window
```

If **behind** is 0, then the window is at the bottom of all other windows on the desktop. If -1, the window is on top. Otherwise, the new window is inserted behind the given "behind" window.

Goawayflag specifies whether the window has a goaway box.

Refcon is the window's reference value that is set and used by the application.

The following example assumes knowledge of the grafport (see the grafport section of QuickDraw) and window record structures (see beginning of this section).

```
integer*4 wstorage
integer*2 boundsrect(4)
character*256 title
logical*4 visible
integer*4 procid
```

```
      integer*4  behind
      logical*4  goawayflag
      integer*4  refcon
      integer*4  mywindow
      wstorage = toolbx  (PTR,windowrecord)
      mywindow = toolbx  (NEWWINDOW,wstorage,boundsrect,title,
     +                    visible,procid,behind,goawayflag,refcon)
```

GETNEWWINDOW is another INTEGER*4 function that returns the absolute address of the window structure. GETNEWWINDOW performs exactly the same function as NEWWINDOW. The only difference is GETNEWWINDOW reads most of its information from an application resource (identified by windowid), which must be on the resource fork of the application file (see Resource Compiler). The parameters **wstorage** and **behind** define the same type of information as with NEWWINDOW.

```
      integer*4  windowid
      integer*4  wstorage,behind
      integer*4  mywindow
      mywindow = toolbx  (GETNEWWINDOW,windowid,wstorage,behind)
```

CLOSEWINDOW is used to terminate a window. The window is deleted from both screen and internal memory. The window record itself, however, is maintained. Any manipulations that are done on the window after the CLOSEWINDOW are ignored. If the deleted window was the active one, then the next window becomes visible and active.

```
      integer*4  mywindow
      call  toolbx  (CLOSEWINDOW,mywindow)
```

DISPOSEWINDOW performs the same functions as **CLOSEWINDOW** and deallocates the memory that was being used to store the window record. This procedure is called when the space for the window record was allocated on the heap by **NEWWINDOW**. If this space is not released, heap fragmentation may occur.

```
      integer*4  mywindow
      call  toolbx  (DISPOSEWINDOW,mywindow)
```

1.13.2 Window Display Routines

These routines manipulate the window's appearance or plane level but not its size or shape.

SETWTITLE sets the given window's title to the string that is passed in title. The first character of the string must contain the length of the title.

```
    integer*4 mywindow
    character*256 title
    title = char(11)//'Window Title'
    call toolbx (SETWTITLE,mywindow,title)
```

GETWTITLE returns the title of the given window. The string returned contains a length byte as the first character.

```
    integer*4 mywindow
    character*256 title
    integer*4 titlelength
    call toolbx (GETWTITLE,mywindow,title)
    titlelength = ichar(title(1:1))
```

SELECTWINDOW causes the given window to be brought to the front and become active. This procedure also causes the given window to be highlighted and unhighlights the window that was active (if there was one).

```
    integer*4 mywindow
    call toolbx (SELECTWINDOW,mywindow)
```

HIDEWINDOW causes the given window to become invisible. If it was the foremost window, the next window is highlighted and made active.

```
    integer*4 mywindow
    call toolbx (HIDEWINDOW,mywindow)
```

SHOWWINDOW causes the given window to become visible. No action is taken to highlight the given window when it becomes visible.

```
    integer*4 mywindow
    call toolbx (SHOWWINDOW,mywindow)
```

SHOWHIDE forces the window to become either visible or invisible depending on the state of the showflag parameter.

```
    integer*4 mywindow
    logical*4 showflag
    showflag = .true.
    call toolbx (SHOWHIDE,mywindow,showflag)
```

HILITEWINDOW forces the window to become either highlighted or unhighlighted depending on the state of the **fhilite** parameter.

```
integer*4  mywindow
logical*4  fhilite
fhilite = .true.
call  toolbx  (HILITEWINDOW,mywindow,fhilite)
```

BRINGTOFRONT forces the given window to the frontmost position on the desktop. This routine does not perform any highlighting operations on the given window.

```
integer*4  mywindow
call  toolbx  (BRINGTOFRONT,mywindow)
```

SENDBEHIND takes the given window and puts it behind behindwindow on the desktop. If the value in behind window is 0, then the given window is put behind all other visible windows. If the given window is the frontmost, then it is unhighlighted and the next window is highlighted and becomes active.

```
integer*4  mywindow
integer*4  behindwindow
call  toolbx  (SENDBEHIND,mywindow,behindwindow)
```

FRONTWINDOW is an INTEGER*4 function that returns the absolute address of the window that is frontmost on the desktop. If there are not any visible windows on the desktop, then FRONTWINDOW returns 0.

```
integer*4  mywindow
mywindow = toolbx  (FRONTWINDOW)
```

DRAWGROWICON draws the size box and scroll bar areas in the given window, if the given window is active. If the window is not active, only the scroll bar areas are drawn. The scroll bar areas are not erased by this routine.

```
integer*4  mywindow
call  toolbx  (DRAWGROWICON,mywindow)
```

1.13.3 Mouse Location Routines

The mouse location routines for windows are used to determine the position of the cursor when a mouse down event occurs.

FINDWINDOW is an INTEGER function that returns a value that describes the location of the given point when the call occurs. Further, if the point lies within a window, the parameter whichwindow will return the absolute address of that window. Whichwindow is set to 0 if the point is not in a window.

The possible values that may be returned by FINDWINDOW are:

```
indesk       = 0   ! none of the following
inmenubar    = 1   ! in menu bar
insyswindow  = 2   ! in system window
incontent    = 3   ! in content region (except grow, if active)
indrag       = 4   ! in drag region
ingrow       = 5   ! in grow region (active window only)
ingoaway     = 6   ! in go-away region (active window only)
```

FINDWINDOW will return the value for indesk when the given point is on the desktop but outside of the menu bar or any of the windows. Indesk may also be returned if the point is inside a window frame, but not in a drag or go-away region.

This routine is especially useful when used in conjunction with GETMOUSE to determine where the cursor is at any given point.

```
integer*2  thept(2)
integer*4  whichwindow
integer*4  where
data thept  /10,20/
where  =  toolbx  (FINDWINDOW,thept,whichwindow)
```

TRACKGOAWAY highlights the go-away region of the window as long as the mouse button is down. When the application detects a mouse-down event inside the go-away region of the window and calls TRACKGOAWAY, control is transferred to the function until the mouse button is released. At that time, TRACKGOAWAY returns a true if the cursor is still inside of the go-away area. If not, false is returned and no action should be taken.

```
integer*4  thewindow
integer*2  thept(2)
logical  mouseflag
data thept  /10,20/
mouseflag  =  toolbx  (TRACKGOAWAY,thewindow,thept)
```

1.13.4 Window Movement and Sizing Routines

These routines cause the given window to alter its position and dimensions. This manipulation is a common occurrence with the Macintosh since these processes are used to move and size the volume windows from within the finder.

MOVEWINDOW uses the top left corner of the given window as a reference point to move the window to a different part of the screen. The new location is defined by the global coordinates hglobal and vglobal. Note that the local coordinates of the top left corner of the given window remain the same. The size and shape of the window also remain constant. If front is true, then the given window is forced to the front and becomes active.

```
     integer*4  mywindow
     integer*4  hglobal,vglobal
     logical*4  front
     data  hglobal,vglobal  /100,150/
     front  =  .true.
     call  toolbx  (MOVEWINDOW,mywindow,hglobal,vglobal,front)
```

DRAGWINDOW tracks a gray outline of the given window with the mouse position while the mouse button is being held down. When the button is released, the window is moved to the gray outline and forced to be active. If the button is released and the mouse position is outside of the boundsrect, the window is returned to its original position, without forcing it to become active.

```
     integer*4  mywindow
     integer*2  point(2)
     integer*2  boundsrect(4)
     data  point  /100,110/
     data  boundsrect  /10,10,210,210/
     call  toolbx  (DRAGWINDOW,mywindow,point,boundsrect)
```

GROWWINDOW is an INTEGER*4 function that pulls a "grow image" from the given window based upon the movements of the mouse while the button is down. Note that this function does not cause the window itself to change size. When the mouse button is released, GROWWINDOW returns the window's size in finalsize, with the vertical size in the first element and the horizontal size in the second.

```
     integer*4  mywindow
     integer*2  startpt(2)
     integer*2  sizerect(4)
     integer*2  finalsize(2)
     data  startpt  /100,150/
     data  sizerect  /40,40,100,150/
     finalsize  =  toolbx  (GROWWINDOW,mywindow,startpt,sizerect)
```

SIZEWINDOW changes the size of the given window to the width and height that are specified in w and h. The parameter fupdate determines whether the area that is affected by this size change is to be updated "automatically". If true, any areas added to the window will be included in the update region, causing them to be included in the next update event.

Otherwise, the application should assume the responsibility for updating the screen.

```
    integer*4  mywindow
    integer*4  w,h
    logical*4  fupdate
    data  w,h  /100,150/
    fupdate  =  .true.
    call  toolbx  (SIZEWINDOW,mywindow,w,h,fupdate)
```

1.13.5 Update Region Maintenance Routines

These routines maintain the update region of the window structure. When a window's dimensions are changed, there exists an area of the desktop that becomes undefined. A data representation of this area is loaded into the update region of the window structure. The following routines provide support in determining the size and position of this area.

INVALRECT defines to the Macintosh that a rectangular area of the desktop has changed and requires updating. The rectangular area is accumulated into the update region of the window whose grafport is the current port. The rectangle lies within the window's content region and is therefore defined in the window's local coordinates.

```
    integer*2  badrect(4)
    call  toolbx  (INVALRECT,badrect)
```

INVALRGN performs the same function as **INVALRECT** with the exception that the work is done using regions.

```
    integer*4 badrgn
    call toolbx (INVALRGN,badrgn)
```

VALIDRECT tells the window manager that the application has already redrawn the rectangle and to cancel any updates that have been accumulated for that area.

```
    integer*2  goodrect(4)
    call  toolbx  (VALIDRECT,goodrect)
```

VALIDRGN performs the same function as **VALIDRECT** with the exception that the work is done using regions.

```
    integer*4  goodrgn
    call  toolbx  (VALIDRGN,goodrgn)
```

BEGINUPDATE is called when an update event is detected (see Miscellaneous Event Utilities). At this time, the window manager has noted that there is an area on the desktop that is undefined, and needs to be updated. This procedure is the first step in updating the desktop. BEGINUPDATE masks out the parts of the window that do not require an update and clears the update region so this update event is not brought up, again. Immediately following this call, the application is given the responsibility to take the undefined area of the window and define it. Since BEGINUPDATE masks out part of the visible window it is important to always call ENDUPDATE when the update is complete.

ENDUPDATE restores the ability to draw in any portion of the window. There should always be a call to ENDUPDATE to offset every call to BEGINUPDATE.

