Chapter Fourteen

THE PROGRAM DEVELOPMENT CYCLE

Introduction

This lesson is designed to introduce you to the system resources available in the OpenVMS programming environment. Each section will discuss a component of the program development cycle or a resource available to the programmer.

The OpenVMS environment offers several resources to provide assistance in the development of sophisticated programs using any of the high-level languages supported by the OpenVMS operating system. These resources include utilities (software programs consisting of one or more modules that provide a basic set of related services and functions) and system facilities.

Objectives

The objectives of this lesson are to provide the programmer with a conceptual knowledge of the following OpenVMS facilities and utilities:

- System services
- Run-time library
- Record Management Services
- Fortran compiler
- Librarian
- Linker
- Debugger

Chapter Terms

The topics discussed in this lesson assume that the student is familiar with the following terms:

Record Management Services	The file and record access subsystem of the OpenVMS operating system
Library	An organized set of routines or text units stored in a file

When the term **FORTRAN** (all in uppercase) appears in the text of this manual, it refers to the DCL command to invoke the Fortran compiler. The term **Fortran** is used to describe language and compiler features. In general, references to the Fortran compiler and language assume the implementation of the Fortran-90 standard unless otherwise stated. PFPC coding standards follow Fortran-77 with some limited implementation of Fortran-90 and Compaq Alpha Fortran extensions and features. These will be noted throughout the course.

Program Development Cycle

It is necessary to be familiar with the typical program development cycle for a high-level language before designing and creating program code. Your background has provided you with a general understanding of the development cycle. The next sections will look at the development cycle as it applies to the OpenVMS environment. The following is a quick overview:

1. Create a text file that contains the source statements of your program.
2. Compile or assemble the text file to produce a file containing object code.
3. Link the object file or files to produce an executable image file.
4. Invoke the image activator to run the executable code produced by the linker.
5. Debug the program to correct errors.

The flow diagram on the following page illustrates the five steps of program development.

