Chapter Seventeen

THE BUILDING-BLOCKS OF A FORTRAN PROGRAM

Introduction

The structure of a Fortran program is no different than the structure of any other high-level language program. It is a mixture of non-executable (declaration) and executable statements which must occur in a specific order.

This chapter examines the basic components used to build Fortran programs, their order, and features.

Objectives

To build a basic Fortran program, a programmer should be able to:

- Specify the main program unit
- Build a set of specification statements and initialize the program
- Issue the necessary executable statements to solve a problem
- Clean up and terminate the program in an orderly manner

The Fortran Program Structure

Overview

A programmer solves a problem in Fortran by following a series of steps to reach the desired solution. While many of the steps are dictated by the programming language, some are dictated by the programmer's solution strategy. This section lays out the steps dictated by the language, allowing the programmer some flexibility in carrying out both sets of steps.

This section discusses:

- Program Statements
- Specification Statements
- Program Code Execution
- Subroutines and Functions
- Termination Statements

Statement Order

There are a number of top level statements that define the structure of the program. The two most obvious are the PROGRAM statement (which specifies the name of the Fortran program) and the END statement (which terminates execution of the program and turns control back to the operating system). The next sections discuss these and other program level statements and their order in a program or program unit.

Placement of Program Statements

In general, all specification and data declaration statements (except FORMAT statements) precede the executable statements in a program or program unit. Fortran also has some detailed rules concerning the sequence of placement of various kinds of declarations. Most of the rules are common sense rules - such as declaring a variable before initializing it and defining a variable's data type before equivalencing it to a variable of another data type.

The following is a suggested sequence of statement placement that will satisfy the requirements of Fortran's rules in most cases. This list applies to both main programs and other program units (unless specified). For completeness, this list includes some statements that have not yet been mentioned. Each of these will be addressed later in this chapter.

1. PROGRAM, SUBROUTINE, FUNCTION, BLOCK DATA statements
2. Explicit data type declarations and PARAMETER declarations
3. Array declarations
4. COMMON declarations
5. EQUIVALENCE declarations
6. EXTERNAL declarations
7. DATA declarations
8. Statement Function Definitions
9. Executable statements and FORMAT statements
10. RETURN statements (subprogram units only) and/or END statements

The Main Program

The main program is the body of code that encompasses all specifications, declarations, and executable statements that make up the tasks necessary to solve the problem or achieve the goal. The main program is defined by the PROGRAM and END statements.

The PROGRAM Statement

The PROGRAM statement is the first line of the program; there can be only one in an executable Fortran program. The PROGRAM statement specifies the name of the Fortran program. As with all variable names, the program name must be unique within the program and must follow all of the rules for building a variable name. It can have a maximum of 31 characters, and the first character must be alphabetic.

The syntax of the PROGRAM statement is as follows:

PROGRAM ***program_name***

Example:
```
PROGRAM Right_Triangle
```

The END Statement

The END statement must be the last statement in a Fortran program. It tells the compiler that there are no further statements to process. Program execution stops when the END statement is reached. The name of the program should be included as a comment to the END statement.

The syntax of the END statement is as follows:

END

Example:
```
END       ! of Program RIGHT_TRIANGLE
```

In this first program example, a simple calculation to determine the size of the hypotenuse of a right triangle is demonstrated. This example uses the formula $c^2 = a^2 + b^2$ and then uses the intrinsic Square Root function (SQRT) to get the resulting value of the side.

```
       PROGRAM Right_Triangle

       IMPLICIT NONE

C      /*******************************************************/
C      /*  This program calculates the hypotenuse of a      */
C      /*  right triangle given the 2 right angle sides.    */
C      /*******************************************************/

C     Variable declarations

       REAL*4   Side_One            ! side one of the triangle
       REAL*4   Side_Two            ! side two of the triangle
       REAL*4   Side_Three          ! side three of triangle
       REAL*4   Hypotenuse_Squared  ! the hypotenude squared
       REAL*4   Hypotenuse          ! the hypotenuse of the
                                    !  triangle

C     End of variable declarations

C     Begin execution - get side sizes
        PRINT *," Enter the size of each of the 2 sides of "
        PRINT *," the right triangle(separate with a comma): "
        READ *,Side_One, Side_Two

C     Calculate hypotenuse
        Hypotenuse_Squared = (Side_One ** 2) + (Side_Two ** 2)
        Hypotenuse = SQRT (Hypotenuse_Squared)    ! intrinsic
                                                  ! function

C     Display the results
        PRINT *," The size of the hypotenuse is: ",Hypotenuse)

       END           ! of the program Right_Triangle
```

Example 17-1 - Calculating the Hypotenuse of a Right Triangle

Specification Statements

Specification statements are non-executable statements that normally occur at the beginning of a program unit and prior to the start of the executable code section. Specification statements primarily provide information to the Fortran processor about the nature of the program. These statements include the declaration statements discussed in Chapter 16 as well as the following:

- The INCLUDE statement
- The COMMON statement
- The EQUIVALENCE statement
- The EXTERNAL statement

The INCLUDE Statement

The INCLUDE statement is a declaration section statement that directs the compiler to read in statements from the referenced or "included" file or library module. When it reaches the end of the included file or module, the compiler resumes compilation with the next statement after the INCLUDE statement. The INCLUDE statement follows one of the following forms:

```
INCLUDE 'filespec [/[NO]LIST]'
   or
INCLUDE '[text-library] (module-name) [/[NO]LIST]'
```

... where ***filespec*** is a character string that specifies the file to be included;

... where ***/[NO]LIST*** specifies whether the included code will appear in the listing file. If it is included, a number precedes each included statement. The number indicates the nesting depth of the code. The default is NOLIST.

... where ***text-library*** is a character string that specifies the file name of the text library to be searched in order to locate the ***module-name*** containing the code to be included. The module name is enclosed in parentheses and can be up to 31 characters long.

INCLUDE Rules

The following rules apply to the INCLUDE statement and included files:

- An included file or module can contain an INCLUDE statement.
- All INCLUDE statements must precede the first executable Fortran statement (SuRPAS)
- Only Fortran declaration statements can appear in an included file or module (SuRPAS)

```
C
C  ============== Common File Definitions ==============
C
           INCLUDE '($FORIOSDEF)'
            INCLUDE 'FAL$LIBRARY:FILES(COMMAREA.CFD)'
            INCLUDE 'FAL$LIBRARY:FILES(FILDIC.CFD)'
            INCLUDE 'FAL$LIBRARY:FILES(MASFAL.CFD)/LIST'
            INCLUDE 'MASFAL_STRUCTURE.INC/LIST'
            INCLUDE 'FAL$LIBRARY:FILES(LEDFAL.CFD)'
C
```

Example 17-2 - Program INCLUDE File Statements

Sharing Data

The ability to share data across program units can be accomplished with a number of Fortran tools. When the amount of information is relatively small it can easily be passed as an argument or arguments to a subroutine or function (as we will learn later in this chapter). As the program becomes more complex and the amount and types of data increase, it becomes more attractive to share the data using Fortran ***common regions*** (also called ***common blocks***). A common region is a defined area of data that is made known to two or more program units, allowing those program units to see, populate, and manipulate the data. The data can be of any size (within reason) and can be of mixed types. The variables and/or constants that define this common region must be defined (both type and size) prior to the declaration of the region; therefore, the region must be declared after the type declarations section of the program unit.