```
      integer*4  mywindow
      integer*2  rect(4)
      integer*1  white(8)
      data  rect  /0,0,342,512/
      data  white  /0,0,0,0,0,0,0,0/
      call  toolbx  (BEGINUPDATE,mywindow)

*  Clear  the  window,  I  didn't  want  it,  anyway.
      call  toolbx  (FILLRECT,rect,white)
      call  toolbx  (ENDUPDATE,mywindow)
```

1.13.6 Miscellaneous Window Utilities

The following window utilities provide a way to define and retrieve certain window attributes, restrict the movement of figures in the window, and reposition the window with the mouse.

SETWREFCON assigns the given window's reference value to the given data.

```
      integer*4  mywindow
      integer*4  data
      call  toolbx  (SETWREFCON,mywindow,data)
```

GETWREFCON returns the given window's reference value.

```
      integer*4  mywindow
      integer*4  data
      data  =  toolbx  (GETWREFCON,mywindow)
```

SETWINDOWPIC loads the windowpic (see Window Initialization and Allocation) of the given window record with the given picture handle. This causes the window manager to draw the given picture into the window's contents, rather than generating an update event (see Update Region Maintenance Routines).

```
integer*4 mywindow
integer*4 pic
call toolbx (SETWINDOWPIC,mywindow,pic)
```

GETWINDOWPIC returns the pointer to the absolute address of the picture that draws the given window's contents as previously stored by **SETWINDOWPIC**.

```
integer*4 mywindow
integer*4 pic
call toolbx (SETWINDOWPIC,mywindow,pic)
```

PINRECT is an INTEGER*4 function that returns a point that is either inside or on one of the borders of the given rectangle. If the given point is inside the given rectangle, then the given point is returned. Otherwise, the nearest boundary of the rectangle is substituted for the illegal part of the coordinate.

```
integer*2 therect(4)
integer*2 thept(2)
integer*2 newpt(2)
data therect /10,10,210,210/
data thept /50,60/
newpt = toolbx (PINRECT,therect,thept)
```

DRAGGREYRGN is an INTEGER*4 function that is used to drag the gray shape of the given region while the mouse button is held down.

The starting point is defined as the upper left hand corner of the region that is being dragged. When the mouse button is released, DRAGGREYRGN returns the offset from the starting point to the current location of the gray region's upper left hand corner.

If your purpose is to move a region from one place to another, you must use COPYRGN to define a disposable version of the region since DRAGGREYRGN will alter its shape and location. When this copied region has been moved, the offset data that is returned can be used to relocate the original region using OFFSETRGN. If the mouse pulls the region outside of the limitrect, then the region is "pinned" to the edge of the limitrect that is closest to the cursor. If the region is pulled outside of the sloprect (which should completely enclose the limitrect), DRAGGREYRGN will return -32768 in both the horizontal and vertical values of the offset point.

The gray region can be restricted to horizontal or vertical movements. The value of the axis determines this. If 0, then there is no restriction to the direction a region may be moved. If 1, then the region can only be moved horizontally. If 2, then the region can only be moved in the vertical direction.

The **actionproc** points to a procedure that is performed over and over again until the user releases the mouse button. If this parameter is 0, then DRAGGREYRGN retains control until the button is released.

The following example shows how DRAGGREYRGN might be used to move a figure from one point to another. The program continually monitors the Macintosh's event queue for a "mouse down" event. If one is detected and if the cursor is within the bounds of the region, DRAGGREYRGN is called. The limitrect and sloprect are displayed to visually delimit the usable area. When the region is moved outside of the sloprect, the program terminates.

```
      program dragtest

* The event record
      logical*4  eventflag
      integer*4  eventmask
      integer*2  myevent(8)
      integer*2  what
      integer*4  message
      integer*4  when
      integer*2  where(2)
      integer*2  modifiers
      equivalence  (myevent(1),what)
      equivalence  (myevent(2),message)
      equivalence  (myevent(4),when)
      equivalence  (myevent(6),where(1))
      equivalence  (myevent(8),modifiers)

* DRAGGREYRGN parameters
      logical*4  inregion
      integer*4  thergn
      integer*2  limitrect(4),sloprect(4)
      integer*4  axis
      integer*4  actionproc
      integer*4  offsetpt

* Miscellaneous utility storage
      integer*4  thergncopy
      integer*2  dh,dv
      integer*4  deltah,deltav
      include  toolbx.par
      integer  toolbx

      common  dv,dh
      equivalence  (offsetpt,dv)

      data  limitrect  /20,20,250,300/
      data  sloprect  /10,10,260,310/
      data  axis  /0/
      data  actionproc  /0/
      data  eventmask  /-1/

*  Graphically  display  the  limits  of  the  DRAG
      call  toolbx  (FRAMERECT,limitrect)
      call  toolbx  (FRAMERECT,sloprect)

*  Define  a  diamond  to  DRAG...
      thergn=toolbx  (NEWRGN)
      call  toolbx  (OPENRGN)
           call  toolbx  (MOVETO,45,40)
           call  toolbx  (LINETO,50,50)
           call  toolbx  (LINETO,45,60)
           call  toolbx  (LINETO,40,50)
           call  toolbx  (LINETO,45,40)
      call  toolbx  (CLOSERGN,thergn)
      call  toolbx  (PAINTRGN,thergn)

*  ...and  allocate  storage  for  a  working  copy  of  it
      thergncopy=toolbx  (NEWRGN)

      do

*  See  if  there  are  any  events  pending  (define  what  and  where)
           eventflag  =  toolbx  (GETNEXTEVENT,eventmask,myevent)
           select  case  (what)

*  Mouse  down  event
                case  (1)
                     call  toolbx  (GLOBALTOLOCAL,where)
                     inregion  =  toolbx  (PTINRGN,where,thergn)
                          if  (inregion)  then
                          call  toolbx  (COPYRGN,thergn,thergncopy)

*  Redefine  the  position  of  the  working  copy  by  dragging  it  with  the  mouse
                          offsetpt  =  toolbx  (DRAGGREYRGN,thergncopy,where,
     +                                 limitrect,sloprect,axis,actionproc)

*  Make  sure  that  it  was  not  moved  out  of  its  limits  (sloprect)
                          if  (dh  =  -32768)  then
                               exit
                          else

*  Assign  the  returned  offsets  to  INTEGER*4  variables  before  using  them
                               deltah  =  dh
                               deltav  =  dv
```

```
                              call  toolbx  (ERASERGN,thergn)
                              call  toolbx  (OFFSETRGN,thergn,deltah,deltav)
                              call  toolbx  (PAINTRGN,thergn)
                           endif
                        endif

* All other events are ignored
               case default
            endselect
         repeat

* Discard the memory that was used to define the 2 regions
         call  toolbx  (DISPOSERGN,thergn)
         call  toolbx  (DISPOSERGN,thergncopy)
         end
```

1.14 Menu Routines

The following routines operate on menus through the Macintosh Menu Manager.

1.14.1 Menu Initialization and Allocation

These routines provide access to the Macintosh's Menu Manager. Menus can be created directly by your application or they can be read as predefined resources. Further information on resources and the resource manager is available in the introduction to the Microsoft FORTRAN Compiler Toolbox Interface, and in the document that describes **RMaker**, the Resource Compiler.

Menus are maintained with a menu record which is allocated in the system heap when a new menu is created. The elements of a menu record are:

Menuid - The unique identifier of an individual menu. Most of the menu manager routines use this value for indicating which menu is being referenced.

Menuwidth - The width of the rectangle that contains the menu items. This value is calculated during the menu definition.

Menuheight - The height of the rectangle the contains the menu items. This value is calculated during the menu definition.

Menuproc - A pointer to the absolute address of the menu's definition procedure. Normally you will use the standard Macintosh menu format, however this location allows you to define custom menus. Refer to "Inside Macintosh" for further information.

Enableflags - An array of flags that indicate which items of the menu are enabled.

Menudata - Stores the menu title, text within the menu and other menu data (see APPENDMENU in this section).

A menu record can be accessed in the Microsoft FORTRAN Compiler in the following manner:

```
integer*4 menu              ! pointer to menu record
integer*2 id, width, height
integer*4 proc
logical*1 flags(32)
character*1 data(256)

id      = word(menu)
```

```
width   = word(menu+2)
height  = word(menu+4)
proc    = long(menu+6)
do  (i=10,41);  flags(i)  = byte(menu+i);  repeat
do  (i=42,297);  data(i)  = byte(menu+i);  repeat
```

INITMENUS initializes the Macintosh's Menu System. INITMENUS is intended to be called once and only once. The Microsoft FORTRAN Compiler calls this procedure before any program execution begins. Therefore, **do not** call this procedure. It is documented for user information only.

NEWMENU is an INTEGER*4 function that returns a pointer to the absolute address of the new menu structure. The memory for the structure is allocated by NEWMENU and is assigned the given menuid and menutitle.

The menuid is an identifier that refers to a particular menu. This can be any number that is greater than 0 and less than 32768. Menus that are stored as resources use the resourceid (see Resource Compiler).

Menutitle defines the character string that will appear in the menu bar. This string may be up to 255 characters in length and must be passed with the length of the string in the first character position.

```
integer*4  menuhandle
integer*4  menuid
character*256  menutitle
menuid  =  30
menutitle  =  char(8)//'Commands'
menuhandle  =  toolbx  (NEWMENU,menuid,menutitle)
```

GETMENU is an INTEGER*4 function that returns a pointer to the absolute address of the menu record having the given menuid. It retrieves this information by reading an open resource file. If **RMaker** was used to add the following menu type as a resource to your application file:

```
TYPE MENU
  ,30                  ;; resource id
Commands               ;; menu title
Type                   ;; item 1
Print                  ;; item 2
Chain                  ;; item 3
Exit                   ;; item 4
                       ;; blank line to terminate
```

GETMENU could be called as follows:

```
    integer*4  menuhandle
    integer*4  menuid
    menuid = 30
    menuhandle = toolbx (GETMENU,menuid)
```

DISPOSEMENU terminates a menu structure by giving the menu's memory space back to the Macintosh. This routine will not remove the menu from the menu list. It is removed from the menu list with **DELETEMENU**. A menu should be deleted, and menu bar updated, before the call to DISPOSEMENU is made. Do not try to use a menu that has been disposed.