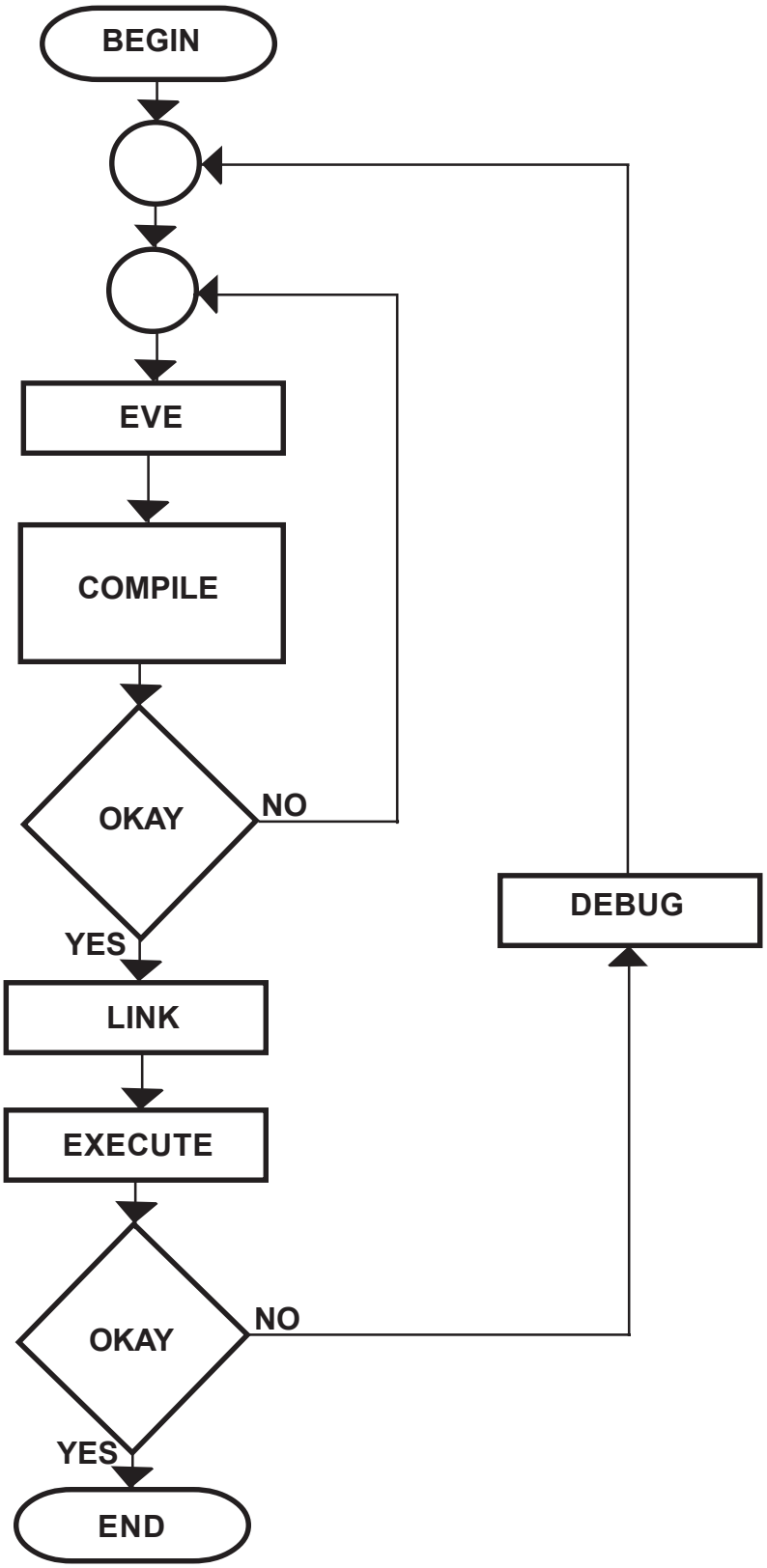

Figure 14-1 - Flow Diagram of the Five Steps of Program Development

Compiling, Linking, and Running a Program

To progress from a coded file to an executable image, the following steps are necessary.

Compile

- Call upon the language compiler to create an object file containing assembled instructions.
- Standard compilers are optimized to create more efficient code. This also means that there may not be a one to one relationship between high-level language statements and assembled instructions.
- OpenVMS high-level language compilers are called with the source code file(s) as parameters to the compiler command. The Fortran compiler is called using the following format (or its shortened version):

```
$ FORTRAN bobs_counter.for
$ FOR bobs_counter
```

- The result of a successful compile is one or more object (or .OBJ) files.

Link

- The linker is called to construct an executable file from one or more object files.
- The linker controls the order of instruction and data blocks (called ***program sections*** or ***psects***).
- The linker is executed with the object files as parameters fed to the LINK command, as in the following:

```
$ LINK bobs_counter.obj
```

- Linker behavior can be modified using option (or .OPT) files.
- The result of a successful link is an executable image (or .EXE) file.

Execute

- The executable image file is activated for execution using the RUN command.
- The RUN command assumes the executable image file has a file type of EXE.
- The following is the format of the RUN command:

```
$ RUN bobs_counter.exe
$ RUN bobs_counter
```

The FORTRAN Compiler

The primary functions of the Alpha Fortran compiler are listed below:

- To verify the Fortran source statements and to issue messages if there are any errors
- To generate machine language instructions from the source statements of the Fortran program
- To group these instructions into an object module for the OpenVMS Linker utility

The FORTRAN command initiates compilation of a source program. The command has the following syntax:

```
$ FORTRAN[/qualifiers]
    file-specification-list[/qualifiers]
```

The *qualifiers* indicate either special actions to be performed by the compiler or special properties of input or output files. The *file-specification-list* specifies the source file(s) containing the program modules to be compiled. The Fortran compiler allows multiple files to be compiled in a single compilation operation. If the source files specified in the *file-specification-list* are separated by commas, the program modules are compiled separately (with separate object code files generated). If they are separated by plus signs, the files are concatenated and compiled as one program (generating a single object code file).

In interactive mode, you can enter the file specification on a separate line by typing the command FORTRAN at the DCL prompt, followed by a carriage return. The system will respond with the following prompt:

```
_File:
```

Type the file specification immediately after the prompt and then press the carriage return key.

Specifying Input Files

When specifying a list of input files to the FORTRAN command, you can use abbreviated file specifications for those files that share common device names, directory names, or file names. The system applies temporary file specification defaults to those files with incomplete specifications; they are based on the previous device name, directory name, or file name encountered in the list (not the current directory default).