The COMMON Statement

Common regions are defined using the COMMON statement. The identical COMMON statement is placed in each program unit that is sharing the common data. Data is allocated to the variables and arrays in the COMMON statement based on the order of their appearance in the common list. This requires that if the number or size of variables or arrays changes, the change must be reflected in each instance of the COMMON statement. If not, the integrity of the data will be lost. The COMMON statement has the following syntax:

```
COMMON common_data_list
```

A simple example of a COMMON statement can be seen below:

```
INTEGER*4   Fund_Id_A, Fund_Id_B, Fund_Id_C
INTEGER*4   Day_Value (3,5)
COMMON      Fund_Id_A, Fund_Id_B, Fund_Id_C, Day_Value
```

The mapping of this simple example brings up an interesting feature of Fortran common regions. This example shows a common region with three integer variables and an integer array. The 3 by 5 array contains 15 integer array elements, so the total size of the common region represented is 18 INTEGER*4 elements or longwords (32-bits each). This region could be mapped in either of the following two ways:

```
1. INTEGER*4   Funds (3)
   INTEGER*4   Day_Value (3,5)
   COMMON      Fund, Day_Value

2. INTEGER*4   Fund_Info (18)
   COMMON      Fund_Info
```

Be careful when mixing character and numeric data in a COMMON statement. When possible, put CHARACTER type variables after INTEGER and REAL type variables.

The Named COMMON Statement

In most programs the need for common regions is minimal, so a single defined common region will suffice. As programs become more complex or have numerous program units, the programmer may find the need to subdivide the common region. This allows the sharing of parts of the region with some program units while other portions of the region are only visible to those program units that have a need to see them.

Common regions can be subdivided by assigning a unique name to each of the smaller portions of the region. This is done in the COMMON statement by attaching the unique name to the beginning of the statement, surrounded by slashes (/.../). The syntax of a named COMMON statement is:

```
COMMON / common_name / common_data_list
```

A simple example of a named COMMON statement can be seen below:

```
INTEGER*4   Funds (3)
INTEGER*4   Day_Value (3,5)
COMMON      /Fund_Info/   Funds, Day_Value
```

A more in-depth set of examples of named COMMONs will be examined in the subroutine portion of the chapter.

The Named COMMON is a PSECT

Fortran named common regions are mapped as OpenVMS program sections (called PSECTs). These are visible on the Fortran compiler listing and can provide some useful debug information.

The EQUIVALENCE Statement

It is frequently desirable to have two or more names referring to the same data (memory). The motivation for building such data or memory sharing may vary. In some cases we may want to map variables to elements of an array. In other cases we may want to change the array mapping from a matrix to a linear array. The Fortran EQUIVALENCE statement provides the ability to double map data or memory. The EQUIVALENCE statement has the following syntax:

EQUIVALENCE (item_a *[(n$_1$:n$_2$)]*, item_b)

In the EQUIVALENCE statement, ***item_a*** and ***item_b*** are variables that have been previously declared with both size and type information. Note that the EQUIVALENCE statement allows the mapping of a portion of the first variable (***item_a***) to the second variable (***item_b***) using the substring specification notation discussed in Chapter 16. This allows you to define the beginning and ending characters of the variable being equivalenced. This substring notation should always be used when using EQUIVALENCE statements in SuRPAS programs as it follows the SuRPAS standard. The EQUIVALENCE statement does not support the mixing of numeric and character type variables when mapping. The EQUIVALENCE statement also assumes that when an unsubscripted array name appears in a list, it starts at the first element of that array.

In this first example, the character array Name is mapped to two character variables - First_Name and Last_Name - resulting in two substrings.

```
CHARACTER*24    Name
CHARACTER*12    First_Name
CHARACTER*12    Last_Name
EQUIVALENCE     (Name(1:12), First_Name)
EQUIVALENCE     (Name(13),   Last_Name)
```

In the next example, two arrays are mapped to access a matrix as a linear array.

```
REAL*4      Parts  (3,3)
REAL*4      Items  (9)
EQUIVALENCE (Parts, Items)
```

Program Code Execution

Executable statements are the statements which cause the computer to carry out the actions of the programmer in building a solution. There are numerous types of executable statements supported in Fortran. The assignment statement was covered in Chapter Sixteen. Conditional selection statements (such as the various versions of the IF statement) and repetition statements (such as the DO and DO WHILE loops) will be covered in Chapter Eighteen. Fortran has built-in I/O statements to support reading and writing files, and PFPC has its own I/O support called FALIOS. These I/O support routines will be covered in Chapter Twenty.

The next sections cover some basic Fortran executable statements that allow you to build a simple Fortran program.

Data and Results

The Fortran assignment statement specifies an expression whose value is computed and then assigned to a designated variable. It is possible to combine several assignment statements into a program, for example:

```
PROGRAM  Average
REAL*4    X, Y, Avg
X = 4.7
Y = 4.9
Avg = (X + Y) / 2.0
END       ! of program Average
```

Example 17-3 - Basic Program Average

Three of the six lines of code in the program example (all of the executable lines of code) are assignment statements. Of course, these assignment statements are relatively simple, the values of the variables in the expressions are hard coded, and there is no way to make the results (AVG) available outside of the computer. Furthermore, it is impractical to develop a program that will calculate the average of only these two constants. It would be more useful if the program could compute the average of any two numbers, with the specific values of **X** and **Y** to be chosen after the program has been written.

To permit development of more useful programs, statements are needed for producing output of results to an external device, as well as for requesting input of data from an external device.

Fortran provides the READ and PRINT statements for these purposes. Using these statements, we can write the following more useful and complete program. Note that at PFPC the READ and PRINT statements are rarely used except for program testing and debugging.

```
PROGRAM  Average
REAL*4    X, Y, Avg

READ *, X, Y
Avg = (X + Y) / 2.0
PRINT *, X, Y, Avg

END       ! of program Average
```

Example 17-4 - More Useful Program AVERAGE

Data Input Using the READ Statement

In the code example above, there are still only three executable lines of code. The first executable line specifies that values for **X** and **Y** are to be obtained from a default input device. In the 1960's and 1970's the default input device would probably have been a card reader. Today the default device is typically a workstation keyboard. The execution of the default READ statement results in the assignment of the input values to the variables **X** and **Y**.

The READ statement specifies the location from which the input information is to be read, the format of the data being read, the status of the I/O operation, and the destination of the data being entered. The syntax of the READ statement is:

READ ***input_device*, *format*, *IOStatus*, *in_data_list***

This READ statement will be discussed fully in Chapter Twenty. The default READ statement requires only the default input device and the input data list. The syntax of the default READ statement is:

READ *, ***in_data_list***

In this example, the asterisk replaces the external device from which the data is being input. The `in_data_list` is **list-directed input**. List-direct input allows the data in the input list to be determined by the data type rather than any required format specification. The READ statement in code example 17-3 indicates that keyboard input of two REAL numbers will be assigned to the REAL variables **X** and **Y.**

Data Display Using the PRINT Statement

Data can easily be displayed to the workstation screen or printer by using the default PRINT statement. In example 17-3 (on page 17-11), the third executable line displays the values of **X**, **Y,** and **Avg** to the screen (assumed to be the default output device).

The PRINT statement specifies the format of the data being written and the location to which the data is being output . The syntax of the PRINT statement is:

```
PRINT  format,  out_data_list
```

The default PRINT statement requires only the default format and the output data list. The syntax of the default PRINT statement is:

```
PRINT  *,  out_data_list
```

In this example, the asterisk replaces the format specification of the data being output. The `out_data_list` is **list-directed output**. List-directed output allows the data in the output list to be determined by the data type rather than any required format specification. The PRINT statement in example 17-3 indicates the screen output of three REAL variables **X, Y,** and **Avg.**

Introduction to the FORMAT Statement

In both the default READ and the default PRINT statements, one of the arguments of the I/O statement is bypassed; that is the **format specification**. The format specification can take on a number of forms. One that we have already seen is the asterisk. The standard method for specifying the format of an input or output specification is the use of the FORMAT specification statement.

