IAS Guide to Program Development

Order Number: AA–PAXVA–TC

Operating System and Version: IAS Version 3.4

May 1990

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

This document was prepared using VAX DOCUMENT, Version 1.2

Contents

FIGURES

TABLES

Preface

Manual Objectives

The *IAS Program Development Guide* introduces the program development environment on the IAS operating system. It provides a synopsis of the information immediately useful in getting started in the program development process. The book also gives an overview of the software environment and some guidelines on program design.

Intended Audience

This book is intended for the person who is already familiar with the general, basic operations of an IAS system: gaining access to the system, using the terminal and related devices, and requesting simple Executive services through the command interface. The greater part of the book addresses assembly language programming, because that language is the one provided with all systems. Included is one chapter summarizing the program development procedures for a high-level language, PDP-11 FORTRAN IV. However, most of the topics covered for the assembly language programmer—using a text editor, creating an executable image, using library facilities—apply to programmers using any computer language.

If you are not familiar with the general, basic operations of the system, you should first read the *IAS V3.4 Release Notes* and the *IAS Installation and System Generation Guide*. These books describe how to access the system, use a terminal, and use the system command interface.

Structure of This Document

This guide is meant to be read as you use the system. For this reason, the examples are presented in an order that you can follow at the terminal. Rather than demonstrate the complexity of the system, these examples are designed to demonstrate practical program development operations.

This guide is also meant to be used with other manuals in your documentation set. Toward this end, a selection of further reading material is listed in the last section of each chapter. By using this guide, then, you can become increasingly familiar with other, more advanced manuals until you need not refer to this introductory text except as a refresher.

The information in this book is organized into seven chapters:

- Chapter 1 introduces the software and hardware on which you develop programs.
- Chapter 2 describes how to create an assembly language source program using a skeleton file and text editor.
- Chapter 3 describes how to use the MACRO-11 Assembler to generate an object module.
- Chapter 4 describes how to use the task builder (TKB) to link object modules to create a loadable task image.
- Chapter 5 introduces debugging aids and discusses how to use them.
- Chapter 6 describes how to create and maintain a library of macro source statements and a library of object module subroutines.
- Chapter 7 briefly introduces the FORTRAN IV program development process.

Associated Documents

As mentioned above, documents recommended for further reading are listed at the end of each chapter. In addition, the *IAS Master Index and Documentation Directory* lists and describes all the documents in the documentation sets for each system.

Conventions Used in This Document

The following conventions are used in this manual:

Convention	Meaning
MCR>	This is the explicit prompt of the monitor console routine (MCR).
PDS>	This is the explicit prompt of the program development system (PDS).
xxx>	Three characters followed by a right angle bracket indicate the explicit prompt for a task, utility, or program on the system.
UPPERCASE	Uppercase letters in a command line indicate letters that must be entered as they are shown. For example, utility switches must always be entered as they are shown in format specifications.
command abbreviations	Where short forms of commands are allowed, the shortest form acceptable is represented by uppercase letters. The following example shows the minimum abbreviation allowed for the PDS command DIRECTORY: `PDS> DIR`
lowercase	Any command in lowercase must be substituted for. Usually the lowercase word identifies the kind of substitution expected, such as a filespec, which indicates that you should fill in a file specification. For example: `filename.filetype;version`This command indicates the values that compose a file specification; values are substituted for each of these variables as appropriate.
/keyword, /qualifier, or /switch	A command element preceded by a slash (/) is an MCR keyword; a DCL qualifier; or a task, utility, or program switch. Keywords, qualifiers, and switches alter the action of the command they follow.
parameter	Required command fields are generally called parameters. The most common parameters are file specifications.
[option]	Square brackets indicate optional entries in a command line or a file specification. If the brackets include syntactical elements, such as periods (.) or slashes (/), those elements are required for the field. If the field appears in lowercase, you are to substitute a valid command element if you include the field. Note that when an option is entered, the brackets are not included in the command line.
[, . . .]	Square brackets around a comma and an ellipsis mark indicate that you can use a series of optional elements separated by commas. For example, (argument[, . . .]) means that you can specify a series of optional arguments by enclosing the arguments in parentheses and by separating them with commas.
{ }	Braces indicate a choice of required options. You are to choose from one of the options listed.

Convention	Meaning
:argument	Some parameters and qualifiers can be altered by the inclusion of arguments preceded by a colon. An argument can be either numerical (COPIES:3) or alphabetical (NAME:QIX). In DCL, the equal sign (=) can be substituted for the colon to introduce arguments. COPIES=3 and COPIES:3 are the same.
( )	Parentheses enclose more than one argument in a command line. `SET PROT = (S:RWED,O:RWED)`
,	Commas are used as separators for command line parameters and to indicate positional entries on a command line. Positional entries are those elements that must be in a certain place in the command line. Although you might omit elements that come before the desired element, the commas that separate them must still be included.
[g,m] [directory]	The convention [g,m] signifies a user identification code (UIC). The g is a group number and the m is a member number. The UIC identifies a user and is used mainly for controlling access to files and privileged system functions. This might also signify a user file directory (UFD), commonly called a directory. A directory is the location of files. Other notations for directories are: [ggg,mmm], [gggmmm], [ufd], and [directory]. The convention [directory] signifies a directory in the same [g,m] form as the UIC. Where a UIC, UFD, or directory is required, only one set of brackets is shown (for example, [g,m]). Where the UIC, UFD, or directory is optional, two sets of brackets are shown (for example, [[g,m]]).
filespec	A full file specification includes device, directory, file name, file type, and version number, as shown in the following example: `DL2:[46,63]INDIRECT.TXT;3` Full file specifications are rarely needed. If you do not provide a version number, the highest numbered version is used. If you do not provide a directory, the default directory is used. Some system functions default to particular file types. Many commands accept a wildcard character (*) in place of the file name, file type, or version number.
.	A period in a file specification separates the file name and file type. When the file type is not specified, the period may be omitted from the file specification.
;	A semicolon in a file specification separates the file type from the file version. If the version is not specified, the semicolon may be omitted from the file specification.
@	The at sign invokes an indirect command file. The at sign immediately precedes the file specification for the indirect command file, as follows: `@filename[.filetype;version]`
. . .	A horizontal ellipsis indicates the following: • Additional, optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis shows where elements of command input or statements in an example or figure have been omitted because they are irrelevant to the point being discussed.

Convention	Meaning
KEYNAME	This typeface denotes one of the keys on the terminal keyboard; for example, the RETURN key.
"print" and "type"	As these words are used in the text, the system prints and the user types.
xxx	A symbol with a one- to three-character abbreviation, such as X or RET, indicates that you press a key on the terminal. For example, RET indicates the RETURN key, LF indicates the LINE FEED key, and DEL indicates the DELETE key.
Ctrl/a	The symbol Ctrl/a means that you are to press the key marked Ctrl while pressing another key. Thus, Ctrl/Z indicates that you are to press the Ctrl key and the Z key together in this fashion. Ctrl/Z is echoed on some terminals as ^Z. However, not all control characters echo.

1 The Program Development Environment

This chapter introduces the software and hardware that you typically need to develop programs on an IAS multiprogramming system. Its aim is to orient you to the environment in which you will be working. The remaining chapters in the guide further describe and illustrate how to use the tools and facilities introduced in the following sections.

1.1 Software Tools

IAS makes software tools available as executable entities called system tasks. The system tasks include one or more editors, the MACRO-11 Assembler, the IAS task builder (TKB), several aids to debugging, and a number of utility tasks. Your system may also include one or more high-level language compilers or interpreters. These elements combined form the program development environment. In general, the system manager makes these tasks accessible to you by installing them on the system.[1]

To invoke a task, you need not know where the task resides. The IAS operating system offers two command line interpreters (CLIs) for communicating with the system and invoking system services. These are the Monitor Console Routine (MCR) and the program development system (PDS). MCR is included in all IAS systems, whereas PDS is optional. Each terminal is set to recognize either MCR or PDS upon logging in.