For example, assume that the current default is FAL$MISC:[9RB]. The following FORTRAN command shows how temporary defaults can be applied to a list of file specifications:

```
$  FOR  FAL$MISC:[254FDK]MAIN,SUB1,[2ST]SUB2
```

The preceding FORTRAN command compiles the following files:

```
FAL$MISC:[254FDK]MAIN.FOR
FAL$MISC:[254FDK]SUB1.FOR
FAL$MISC:[2ST]SUB2.FOR
```

To override a temporary default with your current default directory, specify the directory as a null value. For example:

```
$  FOR  FAL$MISC:[254FDK]MAIN,[]SUB
```

In this case, the empty brackets indicate that the compiler is going to use your current default directory to locate the file SUB.

If text libraries are going to be accessed by programs compiled by the FORTRAN command, you must include those library files in the file specification and use the /LIBRARY qualifier.

Specifying Output Files

The output produced by the compiler includes object files and listing files. You can control the creation of these files by using appropriate qualifiers in the FORTRAN command.

By default the Fortran compiler generates an object file. When you issue the FORTRAN command in interactive mode, the compiler does not generate listing files; you must use the /LIST qualifier to force the compiler to generate the file. When executing in batch mode, the compiler generates listing files by default. If you do not want listing files created in batch mode, use the /NOLIST qualifier in your command procedure.

If you are in the early stages of program development and your code is not ready to be linked (i.e., you are still cleaning up compile errors in your source code), the creation of object code files may be unnecessary and a waste of disk storage space. You can turn off object file creation by attaching the /NOOBJECT qualifier to the FORTRAN command. If object file creation is turned on, the compiler generates object files as follows.

- If you specify one source code file, one object file is generated.
- If you specify multiple source code files separated by commas, each source file is compiled separately and an object file is generated for each source file.
- If you specify multiple source files separated by plus signs, the source files are concatenated and compiled, and one object file is generated.

You can use both commas and plus signs in a single FORTRAN command to produce different combinations of concatenated and separate object files.

Object files are created with the same file name and location as the original source code file. The object file differs from the source code file in that its data type is OBJ rather than FOR. In the case where multiple files are concatenated to generate a single object file, the first source code file name is used for the object file name. To generate an object file with a different name and/or location, you must use the /OBJECT qualifier, which has the following format:

/OBJECT=*file-specification*

Five examples of FORTRAN commands are presented on the next page. They illustrate some of the more common implementations of the Fortran compiler.

1. **$ FORTRAN/LIST PROG01, PROG02, PROG03**

   Source files PROG01.FOR, PROG02.FOR, and PROG03.FOR are compiled as separate files, producing object files (named PROG01.OBJ, PROG02.OBJ, and PROG03.OBJ) and listing files (named PROG01.LIS, PROG02.LIS, and PROG03.LIS).

2. **$ FORTRAN PROG04+PROG05+PROG06**

   Source files PROG04.FOR, PROG05.FOR, and PROG06.FOR are compiled as one file, producing a single object file named PROG04.OBJ. No listing files are generated. (A listing file named PROG04.LIS would be generated in batch mode).

3. **$ FORTRAN/OBJECT=MAIN/NOLIST** Return
   _File: PROG07

   The source file for PROG07.FOR is compiled, producing an object file named MAIN.OBJ; no listing file is generated.

4. **$ FORTRAN PROG08+PROG09,PROG10/LIST**

   In this example two object files are produced.

   - PROG08.OBJ is produced by the compilation of PROG08.FOR and PROG09.FOR (no listing file is generated).
   - The source code file PROG10 produces the object file PROG10.OBJ and the listing file PROG10.LIS.

5. **$ FORTRAN PROG11+PROG12/NOOBJECT+PROG13**

   When you include a qualifier in a list of files to be concatenated, the qualifier affects all files in the list. In this example, the creation of any object files is suppressed. The source files PROG11.FOR, PROG12.FOR, and PROG13.FOR are concatenated and compiled, but no object files are produced.

FORTRAN Command Qualifiers

FORTRAN command qualifiers influence the way that the compiler processes a Fortran source code file. In simple compilations, issuing the FORTRAN command and a file specification of one source code file is sufficient to achieve your compilation goal. The availability of qualifiers allows you to dictate specific processing requirements and produce the correct output files.