```
    integer*4  menuid
    character*256  menutitle
    integer*4  menuhandle
    data menuid /1/
    menutitle = char(8)//'TheTitle'
    menuhandle = toolbx (NEWMENU,menuid,menutitle)
    call toolbx (DELETEMENU,menuid)
    call toolbx (DRAWMENUBAR)
    call toolbx (DISPOSEMENU,menuhandle)
```

APPENDMENU adds one or more menu items to the end of the given menu. The menu record must have been allocated before an item is appended to a menu list. The data can have imbedded meta-characters which control the appearance of the menu items. The meta-characters are:

; or RETURN	Separates multiple items
^	Followed by an icon number, adds that icon to the item
!	Followed by a character, marks the item with that character (a check mark for example)
<	Followed by B, I, U, O, or S, sets the character style of the item, for example:
<B	- **Bold**
<I	- *Italic*
<U	- Underline
<O	- **Outline**
<S	- **Shadow**
/	Followed by a character, associates a keyboard equivalent with the item
(	Disables the item

If the first character of a menu item is a hyphen (-), the item will be a dividing line across the width of the menu.

```
    integer*4  menuhandle
    integer*4  menuid
    character*256  menutitle
    character*256  menuitems
    menuid = 30
```

```
menutitle = char(8)//'Commands'
menuhandle = toolbx (NEWMENU,menuid,menutitle)
menuitems = char(21)//"Type;Print;Chain;Edit"
call toolbx (APPENDMENU,menuhandle,menuitems)
```

ADDRESMENU searches all of the open resource files to locate the resources that are of the given type. When a match is found, the resource is appended to the given menu, and the search continues. The following example builds a menu of all of the character fonts that are available.

```
integer*4 menuhandle
character*4 type
type = 'FONT'
menuhandle = toolbx (NEWMENU,5,char(5)//'FONTS')
call toolbx (ADDRESMENU,menuhandle,type)
```

INSERTRESMENU performs the same operation as **ADDRESMENU** but without restricting the addition of menu items to only the end of the menu. INSERTRESMENU allows an application to position a resource item anywhere in the menu. If the resource types are found, they are inserted in the given menu following the item that is indexed by **afteritem**. If afteritem is 0, the resource items are inserted at the top of the menu.

```
integer*4 menuhandle
character*4 type
integer*4 afteritem
type = 'FONT'
afteritem= 3
menuhandle = toolbx (NEWMENU,5,char(5)//'FONTS')
call toolbx (INSERTRESMENU,menuhandle,type,afteritem)
```

1.14.2 Forming the Menu Bar

These routines perform the operations that build and alter the menu bar.

INSERTMENU puts a menu into the menu list. The new menu is positioned immediately before the current menu that is indexed by **beforeid**. Note that the menu bar is not changed by this procedure. You must call **DRAWMENUBAR** to update the menu bar.

```
integer*4 menuhandle
integer*4 beforeid
data beforeid /2/
call toolbx (INSERTMENU,menuhandle,beforeid)
```

DRAWMENUBAR draws the menu bar based on the information stored in the menu list.

```
    call toolbx (DRAWMENUBAR)
```

DELETEMENU deletes the given menu from the menu list. This procedure does not affect the screen. A call to **DRAWMENUBAR** should follow to make sure the current menu bar is on the screen.

```
    integer*4 menuid
    data menuid /2/
    call toolbx (DELETEMENU,menuid)
    call toolbx (DRAWMENUBAR)
```

CLEARMENUBAR deletes all menus from the current menu list. Note that this procedure does not affect the screen. A call to **DRAWMENUBAR** should follow to make sure the current menu bar is on the screen.

```
    call toolbx (CLEARMENUBAR)
    call toolbx (DRAWMENUBAR)
```

GETNEWMBAR is an INTEGER*4 function that returns the handle of a new menu list. The new menu contains the items that are defined by the menu bar resource having the given **menubarid** and must be defined in an open resource file (see Resource Compiler). You can force the new menu list to become the current menu list by calling **SETMENUBAR**.

```
    integer*4 menubarid
    integer*4 menuhandle
    data menubarid /4/
    call toolbx (GETNEWMBAR,menubarid)
```

GETMENUBAR is an INTEGER*4 function that returns a pointer to the absolute address of a copy of the current menu list. GETMENUBAR creates this copy so you can alter the current menu list and immediately return to the original version.

```
    integer*4 menuhandle
    menuhandle = toolbx (GETMENUBAR)
```

SETMENUBAR forces the given menu list to become the current menu list that was previously saved with **GETMENUBAR**.

```
    integer*4 menuhandle
    menuhandle = toolbx (GETMENUBAR)
    call toolbx (SETMENUBAR,menuhandle)
```

1.14.3 Choosing from a Menu

These routines provide support for detecting the selection of a menu and highlighting the menu when it is selected.

MENUSELECT is an INTEGER*4 function that returns information about the cursor location in the menu area. When a mouse down event is detected in the menu bar area, and MENUSELECT is called, control is transferred to MENUSELECT until the mouse button is released. At that time, the routine returns two INTEGER*2 values packed into a longword (INTEGER*4). The first is the menu ID and the second is the menu item number. If the mouse button was released outside of the menu area, then a 0 is returned in both fields. In practice, the value of **startpt** always comes from an event record (see "Inside Macintosh", available from Apple Computers).

```
integer*2  startpt(2)
integer*2  menudata(2)
integer*4  selectdata
eqivalence  (menudata,  selectdata)
data  startpt  /20,5/
selectdata  =  toolbx  (MENUSELECT,startpt)
```

MENUKEY is an INTEGER*4 function that performs much the same function as MENUSELECT, but using the command key equivalents for menu items. When a key down event with the command key depressed is detected, MENUKEY can be used to return data about the menu item corresponding to the character entered. The 4 byte integer that is returned is of the same format as the one with MENUSELECT. In practice, the value of **ch** comes from the message field of an event record (see the Event Manager Manual of Inside Macintosh).

```
character*4  ch
integer*2  menudata(2)
integer*4  selectdata
eqivalence  (menudata,  selectdata)
data ch /'   C'/
selectdata  =  toolbx  (MENUKEY,ch)
```

HILITEMENU forces the given menu title to become highlighted. If another title was highlighted, it is unhighlighted. After the required processing for a menu selection is complete, the menu title can be unhighlighted by calling HILITEMENU with an argument of zero.

```
integer*4  menuid
data  menuid  /4/
```

```
    call toolbx (HILITEMENU,menuid)
```

1.14.4 Controlling the Appearance of a Menu Item

These routines control the way a menu item is displayed when the menu is chosen. Also, operations are provided to retrieve information about the current menu items.

SETITEM loads the given character string into the given menu title. The given menu is defined as a handle to the menu record. The meta-characters that were used in **ADDRESMENU** are not recognized by this procedure. The first character in itemstring must contain the number of characters in the string.

```
    integer*4 menuhandle
    integer*4 item
    character*256 itemstring
    integer*4 menuid
    character*256 menutitle
    data menuid /1/
    data item /5/
    menutitle = char(10)//'Menu Title'
    menuhandle = toolbx (NEWMENU,menuid,menutitle)
    itemstring = char(9)//'Menu Item'
    call toolbx (SETITEM,menuhandle,item,itemstring)
```

GETITEM returns the text in a menu item. The first character in **itemstring** contains the number of characters in the string.

```
    integer*4 menuhandle
    integer*4 item
    character*256 itemstring
    integer*4 menuid
    character*256 menutitle
    data menuid /1/
    data item /5/
    menutitle = char(10)//'Menu Title'
    menuhandle = toolbx (NEWMENU,menuid,menutitle)
    call toolbx (GETITEM,menuhandle,item,itemstring)
```

DISABLEITEM dims the given menu item and prohibits any action on the item, if chosen. If **item** is 0, then all items under the given menu are disabled.

```
    integer*4 menuhandle
    integer*4 item
    integer*4 menuid
    character*256 menutitle
    data menuid /1/
    data item /0/
    menutitle = char(10)//'Menu Title'
    menuhandle = toolbx (NEWMENU,menuid,menutitle)
    call toolbx (DISABLEITEM,menuhandle,item)
```

ENABLEITEM causes the given menu item to no longer be dimmed and allows it to be selected. If **item** is 0, the ENABLEITEM enables the entire menu.

```
    integer*4 menuhandle
    integer*4 item
    integer*4 menuid
    character*256 menutitle
    data menuid /1/
    data item /0/
    menutitle = char(10)//'Menu Title'
    menuhandle = toolbx (NEWMENU,menuid,menutitle)
    call toolbx (ENABLEITEM,menuhandle,item)
```

CHECKITEM forces the given menu item to either have a check mark to the left of it or not, depending on the state of the logical parameter checked.

```
    integer*4 menuhandle
    integer*4 item
    logical*4 checked
    integer*4 menuid
    character*256 menutitle
    data menuid /1/
    data item /0/
    menutitle = char(10)//'Menu Title'
    menuhandle = toolbx (NEWMENU,menuid,menutitle)
    checked = .true.
    call toolbx (CHECKITEM,menuhandle,item,checked)
```

SETITEMICON associates the given menu item with an **icon**. The given icon number is added to the value 256 to associate the icon with its resource indentification number (see Resource Compiler).

```
    integer*4 menuhandle
    integer*4 item
    integer*4 icon
    integer*4 menuid
    character*256 menutitle
    data menuid /1/
    data item /0/
    menutitle = char(10)//'Menu Title'
    menuhandle = toolbx (NEWMENU,menuid,menutitle)
    icon = 255
    call toolbx (SETITEMICON,menuhandle,item,icon)
```

GETITEMICON returns the **icon** number that is associated with the given menu item.

```
integer*4  menuhandle
integer*4  item
integer*4  icon
integer*4  menuid
character*256  menutitle
data  menuid  /1/
data  item  /0/
menutitle  =  char(10)//'Menu  Title'
menuhandle  =  toolbx  (NEWMENU,menuid,menutitle)
call  toolbx  (GETITEMICON,menuhandle,item,icon)
```

SETITEMSTYLE changes the character style of the given menu item.

```
integer*4  menuhandle
integer*4  item
integer*4  chstyle
integer*4  menuid
character*256  menutitle
data  menuid  /1/
data  item  /5/
menutitle  =  char(10)//'Menu  Title'
menuhandle  =  toolbx  (NEWMENU,menuid,menutitle)
call  toolbx  (SETITEMSTYLE,menuhandle,item,chstyle)
```

GETITEMSTYLE returns the character style of the given menu item.