1.1.1 Command Line Interpreters

IAS systems can have one or more CLIs. All systems include MCR. Many systems include PDS, and some systems include user-written CLIs. Both MCR and PDS include commands to invoke most system tasks and utilities to set and display certain system characteristics. In general, MCR commands invoke tasks such as PIP, a utility used to manipulate files (for example, copying them). PDS commands specify actions directly, as in the COPY command or the TYPE command.

The prompts for PDS are as follows:

```
PDS>
```

MCR is the fundamental CLI for the IAS operating system. From an MCR terminal, tasks installed with names in the form . . . abc can be invoked simply by typing the abc portion of the task name. Most system tasks and utilities are installed with names of that form. MCR also provides commands to set and display certain system and device characteristics. MCR provides the most direct interface with the operating system.

The prompt for MCR is as follows:

```
MCR>
```

PDS is an optional CLI included in most systems with heavy terminal use. Commands in PDS are English-like words and follow well-defined syntax rules.

[1] On systems with fewer resources, some tasks might not be permanently installed. Such systems might require you to use some form of the RUN command to install system tasks temporarily. See your system manager for further information. This manual assumes that all tasks are installed.

You are not required to use the full form of PDS commands, however. Usually, you need type only the command elements required to form a unique command. Most examples in this manual are in full format for clarity, but you should keep in mind that unless you are keeping a copy of your terminal activity for possible future reference, you can use much shorter forms than those in the examples. You will always be able to shorten any command or qualifier to four characters. Most commands and qualifiers can be entered with even fewer characters.

PDS prompts you for all required command elements. If you do not understand a prompt, type a question mark (?). PDS will print HELP text explaining the format and function of the command and then reprompt you for required input.

PDS is actually a CLI task that translates PDS commands into MCR commands for execution by the system. Depending on the kind of use you make of your system and the nature of your system, you may find it more convenient to use one CLI or the other, or both. All nonprivileged system functions are available directly from PDS, but some privileged functions are not. All program development facilities and all common utility functions are available from PDS.

The *IAS Guide to Program Development* concentrates on the actual program development process and not on PDS or MCR. However, some of the program development facilities require different commands in PDS and MCR. In particular, the commands for the MACRO-11 Assembler, the Librarian Utility Program (LBR), and the IAS task builder (TKB) are different. While this will probably not interfere with your use of the system, it may be confusing at first. The programming demonstrations in this manual are documented first using PDS commands, and then once again using MCR commands. This duplication enables you to see the differences between PDS and MCR.

1.1.2 Text Editors

A text editor is the means by which you create a source code file. Most IAS systems include the line text editor (EDI) and the DEC standard editor (EDT). Both of these editors are interactive editing programs that enable you to enter American Standard Code for Information Interchange (ASCII) text at a terminal and store the text in a disk file. They also let you access text in a disk file; examine, delete, and change text; and insert new text. The disk file is then used as input to other tasks in further steps of the program development process.

EDT is documented in the *EDT Editor Manual*.

You can use EDI or EDT or some other editor found at your installation with the examples in this book. PDS users invoke an editor in the following manner:

```
PDS> EDIT/EDI filespec RET
```

MCR users invoke an editor in the following manner:

```
MCR> EDI filespec RET
```

In both PDS and MCR, the EDT command line can include other command elements to invoke special features of EDT.

EDI may be the only editor available on smaller systems.

EDI is a single-pass, line-oriented editor. In its typical mode of operation, called block mode, it reads from a disk file a block of text—as much text as will fit in its text buffer. You perform editing operations on text in the EDI buffer. After editing text in the buffer, you request the editor to renew the buffer with the next block of text. To change text in a previously edited buffer, you must close the current editing session, open the file again by reinvoking EDI, and read from the beginning of the file to the block of text.

Editing functions are on a line-by-line basis. New text is inserted into the buffer one line at a time. Current text in the buffer is changed by your locating the line or lines on which EDI must make the change.

To preserve currently existing text, EDI performs all processing on a temporary copy of the file being edited. As you renew text in the buffer, EDI writes the edited text to a temporary file. This action has two advantages and one drawback. First, the current version of your text file is always left intact. Second, when you exit from the editing session, you have the option of storing the edited file in a new version of the old file or of creating an entirely new file (that is, one with a different name and version number). The drawback of the temporary file is that, in the event of a system crash, edits you are making are lost. After a crash, the new version of the file has a zero length because EDI did not have time to preserve the edits from the temporary file.

1.1.3 Assembly Language

IAS systems support many programming languages. However, the one language distributed on all systems is the PDP-11 assembly language, MACRO-11. MAC is the task that assembles MACRO-11 language files. It accepts a disk source input file in ASCII format and can create a relocatable object module and a listing file of the source language. The object module contains all the object records and relocation information needed to link with other object modules. All symbol definition done by the assembler has a base address of zero. The allocation of virtual addresses and relocation is left for the task-building process.

PDS users invoke MAC with the MACRO command. MCR users invoke MAC with the MAC command.

Source input to MACRO-11 consists of free-format statements, and each line of input contains a single statement. Input statements are either PDP-11 instructions, MACRO-11 Assembler directives, macro calls, comments, or direct assignments. Statements can contain labels to allow control to change locally (within the module) or to enable control to be passed between modules (globally).

Source input usually contains user-defined symbols, which are either local or global. A local symbol is defined in the current source file and is referenced only within the current file. A global symbol is defined in one source file but can be referenced in one or more other source files.

The assembler allows you to use both local and global symbols as labels for statements. When a global symbol appears as a label, the related statement is referred to as an entry point (that is, a point at which other modules can transfer control to the current object module). You can use local symbols as statement labels to define points to which control transfers within an object module.

The assembler evaluates all local symbol definitions in a source file. Any symbols remaining undefined are classified as global. Thus, after an assembly, all local symbols are assigned relative locations, but the module may contain references for which definitions must be supplied. The resolution of these references is left for the task-building process.

Assembler directives in a source file allow you to perform operations such as the following:

- Program sectioning
- Listing control
- Conditional assembly
- Data storage

Program sectioning allows code or data within an object module to be overlaid by, or concatenated with, code or data in other object modules or in noncontiguous locations within the same module. Program sectioning is especially useful where convenient physical ordering differs from logical reference ordering (for example, in table-generating macro statements). Listing control directives enable documentation features such as listing-heading lines, listing-page formatting, and table of contents generation. Conditional assembly directives allow optional omission or inclusion of lines of code or user-defined symbols. You can control the size and contents of data areas by using data storage directives.

Special statements called macro directives allow you to reference a predefined symbol that causes the assembler to expand a single line source statement into multiple lines of code or data and insert the assembled result in the object module. Such macro symbols are typically used for recurring coding sequences. The insertion of the code sequence occurs at each point where you refer to the macro symbol. Definitions for such macro symbols can occur in the source file itself or can reside in a macro library. Generally, you place infrequently used macro definitions in the source file that invokes them, and you store frequently used macro definitions in a macro library. The Executive and file-processing services are made available to the program through macro symbols that are defined in a DIGITAL-supplied macro library.

MACRO-11 is a 2-pass assembler. During the first pass, the assembler groups all symbols as either local or global, performs statement generation, locates all macro symbols, and, if necessary, reads the macro definitions from libraries. At the end of pass 1, the assembler must have processed all local references (such as all undefined global symbols) that are to be resolved by TKB.

During the second pass, the assembler actually generates the object module and listing files, and it flags with an error code in the listing file those source statements that contain errors. If you requested a cross-reference listing of symbols, the assembler also generates a request for the Cross-Reference Processor (CRF) to create the proper information. (CRF is introduced in Section 1.1.6.)

The MACRO-11 listing file provides both documentation for the module and a tool for debugging the code. As a reference aid, the assembler generates and includes line numbers in the listing for each statement in the source file. It also maintains a current location counter for each program section defined in the source file. In addition, the listing includes a symbol table showing symbols, their attributes, and their values if known at assembly time.