The FORMAT statement describes the position and format of the data being read or written. It is a labeled statement; that is, it has a numeric label in columns 1-5 of the statement line. That label is referenced in the I/O statement and acts as a pointer to the FORMAT statement. In example 17-5 on the next page, we see Program AVERAGE modified to issue the READ and PRINT statements with FORMAT statements. Note that label 10 in the first FORMAT statement is referenced in the READ statement and label 20 in the second FORMAT statement is referenced by the PRINT statement.

```
      PROGRAM  Average
      REAL*4    X, Y, Avg

      READ (*,10) X, Y
10    FORMAT  (2(F10.2))
      Avg = (X + Y) / 2.0
      PRINT 20, X, Y, Avg
20    FORMAT  ('X=',F10.2,'  Y=',F10.2,'  AVG=',F10.2)

      END       ! of program Average
```

Example 17-5 - Complete Program Average

The FORMAT statement specifies data type format by way of ***format codes***. The following letters are used to define data type:

- I - Integer
- F - Real
- A - Character Data
- X - skip a character (space)
- / - record terminator (skip a line)
- $ - suppress trailing carriage return

The number that follows the format code is the ***field width***. The field width indicates the size of the variable being specified. Integer variables are described with whole numbers; real variables are described with numbers that have whole number and fraction portions. The whole number portion specifies the size of the real number. The fraction portion describes the number of decimal places that will be stored or displayed.

In the example above, the READ statement inputs two real numbers. The FORMAT statement describes each value to be stored as F10.2. In that FORMAT statement, a ***count*** precedes the format specification indicating the number of times that the specification is duplicated (rather than describing the real number explicitly twice).

The STOP Statement

The STOP statement indicates that execution of the program is to cease immediately. It is normally used within a conditional selection code module or a repetition code module to halt execution prior to arriving at the END statement. The effect of the STOP statement is to return control to the operating system. There are no arguments or parameters in the STOP statement, and its syntax is as follows:

STOP

Subroutines and Functions

Many programs include a series of subtasks to solve a complicated problem. It can be very restrictive to code these subtasks sequentially within the main program code. In some cases these subtasks are executed numerous times at various points within the program. Optimally, we prefer to code these subtasks independent of the main program flow.

Fortran supports a mechanism that is designed to make independent subtask building easy to develop and debug. This mechanism, the ***external procedure***, allows the programmer to code each subtask as a separate program unit that can be ***external*** to the main program. By external, we mean that the code can be compiled separately, allowing for separate and independent testing and debugging prior to combining with the main program. The external procedure also has the feature that it supports a defined interface that allows the caller to pass defined arguments to the subtask and receive results back.

Fortran supports two kinds of external procedures: subroutines and functions.

Fortran Subroutines

A ***subroutine*** is a Fortran procedure that is invoked by referencing it in a CALL statement. It receives its input values either from the argument list passed by the CALL statement or through global variables that are visible to the subroutine. It also returns its results through the CALL statement argument list or through visible global variables. The general format of a Fortran subroutine is as follows:

```
SUBROUTINE subroutine_name ( argument_list )

  < specification and declaration section >

  < code execution section >

RETURN
END    ! of subroutine subroutine_name
```

Fortran Functions

A function is a procedure that results in a single number, logical value, character string, or array. The result of a function is a single value or single array that can be combined with variables and constants to form a Fortran expression. These expressions may appear on the right side of an assignment statement in the calling program so that functions are invoked simply by naming them in the expression. Fortran supports two types of functions: **intrinsic** and **user-defined**.

Intrinsic functions are built into the Fortran language. We have seen some examples of intrinsic functions in the previous chapter. Fortran has a rich intrinsic function set. In addition, the PFPC function library provides additional functions that are available for your use.

User-defined functions are defined by individual programmers to meet a specific need not addressed by the existing set of intrinsic and PFPC functions. The general format of a user-defined Fortran function is:

```
FUNCTION function_name (argument_list )

    < declaration section ... >
    < ... must declare type of function name >

    < code execution section >

RETURN
END      ! of function function_name
```

The SUBROUTINE Statement

The ***SUBROUTINE*** statement marks the beginning of a Fortran subroutine. It specifies the name of the subroutine and the argument list associated with it. The syntax of a SUBROUTINE statement is as follows:

SUBROUTINE ***name*** **(*dummy_argument_list*)**

The subroutine *name* must follow standard Fortran conventions; it may be up to 31 characters long and contain numbers, underscores, and alphabetic characters. The first character must be alphabetic. The *argument_list* contains a list of variables and/or arrays that are passed from the calling program unit to the subroutine. These variables are called ***dummy arguments***, since the subroutine does not actually allocate any memory for them. They are placeholders for the actual arguments which will be passed from the calling program unit when the subroutine is invoked.

Like any Fortran program unit, a subroutine must have a declaration section and an execution section. When a program calls the subroutine, the execution of the calling program is suspended and the execution section of the subroutine runs. When a RETURN or END statement is reached in the subroutine, the calling program starts running again at the line following the subroutine call.

Each subroutine is an independent program unit, beginning with a SUBROUTINE statement and terminating with an END statement. It is compiled separately from the main program and from any other procedures. Because each program unit in a program is compiled separately, statement labels and local variable names may be reused in different routines without causing errors.

Any executable program may call a subroutine, including another subroutine. However, a subroutine may not call itself unless it is declared to be recursive. To call a subroutine, the calling program unit places a CALL statement in its code.

The following is an example of a simple subroutine (named PAGE_HEADER) with a single argument (named PAGE_NUM) in its argument list. Each time this subroutine is called, it prints a page header with the page number as passed in the argument list.

```
   SUBROUTINE Page_Header (Page_Num)

C     /*********************************************/
C    /*   Subroutine to print the header and page    */
C    /*   number at the top of the next page         */
C     /*********************************************/

     INTEGER*4      Page_Num

C    Print the page heading with the supplied page number
     PRINT  '("1",20X,'Handling  Fortran  Data',15X,I3//)',Page_Num

     RETURN
     END       ! of subroutine  Page_Header
```

Example 17-6 - The PAGE_HEADER Subroutine

The RETURN Statement

The RETURN statement is executed as the last statement of a subroutine or function. It precedes the END statement and returns control to the calling program unit that invoked the subroutine or function. The syntax of the RETURN statement is:

RETURN

The END Statement

The END statement signals the termination of a subroutine. Upon arrival at the END statement, control is returned to the next line in the calling program unit. The syntax of the END statement is as follows:

END ! of subroutine *subroutine_name*

The *subroutine_name* should be identical to the name in the SUBROUTINE statement.

The FUNCTION Statement

The Fortran function begins with a FUNCTION statement and terminates with an END statement. The name of the function may contain up to 31 numbers, underscores, and alphabetic characters. The first character must be alphabetic. The name must be specified in the FUNCTION statement; it is optional to include the function name as a comment on the END statement line.

To invoke a function, name it in an expression. When it is invoked, control is transferred to the function code and execution begins at the first line of the code and ends when either a RETURN statement or END statement is reached. When the function completes, control is returned to the expression and the returned value is used to continue evaluating the Fortran expression in which the function was named. The syntax of the FUNCTION statement is as follows:

```
FUNCTION name (dummy_argument_list)
```

The *name* of the function must appear on the left side of at least one assignment statement in the function. The value assigned to the *name* when the function returns to the invoking program unit will be the value of the function.

The *dummy_argument_list* of the function may be blank if the function can perform all of its executing activities with no input arguments. The parentheses are required even if the list is blank.