The following table (continued on the next page) lists the FORTRAN command qualifiers, the negative form of the qualifier (if applicable), and a simple example of the qualifier. The default for the qualifier is noted where appropriate (def.).

Qualifier	Negative Form	Example
/CHECK= [NO]BOUNDS [NO]OVERFLOW [NO]UNDERFLOW ALL NONE	/NOCHECK	/CHECK=(NOBOUNDS, OVERFLOW)
/CONTINUATIONS=n	**None**	**/CONTINUATIONS=19 (def.)**
/CROSS_REFERENCE	/NOCROSS_REFERENCE	/NOCROSS_REFERENCE (def.)
/DEBUG= [NO]SYMBOLS [NO]TRACEBACK ALL NONE	**/NODEBUG**	**/DEBUG=(NOSYMBOLS, TRACEBACK)**
/D_LINES	/NOD_LINES (def.)	/NOD_LINES
/EXTEND_SOURCE	**/NOEXTEND_SOURCE**	**/NOEXTEND_SOURCE (def.)**
/G_FLOATING	/NOG_FLOATING (def.)	/NOG_FLOATING

FORTRAN Command Qualifiers (continued)

Qualifier		Negative Form	Example
/LIBRARY		None	/LIBRARY
/LIST=*file-specification*		**/NOLIST**	**/NOLIST**
/MACHINE_CODE		/NOMACHINE_CODE (def.)	/NOMACHINE_CODE
/OBJECT=*file-specification*		**/NOOBJECT**	**/OBJECT**
/OPTIMIZE		/NOOPTIMIZE	/OPTIMIZE (def.)
/SHOW=	**[NO]DICTIONARY** **[NO]INCLUDE** **[NO]MAP** **[NO]SINGLE** **ALL** **NONE**	**/NOSHOW**	**/SHOW=(NODICTIONARY,** **NOINCLUDE,** **NOMAP,** **SINGLE)**
/WARNINGS=	[NO]DECLARATIONS [NO]GENERAL ALL NONE	/NOWARNINGS	/WARNINGS=(GENERAL, NODECLARATIONS)

The OpenVMS Linker

The primary functions of the OpenVMS link utility are to:

- Allocate virtual memory within the executable image
- Resolve symbolic references among modules being linked together
- Assign values to relocatable global symbols
- Perform relocation

The linker's end product is an executable image that can be run on an OpenVMS system.

The LINK command initiates the link of the object file(s), and has the following syntax:

```
$ LINK[/command-qualifiers]
  file-specification[/file-qualifiers],...
```

The *command-qualifiers* specifies output file options.
The *file-specification* specifies the input object file to be linked.
The *file-qualifiers* specifies input file options.

In interactive mode, you can issue the LINK command with no accompanying file specifications. The system then requests the file specifications with the following prompt:

```
_File:
```

You can enter multiple file specifications by separating them with commas or plus signs. When used with the LINK command, the comma has the same effect as the plus sign; a single image is created from the input files specified. If no output file is specified, the linker produces an executable image file with the same name as that of the first input object module and a file type of EXE.

LINK Command Qualifiers

The following table lists the linker qualifiers of particular interest to Fortran programmers.

Function	Qualifiers	Defaults
Request output file and define a file specification	/EXECUTABLE[=file-spec] /SHAREABLE[=file-spec]	/EXECUTABLE=name.EXE, where name is the name of the first input file /NOSHAREABLE
Request and specify the contents of a memory allocation listing	**/BRIEF** **/[NO]CROSS_REFERENCE** **/FULL** **/[NO]MAP**	**/NOCROSS_REFERENCE** **/NOMAP (interactive)** **/MAP=name.MAP (batch), where name is the name of the first input file**
Specify the amount of debugging information	/[NO]DEBUG /[NO]TRACEBACK	/NODEBUG /TRACEBACK
Indicate that the input files are libraries and to specifically include certain modules	**/INCLUDE=(module-name...)** **/LIBRARY** **/SELECTIVE_SEARCH**	**Not applicable**
Request or disable the searching of default user libraries and system libraries	/[NO]SYSLIB /[NO]SYSSHR /[NO]USERLIBRARY[=table]	/SYSLIB /SYSSHR /USERLIBRARY=ALL
Indicate that the input file is a linker options file	**/OPTIONS**	**Not applicable**

Linker Output File Qualifiers

Qualifiers can be used in the linker command to influence the output of the linker. These qualifiers specify whether the debugging or the traceback facilities are to be included. The qualifiers are:

/[NO]DEBUG (required for run-time program debugging)
/[NO]TRACEBACK

Image File Qualifiers

The linker outputs an image file (.EXE) and, optionally, a map file (.MAP). The image file qualifiers are /[NO]EXECUTABLE and /[NO]SHAREABLE.