The location counter value given in the listing file is important in debugging because it provides the offsets into the module for each program section. An offset, combined with the base load address for a program section (from the TKB map), allows you to access locations in the memory-resident task image during debugging.

1.1.4 Task Creation

The task builder (TKB) on IAS systems is a multipurpose tool. It allows you to create a loadable entity (called a task image), define and structure a shared area of memory (called a resident common), and arrange shareable routines to reside in memory (called resident libraries). TKB has many complex aspects but this guide introduces only its most frequent usage: building a task image.

PDS users invoke TKB with the LINK command. MCR users invoke TKB with the TKB command.

To build a task image, TKB accepts, as basic input, the output of a language processor: an object module or multiple object modules. TKB can optionally generate a file of executable code (the task image), a file of memory allocation information (a map), and a special file of symbol definitions used in constructing the task (the symbol definition file). The task image, residing on disk, is in a

format suitable to be loaded into memory and executed. If you generate a cross-reference listing, the listing itself contains only global symbols and is appended to the map file.

In creating a task image, TKB's primary functions are linking, address binding, and building system data structures. Linking involves resolving global references in all object modules and resolving program section references among all object modules. Address binding is assigning virtual address space within the task. Building system data structures involves creating elements that the system requires to load the task image into memory and to execute the task. To resolve global symbols that are not defined in any of the input object modules, TKB searches any object libraries you specify and, as a default condition, searches the system object library.

Because the PDP-11 processor can address only 32K words (the address limit of 16 bits) at any one time, a task cannot reference more than 32K words at a time. However, if you use certain advanced programming techniques, TKB allows a task to access more code or data than can fit within the address limits. Techniques to overcome the addressing limits include the following:

- Overlaying segments of a task with either disk-resident or memory-resident code
- Mapping to different regions of memory outside the physical limits of the current task space

Because these are advanced techniques, they are not shown in the examples in this guide. For more information on them, refer to the *IAS Task Builder Reference Manual*.

The memory allocation information, or map, produced by TKB shows you how program sections are arranged in task memory (their starting virtual addresses and extents on mapped systems and physical addresses and extents on unmapped systems), what contributions are in a program section, any undefined symbols, and the optional cross-reference listing of global symbols. You can use the starting virtual addresses, combined with the current location counter values (provided by the assembler), as offsets to access locations within the memory-resident task during debugging.

1.1.5 Debugging Aids

This section introduces the debugging aids provided with IAS systems to assist in identifying faulty code.

1.1.5.1 On-Line Debugging Tool (ODT)

The On-Line Debugging Tool (ODT) allows interactive control of task execution. You specify to TKB that you want a debugging aid included in a task. TKB then inserts the module LB:[1,1]ODT.OBJ into the task.

When using the separate instruction and data space capabilities found in some IAS operating systems, TKB inserts the module LB:[1,1]ODTID.OBJ into the task.

When you run a task that includes ODT, execution begins at the ODT transfer address rather than at the task starting address. Therefore, ODT gains control and allows you to type special commands that establish base addresses and that set breakpoint locations within the task. After you tell ODT to begin task execution, ODT saves the instructions at breakpoint locations you specified and replaces them with PDP-11 breakpoint (BPT) instructions. Upon encountering a BPT instruction in the task, the Executive passes control to ODT at its breakpoint routine. ODT saves task registers in special locations, restores instructions to the breakpoint locations, and transfers control to the user task terminal. By typing ODT commands, you can examine and alter any instructions or data within task memory.

ODT also enables the BPT synchronous system trap (SST) entry point in the task. If a task generates an SST error, ODT gains control at its SST entry point, prints a notice at the user terminal, and passes control to the terminal. You can use the ODT commands to discover the cause of the error, correct it, and perhaps continue executing the task.

To successfully modify instructions, you must have a thorough understanding of the PDP-11 instruction set. If you are programming in a high-level language, you should avoid interactive debugging whenever possible.

1.1.6 General Utilities

This section introduces the general-purpose utility programs that are mentioned in this guide.

1.1.6.1 Cross-Reference Processor

The Cross-Reference Processor (CRF) is an installed task that receives requests from MACRO-11 and TKB to generate cross-reference listings of symbols. CRF generates a specially formatted file containing the cross-reference data and appends that file to the assembler listing or the task map file. Therefore, if you request a cross-reference listing of symbols, it always appears at the end of a listing or map file.

A request for the services of the CRF is included in your command line to the MACRO-11 Assembler and TKB. From PDS, use the /CROSS_REFERENCE qualifier to the MACRO and LINK commands. From MCR, use the /CR switch on the proper file specification in the MAC and TKB command lines.

1.1.6.2 Peripheral Interchange Program

The Peripheral Interchange Program (PIP) is the standard DIGITAL program for performing file- and device-related functions: transferring files from one medium or directory to another, obtaining directory listings, renaming files, deleting files, and changing file protection codes. PIP handles all Files-11 file-structured devices and is used for almost all file operations. The noteworthy exception to PIP capabilities is certain PDP-11 Record Management Services (RMS-11) file operations, for which DIGITAL supplies special RMS-11 utilities.

MCR users access PIP services through various PIP commands. PDS includes the DIRECTORY, DELETE, PURGE, COPY, RENAME, TYPE, APPEND, and SET PROTECTION commands, which give you transparent access to PIP services.

1.1.6.3 Queuing and Spooling

In IAS systems, almost all program development tasks automatically generate requests to the proper queuing tasks to print an ASCII output file on the system's default printer. If your installation has the proper tasks installed, the spooling task dequeues such requests and prints the requested output file on the proper device. You should consult the system manager at your installation for the exact details.

1.1.6.4 Librarian Operations

The Librarian Utility Program (LBR) can create and maintain specially formatted library files on disk: one for macro call definitions and one for object module subroutines.[2] The MACRO-11 Assembler and TKB can access these library files and extract the proper code from them. Libraries are convenient to use because they encourage sharing of code, provide faster access to multiple modules (only one file need be opened and closed), occupy less space than the equivalent number of separate modules, and impose a coding standard. The library files you create using LBR are in the

[2] The Librarian can also create a universal library file to contain any of one file type you prefer.

same format as those that DIGITAL supplies with the operating system. For more information, refer to the *IAS Utilities Manual*.

PDS includes a series of LIBRARY commands that give you access to LBR services. MCR users access LBR services through various LBR commands.

1.2 DIGITAL-Supplied System Software

DIGITAL supplies system software in two standard library formats: macro call definitions and object module subroutines. You use macro libraries as input to the assembler, and you use object libraries as input to TKB. The following two subsections describe these system libraries.

1.2.1 System Directives—Macro Libraries

DIGITAL makes available system directives and system-related features through calls; definitions for these calls reside in macro libraries. The libraries are stored in a predefined file area known as a directory. The directory is [1,1] on the system library device (referenced explicitly by the device-independent designation LB).

Table 1–1 DIGITAL-Supplied Macro Libraries

File Name and Type	Description of Contents
IASMAC.SML	System Macro Library. Contains the macro definitions for all IAS system directives and file control service (FCS) file-processing calls. Default library for the assembler.
EXEMC.MLB	Executive Macro Library. Contains the symbol and offset definitions for the Executive data structures.
RMSMAC.MLB	PDP-11 Record Management Services (RMS-11) Library. Contains the definitions for the RMS-11 calls for sequential, relative, and indexed file I/O. If your system has the optional RMS-11K software, this library will also contain calls for indexed file operations.

To use these libraries, you should follow the specific procedures described in the system documentation. Typically, you supply in the source code the appropriate names of the modules as parameters of a .MCALL MACRO-11 directive. This action tells the assembler to generate an entry for that call in its macro symbol table and to search the appropriate library for the definition of the macro symbol.