It is necessary to assign a data type to a function since it returns a value. The data type must be declared both in the function specification section and in the calling program units. The type declaration of a user-defined Fortran function can take one of two forms. It is SuRPAS practice to declare the function type and size as a part of the FUNCTION statement itself (example 1). Fortran also supports defining the function in a separate data declaration line (example 2). Both forms are shown in the examples that follow:

```
1.   INTEGER*4 FUNCTION My_Function (I,J)

2.   FUNCTION My_Function (I,J)
     INTEGER*4 My_Function
```

An example of a user-defined function is shown in example 17-7. In this example, function Poly evaluates a quadratic polynomial equation with user-specified coefficients for x, a, b, and c. The formula used is the standard $ax^2 + bx + c$.

```
    REAL FUNCTION Poly (X,A,B,C)

C     /*****************************************************/
C    /*    Function to evaluate the quadratic polynomial   */
C    /*    of the form POLY=A * X ** 2 + B * X + C         */
C     /*****************************************************/

      REAL*4   X          ! Value to evaluate expressions
      REAL*4   A          ! Coefficient of X**2 term
      REAL*4   B          ! Coefficient of X term
      REAL*4   C          ! Coefficient of the constant term

C     Evaluate the expression
      Poly = A * X ** 2 + B * X + C

      RETURN
      END      ! of function Poly
```

Example 17-7 - The Poly Function

The END Statement

The END statement signals the termination of a function. Upon arrival at the END statement, control is returned to the next line in the calling program unit. The syntax of the END statement is as follows:

END ! of function *function_name*

The *function_name* should be identical to the name in the FUNCTION statement.

The CALL Statement

Subroutines are accessed by the CALL statement, which provides the name of the subroutine being called and a list of arguments which are used to pass information between the calling program unit and the subroutine. The syntax of the CALL statement is as follows:

CALL *subroutine_name (actual_argument_list)*

The CALL statement transfers control so that the Alpha executes code in the subroutine named in the CALL statement (instead of the calling program unit executing the next line of code). When the subroutine completes, control is returned to the calling program unit at the line after the CALL statement.

Unlike a function which always returns the result of its execution as the value of the function name, a subroutine is not required to return anything to the calling program unit. However, if it does return any values, they are returned as arguments in the CALL statement *actual_argument_list.*

Variables internal to a subroutine are initialized to zero when the main program starts up and are not initialized each time the subroutine is called. The variables retain their values as defined when the subroutine last reached its RETURN or END statement, and control is returned to the calling program unit. All variables should be initialized as a part of the declaration or initialization portions of the subroutine code to guarantee the predictability of execution. These variables are referred to as ***static variables*** because memory is allocated for each variable and they are therefore, able to retain their value.

Argument Passing

When a subroutine or function is accessed, information is passed to it through its arguments. In the case of the subroutine, these arguments may be used to return values to the calling program unit. If the execution of the procedure is to be successful, the relationship between the actual list of arguments in the calling program and the dummy arguments in the subroutine or function is critical; there is a one to one relationship in the quantity of arguments as well as the order and type. The order and type of the actual arguments must correspond exactly with the order and type of the corresponding dummy arguments. In the following example, the CALL statement (#1) and the SUBROUTINE statement (#2) for the GRADES subroutine have identical arguments.

```
1. CALL Grades (Gr_Eng,Gr_Sci,Gr_Math,Gr_Hist,Average)
2. SUBROUTINE Grades (Gr_1,Gr_2,Gr_3,Gr_4,Avg)
```

Fortran passes the 32-bit value of each of the arguments in the CALL statement or FUNCTION argument list to its subroutines and functions. Once the subroutine or function begins to execute the argument value does not change, unless changed by the routine code. If data is passed into a subroutine or function using the COMMON statement, variable values in the common area may change dynamically as the result of external activities (e.g. I/O, multithreaded code).

The Calling Standard

The OpenVMS operating system supports a calling standard that is followed by all OpenVMS languages and the operating system itself. This allows a program written in one language to call a system service, a library service, or a subprogram written in another language. Fortran functions and subroutines follow this standard in both the CALL statement and function expressions. Fortran passes its arguments by value - except character strings, which are passed by descriptors (data structures containing information about the variable). Other languages and the operating system may require arguments to be passed in another form. To support these forms, Compaq Alpha Fortran uses a prefix attached to the argument being passed. The argument is surrounded in parentheses and preceded by the prefix. The three methods (and an example of the argument format) follow.

Passing Method	Prefix	Example
By reference (address or pointer is passed)	%REF	%REF(MFUND)
By value (the contents or value is passed)	**%VAL**	**%VAL(MFUND)**
By descriptor (the data structure that describes the argument is passed)	%DESCR	%DESCR(MFUND)

Table 17-8 - Standard Argument Passing Methods

Defining Global Variables

Arguments that are passed to a subroutine or function are global in the sense that they are visible to both the calling program unit and the subroutine or function being called. Other variables defined in the calling program or those defined in the subroutine or function called are not visible across program unit boundaries. They have local visibility only - unless a global structure is defined. The COMMON region is an example of one such global structure. The common structure can be either named or unnamed, but it must be defined identically in both the calling program unit and the subroutine or function being called.

The SuRPAS Programming Template

Writing Fortran program code for the SuRPAS product requires the programmer to follow the SuRPAS Programming Standards (Best Practice available) and conforming to the SuRPAS templates. In Chapter Thirteen there is a discussion about the SuRPAS command procedure template (`fal$com:template.com`). This section discusses two additional templates: the main program template (`fal$main:template.for`) and the routine template (`fal$sub:template.for`).

The Main Program Template

The main program template, **`template.for`**, is found in the directory **`fal$main`**, It contains a structure that must be strictly followed when creating main program code for execution within SuRPAS. The template is found in Example 17-9 on the next page. This example has been compressed (blank comment lines removed) so that the content of the template can be viewed on one page.

When using this template be careful to leave the Modification History portion of the template as you found it. This portion is filled in by the SuRPAS Code Management System (CMS) when it is checked in. If you are creating new code, this section will be blank when you copy the template from fal$main. If you are updating the code, this section will already reflect previous modification history entries. Do not attempt to make manual modifications.

Following the Modification History section is the first executable line of code, the PROGRAM statement. Replace the X's with the program name and be sure to leave the IMPLICIT NONE statement. This is required to meet the SuRPAS coding standards.

The next sections are declaration sections. The basic common areas are included in the template. You will need to add any additional common areas required by your program by placing the appropriate INCLUDE statements in the common file and common area sections. Variable type declarations and FORMAT statements are also placed in this portion of the template.

The execution sections follow. The first executable line is normally the initialization of the SuRPAS file I/O system (CALL FALINI). This is followed by four reserved CONTINUE statements that define sections of program code. Label 1000 indicates the beginning of initialization code; label 2000 indicates beginning of main program code; label 8000 signifies the end of main program code and the beginning of cleanup (ENDTRP); and label 9000 signifies the beginning of error handling code (ERRTRP).

```
C       PROGRAM         :       xxxxxxxxx.FOR
C
C       COPYRIGHT 2002  :       PFPC Global Fund Services
C                               King of Prussia, PA  19312
C       AUTHOR          :
C       DESCRIPTION     :
C       INPUT FILES     :
C       OUTPUT FILES    :
C       REPORT FILES    :
C
C       ------------MODIFICATION  HISTORY-------------
C       WHO    DATE    RELEASE  SEQ#    FTK#  NOTE#
C       ===  ========  ======= =====  ======= =====
C       ----------------------------------<EOM>
        PROGRAM xxxxxxxxx

        IMPLICIT NONE
C       -----------------------------------
C       COMMON FILE DEFINITIONS:

        INCLUDE '($FORIOSDEF)'
        INCLUDE 'FAL$LIBRARY:FILES(COMMAREA.CFD)'
        INCLUDE 'FAL$LIBRARY:FILES(FILDIC.CFD)'
C       -----------------------------------
C       LOCAL VARIABLES:
C       -----------------------------------
C       COMMON AREA DEFINITIONS:
C       -----------------------------------
C       VARIABLE ASSIGNMENTS:
C       -----------------------------------
C       FORMAT STATEMENTS:
C       -----------------------------------
C       FILE CABINET:

        CALL FALINI
C       ===================================================================
C       ===================================================================
1000    CONTINUE
C       /***********************/
C       /* Initialization area */
C       /***********************/
2000    CONTINUE
C       -----------------------------------
8000    CONTINUE
C       -----------------------------------
9000    CONTINUE
C       /***************/
C       /* end of MAIN */
C       /***************/

        END
```

Example 17-9 - The Main Program Template

The Subroutine Template

The subroutine template, **template.for**, is found in the directory **fal$sub**. It contains a structure that must be strictly followed when creating separately compiled subroutine code for inclusion within SuRPAS programs. The majority of the template is found in Example 17-10 on the next two pages.