```
integer*4  menuhandle
integer*4  item
integer*4  chstyle
integer*4  menuid
character*256  menutitle
data  menuid  /1/
data  item  /0/
menutitle  =  char(10)//'Menu  Title'
menuhandle  =  toolbx  (NEWMENU,menuid,menutitle)
call  toolbx  (GETITEMSTYLE,menuhandle,item,chstyle)
```

SETITEMMARK places the given mark character to the left of the given menu item. This is a more general case than the check mark used by **CHECKITEM**. The following are some marks that may be used:

```
cloverleaf   = 17  ! Command Symbol
checkmark    = 18  ! Check Mark
diamond      = 19  ! Diamond
```

```
     applesymbol = 20  ! Apple Symbol
     nomark      = 0   ! Nothing, to remove a mark
```

```
    integer*4  menuhandle
    integer*4  item
    character*4  markchar
    integer*4  menuid
    character*256  menutitle
    data  menuid  /1/
    data  item  /3/
    menutitle  =  char(10)//'Menu  Title'
    menuhandle  =  toolbx  (NEWMENU,menuid,menutitle)
    markchar  =  '    ' // char  (20)          !  Only  the  last  character
                                               !   is  significant.
    call  toolbx  (SETITEMMARK,menuhandle,item,markchar)
```

GETITEMMARK returns the mark character of the given menu item. Returns 0 if there is no mark.

```
    integer*4  menuhandle
    integer*4  item
    character*1  markchar
    integer*4  menuid
    character*256  menutitle
    data  menuid  /1/
    data  item  /3/
    menutitle  =  char(10)//'Menu  Title'
    menuhandle  =  toolbx  (NEWMENU,menuid,menutitle)
    call  toolbx  (GETITEMMARK,menuhandle,item,markchar)
```

1.14.5 Miscellaneous Menu Utilities

The following routines perform special operations ranging from the way a menu bar flashes to retrieving information about the menus.

SETMENUFLASH causes the menu item under the given menu to flash when the mouse button is released. A value of 0 for count disables the flash. The standard Macintosh default is 2 (2 tenths of a second). Anything greater than 3 is not recommended due to speed considerations.

```
    integer*4  menuhandle
    integer*4  count
    integer*4  menuid
    data  menuid  /1/
    data  count  /2/
    menuhandle  =  toolbx  (GETMHANDLE,menuid)
    call  toolbx  (SETMENUFLASH,menuhandle,count)
```

CALCMENUSIZE calculates and sets the horizontal and vertical size fields in the given menu record.

```
integer*4  menuhandle
integer*4  menuid
data  menuid  /1/
menuhandle  =  toolbx  (GETMHANDLE,menuid)
call  toolbx  (CALCMENUSIZE,menuhandle)
```

COUNTMITEMS is an INTEGER function that returns the number of menu items under the given menu.

```
integer*4  menuhandle
integer*4  itemcount
integer*4  menuid
data  menuid  /1/
menuhandle  =  toolbx  (GETMHANDLE,menuid)
itemcount  =  toolbx  (COUNTMITEMS,menuhandle)
```

GETMHANDLE is an INTEGER*4 function that returns the pointer to the absolute address of the menu that is assigned the given menu ID. If the menu is not currently installed in the menu list, then GETMHANDLE returns a 0.

```
integer*4  menuid
integer*4  menuhandle
data  menuid  /3/
menuhandle  =  toolbx  (GETMHANDLE,menuid)
```

FLASHMENUBAR inverts the entire menu bar, if menu ID is 0 or the ID of any menu in the menu list. Otherwise, it inverts only the title of the given menu bar.

```
integer*4  menuid
data  menuid  /3/
call  toolbx  (FLASHMENUBAR,menuid)
```

Microsoft® FORTRAN Compiler
for the Apple® Macintosh™

Event Manager

2.0 About This Chapter

This chapter describes the Event Manager, the part of the Macintosh User Interface Toolbox that allows your program to monitor the user's actions with the mouse, keyboard, and keypad. The Event Manager is also used for various purposes within the Toolbox itself, such as to coordinate the ordering and display of windows on the screen. Finally, you can use the Event Manager as a means of communication between parts of your own program.

Actually, there are two Event Managers: one in the Operating System and one in the Toolbox. The Toolbox Event Manager calls the one in the Operating System and serves as an interface between it and your application program; it also adds some features that aren't present at the Operating System level, such as the window management facilities mentioned above. This chapter describes the Toolbox Event Manager, which is ordinarily the one your program will be dealing with. All references to "the Event Manager" should be understood to refer to the Toolbox Event Manager. For information on the Operating System's Event Manager, see the Macintosh Operating System Reference Manual in Inside Macintosh.

The chapter begins with an introduction to the Event Manager and what you can do with it. It then discusses the various types of events, their relative priority, and, in particular, how the user's keyboard actions are reported in the form of events. Next are sections on the structure of event records which contain all the pertinent information about each event, and event masks, which some of the Event Manager routines expect as parameters.

A section on using the Event Manager introduces its routines and tells how they fit into the flow of your application program. This is followed by detailed descriptions of all Event Manager procedures and functions, their parameters, calling protocol, effects, side effects, and so on.

Finally, there is an appendix containing information on the standard Macintosh character set and keyboard configuration.

2.1 About the Event Manager

The Macintosh Event Manager is your program's link to its user. Whenever the user presses the mouse button, types on the keyboard or keypad, or inserts a disk in a disk drive, your program is notified by means of an event. A typical Macintosh application program is event-driven: it decides what to do from moment to moment by asking the Event Manager for events and responding to them one by one, in whatever way is appropriate.

Although the Event Manager's primary purpose is to monitor the user's actions and pass them to your program in an orderly way, it also serves as a convenient mechanism for sending signals from one part of a program to another. For instance, the Window Manager uses events to coordinate the ordering and display of windows as the user activates and deactivates them and moves them around on the Macintosh screen. You can also define your own types of events and use them in any way your application calls for.

Events waiting to be processed are kept in the event queue. In principle, the event queue is a FIFO (first-in-first-out) list: events are added to the queue (posted) at one end and retrieved from the other. You can think of the queue as a funnel that collects events from a variety of sources and feeds them to your program on demand, in the order they occurred. (There are a few exceptions to the strict FIFO ordering, which will be discussed later.)

Note: The event queue has a limited capacity (currently 20 events, but this may change). When the queue becomes full, the Event Manager begins throwing out old events to make room for new ones as they are posted. The event thrown out is always the oldest one in the queue.

Using the Event Manager, your program can:

- Retrieve events one at a time from the event queue
- Control which types of events get posted and which are ignored
- Post events of its own (only using the Operating System Event Manager, documented in Inside Macintosh.)
- Read the current state of the keyboard, keypad, and mouse button
- Monitor the location of the mouse
- Read the system clock to find out how much time has elapsed since the system was last started up

Another service provided by the Event Manager is journaling. This feature enables your program to record all its interactions with the Event Manager and play them back later (see the Event Manager Manual in Inside Macintosh for details).

2.2 Event Types

Events are of various types, depending on their origin and meaning. Some report actions by the user, some are generated by the Window Manager, and some may be generated by your program itself for its own purposes. Some events are handled by the Desk Manager before your program ever sees them; others are left for your program to handle in its own way.

The most important event types are those that record user actions:

- Mouse down and mouse up events occur when the user presses or releases the mouse button.
- Key down and key up events occur when the user presses or releases a key on the keyboard or keypad. The Event Manager also automatically generates auto-key events when the user presses and holds down a repeating key. Together, these three event types are called keyboard events.
- Disk inserted events occur when the user inserts a disk into a disk drive.

Note: Mere movements of the mouse are not reported as events. If necessary, your program can keep track of them by periodically asking the Event Manager for the current location of the mouse.

The following event types are used by the Window Manager to coordinate the display of windows on the screen:

- Activate events are generated whenever an inactive window becomes active or vice versa. They generally occur in pairs (that is, one window is deactivated and another activated at the same time).
- Update events occur when a window's contents need to be redrawn, usually as a result of the user's opening, closing, activating, or moving a window.

Another event type (device driver event) may be generated by device drivers in certain situations; for example, a driver might be set up to report an event when its transmission of data is interrupted. The documentation for the individual drivers (in Inside Macintosh) will tell you about any specific device driver events that may occur.

A network event may be generated by the AppleBus Manager. Documentation for this event type is not yet available.

In addition, your application can define as many as four event types of its own and use them for any desired purpose.

Note: You place application-defined events in the event queue with the Operating System Event Manager procedure POSTEVENT.

See the Operating System Event Manager manual in Inside Macintosh for details.

One final type of event is the null event, which is what the Event Manager returns if it has no other events to report.

2.3 Priority of Events

It was stated earlier that in principle the event queue is a FIFO list; events are retrieved from the queue in the order they were posted. Actually, the way in which various types of events are generated and detected causes some to have higher priority than others. Furthermore, when you ask the Event Manager for an event, you can specify a particular type or types that are of interest. This can also alter the strict FIFO order, by causing some events to be passed over in favor of others that were actually posted later. Everything said in the following discussion is understood to be limited to the event types you've specifically requested in your Event Manager call.

The Event Manager always returns the highest-priority event available of the requested type(s). The priority ranking is as follows:

1. Activate (window becoming inactive before window becoming active)

2. Mouse down, mouse up, key down, key up, disk inserted, network, I/O driver, application-defined (all in FIFO order)

3. Auto-key

4. Update (in front-to-back order)

5. Null

Activate events take priority over all others; they are detected in a special way, and are never actually placed in the event queue. The Event Manager checks for pending activate events before looking in the event queue, so it will always return such an event if one is available. Because of the special way activate events are detected, there can never be more than two such events pending at the same time: one for a window becoming inactive and another for a window becoming active. If there's one of each, the event for the window becoming inactive is reported first.

Category 2 includes most of the possible event types. Within this category, events are normally retrieved from the queue in the order they were posted.

If no event is available in categories 1 and 2, the Event Manager next checks to see whether the appropriate conditions hold for an auto-key event. (These conditions are described in detail in the next section.) If so, it generates one and returns it to your program.

Next in priority are update events. Like activate events, these are not placed in the event queue, but are detected in another way. If no higher-priority event is available, the Event Manager checks for windows whose contents need to be redrawn. If it finds one, it generates and returns an update event for that window. Windows are checked in the order in which they're displayed on the screen, from front to back, so if two or more windows need to be updated, an update event will be generated for the front-most such window.

Finally, if no other event is available, the Event Manager returns a null event.

2.4 Keyboard Events

Every key on the Macintosh keyboard and the optional keypad generates key down and key up events when pressed and released. (Exceptions are the modifier keys--Shift, Caps Lock, Command, and Option. These keys are treated specially, as described below, and generate no keyboard events of their own.) In addition, the Event Manager itself generates auto-key events whenever you request an event and all of the following conditions apply:

- No higher-priority event of the requested type(s) is available
- The user is currently holding down a key other than a modifier key
- The appropriate time interval (see below) has elapsed since the last keyboard event
- Auto-key events are one of the types you've requested
- Auto-key events are one of the types currently being posted into the event queue

Two different time intervals are taken into account. Auto-key events begin to be generated after a certain initial delay has elapsed since the original key down event (that is, since the key was originally pressed); this is called the auto-key threshold. Thereafter, they are generated each time a certain repeat interval has elapsed since the last auto-key event; this is called the auto-key rate. The initial settings for these two intervals are 16 ticks (sixtieths of a second) for the initial delay and 4 ticks for the repeat interval. The user can adjust these settings to individual preference with the Control Panel desk accessory, by adjusting the keyboard touch and the rate of repeating keys.

Note: The current values for the auto-key threshold and rate are stored in system global variables located at absolute memory locations z'18E' and z'190', respectively. The following statements could be used if you need these values:

```
integer*2  keythresh                   ! Auto-key threshold
integer*2  keyrepthresh                ! Auto-key rate

keythresh = word(z'18E')
keyrepthresh = word(z'190')
```

When the user presses, holds down, or releases a key, the resulting keyboard event identifies the key in two different ways: with a key code designating the key itself and a character code designating the character the key stands for. Character codes are given in the extended version of ASCII (the American Standard Code for Information Interchange) used by Macintosh and Lisa; see the Appendix for further information.

The association between keys and characters is defined by a keyboard configuration. The particular character a key generates depends on three things:

- The key itself
- The keyboard configuration currently in effect
- Which, if any, of the modifier keys were held down when the key was pressed

As mentioned earlier, the modifier keys don't generate keyboard events of their own. Instead, they modify the meaning of the other keys by changing the character codes that those keys generate. For example, under the standard Macintosh keyboard configuration, the "C" key generates a lowercase letter c when pressed by itself; when pressed with the Shift or Caps Lock key down, it generates a capital C; with the Option key down, a lowercase c with the cedilla (ç), used in French, Portuguese, and a few other foreign languages; and with the Option and Shift or Option and Caps Lock keys down, a capital C with a cedilla (Ç). The state of each of the option keys is also reported in a field of the event record (see next section), where your program can examine it directly.

The standard keyboard configuration gives each key its normal ASCII character code according to the standard Macintosh keyboard layout. When the Option key is held down, most keys generate special characters with codes between 128 and 255 ($80 and $FF), included in the extended character set for business, scientific, and international use.

Note: Under the standard keyboard configuration only the Shift, Caps Lock, and Option keys actually modify the character a key stands for: the Command key has no effect on the character code generated. (Keyboard configurations other than the standard may take the Command key into account.) Similarly, character codes for the keypad are affected only by the Shift key. To find out whether the Command key was down at the time of an event (or Caps Lock or Option in the case of one generated from the keypad), you have to examine the appropriate field of the event record.

Normally you'll just want to use the standard keyboard configuration, which is read from the system resource file every time the Macintosh is started up. Other keyboard configurations can be used to reconfigure the keyboard for foreign use or for nonstandard layouts such as the Dvorak arrangement. In rare cases, you may want to define your own keyboard configuration to suit your program's special needs. For information on how to install an alternate keyboard configuration or define one of your own, see Inside Macintosh (this documentation is currently unavailable).

2.5 Event Records

Every event is represented internally by an event record containing all pertinent information about that event. The event record includes the following information:

- The type of event
- The time the event was posted (in ticks since system startup)
- The location of the mouse at the time the event was posted (in global coordinates)
- The state of the mouse button and modifier keys at the time the event was posted
- Any additional information required for a particular type of event, such as which key the user pressed or which window is being activated

This information is filled into the event record for every event--even for null events, which just means that nothing special has happened.

An event record can be defined in the Microsoft FORTRAN Compiler as follows:

```
integer*2  eventrecord(8)
integer*2  what
```

```
    integer*4  message
    integer*4  when
    integer*2  where(2)
    integer*2  modifiers

    equivalence  (eventrecord(1),what)
    equivalence  (eventrecord(2),message)
    equivalence  (eventrecord(4),when)
    equivalence  (eventrecord(6),where(1))
    equivalence  (eventrecord(8),modifiers)
```

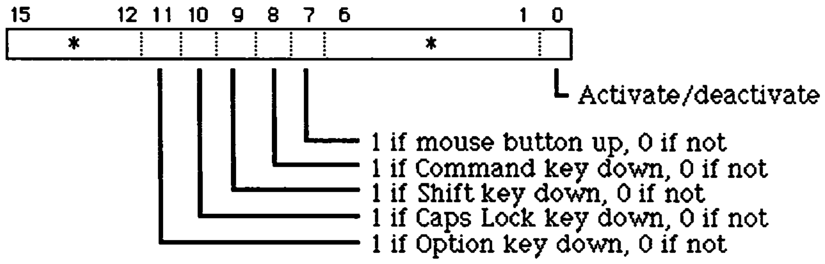
This definition is assumed in the examples given with the description of individual Event Manager procedures and functions below. If more than one event record is necessary, different field names will be required for each one, since FORTRAN lacks true records. However, one record definition will usually suffice in a Macintosh application; generally your program will not ask for another event record until it has responded to the current one.

The what field contains an event code identifying the type of the event. The Event Manager can handle a maximum of 16 different event types, denoted by event codes from 0 to 15. The standard event codes built into the Event Manager are as follows:

```
        nullEvent      = 0      ! null
        mouseDown      = 1      ! mouse down
        mouseUp        = 2      ! mouse up
        keyDown        = 3      ! key down
        keyUp          = 4      ! key up
        autoKey        = 5      ! auto-key
        updateEvt      = 6      ! update
        diskEvt        = 7      ! disk inserted
        activateEvt    = 8      ! activate
        networkEvt     = 10     ! network
        driverEvt      = 11     ! I/O driver
        app1Evt        = 12     ! application-defined
        app2Evt        = 13     ! application-defined
        app3Evt        = 14     ! application-defined
        app4Evt        = 15     ! application-defined
```

The when field contains the time the event was posted, in ticks (sixtieths of a second) since the system was last started up.

The where field gives the location of the mouse at the time the event was posted, expressed in global coordinates.


15
12 11 10 9 8 7 6
1 0
*
*
Activate/deactivate
1 if mouse button up, 0 if not
1 if Command key down, 0 if not
1 if Shift key down, 0 if not
1 if Caps Lock key down, 0 if not
1 if Option key down, 0 if not

* reserved for future use

Figure 1. Modifier Bits

The modifiers field gives the state of the mouse button and the modifier keys at the time the event was posted, as shown below and in Figure 1. (Following the customary convention, the bit positions are numbered from the right to left, starting from 0 at the low order end; see Figure 1.)

Bit	Meaning
15-12	Reserved
11	1 if option key down, 0 if up
10	1 if caps Lock key down, 0 if up
9	1 if shift key, 0 if not
8	1 if command key down, 0 if not
7	1 if mouse button up, 0 if not
6-2	Reserved
1-0	Used only by activate events (see below)

For activate events, the low-order bit of the modifiers field (bit 0) is set to 1 if a window is being activated, or to 0 if it is being deactivated. The remaining bits indicate the state of the mouse button and modifier keys. Notice that the mouse button bit is set if the mouse button is **up**, whereas the bits for the four modifier keys are set if their corresponding keys are **down**.

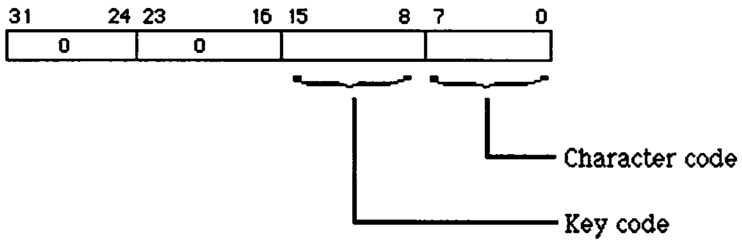


Figure 2. Event Message Format for Keyboard Events

The message field contains the event message, which conveys extra information specific to a particular event type:

- For **keyboard** events, the event message identifies the key that was pressed or released, as shown in Figure 2. The low-order byte (message .AND. 255) contains the character code for the key, depending on the keyboard configuration currently in effect and on which, if any, of the modifier keys were held down. Under the standard keyboard configuration this is just the normal ASCII code associated with the key, which is usually the information your program needs. The third byte (message / 256) gives the key code, useful in special cases (a music generator, for example) where you want to treat the keyboard as a set of buttons unrelated to specific characters. Detailed information on key codes for the standard Macintosh keyboard configuration is given in the Appendix. The first two bytes of the message are set to 0.

- For **disk inserted** events, the event message gives the drive number of the disk drive: 1 for the Macintosh's built-in drive, 2 for the external drive, if any. Numbers greater than 2 denote additional disk drives connected through the serial port. By the time your program receives a disk inserted event, the system will already have attempted to mount the volume that was inserted. If for any reason the attempt was unsuccessful (the user inserted an unformatted disk, for example), the high-order word of the event message will contain the error code returned by the Operating System;

see the Operating System Manual in Inside Macintosh for further details.

- For **activate** and **update** events, the event message is a pointer to the window affected.

- For **application-defined** event types, you can use the event message for whatever information your application calls for.

- For **mouse down, mouse up**, and **null** events, the event message is meaningless and should be ignored. For network and device driver events, the contents of the event message depend on the situation under which the event was generated; the documentation describing those situations will give the details.

2.6 Event Masks

Several of the Event Manager routines can be restricted to a specific event type or group of types. For instance, instead of just requesting the next available event, you can ask specifically for the next keyboard event.

You specify which event types a particular Event Manager call applies to by supplying an event mask as a parameter. This is an integer in which each of the least significant 16 bit positions stands for an event type, as shown in Figure 3. Notice that the bit position representing a given type corresponds to the event code for that type. For example, update events (type code 6) are specified by bit 6 of the mask, counting from 0 at the right (low-order) end. A 1 bit at the position means that this Event Manager call applies to update events; a 0 means it doesn't.

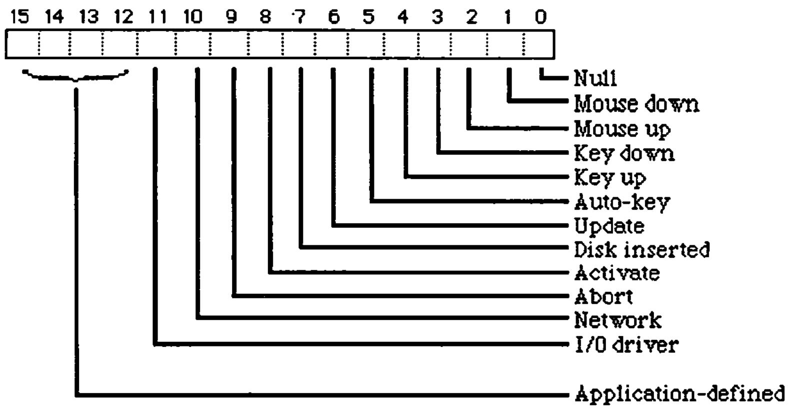


Figure 3. Event Mask

Masks for each single event type are as follows:

```
nullMask      = 1       ! null
mDownMask     = 2       ! mouse down
mUpMask       = 4       ! mouse up
keyDownMask   = 8       ! key down
keyUpMask     = 16      ! key up
autoKeyMask   = 32      ! auto-key
updateMask    = 64      ! update
diskMask      = 128     ! disk inserted
activMask     = 256     ! activate
networkMask   = 1024    ! network
driverMask    = 2048    ! device driver
app1Mask      = 4096    ! application-defined
app2Mask      = 8192    ! application-defined
app3Mask      = 16384   ! application-defined
app4Mask      = 32768   ! application-defined
```

Note

Null events can't be disabled; a null event will always be reported when none of the enabled types of events is available.

To request all events, you can use an event mask of -1.

You can form any mask you need by combining these mask constraints with logical operations. For example, to specify any keyboard event, you can use a mask of

```
8 .OR. 16 .OR. 32      ! keyDownMask + keyUpMask + autoKeyMask
```

For any event except an update, you can use

```
    .NOT. 64            ! everyevent - updateMask
```

Caution

Recommended programming practice is always to use an event mask of -1 unless there is a specific reason not to. This ensures that all events will be processed in their natural order.

In addition to the mask parameters to individual Event Manager routines, there's also a global system event mask, which controls which event types get posted into the event queue. Only those events corresponding to "1" bits in the system event mask are posted; those with "0" bits are ignored. When the system is started up, the system event mask is initially set to post all except key up events--that is, it is initialized to

```
    .NOT. keyUpMask
```

(Key up events are meaningless for most applications, and your program will usually want to ignore them anyway.) The system event mask is a Macintosh system global in low memory, and resides at location z'144'. If necessary for your particular application, your can change the setting of the system event mask with the Microsoft FORTRAN Compiler WORD operator:

```
    * Post all but mouse up and key up events.
    word(z'144') = .NOT. (4 .OR. 16)
```

2.7 Using the Event Manager

This section discusses how the Event Manager routines fit into the general flow of your program and gives you an idea of which routines you'll need to use. The routines themselves are described in detail in the next section.

As noted earlier, most application programs are event-driven. Such programs typically have a main loop that repeatedly calls GETNEXTEVENT to retrieve the next available event, then uses a SELECT CASE statement to decide what type of event it is and take whatever action is appropriate. Two Microsoft FORTRAN programs which demonstrate this structure are included in the source code folder: Demo.for and Edit.for. Note that a CASE statement does not have to be provided for all events that you expect to receive; you can put an empty CASE DEFAULT statement in the SELECT CASE instead. This can be useful during program development to implement and test events a few at a time.

Your program is only expected to respond to those events that are directly related to its own operations. Events that are of interest only to the system, or that pertain only to system windows, are intercepted and handled by the Desk Manager, but are still reported back to your program by GETNEXTEVENT. After calling GETNEXTEVENT, you.should test its LOGICAL result to find out whether your program needs to respond to the event: .TRUE. means the event is of interest to your program, .FALSE. means you can ignore it.

Note: Events handled by the system include activate and update events for system windows; all keyboard and mouse up events when a system window is active, if the window contains a desk accessory that is prepared to handle the event; and network events if there's a disk accessory present that will handle them. Further details are given in the Desk Manager Manual in Inside Macintosh.

2.8 Responding to Mouse Events

On receiving a mouse down event, you should first call the Window Manager function FINDWINDOW to find out where on the screen the mouse button was pressed; you can then respond in whatever way is appropriate. Depending on the part of the screen the button was pressed in, this may involve calls to Toolbox routines such as the Menu Manager function MENUSELECT, the Disk Manager procedure SYSTEMCLICK, the Window Manager routines SELECTWINDOW, DRAGWINDOW, GORWWINDOW, and TRACKGOAWAY, and the Control Manager routines FINDCONTROL, TRACKCONTROL, and DRAGCONTROL. See the relevant Toolbox manuals for details (Toolbox manuals pertaining to the Desk and Control managers can be found in Inside Macintosh).

If your application attaches some special significance to pressing a modifier key along with the mouse button, you can discover the state of that modifier key while the mouse button is down by examining the appropriate flag in the modifiers field.

If you're using the TextEdit part of the Toolbox to handle text editing, mouse double-clicks will work automatically as a means of selecting a word; to respond to double-clicks in any other context, however, you'll have to detect them yourself. You can do so by comparing the time and location of a mouse-up event with those of the immediately following mouse-down event. You should assume a double-click has occurred if both of the following are true:

- The times of the mouse-up event and the mouse-down event differ by a number of ticks less than or equal to the value in the system global at absolute location z'2F0':

  ```
  integer doubletime
  doubletime = long(z'2F0')
  ```

- The locations of the two mouse-down events separated by the mouse-up event are sufficiently close to each other. Exactly what this means depends on the particular application. For instance, in a word-processing application, you might consider the two locations essentially the same if they fall on the same character, whereas in a graphics application you might consider them essentially the same if the sum of the horizontal and vertical changes in position is no more than five pixels.

Mouse-up events may be significant in other ways; for example, they might signal the end of dragging in a graphics or spreadsheet application. Many simple applications, however, will ignore mouse-up events.

2.9 Responding to Keyboard Events

When one of your own windows is active, you should respond to the keyboard in whatever way your application calls for. For example, when the user types a character on the keyboard, you might want to insert that character into the document displayed in an active document window. For keyboard events, you should first check the modifiers field to see whether the character was typed with the Command key held down: if so, the user may have been choosing a menu item by typing its keyboard equivalent. To find out, pass the character that was typed to the Menu Manager function MENUKEY. If that character, combined with the Command key, stands for a menu item, MENUKEY will return a nonzero result identifying the item. You can then do whatever is appropriate to respond to that menu item, just as if the user had chosen it with the mouse. If MENUKEY's result is 0, the user has typed a key combination that has no menu equivalent; your program may then want to respond in some other way.

Usually your application can handle auto-key events the same as key-down events. You may, however, want to ignore auto-key events that invoke commands that shouldn't be continually repeated.

Note: Remember that most applications will want to ignore key-up events; with the standard system event mask you won't get any.

If you wish to periodically inspect the state of the keyboard or keypad--say, while the mouse button is being held down--use the procedure GETKEYS; this procedure is also the only way to tell whether a modifier key is being pressed alone.

2.10 Responding to Activate and Update Events

When you receive an activate event for one of your own windows, the Window Manager will already have done all of the normal "housekeeping" associated with the event, such as highlighting or unhighlighting the window. You can then take any further action that your application may require, such as showing or hiding a scroll bar or highlighting or unhighlighting a selection.

On receiving an update event for one of your own windows, you should usually call the Window Manager procedure BeginUpdate, redraw the window's contents, then call EndUpdate.

2.11 Responding to Disk-Inserted Events

When you receive a disk inserted event, the Disk Manager will already have responded to the event by attempting to mount the new volume just inserted in the disk drive. Usually there's nothing more for your program to do, but GETNEXTEVENT returns .TRUE. anyway, giving you an opportunity to take some further action if your application demands it. If the attempt to mount the volume was unsuccessful, there will be a nonzero error code in the high-order word of the event message; in this case you might want to take some special action, such as displaying an alert box containing an error message.

2.12 Other Operations

If you're using your own event types for internal communication between parts of your program, you can use POSTEVENT to post them into the event queue. (POSTEVENT is part of the Operating System Event Manager, and is documented in Inside Macintosh.) When you receive them back from GETNEXTEVENT, you can respond in whatever way is appropriate for your application.

To "peek" at pending events without removing them from the event queue, use EVENTAVAIL instead of GETNEXTEVENT. To control which event types get posted into the queue, or to cause certain types to be ignored, set the system event mask; a word at absolute location z'144' in memory (see 'Event Masks', above).

In addition to receiving the user's mouse and keyboard actions in the form of events, you can directly read the keyboard (and keypad), mouse location, and state of the mouse button by calling GETKEYS, GETMOUSE, and BUTTON, respectively. To follow the mouse when the user drags it with the button down, use STILLDOWN or WAITMOUSEUP.

The function **TICKCOUNT** returns the number of ticks since the last system startup; you might, for example, compare this value to the when field of an event record to discover the delay since that event was posted.

Finally, the system global at absolute memory location z'**2F4**' contains the number of ticks between blinks of the "caret" (usually a vertical bar) marking the insertion point in editable text. If you aren't using TextEdit and therefore need to cause the caret to blink yourself, you can get this value with the following FORTRAN statements:

```
    integer  carettime
    carettime  =  long(z'2F4')
```

You would check this value each time through your program's main event loop, to ensure a constant frequency of blinking.

2.13 Event Manager Routines

This section describes all the Event Manager procedures and functions. They are presented as calls to toolbx.sub, an external subroutine supplied with the Microsoft FORTRAN Compiler. This subroutine must reside on the same disk as your application when it is run, or must have been linked into your application with **link**. This subroutine can be invoked by using a FORTRAN CALL statement or by using it as a function in a FORTRAN expression. It should be declared INTEGER in every program unit that uses it as a function, even if its value is to be assigned to LOGICAL variables.

2.14 Accessing Events

GETNEXTEVENT is a LOGICAL function whose primary purpose is to notify the application what the next event is. After reporting on an event, the event is deleted from the event queue. The current event is returned in eventrecord. If the system intercepts the event, or the event being reported is a null event, then GETNEXTEVENT will return .FALSE. as its value. The events that will be intercepted by the system include activate and update events directed at the system window, keyboard and mouse up events in the system window, and all network events if there is a desk accessory present to handle them. If the system deems that the application should take care of the current event, then GETNEXTEVENT returns .TRUE. The following example assumes the record definition given under the Event Records section (above).