In translating source code, the assembler first checks for macro symbols. When the assembler finds an operator on a source line, it searches its macro symbol table to see whether the operator is a macro symbol. An operator is any PDP-11 operation code, MACRO-11 Assembler directive, or macro symbol. If the operator is a macro symbol, the assembler applies the local definition for the macro symbol or extracts the definition from a library you specified or from the system library. By searching the user-supplied library first, the assembler allows you to tailor the definitions of system macro calls or PDP-11 instructions. MACRO-11 assembles the macro definition with any accompanying parameters and includes the assembled code in the object module. As a result, the proper code is included from a library.

Through the use of the System Macro Library, you are provided with the code that enables a task to issue system directives and to obtain the File Control Services (FCS). These services enable a task to obtain run-time and system information, perform input/output functions, communicate with other tasks, manipulate logical and virtual address space, control execution, and properly exit. In general, most IAS features are made available to a task through macro calls to the System Macro

Library. For the system macro library RSXMAC, you need not designate the library name to the assembler. As a default condition, the assembler automatically searches the System Macro Library.

Through the use of the Executive macro library EXEMC.MLB, you are provided with code to allow software to refer to offsets within the Executive and system definitions of the Executive data structures. This library is provided for assembling privileged tasks and for incorporating specially written device drivers into the system.

The Record Management Services library RMSMAC.MLB is provided to support file and record access to RMS-11 data. RMS-11 is an upward-compatible extension of FCS and offers more functions such as indexed sequential (keyed) access to data. You include the RMS-11 macro symbols in the source code and supply to the assembler the name of the RMS-11 library to use. The assembler extracts the definitions from the library and includes the RMS-11 code in the object module.

1.2.2 System Subroutines—Object Libraries

On IAS systems, system object libraries provide general utility functions and special-purpose Executive features. These libraries, like the macro libraries, reside in directory [1,1] on the system library device (LB). Table 1–2 lists and describes the object libraries that DIGITAL supplies.

Table 1–2 DIGITAL-Supplied Object Libraries

File Name and Type	Description of Contents
SYSLIB.OLB	System Library. Contains register handling, arithmetic, data conversion, output formatting, File Control Services (FCS), and FCS command line processing subroutines. Optionally contains a set of real-time data acquisition routines. Default library for TKB.
VMLIB.OLB	Virtual Memory Management Library. Contains dynamic memory, core allocation, virtual memory, and page management subroutines.
EXEC.OLB	Executive Library. Contains the definitions of the Executive symbols.
RMSLIB.OLB	Record Management Services Library. Contains the routines for RMS-11 sequential, relative, and indexed file I/O.

You typically include system object routines in a task by specifying the routine name as the operand of a CALL macro or Jump To Subroutine (JSR) instruction in the source code. The language processor, at the point of the reference, generates the instructions to transfer control to the external subroutine. The name of the subroutine is left as an externally defined global symbol for TKB to resolve.

To ensure that subroutines are placed in the task image, TKB, as a default operation, searches the library SYSLIB.OLB for routine names that remain undefined after the search of any user-specified libraries. TKB attempts to match the undefined global reference (the subroutine name in a module) with an entry point name in the SYSLIB library. When it finds a match, TKB extracts a copy of the module that defines the symbol from SYSLIB and inserts the subroutine in the task image. Any further references to that symbol in the task are defined by the subroutine, and TKB need not add any code to resolve further references.

If a module references routines that are in an object library other than SYSLIB.OLB, you must specify that library when you build the task. TKB performs the same search operations on user-supplied libraries as it does on the default search of SYSLIB. TKB also searches any user-specified libraries in the order in which you specify them before it searches the system library.

1.3 Hardware for Program Development

Basically, you need three types of devices for program development: disks, terminals, and printers. This section briefly introduces these devices and tells where you can find further information. In general, each hardware unit on the system is delivered with relevant hardware documentation that provides programming information in addition to operational instructions. Your installation should have a library of such hardware documentation. If you are not writing any specially tailored code for these devices, the system software handles them transparently through such mechanisms as the print spooler and PIP.

1.3.1 Disks

Disks are the main storage media on IAS systems. Disk drives are either public (that is, accessible to all users) or private (that is, accessible to a restricted set of users). Almost all utility programs work with disks as their default device. You can share public disk resources to create source program files and, as needed, allocate your own private drive to store reserved copies of source and documentation files.

1.3.2 Terminals

Terminals are the means by which you communicate with the system. DIGITAL terminals handle 7-bit ASCII characters, and system software usually ignores any eighth, or parity, bit. You perform input to the system through a typewriter-like keyboard; the system returns output to you either on a screen at a video-display terminal or on paper at a hardcopy terminal. Video-display terminals are more convenient because they typically operate at faster rates than hardcopy devices. Hardcopy terminals, however, have the advantage of providing a record of what transpired during a session on the system.

Terminals are connected to the computer through either a direct line or a modem unit over a dial-up telephone line. The *IAS MCR User's Guide* explains how to access the system and basic system functions using MCR.

1.3.3 Printers

Printers provide hardcopy output of data. MCR provides a QUE command and PDS provides a PRINT command on systems with the QMG. On smaller systems, you may have to specify explicitly that output is to go to a line printer. All systems have a terminal or other output device serving as a line printer. All listings from the MACRO-11 Assembler or TKB are queued to the system line printer.

1.4 The Program Development Process—Overview

Figure 1–1 illustrates the steps in the program development process. The following paragraphs briefly describe these steps, which are treated in greater detail in Chapters 2 to 7.

The steps normally taken to prepare a program to run on the system are as follows:

1 Create a source program in a file on disk.

2 Submit the source file to a language processor (assembler or compiler) to produce an object module.

3 Submit the file (or files) containing the object or modules to TKB to create a file containing a loadable task image.

4 Request the Executive to execute the task.

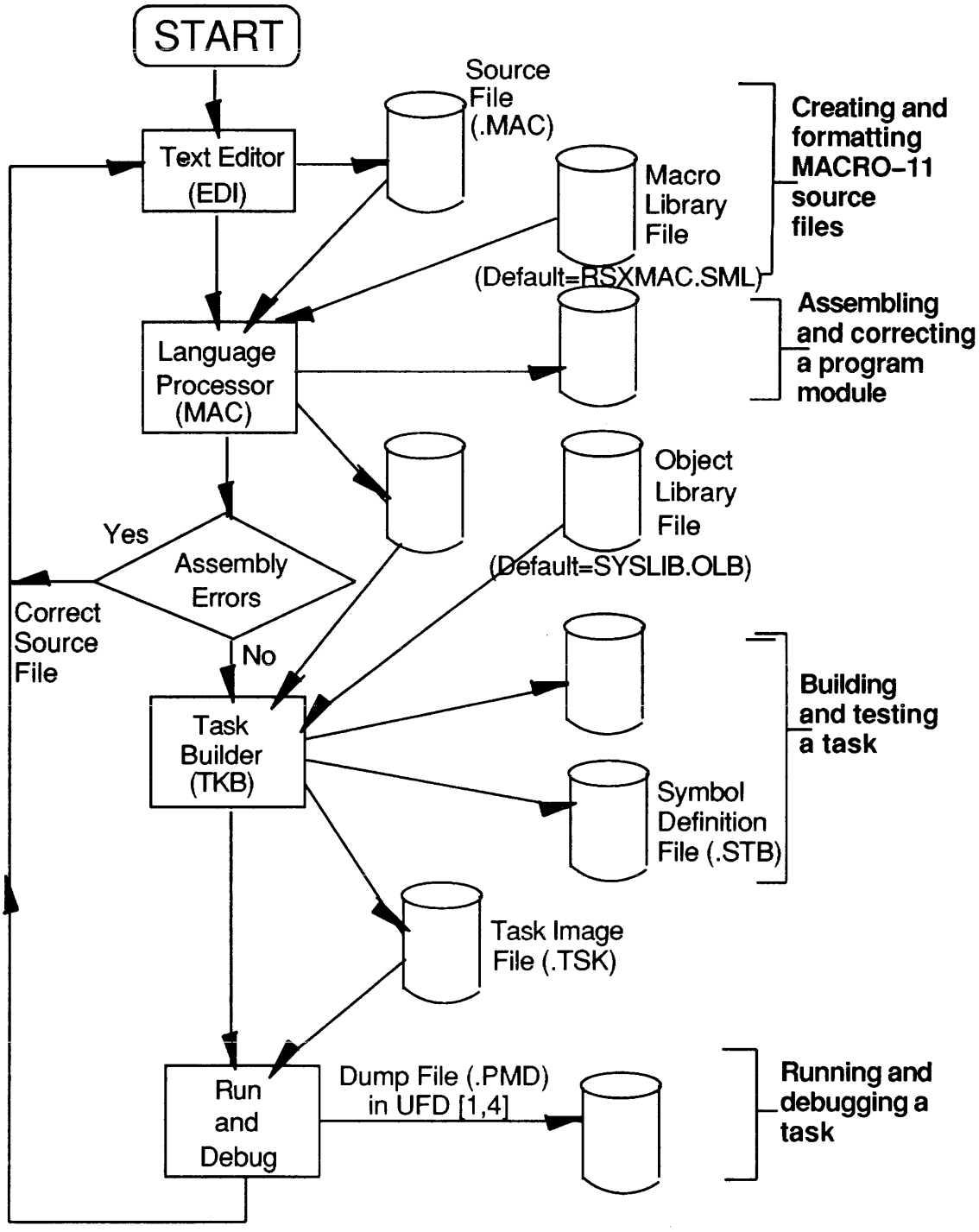
You use a text editor to create the source file. For MACRO-11 programmers, this guide suggests a skeleton format for source files and shows how to replicate and modify the skeleton file. The skeleton file becomes a common base from which you create each new source file.