When using this template fill in the commented subroutine name on the first line (replace the X's). Be careful to leave the Modification History portion of the template as you found it. This portion is filled in by the SuRPAS Code Management System (CMS) when it is checked in. If you are creating new code, this section will be blank when you copy the template from fal$sub. If you are updating the code, this section will already reflect previous modification history entries. Do not attempt to make manual modifications.

Following the Modification History section is the first executable line of code, the SUBROUTINE statement. Replace the X's with the subroutine name and be sure to leave the IMPLICIT NONE statement. As in the program template this is required to meet the SuRPAS coding standards.

The declaration sections follow. These sections allow you to supply necessary INCLUDE statements to bring in common areas and definition files. There are also sections for the type declaration of variables that will be used in this subroutine. Notice that a state change variable has been predefined for you. The setting of this integer variable is used to determine if this execution pass is the first iteration within the program. It allows you to provide first pass initialization of variables, etc.

The execution sections follow. The first executable line is normally the label 1000 CONTINUE statement that signals the beginning of initialization code. The template provides a conditional test for first pass processing based on the state change variable mentioned earlier. This is followed by the three additional reserved CONTINUE statements that define sections of program code. Once again label 2000 indicates beginning of subroutine code; label 8000 signifies the end of subroutine code and the beginning of cleanup (ENDTRP); and label 9000 signifies the beginning of error handling code (ERRTRP).

The template ends with the standard RETURN and END statements.

```
C       SUBROUTINE      :       xxxxxxxxx.FOR
C
C       COPYRIGHT 2003  :       PFPC Global Fund Services
C                               King of Prussia, PA  19312
C
C       AUTHOR          :
C
C       DESCRIPTION     :
C
C       INPUT FILES     :
C
C       OUTPUT FILES    :
C
C       REPORT FILES    :
C
C       ------------MODIFICATION  HISTORY-------------
C       WHO    DATE    RELEASE  SEQ#    FTK#  NOTE#
C       ===  ========  ======= =====  ======= =====
C       ----------------------------------<EOM>

        SUBROUTINE      xxxxxxxxx()

        IMPLICIT NONE
C       ------------------------------------
C       PASSED VARIABLES:

C       xxxx*#                  xxxxxxx                 !IN>
C       xxxx*#                  xxxxxxx                 !OUT<

C       ------------------------------------
C       COMMON FILE DEFINITIONS:

        INCLUDE '($FORIOSDEF)'
        INCLUDE 'FAL$LIBRARY:FILES(COMMAREA.CFD)'
        INCLUDE 'FAL$LIBRARY:FILES(FILDIC.CFD)'

C       ------------------------------------
C       LOCAL VARIABLES:

        INTEGER*4       STATE_CHG_PREV /0/ !"first pass" processing

C       ------------------------------------
C       COMMON AREA DEFINITIONS:

C       ------------------------------------
C       VARIABLE ASSIGNMENTS:

C       ------------------------------------
C       FORMAT STATEMENTS:

C       ------------------------------------
```

Example 17-10 - The Subroutine Template

```
1000    CONTINUE

C       /**********************/
C       /* Initialization area */
C       /**********************/

        IF (STATE_CHG_PREV .NE. STATE_CHG) THEN

C           /***************************/
C           /* "FIRST PASS" processing */
C           /***************************/

            STATE_CHG_PREV = STATE_CHG

        ENDIF !END OF "FIRST PASS" processing


2000    CONTINUE


C       ___________________________________
8000    CONTINUE

C       ___________________________________
9000    CONTINUE

C       /**********************/
C       /* end of SUBROUTINE */
C       /**********************/

        RETURN
        END
```

Example 17-10 - The Subroutine Template - continued

Getting Started with the OpenVMS Debugger

Overview

The OpenVMS symbolic debugger allows the programmer to enter commands from the keypad as well as from the command line. The debugger is most easily used in screen mode; this displays the source code, the debug command, and system messages. This section discusses the following:

- Entering debugger commands
- Using screen mode
- Obtaining online HELP
- Terminating a debugging session

For information on how to build a program that can be run with the debugger built in, please refer to Chapter 14: The Program Development Cycle. In chapter 14, the section “Using the OpenVMS Debugger” walks you through the steps for compiling and linking with the debugger.

Entering Debugger Commands

The OpenVMS debugger allows you to enter commands in two ways:

- Keyboard entry - enter debugger commands in response to the DBG> prompt.
- Keypad entry - press the keypad key that corresponds to the intended debugger command.

When you invoke the debugger, keypad mode is the default.

- If you do not want the debugger keypad enabled, enter the following command: DBG> SET MODE NOKEYPAD
  The debugger is now in NOKEYPAD mode. You can assign debugger functions to those keys with the DEFINE/KEY commands
- To return to keypad mode, enter the following command:
  DBG> SET MODE KEYPAD

Most keypad keys have three predefined functions: DEFAULT, GOLD, and BLUE.

- To enter a key's DEFAULT function, press the key.
- To enter a key's GOLD function, first press and release the PF1 key, and then press the key.
- To enter a key's BLUE function, first press and release the PF4 key, and then press the key.

In Figure 17-11 on the next page, the DEFAULT, GOLD, and BLUE functions are listed within each key's outline from top to bottom, respectively.

For example:

- Pressing KP0 (keypad key 0) enters the STEP command (DEFAULT).
- Pressing the PF1 and KP0 enters the STEP/INTO command (GOLD).
- Pressing the PF4 and KP0 enters the STEP/OVER command (BLUE).