```
    integer  eventmask
    logical  eventflag

    eventmask  =  -1                    ! Get all event types.
    eventflag  =  toolbx(GETNEXTEVENT,  eventmask,  eventrecord)
```

EVENTAVAIL is identical to GETNEXTEVENT above, except that it does not remove the event that it returns from the event queue. This allows you to "peek" at pending events while still leaving them in the queue for later processing.

GETMOUSE loads the current mouse position into mouseloc. The location is given in local coordinates. Notice that this differs from the mouse location stored in the where field of an event record; that location is always in global coordinates. Mouseloc is returned in the form of the vertical value in the first element, and the horizontal value in the second.

```
    integer*2  mouseloc(2)
    call  toolbx  (GETMOUSE,mouseloc)
```

BUTTON is a LOGICAL function that returns .TRUE. if the mouse button is down, and .FALSE. otherwise.

```
    logical*4  buttonflag
    buttonflag  =  toolbx  (BUTTON)
```

STILLDOWN is a LOGICAL function that is to be called after a mouse event has been detected. STILLDOWN returns .TRUE. if the button is still down and no other mouse events are in the event queue.

```
    logical*4  buttonflag
    buttonflag  =  toolbx  (STILLDOWN)
```

WAITMOUSEUP is a LOGICAL function that operates exactly the same as STILLDOWN (above), except that if the button is not still down from the original press, WAITMOUSEUP removes the preceding mouse-up event before returning .FALSE. If, for instance, your application attaches some special significance to mouse double-clicks and to mouse-up events, this function would allow your application to recognize a double-click without being confused by the intervening mouse-up.

```
    logical*4  buttonflag
    buttonflag  =  toolbx  (WAITMOUSEUP)
```

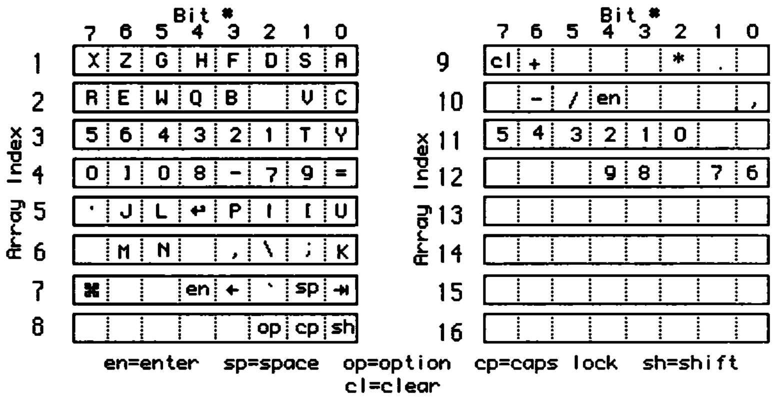
GETKEYS reads the current state of the keyboard (and keypad, if any) and returns a 128 bit map indicating the status of each key. A 16 element INTEGER*1 array can be used to hold the bitmap. Each byte of this array corresponds to 8 keys, 1 bit per key, as shown in Figure 4. If a bit is 1, the corresponding key is down; if it is 0, that key is up. The maximum number of keys that can be down simultaneously is two character keys plus any combination of the four modifier keys. For example, the following code will wait until the Shift key is pressed:

```
      integer*1  keymap(16)
      logical  shiftdown
      integer*2  syseventmask

*  Disable posting of all events.
      syseventmask  =  0                    !  No events.
      word(z'144')  =  syseventmask
```

```
* Wait for the shift key to be pressed.
     do
         call toolbx(GETKEYS, keymap)
         shiftdown = keymap(8) .AND. z'1'     ! Least significant bit.
         if (shiftdown) exit
     repeat

* Turn event posting back on.
     syseventmask = .NOT. 16                 ! Post all but key up events
     word(z'144') = syseventmask
```



Array Index	Bit # 7	6	5	4	3	2	1	0
1	X	Z	G	H	F	D	S	A
2	R	E	W	Q	B		V	C
3	5	6	4	3	2	1	T	Y
4	O	]	0	8	-	7	9	=
5	'	J	L	↵	P	I	[	U
6		M	N		,	\	;	K
7	⌘			en	←	`	sp	⇥
8						op	cp	sh

Array Index	Bit # 7	6	5	4	3	2	1	0
9	cl	+				*	.	
10		-	/	en				,
11	5	4	3	2	1	0		
12				9	8		7	6
13								
14								
15								
16								

en=enter sp=space op=option cp=caps lock sh=shift
cl=clear

Figure 4. Correspondence between keymap array and keys. Elements 1-8 correspond to the keyboard, 9-16 to the keypad.

TICKCOUNT is an INTEGER*4 function that returns the current tick count from the system clock. This value represents the elapsed time since the Macintosh was turned on. The value returned is in sixtieths (1/60) of a second.

```
integer*4 ticks
ticks = toolbx (TICKCOUNT)
```

This value is also available in a system global at absolute memory location z'16A'. This operation is somewhat faster.

```
ticks = long(z'16A')
```

2.15 Appendix: Standard Key and Character Codes

The following describes the key and character codes used by the Macintosh and the characters assigned to keys on the keyboard and keypad under the standard Macintosh keyboard configuration. All key and character codes are given in hexadecimal.

Character codes between z'20' and z'7E' have their normal ASCII meanings (see Appendix L of the Microsoft FORTRAN Compiler language manual). Codes between z'80' and z'CA' denote special characters included in the extended character set for business, scientific, and international use; codes from z'CB' to z'FF' are unassigned. Most of the control characters have no special meaning on Macintosh and cannot be generated from the Macintosh keyboard under the standard keyboard configuration. The exceptions are the following:

Code	Character	Key
z'03'	ETX	Enter (keyboard and keypad)
z'08'	BS	Backspace
z'09'	HT	Tab
z'0D'	CR	Return
z'1B'	ESC	Clear (keypad)
z'1C'	FS	Left arrow (keypad)
z'1D'	GS	Right arrow(keypad)
z'1E'	RS	Up arrow (keypad)
z'1F'	US	Down Arrow (keypad)
z'20'	SP	Space bar

In addition, as show in the table, codes from z'11' to z'15' denote special characters used on the Macintosh screen, such as the open and solid Apple characters. These characters are intended exclusively for use on the screen, and have no keyboard or keypad equivalents under the standard keyboard configuration.

Under the standard keyboard configuration, characters with accents or diacritical marks cannot be typed directly from the keyboard. Instead, they are generated by first typing the accent or diacritical mark alone, followed by the letter to be accented. For example, a lowercase letter e with the grave accent (è, character code $8F) is produced by typing a grave accent (`, code z'60') followed by a lowercase e (code z'65'). The Macintosh keyboard driver will translate such two-character sequences involving diacriticals into the corresponding single accented letters.

Figure 5 shows the hexadecimal key codes corresponding to keys on the Macintosh keyboard and keypad, respectively. Modifier keys are not shown, since they never generate keyboard events of their own.

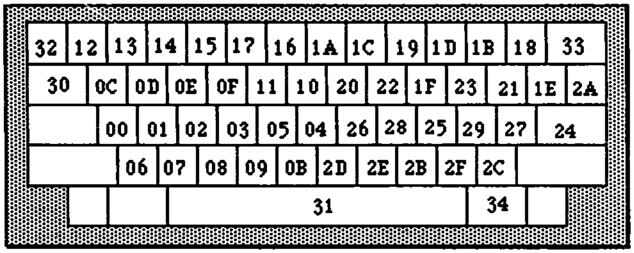


Figure 5. Hexadecimal key codes for the Macintosh keyboard.

Microsoft® FORTRAN Compiler
for the Apple® Macintosh™

Desk Manager

The Desk Manager

This chapter describes the Desk Manager, the part of the Macintosh User Interface Toolbox that supports the use of desk accessories from an application; the Calculator, for example, is a desk accessory. In particular, it tells you how to make the standard desk accessories available in your application.

3.1 About the Desk Manager

The Desk Manager enables your application to support desk accessories, which are "mini-accessories" that can be run at the same time as a Macintosh application. There are a number of standard desk accessories, such as the Calculator shown in Figure 1.

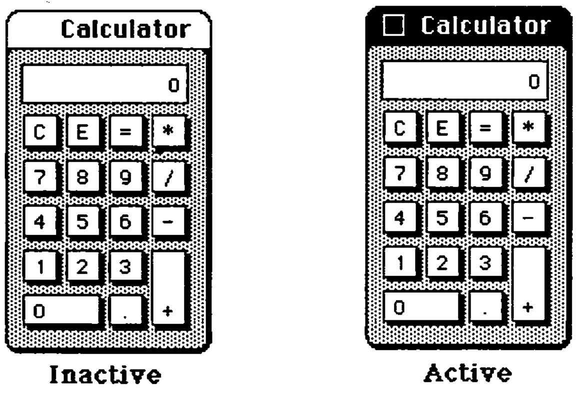


Figure 1. The Calculator Desk Accessory.

The Macintosh user opens desk accessories by choosing them from the standard Apple menu (whose title is an apple symbol), which by convention is the first menu in the menu bar. When a desk accessory is chosen from the menu, it's usually displayed in a window on the desktop, and that window becomes the active window.

After being selected, the accessory may be used as long as it's active. The user can activate other windows and then reactivate the desk accessory by clicking inside it. Whenever a standard desk accessory is active, it has a close box in its title bar. Clicking the close box makes the accessory disappear, and the window that's then front most becomes active.

The window associated with a desk accessory is usually a round-corner window (as shown in Figure 1) or a standard document window, although it can be any type of window. It may even look and behave like a dialog window; the accessory can call on the Dialog Manager to create the window and then use Dialog Manager routines to operate on it. In any case, the window will be a system window, as indicated by the fact that its windowKind field contains a negative value.

The Desk Manager provides a mechanism that lets standard commands chosen from the Edit menu be applied to a desk accessory when it's active. Even if the commands aren't particularly useful for editing within the accessory, they may be useful for cutting and pasting between the accessory and the application or even another accessory. For example, the result of a calculation made with the Calculator can be copied into a document prepared in MacWrite.

3.2 Using the Desk Manager

To allow access to desk accessories, your application must do the following:

- Initialize TextEdit and the Dialog Manager, in case any desk accessories are displayed in windows created by the Dialog Manager (which uses TextEdit).

- Set up the Apple menu as the first menu in the menu bar. You can put the names of all currently available desk accessories in a menu by using the Menu Manager procedure ADDRESMENU (see the Menu Manager manual for details).

- Set up an Edit menu that includes the standard commands Undo, Cut, Copy, Paste, and Clear (in that order, with a gray line separating Undo and Cut), even if your application itself doesn't support any of these commands.

When a user chooses a desk accessory from the Apple menu, call the Menu Manager procedure GETITEM to get the name of the desk accessory, and then the Desk Manager function OPENDESKACC to open and display the accessory. When a system window is active and the user chooses Close from the File Menu, close the desk accessory with the CLOSEDESKACC procedure.

Warning

Most desk accessories allocate nonrelocatable objects (such as windows) on the heap, resulting in fragmentation of the heap area. Before beginning an operation that requires a large amount of memory, your application may want to close all open desk accessories.

When the Toolbox Event Manager function GETNEXTEVENT reports that a mouse-down event has occurred, your application should call the Window Manager function FINDWINDOW to find out where the mouse button was pressed. If FINDWINDOW returns 2 (in a system window), call the Desk Manager procedure SYSTEMCLICK. SYSTEMCLICK handles mouse-down events in system windows, routing them to desk accessories where appropriate.

Note: The application needn't be concerned with exactly which desk accessories are currently open.

When the active window changes from an application window to a system window, the application should disable any of its menus or menu items that don't apply while an accessory is active, and it should enable the standard editing commands Undo, Cut, Copy, Paste, and Clear, in the Edit menu. An application should disable any editing commands it doesn't support when one of its own windows becomes active.

When a mouse-down event occurs in the menu bar, and the application determines that one of the five standard editing commands has been invoked, it should call SYSTEMEDIT. Only if SYSTEMEDIT returns .FALSE. should the application process the editing command itself; if the active window belongs to a desk accessory, SYSTEMEDIT passes the editing command on to that accessory and returns .TRUE.

Keyboard equivalents of the standard editing commands are passed on to desk accessories by the Desk Manager, not by your application.

Warning

The standard keyboard equivalents for the commands in the Edit menu must not be changed or assigned to other commands; the Desk Manager automatically interprets Command-Z, X, C, and V as Undo, Cut, Copy, and Paste, respectively.

Certain periodic actions may be defined for desk accessories. To see that they're performed, you need to call the SYSTEMTASK procedure at least once every time through your main event loop.

3.3 Desk Manager Routines

This section describes the routines required to access desk accessories in your application. An example program is included at the end of this chapter demonstrating the use of each routine.

3.3.1 Opening and Closing Desk Accessories

OPENDESKACC is an integer function which takes the name of a desk accessory, opens that accessory, and returns its refnum. The name is a Pascal-style string (with the string size in the first byte) containing the accessory's resource name, which you get by calling the Menu Manager procedure GETITEM. You should ignore the value returned by this function; you will not need it for other desk accessory routines.

3.3.2 Handling Events in Desk Accessories

SYSTEMCLICK processes mouse-down events in system windows. When a mouse-down event occurs and the Window Manager function FINDWINDOW reports that the mouse button was pressed in a system window, the application should call SYSTEMCLICK with the event record and the window pointer. If the given window belongs to a desk accessory, SYSTEMCLICK will see that the event gets handled properly.

SYSTEMEDIT is a LOGICAL function used to process editing commands in desk accessories. When the user chooses one of the five standard editing commands from the edit menu, call SYSTEMEDIT with one of the following integers:

Editing command	Parameter
Undo	0
Cut	2
Copy	3
Paste	4
Clear	5

If your Edit menu contains these five in the standard arrangement (the order listed above, with a gray line separating Undo and Cut), you can simply call