A language processor creates the file of relocatable object code. For assembly language processing, MACRO-11 also accesses the System Macro Library to include code for system directives in the object file. For compilers, system directives are invoked by calls to subroutines in the system object library SYSLIB.

TKB creates the file of loadable code, which assumes certain default conditions about the run-time environment and builds these characteristics into the task. TKB also accesses system and user-specified libraries to resolve references in the task.

Once you have a task image, you request the Executive to run the program. If any errors occur, recompile, build a new task image file, and try again.

Figure 1–1 The Program Development Process



2 Creating MACRO-11 Source Files

Your first step in program development is to create a file that contains MACRO-11 source statements. One way to do this is to create a skeleton source file that you can use as a framework for all your source programs. This chapter performs the following functions:

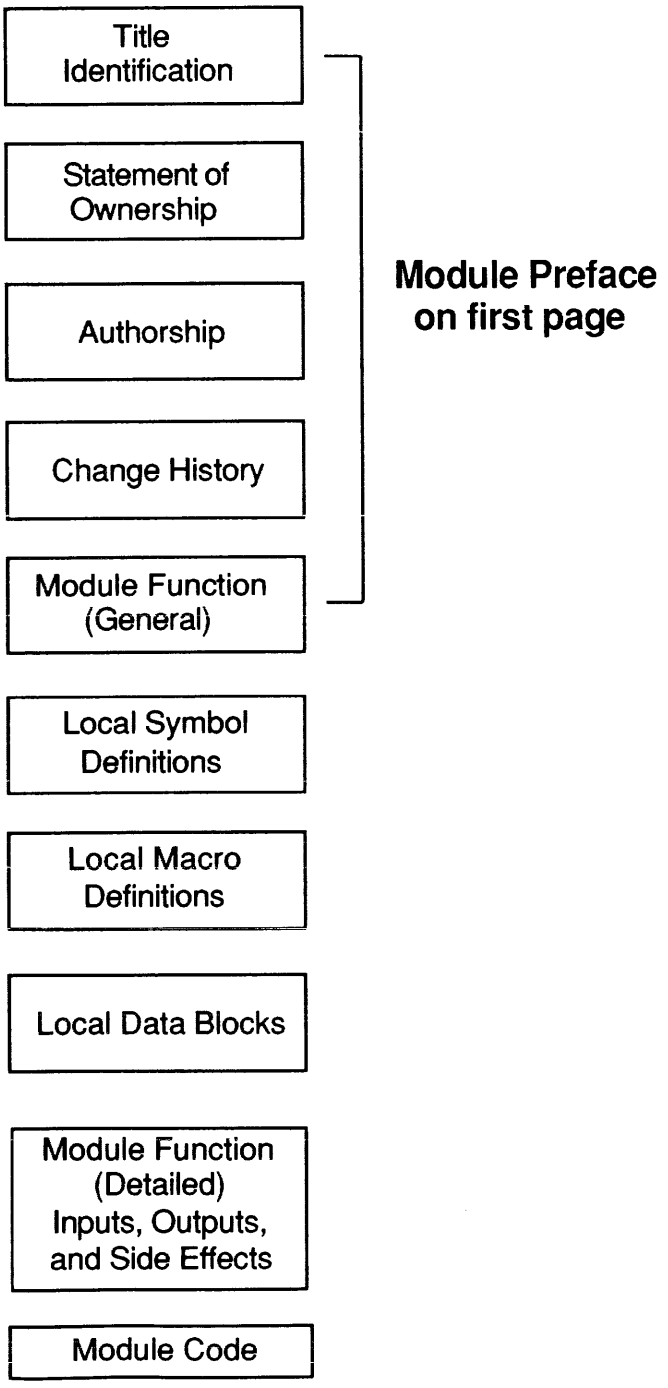
- Describes a source file format you can use as a guideline to create your own skeleton file.
- Presents some MACRO-11 statements to include in the file.
- Explains some elementary editing commands that you can use to create and modify source files.

Digital has established a coding standard to enhance the readability and maintainability of its MACRO-11 source programs. That standard is outlined in an appendix of the *PDP-11 MACRO-11 Language Reference Manual.*

2.1 MACRO-11 Skeleton Source File Format

This section presents the skeleton and source statement formats and discusses each of the elements in the skeleton. Figure 2–1 illustrates the basic elements of the skeleton: a preface, definitions, functional descriptions, and the code itself.

Figure 2–1 MACRO-11 Source File Format

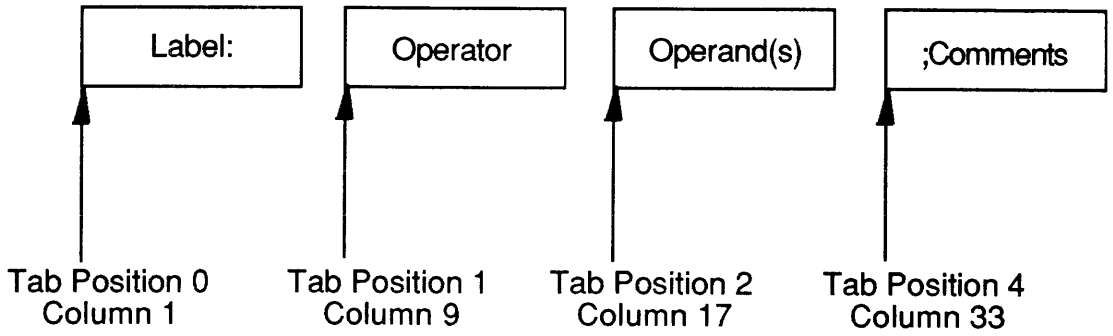


The source file preface, or preamble, should be on the first page. The preface essentially describes the code, states its ownership, identifies the author, defines the changes to the code, and gives a brief description of the module's function.

After the preface of the module comes the detail of the code. Declarations (such as local symbol, macro, and data definitions) that appear toward the beginning of the code make reading the code easier. Preceding the routines in the module, you should place detailed descriptions of what the routines do and define what is required for input to the routines, what the routines produce, and what effects result from execution.

Each statement line in a source file should follow a consistent format, as shown in Figure 2–2.

Figure 2–2 MACRO-11 Source Statement Format



Although the assembler enables free formatting of statements, you should follow the recommended format because it is easy to follow and creates readable, consistent code.