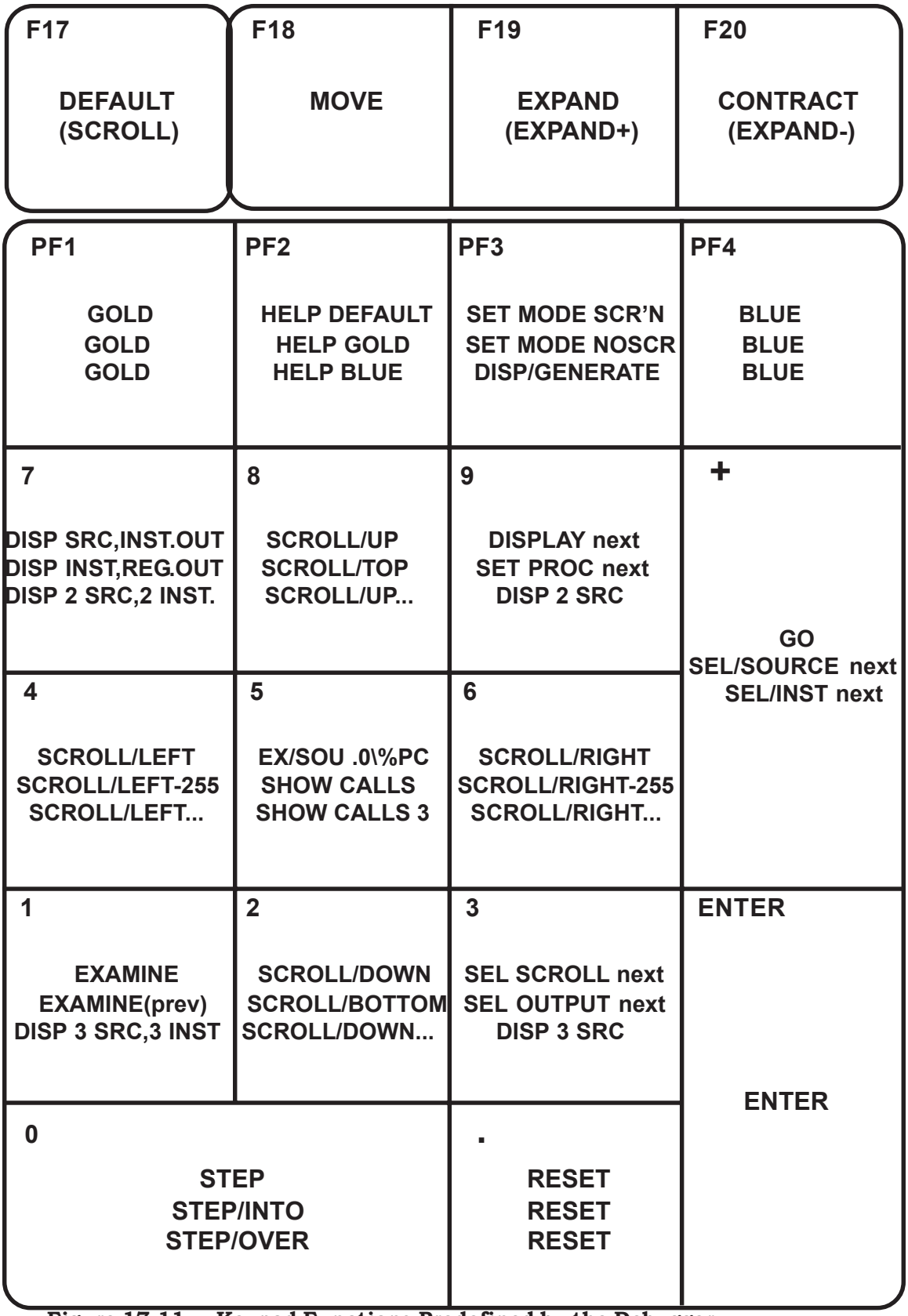

Figure 17-11 - Keypad Functions Predefined by the Debugger

Using Screen Mode

The debugger provides two modes for displaying information: screen mode and noscreen mode (default).

Screen mode provides the easiest way to view your source code. To use screen mode, press PF3 or enter the following command:

```
SET MODE SCREEN
```

Three displays appear on the screen by default:

- **SRC** (source display) automatically shows the current location of your program source code where execution is paused. An arrow in the left column points to the source line corresponding to the current value of the program counter (the PC), the next instruction to execute. The line numbers, assigned by the compiler, match those in a listing file. As you execute the program, the arrow moves down and the source code is scrolled vertically to center the arrow on the display.

- **OUT** (output display) shows all output generated by the debugger commands you issue.

- **PROMPT** shows the debugger prompt, your input, debugger diagnostic messages, and program output.

```
SRC module SWAP_ROUTINES-scroll-source
   2: PRINT *," Enter the size of each of the 2 sides of "
   3: PRINT *," the right triangle(separate with a comma): "
   4: READ *,SIDE_ONE, SIDE_TWO
   5: HYPOTENUSE_SQUARED = (SIDE_ONE ** 2) + (SIDE_TWO **
   6: HYPOTENUSE = SQRT (HYPOTENUSE_SQUARED)
OUT - output
 stepped to SWAP_ROUTINES\SWAPI\%LINE 4
 SWAP_ROUTINES\SWAPI\A: 35


PROMPT -error-program-prompt
 DBG> STEP
 DBG> EXAMINE A
 DBG>
```

Figure 17-12 - Debugger Screen Mode Display

Obtaining Online HELP

While you are using the debugger, you can obtain detailed online help:

- To list help topics, type **HELP** at the prompt.
- For an explanation of the help utility, type **HELP HELP.**
- For complete rules on entering commands, type **HELP Command_Format.**
- To display help for a particular command, type **HELP** *<command>*. For example, to display help on the SET BREAK command, type **HELP SET BREAK**.
- To list commands grouped by function, type **HELP Command_Summary.**

Terminating a Debugging Session

You can terminate a debugging session at any time; you do not have to allow the program to finish executing.

- To terminate a debugging session, use the EXIT command or the CTRL/Z key sequence.
- When you issue either of these commands (EXIT or CTRL/Z):
  - Your program's exit handlers (if any) are executed.
  - Control is returned to DCL command level.
- Once you return to DCL command level, you cannot resume the previous debugging session by entering the DEBUG command. To restart the debugger, you must run the program again.

Debugging Executable Lines and Variables

Overview

The OpenVMS symbolic debugger allows the programmer to move through the code step by step, and to set breakpoints at certain critical locations so the code will stop at those points. At any time the programmer can examine the contents of variables and, if necessary, change the value of a variable.

This section discusses:

- Stepping through code
- Stepping into subprograms
- Examining and manipulating variable contents
- Setting breakpoints

Starting and Resuming Program Execution

The GO command is used to:

- Start program execution after invoking the debugger with the DCL RUN command
- Resume program execution whenever the program has been suspended for any reason and the debugger prompt is displayed

The format of the GO command is as follows:

```
DBG> GO (address_expression)
```

The address_expression parameter specifies the location at which program execution should resume.

After the program is started with the GO command, program execution continues until one of the following events occurs:

- The program completes execution
- A breakpoint is reached
- A watchpoint is triggered
- An exception is signaled
- You press the Ctrl/C key combination

When you bring a program under debugger control, execution is paused at the beginning of the main program. Entering the GO command enables you to quickly test for an infinite loop or an exception.

The most common use of the GO command is in conjunction with breakpoints, tracepoints, and watchpoints:

- If you set a breakpoint in the path of execution and enter the GO command, execution is paused at that breakpoint.
- If you set a tracepoint, execution is monitored through that tracepoint.
- If you set a watchpoint, execution is paused when the value of the watched variable changes.

When your program runs to completion, you will receive the EXITSTATUS message.

```
DBG> GO
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
DBG>
```

Example 17-13 - EXITSTATUS Message

Executing a Single Line

When you run a program under the control of the debugger, the program statements are not executed automatically. You must enter debugger commands to execute your program.

Using the STEP Command

When you want to execute a specified number of program statements, use the STEP command. The syntax of the STEP command is as follows:

STEP [*integer*]

The optional parameter *integer* specifies the number of lines (or instructions) to be executed by the STEP command. For example:

```
DBG> STEP 3
```

By default, the STEP command causes the debugger to execute your program one line at a time. The example at the bottom of this page uses the following source code to demonstrate the use of the STEP command:

```
1: PROGRAM MAIN
2: INTEGER I
3: INTEGER J
4: J = 0
5: I = J
6: CALL SUB1 (I)
7: J = 2
8: END
```

```
DBG> STEP
stepped to MAIN\%LINE 4
4:        J = 0
DBG> STEP
stepped to MAIN\%LINE 5
5:        I = J
```

Example 17-14 - Using the STEP Command

STEP Command Qualifiers

Using qualifiers overrides the default STEP condition for a single command. The SET STEP command parameters specify how the debugger responds whenever the STEP command is issued. For example:

```
DBG> SET STEP INSTRUCTION
```

The parameters for SET STEP correspond to the qualifiers for STEP. The following lists some of the more popular STEP qualifiers and SET STEP parameters that affect the action of the STEP command.