To suppress the generation of an image file, specify the /NOEXECUTABLE qualifier. A map file can be produced even if no image file is generated.

The /SHAREABLE qualifier specifies that the linker will produce a shareable image as its output. A shareable image is one that has all of its internal references resolved, but must be linked with one or more object modules to produce an executable image. This allows for the creation of global sections shared across multiple applications.

Map File Qualifiers

The map file qualifiers tell the linker whether a map file is to be generated and, if so, what information is to be included. The map qualifiers are specified as follows:

```
/MAP[=file-spec]
                        {/BRIEF or /FULL}
                        [/CROSS_REFERENCE]
```

The linker uses the following map file defaults.

- In interactive mode, the default is to suppress a map.
- In batch mode, the default is to generate a map.

The /BRIEF and /FULL qualifiers define the amount of information included in the map file. If neither the /BRIEF nor the /FULL qualifier is specified, the map file contains the image's characteristics, a list of contributing modules (as produced by /BRIEF), and a list of global symbols and values in symbol name order. You can use the /CROSS_REFERENCE qualifier with either the default or /FULL map qualifiers to request cross reference information for global symbols.

Linker Input File Qualifiers

Input file qualifiers affect the file specifications of input files. These files can be object files, shareable files previously linked, or library files. The qualifiers that control linker input files are the /LIBRARY qualifier and the /INCLUDE qualifier.

The /LIBRARY qualifier has the following syntax:

```
/LIBRARY
```

The /LIBRARY qualifier specifies that the input file is an object module or shareable-image library that is to be searched to resolve undefined symbols referenced in other input modules The default file type of input files with the /LIBRARY qualifier is OLB.

The /INCLUDE qualifier has the following syntax:

```
/INCLUDE=module-name(s)
```

The /INCLUDE qualifier specifies that the input file is an object module or shareable-image library and that the modules named are the only modules in the library to be explicitly included as input to the link activity. In the case of shareable-image libraries, the module is the shareable-image name. The default file type is OLB.

When supplying multiple names as *module-names* to the /INCLUDE qualifier, the module names should be surrounded in parentheses and separated by commas. At least one module name is required to use the /INCLUDE qualifier.

The /LIBRARY qualifier can also be used on the input file that has an /INCLUDE qualifier attached to it. In this case, the same library will be searched for undefined symbols.

FALLINK - The SuRPAS Linker Tool

The FALLINK Command procedure is used to link SuRPAS main programs. This procedure has two parameters, the name of the program to be linked (and any accompanying object files), and any link options that are to be added to the LINK command. The FALLINK command procedure has the following syntax:

```
$   @FAL$COM:FALLINK program-name link-options
```

In this call ***program-name*** is a character string parameter containing the name of the program and any object code files (not libraries) to be included in the resulting executable when the linker creates the program image file. When creating the string to be passed as the first parameter, the string must be surrounded by a set of double-quotes ("<string>") with each object file separated by a plus sign (+) and no spaces in between the file names or the plus signs.

The second parameter, ***link-options***, is also a string parameter. It contains the link options to be concatenated to the end of the linker call. The string should be surrounded by double-quotes ("<string>"), contain no spaces and include a series of qualifier keywords, each preceded by a slash (/), followed by an equal sign (=), and then by the accompanying information (unless the link option is not qualifier-based, such as the inclusion of additional object libraries).

This command procedure contains a series of checks that are SuRPAS related. They include determining:

- if the module will run on a VAX rather than an Alpha processor
- if an installed image will include the /TRACEBACK qualifier
- if the program requires system libraries that are not at a specific client site causing the program not to be linked
- if the program requires special options that are known to the procedure (they will be automatically added)

Program Execution

The RUN command initiates execution of a program. The command has the following syntax:

```
$ RUN[/[NO]DEBUG]   file-specification
```

You must specify the file name when executing the RUN command. If you omit optional elements of the file specification, the system automatically provides a default value. The default file type is EXE.