In the source statement format shown in Figure 2–2, the label is any user-defined symbol that identifies a reference location in the code. An operator is any PDP-11 operation code, MACRO-11 Assembler directive, or macro symbol. An operand is any argument(s) or parameter(s) of an operator. Comments consist of information you provide to describe what effect you desire from the execution of the instruction. Comments do not affect program execution; the assembler merely transfers them to the listing file produced during the assembly.

Comments, accompanied by selected MACRO-11 Assembler directives, constitute the source file skeleton. This skeleton provides the structure on which you build the source file. Directives in the source file skeleton identify the code and control the format of the listing. Example 2–1 shows a sample skeleton.

Notes on Example 2–1:

❶ .TITLE Directive

The .TITLE directive enables you to name the module. The assembler takes the first six nonblank (alphanumeric) characters, up to the first blank or horizontal tab character, as the module name. Following the name in the .TITLE directive, you can use up to 24 characters to describe the function of the module. The name and the description appear as the first entry in the header line of each page in the assembly listing. For example, consider the following .TITLE directive:

```
.TITLE SKELTN SOURCE FILE SKELETON
```

Example 2–1 Sample Source File Skeleton

```
        .TITLE  SKELTN SOURCE FILE SKELETON  ❶
        .IDENT  /01/  ❷
;
;
; AUTHOR: Z  ❸
;
;
; CHANGES:  ❹
;
;
; MODULE FUNCTION GENERAL:  ❺
;
;
        .PAGE                   ; BREAK PAGE FOR PREFACE  ❻
        .SBTTL  SYMBOL, MACRO, DATA DEFINITIONS  ❼
        .LIST   TTM  ❽
        .NLIST  BEX             ;SUPPRESS BIN EXTENSION  ❾
        .MCALL  EXIT$S          ;EXEC'S EXIT MACRO  ❿
;
; LOCAL SYMBOL DEFINITIONS:  ⓫
;
;
; LOCAL MACROS:  ⓬
;
;
; LOCAL DATA BLOCKS:  ⓭
;
        .PSECT  DATA,D,RW  ⓮
;
; MODULE FUNCTION DETAILED:  ⓯
;
;
;       INPUTS:
;
;       OUTPUTS:
;
;       SIDE EFFECTS:
;
; START CODE HERE
        .PAGE
        .SBTTL
        .PSECT
START:
END:    EXIT$S                  ;EXIT CLEANLY TO EXEC
        .END                    ;TELL ASSEMBLER END OF CODE  ⓰
```

The assembler takes the characters SKELTN as the module name. The remaining characters up to the 30th character are taken as the description. Any remaining characters after the 30th character would be discarded.

The assembler does not relate the name you specify in the .TITLE directive to the name you specify for the source or object files. To minimize confusion, however, it is helpful to apply the name specified in the .TITLE directive to the source file. (Note that the sample code and commands shown in this guide use different names to help you distinguish their usage.)

The name the assembler extracts from the .TITLE directive is also important in subsequent steps of program development. The task builder (TKB) lists this name in its memory allocation synopsis to show which object modules made contributions to each program section in the task image. In addition, if the LIBRARY command is used to insert the object module in an object library, this name is kept in the directory of the library to refer to the object module.

❷ .IDENT Directive

The .IDENT directive records the version of the module. You can establish your own version identification conventions. The identification follows the module into the task image and is displayed in the map. Knowing whether the correct version of the module was linked into the task image helps in the debugging and maintenance process.

❸ Author Line

The author line identifies the originator of the code.

❹ Changes

This section of the source file describes any modifications that have been made to the module. You can develop a convention whereby the author's initials and a number can indicate a change. The author of the modification can identify the change in this section and flag each line of code with an additional comment, such as the following:

```
; TOM JONES      8-AUG-90        1.01
; TJ001          ADD STATE TAX TO TOTAL
```

A changed or added line in the code can be flagged with the notation TJ001 as follows:

```
ADD    A,B             ;TOTAL WITH TAX   ;TJ001
```

This procedure helps the author recall what changes were made to the module and assists others in determining the extent of changes.

❺ Module Function General

In the module function part of the source file, you can describe the general processing operations that the code performs. This description can include how the module relates to the system or specific application, that is, what type of processing precedes and follows the execution of this module.

❻ .PAGE Directive

The .PAGE general-purpose directive causes a page break in the assembly listing. It appears as shown to keep the preamble alone on the first page of the listing (after the table of contents). You can use the .PAGE directive throughout the module to generate page breaks for different subroutines.

❼ .SBTTL Directive

The .SBTTL general-purpose directive creates an entry for the assembly listing table of contents printed at the front of the listing. A table of contents is helpful in summarizing the subroutines in a large module. Therefore, the text you supply with the directive should describe what the related subroutine does. In addition to appearing in the table of contents, the text appears on the second line of the heading at the top of each listing page. If your modules typically contain only a small number of subroutines, you might not find the table of contents feature very useful.

❽ .LIST TTM Directive

The .LIST TTM general-purpose directive creates a listing formatted more conveniently for output on a terminal. (Chapter 3 shows how to display a listing at a terminal.) You can include the directive during the early stages of program development and later remove it from the stabilized code.

❾ .NLIST BEX Directive

The .NLIST BEX general-purpose directive suppresses the binary extension of statements beyond what can fit on one source statement line. Using this directive saves excess printing in the assembly listing. For example, only the binary value of the first characters of an ASCII string would appear in the listing. The directive simply makes the listing more readable, and it saves paper.

❿ .MCALL Directive

Use the .MCALL general-purpose directive to tell the assembler the names of the externally defined macro calls that appear in the source file. The directive causes the assembler to create entries in its macro symbol table for the macro names and to look up the definitions of the related calls in either a user or a System Macro Library. The assembler includes the definitions from the library in the module where the calls themselves appear.[1]

The EXIT$S directive (shown in the .MCALL statement) should be in every user program for a clean exit. It is the last statement the program (task) executes before it returns control to the Executive. (The EXIT$S directive performs important system housekeeping operations for the task.) The related definition for EXIT$S resides in the file IASMAC.SML in system directory [1,1] on the library device (LB). Digital recommends that user tasks exit by using the EXIT$S directive. (An alternative form of exiting enables a task to exit and post status.)

If a call for an externally defined macro statement appears in the source file but is not preceded by a .MCALL directive and the macro name, the assembler treats the unrecognized macro call as an implicit .WORD data storage directive. (If the macro call has parameters, the assembler might generate an error because of illegal syntax for a .WORD directive.) Later, when you build the task with the related object module and the macro name is not a valid symbol, TKB flags the name as an undefined reference. Thus, without the .MCALL directive, the assembler does not know that it must search libraries to resolve the macro symbol.

⓫ Local Symbol Definitions

In this section of your source file, you collect symbols in direct assignment statements. Because symbols in MACRO-11 can be defined as expressions of other symbols, having the definitions in one place is an advantage. In addition, good programming practice encourages using symbols instead of simply supplying a numeric constant.

For example, in defining a 10-byte buffer, the best method is to define a symbol, then use the symbol in the buffer definition, as follows:

[1] If you do not include the directive .LIST ME (list macro expansions) or .LIST MEB (list macro expansion lines that generate object code) in the source file, the assembler does not insert in the listing the expanded source code of the macros it assembles.

```
;
; LOCAL DEFINITIONS
;
SIZB = 10.
.
.
.
;
; LOCAL DATA BLOCKS
;
BUFB:   .BLKB    SIZB
```

This method has the following advantages:

- If a single constant that is referred to in numerous places in the code must be altered, you need perform only one edit (to the symbol definition) to effect the change.
- If all the symbols are gathered in one place in alphabetical order, reading the code is simplified.
- You can find all references to a symbol in a cross-reference listing. The cross-reference capability allows you to examine all the references to a symbol and confidently assess the effects of altering the symbol definition.

These advantages are lost if you use constants. Thus, the symbol list would contain such local symbol definitions as SIZB = 10. The symbols themselves would appear in the module code.