Debugger Action	STEP Qualifier	SET STEP Parameter
Steps to the next exception condition (if any)	/EXCEPTION	EXCEPTION
Steps to the next instruction	**/INSTRUCTION**	**INSTRUCTION**
Steps into a called routine	/INTO	INTO
Steps to the next line	**/LINE**	**LINE**
Executes a called routine up to and including that routine's return instruction (opposite of /INTO)	/OVER	OVER
Steps to the return of the current routine	**/RETURN**	**RETURN**

Table 17-15 - Qualifiers and Parameters Used to Control STEP Action

Suspending Program Execution at a Specific Point

You can suspend program execution at specified points during a debugging session. For example, if you know the approximate location of a program bug, you can direct the debugger to suspend execution when it reaches the code preceding the suspected logic error. You can then enter debugger commands to determine the cause of the problem.

Breakpoints

Breakpoints allow a program to run until it reaches a specified statement. At that point you can examine and/or modify variables or arrays. The commands used to control breakpoints are summarized as follows:

SET BREAK	Defines a location to suspend program execution
SHOW BREAK	Displays all currently established breakpoints
CANCEL BREAK	Removes currently established breakpoints

Setting Breakpoints

Use the SET BREAK command to set a breakpoint at a specified location in your program. When a breakpoint is activated, the debugger suspends program execution, reports the breakpoint, and prompts you for commands. The syntax of the SET BREAK command is as follows:

```
SET BREAK [/qualifier] address_expression [,...]
    [WHEN (conditional_expression)]
    [DO (command [;...])]
```

The *address_expression* parameter specifies the location(s) at which the breakpoint is to be set. Some qualifiers are used instead of an address. An optional *WHEN clause* allows you to conditionally activate a breakpoint. An optional *DO clause* allows you to specify one or more debugger commands to be executed when a breakpoint is activated. The *conditional_expression* parameter is a language expression that is evaluated when the breakpoint is about to be activated. The breakpoint is then activated if *and only if* the expression evaluates to TRUE.

After entering a SET BREAK command you can resume program execution with the GO command. Your program runs until it reaches the specified location. The following are examples of how you can use the SET BREAK command to control your debugging session:

Command Line	Explanation of Action
`DBG> SET BREAK %LINE 4`	Sets a breakpoint at line 4 of the program.
`DBG> SET BREAK SUB1`	**Sets a breakpoint at routine SUB1**
`DBG> SET BREAK SUB1\%LINE 3`	Sets a breakpoint at line 3 of routine SUB1
`DBG> SET BREAK/CALL`	**Sets a breakpoint that takes effect on any call or return statement. In this case, no address is specified.**

Example 17-16 - Using the SET BREAK Debug Command

Examples 17-17 and 17-18 use the following source code to demonstrate the SET BREAK command:

```
1:   PROGRAM MAIN
2:   INTEGER I
3:   INTEGER J
4:   J = 0
5:   I = J
6:   CALL SUB1 (I)
7:   J = 2
8:   END
```

```
DBG> SET BREAK %LINE 6
DBG> GO
break at MAIN\%LINE 6
6:        CALL SUB1 (I)
DBG>
```

Example 17-17 - Breaking at a Line

Notes on Example 17-17:

- The breakpoint is set at line 6 of the program.
- Execution begins with the GO command.
- The breakpoint is encountered and execution is suspended; line 6 is displayed.

```
DBG> SET BREAK %LINE 6 WHEN (I .EQ. 0) DO (STEP); GO
break at MAIN\%LINE 6
6:        CALL SUB1 (I)
stepped to MAIN\%LINE 7
7:        J = 2
DBG>
```

Example 17-18 - Breaking with a Condition

Notes on Example 17-18:

- The breakpoint is set at line 6 of the program and stepped to the next line when the condition I .EQ. 0 is true; execution begins.
- The breakpoint is encountered; line 6 is displayed.
- The debugger steps to line 7 because the WHEN condition is true; line 7 is displayed.

The following is a partial list of qualifiers that are available for the SET BREAK command:

Command Qualifier	Resulting Debugger Action
/AFTER: n	Takes break action after the nth time the designated breakpoint is encountered
/CALL	**Breaks on every call or return instruction**
/EXCEPTION	Breaks every time an exception is signaled *
/INSTRUCTION	**Breaks on every instruction executed**
/LINE	Breaks at the start of each new line
/MODIFY	**Breaks at every instruction that writes to and modifies the value of a specified location**
/OVER	Breaks at the specified points only within the routine where the execution is currently suspended (not within called routines)
/RETURN	**Breaks on the return instruction of the routine associated with a specific address expression**

Table 17-9 - SET BREAK Command Qualifiers

* On Alpha processors, an exception may not be delivered to the program or debugger immediately after the execution of the instruction that caused the exception.

Examining Program Locations

The debugger allows you to examine and manipulate data as your program executes. You can use debugger commands to display the contents of:

- Program locations and variables
- Composite data structures (arrays and record structures)
- Arrays or portions of arrays (array slices)

The EXAMINE Command

Use the EXAMINE command to view the values of variables or memory locations. The syntax of the EXAMINE command is as follows:

```
EXAMINE [/qualifier]
   address_expression[:address_expression[,...]]
```

The ***address_expression*** represents either the name of the object to be examined or an expression specifying the memory location to be examined. Include the optional address_expression if you want to examine a range of addresses.

By default, EXAMINE causes the debugger to display a data item according to its data type (as it is defined in the program). You can use command qualifiers to display variables in a data type of your choice. Table 17-20 lists the EXAMINE command qualifiers used to specify the data type of variables.

Qualifier	Action
/ASCID	Interprets address_expression as the address of a string descriptor pointing to an ASCII string, and displays the string
/DFLOAT	**Displays d_floating point data**
/FLOAT	Displays f_floating point data
/GFLOAT	**Displays g_floating point data**
/HFLOAT	Displays h_floating point data
/HEXADECIMAL	**Displays data in hexadecimal radix**
/LONG	Displays long integer (32-bit) data

Table 17-20 - EXAMINE Qualifiers for Specifying Data Type

Using the EXAMINE Command

When examining a specific program line, the debugger displays the MACRO instruction generated by that line. When the debugger examines a program variable, it recognizes the variable type and displays its value accordingly. The following source code is used to demonstrate the EXAMINE command in the next five examples (17-21 through 17-25).

```
PROGRAM  MAINPROG
REAL*4          REALNUM         /6./
INTEGER*4       NUMBER          /2/
INTEGER*4       ONE_DIM(2)      /3,6/
INTEGER*4       TWO_DIM(2,2)    /5,10,15,20/
CHARACTER*5     STRING          /'HELLO'/
CALL  SUBPROG1
CALL  SUBPROG2
END         ! of program MAINPROG

SUBROUTINE  SUBPROG1
INTEGER*4       NUMBER
INTEGER*4       ONE_DIM(2)
INTEGER*4       TWO_DIM(2,2)
CHARACTER*5     STRING
RETURN
END         ! of subroutine SUBPROG1

SUBROUTINE  SUBPROG2
INTEGER         X           /4/
RETURN
END         ! of subroutine SUBPROG2
```

In the example below, the EXAMINE command displays the current value for different variables. NUMBER is an integer variable; STRING is a character string variable; and REALNUM is a floating-point variable.

```
DBG>  EXAMINE  NUMBER
MAINPROG\NUMBER:          2
DBG>  EXAMINE  STRING
MAINPROG\STRING:          'HELLO'
DBG>  EXAMINE  REALNUM
MAINPROG\REALNUM:         6.000000
DBG>
```

Example 17-21 - Examining Different Types of Program Variables

The following example shows how to examine one element of the one dimensional array ONE_DIM.

```
DBG> EXAMINE ONE_DIM (1)
MAINPROG\ONE_DIM
(1):           3
DBG>
```

Example 17-22 - Examining an Array Element

To display all of the elements of the one dimensional array ONE_DIM, enter EXAMINE without the array subscript:

```
DBG> EXAMINE ONE_DIM
MAINPROG\ONE_DIM
(1):           3
(2):           6
DBG>
```

Example 17-23 - Examining an Entire Array

You can use EXAMINE to display a specific portion of an array's elements, called an ***array slice***, without having to specify and examine the entire array. You can specify a full subscript range with a single asterisk. Example 17-24 displays the contents of column one of the two dimensional array TWO_DIM.

```
DBG> EXAMINE TWO_DIM (*,1)
MAINPROG\TWO_DIM
(1,1):             5
(2,1):             10
DBG>
```

Example 17-24 - Examining an Array Slice

The example below displays the contents of row two of the same array.

```
DBG> EXAMINE TWO_DIM (2,*)
MAINPROG\TWO_DIM
(2,1):             10
(2,2):             20
DBG>
```

Example 17-25 - Examining Another Array Slice

Modifying Program Variables

The debugger allows you to change the value of program locations during a debugging session.