The DEBUG qualifier allows you to use the symbolic debugger, even if you omitted this qualifier from the FORTRAN and LINK commands. In order to achieve the best results when debugging a program, the debug qualifier should be included when compiling and linking. This forces the inclusion of critical symbol tables and allows access to source code listings. The /NODEBUG qualifier allows you to run a program that has debugging built in without the debugger.

Before the image is activated, the system initializes all variables and arrays that are initialized by means of DATA statements to zero. However, it is not considered good programming practice to rely on this.

Using The OpenVMS Debugger

Software debugging is the process of locating functional programming errors in a program that has compiled and linked successfully, but does not run correctly. For example, the program may terminate abnormally, produce incorrect output, or execute an infinite loop.

To help locate errors that occur during program execution (run-time), the OpenVMS Symbolic Debugger allows you to maintain some control while the program is running. You can observe and control program execution, manipulate program locations and variables, display source code, and observe program input and output. Once you have found the error in your program, you can edit the source code and compile, link, and execute the corrected version.

In this course we will use the debugger to aid in creating each of our Fortran class exercises. The following paragraphs get us started. Prior to each of our class exercises, additional features of the OpenVMS Symbolic Debugger will be introduced.

Compiling and Linking Your Program for Debugging

To bring your program under debugger control, you must first compile and link the program's modules.

When compiling your program, you must attach the /DEBUG qualifier to the FORTRAN command. This directs the compiler to write symbol table information for the modules of your program. It is also recommended that you use the /NOOPTIMIZE qualifier. Compiler optimization often removes, merges, or changes the order in which code is executed and variables are managed. This can cause problems and confusion as you attempt to debug your program.

If your program's source code is in several files, you must compile each file whose symbols you want to reference by specifying the /DEBUG qualifier for each compilation.

The LINK/DEBUG command passes symbol table information to the executable image. It also causes the debugger to be invoked when the user follows the LINK/DEBUG command with RUN/DEBUG.

Invoking the OpenVMS Debugger

The following information explains how to invoke the OpenVMS debugger and bring a program under debugger control.

- Verify that you have compiled and linked the program with the /DEBUG qualifier as discussed in the previous sections. Using this information, your compile and link command should be similar to these example commands in which the MONTHLY_REPORT program is prepared for program execution:

```
$ FORTRAN/DEBUG/NOOPTIMIZE MONTHLY_REPORT.FOR
$ LINK/DEBUG MONTHLY_REPORT.OBJ
```

- If you specified the /DEBUG qualifier when compiling and linking, the debugger will be invoked as a part of program activation (the RUN command). To execute the program without invoking the debugger you will need to issue a RUN/NODEBUG command. If you did not specify the /DEBUG qualifier when you compiled and linked your program, specify the qualifier when you run the program, as illustrated below:

```
$ RUN/DEBUG MONTHLY_REPORT
```

The following message will be displayed when you enter the program:

```
                OpenVMS Debugger Version 7.1
%DEBUG-I-INITIAL, language is FORTRAN, module set to MONTHLY_REPORT
DBG>
```

OpenVMS Program Development Facilities

The sections that follow introduce OpenVMS facilities used in program development. The resources provided in the OpenVMS environment include:

- System services
- Run-time libraries
- Record Management Services (RMS)

System Services

System services are procedures that the OpenVMS operating system uses to:

- Control resources available to processes
- Provide for communication among processes
- Perform other operating system functions

Although most system services are used primarily by the operating system on behalf of users, many are available for general use and provide techniques that can be used in application programs.

- A user references a system service from a program (also known as a process). The process code is loaded into the program region (P0 space).
- System services are referenced from the control region (P1 space).

System services are divided into the following functional groups:

- Security
- Event flag
- Asynchronous System Trap (AST)
- Logical Names
- Input/Output (I/O)
- Process control
- Timer and time conversions
- Condition-handling
- Memory management
- Change mode
- Lock management

System services use a structural format called the routine template, which includes the following main headings:

- Routine name
- Format
- Arguments
- Description

Run-time Library Routines

The OpenVMS run-time library contains sets of general purpose and language support procedures. Run-time library procedures are provided to do the following:

- Manipulate strings
- Perform data conversions
- Evaluate mathematical functions
- Control terminal I/O
- Control other OpenVMS features such as logical names

User programs can call these procedures in any combination to perform tasks required for program execution.