⓬ Local Macro Definitions

The definition of a macro statement can appear anywhere in the source file as long as the definition appears before the first occurrence of the macro statement. It is better programming practice to place all macro definitions in a standard place near the front of the source file.

⓭ Local Data Blocks

This section of the source file defines such data as buffers, status words, and status bytes. Generally, it describes the local storage that the module references. It is good programming practice to use a separate .PSECT directive for data.

⓮ .PSECT Directive

The .PSECT directive establishes a name and attributes for a program section. A program section is a unit allocation of memory reserved for either code or data. For example, you can establish a program section to contain data for your program as follows:

```
.PSECT DATA,D,RW
```

The .PSECT directive creates the program section named DATA with the attributes data (D) and read/write (RW). You can give a program section for data either the read-only (RO) or the read/write (RW) attribute. (The assembler applies other default attributes not relevant to this discussion.) Consult the *IAS Task Builder Reference Guide* for a discussion of program section allocation in multiuser tasks.

The three most important aspects of the .PSECT directive are as follows:

- Contributions defined for a specific program section can be in separate places in a source file or in separate source files.
- Attributes of the program section are passed to TKB.
- Contributions for a specific program section with the same attributes are collected in one continuous allocation of memory space by TKB.

In the skeleton file, it is useful to define one program section to contain the data elements referenced in the task and to define another program section to contain the code.

⑮ Module Function Detailed

This section of the source file can be as general or specific as necessary to describe the functions of the module. A complex module should have a lengthy discussion; a simple module need not have as much. At the minimum, this section should state the register usage on input to and output from the module.

⑯ .END Directive

The .END directive in a module signals the logical end of source input and optionally specifies the task transfer address. The transfer address is the location where program execution begins. Although each source file should contain an .END directive, only one source file should define the transfer address. The assembler does not process lines beyond the one where the .END directive appears.

2.2 Creating a Source File from a Skeleton File

This section describes how to use the line text editor (EDI) to create a skeleton file, then to create a source file from the skeleton. If you are using the DEC standard editor (EDT) or some other editor, follow the text for EDI and perform the functions using your editor.

2.2.1 Performing the Initial Input

To create the skeleton file using MCR, type EDI and the specification of a new file (that is, one not in your directory). For DCL, use the command EDIT/EDI and the specification of a new file.

The sequence from a DCL terminal is as follows:

```
DCL> EDIT/EDI SKEL.MAC RET
[CREATING NEW FILE]
INPUT
```

The sequence from an MCR terminal is as follows:

```
MCR> EDI SKEL.MAC RET
[CREATING NEW FILE]
INPUT
```

The editor performs the following functions:

- Runs.
- Determines that the file does not exist.
- Creates the file.
- Tells you to begin typing the input.

Type the input as in Example 2–2. Leave any typographical errors until after you have become familiar with the editing commands described in Section 2.3. The notation conventions for the figure are described in the Preface at the front of this guide.

2.2.1.1 Inserting Blank Lines in Text

To insert a blank line in the source file as shown in Example 2–2, press the space bar or [TAB] key on a new line followed by the [RETURN] key. If you press the <BPX>(RETURN) key twice in succession (that is, press the [RETURN] key to enter a line of text and immediately press the [RETURN] key again on the new line, EDI terminates the input. Thus, to enter a blank line, press only one nonprinting character, such as the [TAB], on a new line.

2.2.1.2 Terminating the Input and the EDI Program

To terminate the input, press the [RETURN] key twice in succession. EDI prints the asterisk (*) to request a command. Type the EXIT command to close the file and terminate EDI. For example:

```
last line of text [RET]
[RET]
* EXIT
[EXIT]

MCR>
```

When EDI exits, it prints the message [EXIT] and returns control to the operating system. The implicit prompt (>) indicates that the command line interpreter (CLI) is ready to accept a new command. The remainder of this chapter makes changes in the source file SKEL.MAC and demonstrates EDI commands at the same time. If you are using EDT or some other editor, follow the changes as demonstrated and make them in your file using your editor. If you are using DCL, remember to use the EDIT/EDI or EDIT/EDT command.

Example 2–2 Creating the Skeleton File SKEL.MAC

```
$ EDT SKEL.MAC [RET]
Input file does not exist
[EOB]
* c [RET]
[TAB]       .TITLE [TAB] SKELTN SOURCE FILE SKELETON [RET]
[TAB]       .IDENT [TAB] /01/ [RET]
; [RET]
; [RET]
; AUTHOR: Z [RET]
; [RET]
[TAB][RET]
; [RET]
; CHANGES: [RET]
; [RET]
[TAB][RET]
; [RET]
; MODULE FUNCTION GENERAL: [RET]
; [RET]
[TAB][RET]
; [RET]
[TAB]       .PAGE [TAB] [TAB] [TAB]          ; BREAK PAGE FOR PREFACE [RET]
[TAB]       .SBTTL [TAB] [TAB] [TAB]         ; SYMBOL, MACRO, DATA DEFINITIONS [RET]
[TAB]       .LIST [TAB] TTM [TAB] [TAB]      ;TERMINAL LISTING MODE [RET]
[TAB]       .NLIST [TAB] BEX [TAB][TAB]      ;SUPPRESS BIN EXTENSION [RET]
[TAB]       .MCALL [TAB] EXIT$S [TAB][TAB]   ;EXEC'S EXIT MACRO [RET]
[TAB][RET]
; [RET]
; LOCAL SYMBOL DEFINITIONS: [RET]
; [RET]
[TAB][RET]
; [RET]
; LOCAL MACROS: [RET]
; [RET]
[TAB][RET]
; [RET]
; LOCAL DATA BLOCKS: [RET]
; [RET]
[TAB]       .PSECT [TAB] DATA,D,RW [RET]
[TAB][RET]
; [RET]
; MODULE FUNCTION DETAILED: [RET]
[TAB][RET]
; [RET]
; [TAB]   INPUTS: [RET]
; [RET]
; [TAB]   OUTPUTS: [RET]
; [RET]
; [TAB]   SIDE EFFECTS: [RET]
; [RET]
[TAB][RET]
[TAB]       .PAGE [RET]
[TAB]       .SBTTL [RET]
[TAB]       .PSECT [RET]
```

Example 2–2 Cont'd on next page

Example 2–2 (Cont.) Creating the Skeleton File SKEL.MAC

```
; START CODE HERE [RET]
START: [RET]
[TAB][RET]
END: [TAB]  EXIT$S [TAB] [TAB] [TAB]        ;EXIT CLEANLY TO EXEC [RET]
[TAB]       .END [TAB] [TAB] [TAB]          ;TELL ASSEMBLER END OF CODE [RET]
[RET]
[GOLD][COMMAND]
Command:  EXIT
[ENTER]
$
```

2.2.2 Creating a Source File from the Skeleton

After you create the skeleton file, you can use it many times to create different source files by running the editor again as described in Section 2.2.1. For example:

```
MCR> EDI SKEL.MAC [RET]
[00054 LINES READ IN]
[PAGE 1]
*
```

This time EDI finds the file you just created, reads it into memory, and prints an asterisk to request a command.

The EXIT command with a file specification creates a new file that has that name and contains all the text in your skeleton. For example:

```
* EXIT FILE.MAC [RET]
[EXIT]

MCR>
```

EDI creates either the new file FILE.MAC;1 in your directory or, if the file already exists, a new version of the file. It retains the input file SKEL.MAC. You can repeat this process to create as many new source files as you need.

At this point, the contents of SKEL.MAC and your new file are exactly the same—typographical errors and all. Now you must use editing commands to change your new file to make it unique. Section 2.3 describes some of these commands and gives examples of their usage to enable you to perform the most common editing functions.

By using the same skeleton file each time you want to create a new source file, you save typing time and have a better chance of creating consistent, easily readable, and well-documented code. After you have gone through Section 2.3 and have learned the editing commands, you can correct the errors in the skeleton file.