```
toolbx(SYSTEMEDIT, menuitem - 1)
```

If the active window doesn't belong to a desk accessory, SYSTEMEDIT returns .FALSE.; the application should then process the editing command as usual. If the active window does belong to a desk accessory, SYSTEMEDIT asks that accessory to process the command and returns .TRUE.; in this case, the application should ignore the command.

Note: It's up to the application to make sure desk accessories get their editing commands that are chosen from the Edit menu. In particular, make sure your application hasn't disabled the Edit menu or any of the five standard commands when a desk accessory is activated.

3.3.3 Performing Periodic Actions

Some desk accessories have to perform tasks at certain time intervals. These might include communications accessories that have to check for new messages, scheduling accessories that have to remind the user of appointments, or utilities which have to check periodically for the status of peripheral devices (such as the printer). The Alarm Clock standard desk accessory updates its time display once a second; this update is defined as a periodic task for that accessory.

SYSTEMTASK allows the Desk Manager access to the processor so that it can check to see if any desk accessories are ready to perform their periodic task. You should call SYSTEMTASK as often as possible, usually once every time through your main event loop. Call it more than once if your application does an unusually large amount of processing each time through the loop.

Note: SYSTEMTASK should be called at least every sixtieth of a second.

3.4 An Example Program

This Microsoft FORTRAN program implements the standard Apple menu with the desk accessories available in the current system file. It is not useful as such, since this menu is available under the finder, but it illustrates the use of the Desk Manager calls described above. It is also serves to demonstrate event processing in an event loop; the basis for all interactive programs under the toolbox. This is described in more detail in the Event Manager Manual.

```
      program applemenu

      implicit none           ! enforce strong typing

* Toolbox definitions.
      include toolbx.par

      logical temp
      integer code

      integer*4 eventmask     ! specifies the events of interest
      integer*2 myevent(8)    ! overlying structure
      integer*2 what          ! type of event.
      integer*4 message       ! extra event information.
      integer*4 when          ! time of event in 60ths of seconds
      integer*2 where(2)      ! mouse location in global coordinates
      integer*2 modifiers     ! state of mouse button
                              !   and modifier keys.
      equivalence  (myevent(1),what)
      equivalence  (myevent(2),message)
      equivalence  (myevent(4),when)
      equivalence  (myevent(6),where(1))
      equivalence  (myevent(8),modifiers)

      integer mywindow, whichwindow
      integer refnum
      character*20 name

* Structure returned by MenuSelect - menu number in high word,
* item number in low word.
      integer*2 menuitem(2)
      integer*4 menusel
      equivalence (menuitem, menusel)
      integer item4            ! 4 byte version of item number.

* Declare the toolbox interface as an integer function.
      integer toolbx

      integer menuhandle(3)
      logical doneflag  ! Exit if true.
      logical editflag  ! Flag for SYSTEMEDIT.


*
*  Close Microsoft FORTRAN I/O window
* (never make a DISPOSEWINDOW call on this window):
*
      mywindow = toolbx(FRONTWINDOW)
      call toolbx(CLOSEWINDOW,mywindow)

      eventmask = -1           ! Every event.

* Most toolbox initialization required is done by runtime.  Only Text
* Edit needs to be initialized here.
      call toolbx(TEINIT)

* Set up the apple menu with the apple character for a title and add
* all available desk accessories to it.
      menuhandle(1) = toolbx(NEWMENU, 1, char(1) // char(z'14'))
      call toolbx(ADDRESMENU, menuhandle(1), 'DRVR')
      call toolbx(INSERTMENU, menuhandle(1), 0)

* Set up the file menu.
      menuhandle(2) = toolbx(NEWMENU, 2, char(4) // 'File')
      call toolbx(APPENDMENU, menuhandle(2), char(4) // 'Quit')
      call toolbx(INSERTMENU, menuhandle(2), 0)

* Set up the edit menu.
      menuhandle(3) = toolbx(NEWMENU, 3, char(4) // 'Edit')
      call toolbx(APPENDMENU, menuhandle(3),
     +         char(28) // 'Undo;(-;Cut;Copy;Paste;Clear')
      call toolbx(INSERTMENU, menuhandle(3), 0)
      call  toolbx(DRAWMENUBAR)

      doneflag = .false.

* Main event loop.  Get and process events  until doneflag
* is set to .true. by the Quit option under the File menu.
      do

* Allow desk accessories to do periodic processing.
         call  toolbx(SYSTEMTASK)

* Get the next event and process it.
         temp = toolbx(GETNEXTEVENT, eventmask, myevent)
         select case (what)
           case (1)            ! Mouse down.
              code = toolbx(FINDWINDOW, where, whichwindow)
             select case (code)
              case (1)         ! In menu bar
                 menusel = toolbx(MENUSELECT, where)
                 item4 = menuitem(2)! Convert to 4 bytes.
                 select case (menuitem(1))
                  case (1)     ! applemenu
                     call toolbx(GETITEM, menuhandle(1), item4, name)
                     refnum = toolbx(OPENDSKACC, name)
                  case (2)     ! file menu
                    doneflag = .true.
                  case (3)     ! edit menu
* Ignore the value of editflag;
* if no desk accessory processes the command we ignore it.
                     editflag = toolbx(SYSTEMEDIT, item4 - 1)
                  case default       ! Ignore any other menu.
                end select

* Unhilite the selected menu.
                 call toolbx(HILITEMENU, 0)
              case (2)         ! In system window
                  call toolbx(SYSTEMCLICK, myevent, whichwindow)
              case default     ! Ignore other window types.
             end select
           case default        ! Ignore all other events.
         end select
         if (doneflag) exit
      repeat
      end
```

Documentation Report Form

Please tell Microsoft what you think about the documentation that accompanies the software. Your comments and suggestions help us improve our products. Mail this questionnaire to:

MICROSOFT Corporation
10700 Northup Way
Box 97200
Bellevue, WA 98009
Attn: Languages User Education

Respond to any or all of the following questions. If you want to make additional comments, use the back of this page or a separate page.

Product Name: **Version Number:**

1. Did you find errors in the documentation? Please give the document title, page number, and a description of the error.

2. Which parts of the documentation do you consider most important? Do you have any suggestions for improving these parts?

3. Is it easy to find the information you need? Is anything missing?

4. Is the documentation clear and easy to read?

5. What type of user are you?
   _____ First-time programmer
   _____ First-time programmer in this language, but experienced in at least one other language
   _____ Experienced programmer

MICROSOFT®
10700 Northup Way, Bellevue, WA 98004

Software Problem Report

Name ______________________________

Street ______________________________

City ______________ State __________ Zip __________

Phone ______________________ Date __________

Instructions

Use this form to report software bugs, documentation errors, or suggested enhancements. Mail the form to Microsoft.

Category

_____ Software Problem	_____ Documentation Problem (Document #__________)
_____ Software Enhancement	_____ Other

Software Description

Microsoft Product ______________________________

Rev. __________ Registration # ______________________

Operating System ______________________________

Rev. __________ Supplier ______________________

Other Software Used ______________________________

Rev. __________ Supplier ______________________

Hardware Description

Manufacturer ______________ CPU __________ Memory ________ KB

Disk Size ________" Density: Sides:

Single _____ Single _____

Double _____ Double _____

Peripherals ______________________________

Problem Description

Describe the problem. (Also describe how to reproduce it, and your diagnosis and suggested correction.) Attach a listing if available.

Microsoft Use Only

Tech Support ______	Date Received ______
Routing Code ______	Date Resolved ______
Report Number ______	

Action Taken:

MICROSOFT®