Run-time Library Facilities

- LIB$ - general purpose procedures
- MTH$ - mathematics procedures
- OTS$ - language-dependent support procedures
- SMG$ - screen management procedures
- STR$ - string manipulation procedures

Run-time Library Access

- System services
- The command language interpreter
- Some OpenVMS operating system machine instructions

Run-time Library Operations

- Allocate resources that your process needs (virtual memory and event flags)
- Display time information while your program is running
- Fetch and modify some default characteristics of the line printer output
- Present the system date and time in various formats
- Set up and use binary tree
- Search for specified files
- Get and put string in the process common storage area

Libraries

High-level language routines are often used by more than one program. Routines can be invoked from any program now or in the future by storing them in a special file called a library. The OpenVMS Librarian utility creates and maintains five types of libraries.

- **Object Libraries** - store compiled routines (object modules) created in the MACRO assembler or high-level languages such as Fortran or C. The OpenVMS linker uses object libraries to resolve undefined symbols and routines called in a program.
- **Macro Libraries** - store MACRO assembler macros. The MACRO assembler reads them to resolve undefined references to macros in MACRO source programs.
- **Help Libraries** - store help messages. The OpenVMS Help utility reads these files.
- **Text Libraries** - store textual information. Various utilities and user programs read them.
- **Shareable Libraries** - contain symbol tables for shareable images (not the images themselves) that can be accessed by user programs.

The OpenVMS operating system provides two system-defined libraries:

- VMSRTL.EXE (the common run-time library)
- STARLET.OLB (the default system object library)

Each high-level language has an installed standard, intrinsic, or run-time library for routines supported by that language. The Fortran libraries are:

- FORSYSLIB.OLB (the system routine object library)
- FORSYSDEF.TLB (the language definitions default text library)

Object Libraries

The OpenVMS librarian enables you to store frequently used segments of code (such as procedures and functions) in specially indexed files called libraries. A library is a file with an internal index that points to the location for each of the units it contains. An object library stores compiled routines (object modules) created by the MACRO assembler or high-level languages such as Fortran.

When you build a routine that you wish to use in more than one application, it should be placed in an object module library once it has been successfully compiled. The OpenVMS Linker utility can read this type of library by extracting the routines from the library in order to support calls made in the program.

Linking Object Libraries

The OpenVMS Linker utility uses the object library(ies) to resolve undefined symbols in a program. It combines the code from the library(ies) with your source code object modules to produce a single executable image file.

To make an object library known to the linker, use one of the following techniques.

- The most common method is to specify the library in a LINK command line, using the /LIBRARY qualifier on the file specification:

```
$ LINK BOBS_COUNTER.OBJ, IO_ROUTINES.OLB/LIB
```

- The user can also define the library as a default library for the linker using the DEFINE command. The linker will search the default libraries in the event that it cannot resolve symbolic references using the input modules alone.

```
$ DEFINE LNK$LIBRARY IO_ROUTINES.OLB
```

  This allows you to issue the LINK command without referencing the object library.

```
$ LINK BOBS_COUNTER.OBJ
```

NOTE: The Linker utility has been modified in the SuRPAS environment to perform an automatic search for all modules which are referenced in the code but not explicitly supplied on the LINK command line. This permits a developer to specify only the main program and their changed modules in the LINK statement. The linker will resolve the other references.