- This feature is helpful if a program variable contains incorrect data, and you suspect that this is causing the program to produce incorrect output.
- You can use the debugger to place the correct value in the variable and then observe the results to see whether the correct output is produced.

The DEPOSIT Command

Use the DEPOSIT command to change the value of a program variable. The DEPOSIT command places a specified value into a specified program location. The syntax of the DEPOSIT command is as follows:

```
DEPOSIT [/qualifier]
        address_expression = value_expression
```

The ***address_expression*** parameter specifies the location into which the data should be deposited. The ***value_expression*** specifies the value to be deposited into the location.

The DEPOSIT command is like an assignment statement. The value of the expression specified to the right of the equal sign is assigned to the variable specified to the left of the equal sign. DEPOSIT accepts most of the same qualifiers as EXAMINE.

In Example 17-26, DEPOSIT is used to assign new values to program variables.

```
DBG> DEPOSIT REALNUM = 3.14
DBG> DEPOSIT STRING = 'HI'
DBG> EXAMINE REALNUM
MAINPROG\REALNUM:       3.140000
DBG> EXAMINE STRING
MAINPROG\STRING:        'HI   '
DBG>
```

Example 17-26 - Changing Values in Program Variables

SuRPAS Fortran Coding Standards for Programs

Overview

The SuRPAS Fortran coding standards that are addressed in this chapter include:

- Data types
- Data declarations
- Program sizes
- Use of commons

Data Types

There are 3 basic classes of data types used in SuRPAS programming: character, numeric, and boolean. The character declaration includes any type of alphanumeric value. Character declarations are either fixed or assumed size:

```
CHARACTER*3     TRAN_CODE
CHARACTER*(*)   PAS_BUF
```

Numeric data types commonly used in SuRPAS are REAL*8 and INTEGER*4. INTEGER*4 represents a signed integer and holds values from -2147483648 to 2147483647. REAL*8 represents a double precision real number and holds values from 0.56D-308 to 0.9D308. Some examples follow.

```
INTEGER*4 COUNTER
REAL*8    AMOUNT
```

The boolean data type most commonly used is the LOGICAL*4 declaration. A typical use of the LOGICAL data type in SuRPAS is the FSTPAS test (see the section in the complete coding standards on FSTPAS logic). A LOGICAL is simply used to set or test a condition for a value of true or false.

Data Declarations

Argument variables should be declared separately from variables which are local to a routine. The TEMPLATE.FOR programs have separate precoded sections. *Do not attempt to defeat their purpose.*

Each variable should be declared on a separate line. The only exceptions may be simple loop variables (the familiar I, J, K,...). Whenever practical, use an in-line comment to the right of the name to describe the variable function. Descriptions of argument variables which are input-only or output-only should include an arrow (“>” or ”<“) to indicate the usage.

Program Sizes
It can be difficult to understand and maintain routines when they become too large. If you end up with more than 1,000 lines or so, try to break up the routine into smaller pieces. You may be able to take a section of code which is repeated in a few places and move it to a separate routine which can be called repeatedly. This has a twofold advantage; it makes the main process easier to follow and it reduces the amount of duplicate code written (as well as the duplicate code that must be tested).

Of course, there are many cases where breaking code apart may not be practical. Use your judgement when deciding whether to break up code or keep it in one program.

Use of Commons
In most cases in SuRPAS, data is passed as arguments. Elements of an EQUIVALENCE or record structure are assigned in the caller, the associated data buffer is passed to the receiver, and the elements are recovered by assigning the received buffer back to the EQUIVALENCE or record. Look for uses of variables such as PAS_MASIOB and PAS_LEDIOB as examples.

A COMMON statement defines one or more contiguous blocks of storage shared among separate subprograms. Variables declared in a COMMON statement are available to any program which includes that COMMON's declaration and do not have to be passed as arguments.

The major use of the COMMON statement is FALCOM.CFD, which holds most of the "global" variables needed throughout SuRPAS. The values in FALCOM are initialized by FALINI.FOR. A call to FALINI should appear in the main program of any executable. Any underlying subroutine which requires access to those global values simply needs to include FALCOM.CFD. The use of FALCOM saves passing the global variables as arguments among the hundreds of subroutines that may reference them.

Commons may be used on a smaller scale to manage variables needed among a group of related subroutines. A common works more efficiently in this case than passing a buffer back and forth. Examples of such commons include TAXTAPCMN.CMN and COMMSTM.CFD. It is best to follow the same naming conventions as a CFD; that is, use the same prefix for all of the variables in the common. Commons should only be used in rare circumstances and only with SA approval.

Summary

Concepts

The Fortran Program Structure

- In general, all specification and data declaration statements (except FORMAT statements) precede the executable statements in a program unit.
- The PROGRAM statement is the first line of the program; there can be only one in an executable Fortran program.
- Data can be shared across program units through common regions. The common region is defined using the COMMON statement.
- The EQUIVALENCE statement provides the ability to double map data.
- The external procedure or subprogram allows the programmer to code and test subtasks separately. This is supported by Fortran subroutines and functions.

Getting Started with the OpenVMS Debugger

- The debugger allows a user to enter commands by way of the keypad or the keyboard.
- Information can be displayed either in command line mode or in screen mode.
- You can obtain detailed online help through the debugger's HELP utility.

Debugging Executable Lines and Variables

- Use the GO command to start or resume program execution.
- Execute a single or a specific number of program lines using the STEP command.
- Program execution can be suspended at a specified point using a breakpoint.
- Variables and memory locations can be viewed using the EXAMINE command.
- Data variables and memory locations can be modified using the DEPOSIT command.
- The DEPOSIT command can examine array elements, array slices, and entire arrays.

Commands

The Fortran Program Structure

PROGRAM ... END
Main program structure definition statements

INCLUDE
Imports source code or data into the program prior to compile

COMMON
A defined region of data that is shared across program units

EQUIVALENCE
The ability to double map data or memory

EXTERNAL
Link-time resolution of symbolic naming

READ *
Accepts input from a default keyboard device

PRINT *
Displays data to a default output device

FORMAT
Describes the position and format of data in an I/O statement

STOP
Execution of the program is to cease immediately

SUBROUTINE ... END
Subroutine structure definition statements

FUNCTION ... END
Function structure definition statements

RETURN
Returns control to the calling program unit at its next line of code

CALL
Invokes a subroutine and passes an argument list

Getting Started with the OpenVMS Debugger

SET MODE (NO)KEYPAD
Enables (disables) the keypad and its debugger functions

SET MODE (NO)SCREEN
Enables (disables) screen supported debugging

HELP
Gets debugger command help

EXIT or Ctrl/Z
Exits the debugger and returns control to the operating system

Debugging Executable Lines and Variables

GO
Starts or resumes program execution under the debugger

STEP
Executes a program statement, line, or instruction

SET BREAK
Defines a location at which to suspend program execution

SHOW BREAK
Displays all currently established breakpoints

EXAMINE
Views the value of a variable, instruction, or memory location

DEPOSIT
Inserts or modifies the value of a variable or memory location