Building or Adding a Module to a Library

To build or add a module to a new or existing library, or to do both, follow these steps:

1. Create a routine in the source file using an interactive editor.

2. Compile (or assemble) the routine to create one or more object modules. This example calls the Fortran Compiler to compile the FIND routine:

```
$ FORTRAN FIND.FOR
```

3. Add the module to a library by invoking the OpenVMS Librarian. The Librarian manages object libraries and uses the /CREATE qualifier to create new libraries. The syntax for the LIBRARY command is:

```
$LIBRARY[/qualifiers]  library-name
                       routine-name
```

   - This example shows how to create a new library named MISSION and add the SEARCH object module to it:

```
$LIBRARY/CREATE MISSION.OLB SEARCH.OBJ
```

   - The following example illustrates the LIBRARY command used to add the FIND module to an existing library named MISSION:

```
$ LIBRARY MISSION.OLB FIND.OBJ
```

   - The following example illustrates the LIBRARY command used to list the contents of an existing library named FAL$LIBRARY:FILES.CFD. The listing is output to a file named FILES.LIS in the default directory:

```
$ LIBRARY/LIST=FILES.LIS
                          FAL$LIBRARY:FILES.TLB
```

Extracting a File from a Library - An Interactive Exercise

Extract the FILDIC.CFD and COMMAREA.CFD files from the FILES.TLB text library. FILES.TLB is located in FAL$LIBRARY. Place the two files in your [.CFD] subdirectory. The format of this LIBRARY command is:

```
$ LIBRARY/EXTRACT=file-spec
      /OUTPUT=destination-file-spec library-name
```

Appendix: OpenVMS Record Management Services (RMS)

OpenVMS ***Record Management Services (RMS)*** is the file and record access subsystem of the OpenVMS operating system. RMS contains general routines that assist user programs in processing and managing files and their contents. RMS can be utilized directly or transparently.

- The MACRO assembly or C high-level language programmer generally calls RMS procedures directly to perform file I/O.
- High-level language (e.g., Fortran) programmers use RMS indirectly by calling the language-specific file I/O routines provided by the language syntax. These programmers can also call the RMS procedures directly to implement specific functionality not supported by the high-level language.

A user calls a RMS procedure in a program. The program is loaded into the program region of memory (P0 space). RMS vectors to RMS procedures stored in the program's control region of memory (P1 space). The RMS procedures are actually located in the operating system's region of memory (S0 space).

When the operating system transfers control to a RMS procedure, the access mode is elevated to executive mode. The system executes the procedure in executive mode on behalf of the calling program. When the procedure terminates, the access mode is lowered and control is returned to the calling program.

File Organization

There are three types of files supported by RMS.

- Sequential files - one record is defined after another
- Relative files - records are defined in cells
- Indexed files - keys are used to access records

RMS User Control Blocks

One of the important uses for RMS in programming is ***User Control Blocks*** **(UCB's)**. Programs communicate with RMS through the user control blocks (structures), and space must be allocated in the program for them. Usually this is done at compile time (as with Fortran). Optionally, values can also be set for the fields in the UCB's at run time. After the appropriate control blocks have been allocated in the program, RMS procedures can be called. In many high-level languages this is transparent to the programmer, as the structures are defined by the compiler in response to a RMS file creation or open call.

RMS provides a set of procedures that enables you to invoke file and record processing services and to allocate user control blocks. RMS supports two kinds of records: ***fixed-length*** and ***variable-length***. As mentioned, RMS supports three types of file organization: sequential, relative, and indexed.

There are four user control blocks that are created for files and interface with the program.

- **FAB** - file access block
- **RAB** - record access block
- **NAM** - name block
- **XAB** - the extended access block

Summary

Concepts

An efficient program development cycle includes the following steps:

1. Create the source code
2. Compile or assemble the source code
3. Link the object files and object libraries to produce an executable image file
4. Invoke the image activator to run the executable code produced by the linker
5. Debug the program to correct errors

The following OpenVMS system services and utilities can be used in program development:

- Resources provided in the OpenVMS environment:
    - System services
    - Run-time library (RTL)
    - Record Management Services (RMS)
- User programming tools:
    - Librarian
    - OpenVMS Debugger

Commands

Compiling FORTRAN Source Code

```
$ FORTRAN[/qualifiers]
  file-specification-list[/qualifiers]
```

Linking Object Code Files

```
$ LINK[/command-qualifiers]
  file-specification[/file-qualifiers],...
```

FALLINK

```
$ @FAL$COM:FALLINK program-name link-options
```

Program Code Execution

```
$ RUN[/[NO]DEBUG] file-specification
```

Creating and Using OpenVMS Libraries

```
$ LIBRARY[/qualifiers] library-name
                       routine-name
```

Extracting a File from a Library

```
$ LIBRARY/EXTRACT=file-spec
    /OUTPUT=destination-file-spec library-name
```