2.3 Editing the Source File

This section describes how to use a subset of EDI commands to edit a source file. By following the examples in this section, you create three source files that you can use in subsequent stages of the program development cycle.

You can abbreviate most of the commands in EDI. For example, the EXIT command can be abbreviated EX. The descriptions of each command include (within parentheses) the accepted abbreviation, if one exists.

2.3.1 Displaying Text

Use the EDI command to access a source file to edit. For example:

```
MCR> EDI FILE.MAC [RET]
[00054 LINES READ IN]
[PAGE 1]
*
```

Two keys, [RETURN] and [ESCAPE], cause EDI to move forward and backward respectively, one line, and to display the new line. By using these two keys, you can step line by line through a file. For example:

```
*  [RET]
     .TITLE  SKELTN   SOURCE FILE SKELETON
*  [RET]
     .IDENT  /01/
*  [ESC]
     .TITLE  SKELTN   SOURCE FILE SKELETON
*
```

Pressing the [RETURN] key twice advances the pointer twice and displays each line. Pressing the [ESCAPE] key moves the pointer back to the previous line and displays the line.

The following sections describe several of the EDI commands used to display text.

2.3.1.1 TYPE Command

The TYPE command (TY n) displays n lines at a time but does not alter the line position. For example:

```
* TY 2
     .TITLE SKELTN  SOURCE FILE SKELETON
     .IDENT /01/
*
```

The 2 in the TYPE command causes EDI to display the current line and the next line. If you give the TYPE command without a number, EDI displays the current line (that is, one line).

2.3.1.2 LIST Command

The LIST command (LI) displays all lines in the buffer starting at the current line and stopping at the last line in the buffer (that is, end-of-buffer). For example:

```
* LI [RET]
     (all lines are listed)
* TY [RET]

[*BOB*]
* EX [RET]
[EXIT]

MCR>
```

The LIST command positions the line pointer at the beginning of the buffer. The TYPE command shows the position of the line pointer. EDI prints the blank line it maintains at the beginning of the buffer and the message [*BOB*] to remind you that the line pointer is at the beginning of the

buffer. EDI always keeps a blank line at the beginning of the buffer to allow you to insert lines before the first line of text in the buffer.

2.3.2 Locating Text and Positioning the Line Pointer

Editing a file requires you to locate a line of text in the buffer and to position the pointer to that line. The following sections describe several of the commands used in editing files.

2.3.2.1 BEGIN and END Commands

The BEGIN command (B) and END command (E) position the pointer to fixed lines in the buffer–the beginning and ending lines. The END command also prints the last line of the buffer. For example:

```
MCR> EDI FILE.MAC [RET]
[00054 LINES READ IN]
[PAGE 1]
* E [RET]
[TAB] .END [TAB] [TAB] [TAB]; TELL ASSEMBLER END OF CODE
*
```

The END command is useful for quickly assessing what the last line in the buffer is. The BEGIN command is helpful in quickly positioning the pointer at the beginning (or top) line of the buffer, thus enabling you to make multiple passes over a buffer. For example:

```
* B [RET]
* TY [RET]

[*BOB*]
*
```

Because the BEGIN command does not display any text, you can use the TYPE command to display the first line in the buffer. The command in the example shows the blank line at the beginning of the buffer. EDI prints [*BOB*] to show you that it is positioned at the beginning of the buffer. For example:

```
* [RET]
[TAB] .TITLE [TAB] SKELTN [TAB] SOURCE FILE SKELETON
*
```

Pressing the RETURN key advances the pointer and displays the line, as follows:

```
* [RET]
[TAB] .TITLE [TAB] SKELTN [TAB] SOURCE FILE SKELETON
* [RET]
    4
*
```

2.3.2.2 LOCATE Command

If the text you want to examine is within the buffer, you can type the LOCATE command (L) with a string to be located, as follows:

```
* L MODULE [RET]
; MODULE FUNCTION:
*
```

A space should separate the command and the search string to be located. EDI displays the line where it found the first occurrence of the string. If EDI does not find the string, it prints a message indicating that the end-of-buffer has been reached. For example:

```
* L MODULE [RET]
[*EOB*]
*
```

After an unsuccessful search, EDI leaves the line pointer at the last line of the buffer.

2.3.2.3 PLOCATE Command

If the string you want is not in the buffer, you can use the PLOCATE command (PL) to tell EDI to search successive buffers until it locates the string. For example:

```
* B [RET]
* PL .END [RET]
    .END [TAB] [TAB] [TAB]; TELL ASSEMBLER END OF CODE
*
```

EDI searches the buffer starting at the current line. If the string is not found in the buffer, EDI preserves the contents of the buffer and reads in more lines from the input file to fill the buffer again. It prints a message telling the number of lines searched. When EDI finds the string, it displays the line where the string occurs. If EDI does not find the string, it prints a message indicating that the end-of-file has been reached. For example:

```
* PL .ENDR [RET]
[*EOF*]
*
```

At the end-of-file (signaled by [*EOF*]), EDI leaves an empty buffer where you can either insert new text (which follows all the text currently in the file) or exit to preserve any changes made and to start at the beginning of the file again. Note that, once EDI has preserved a buffer, you cannot go back to it except by starting at the beginning of the file again. For example:

```
* EX [RET]
[EXIT]

MCR>
```

You can also use the PLOCATE command with a string known not to exist in the file to position EDI after the last line of the file.

2.3.2.4 RENEW Command

The RENEW command (REN) enables you to read new lines from the input file. For example:

```
* EDI FILE.MAC [RET]
[00054 LINES READ IN]
[PAGE 1]
* REN [RET]
[*EOF*]
[PAGE 2]
* EX [RET]
[EXIT]

MCR>
```

The RENEW command writes the lines in the buffer to the temporary output file before it reads in new lines from the input file. If there are no more lines left in the file, EDI signals the end-of-file. This command is useful for casually inspecting the contents of a file.

2.3.3 Changing Text and Exiting from EDI

While editing a file, you can alter text by replacing, adding, or deleting strings. The following sections describe several of the commands most commonly used in changing text.

2.3.3.1 CHANGE Command

The CHANGE command (C) alters text on the current line and enables you to perform the following tasks:

1 Replace an old string with a new string.

2 Add a string at the start of a line.

3 Delete a string from a line.

The command requires that you type, within character delimiters, the old string (the text to be altered) followed by the new string. The only requirement for the delimiting character is that it does not appear in either the old or the new string.[2]

A convenient character to use as a delimiter is the slash character (/). For example:

```
MCR> EDI FILE.MAC [RET]
[00054 LINES READ IN]
[PAGE 1]
*  [RET]
     .TITLE SKELTN    SOURCE FILE SKELETON
* C /SKELTN/NUMA/ [RET]
     .TITLE NUMA      SOURCE FILE SKELETON
```

After you enter the C command, EDI searches the line for the old string (SKELTN) and replaces it with the new string (NUMA). EDI then prints the changed line to allow you to verify the operation. If EDI cannot locate the old string, it prints the message [NO MATCH] and reprints the prompt.

To save typing long strings, EDI enables you to include an ellipsis (. . .) in the old string, as follows:

```
* C /SO...ON/COUNT NUMBER OF A'S/ [RET]
     .TITLE [TAB] NUMA [TAB] COUNT NUMBER OF A'S
*
```

EDI takes the characters SO, all intervening characters, and the characters ON as the old string. The ellipsis, used in this manner, reduces the amount of typing required to specify a string to be changed. Three other forms of the ellipsis enable variations of the abbreviation:

/ . . . /	By itself, the ellipsis means the entire line.
/old string . . . /	From old string to the end of the line.
/ . . . old string/	From the beginning of the line to old string.

The slash characters shown as delimiters with the ellipsis can be any unique character. To place a string at the beginning of a line, specify the null string as the old string. For example:

```
* C //OLD STRING/ [RET]
OLD STRING [TAB] .TITLE [TAB] NUMA [TAB] COUNT NUMBER OF A'S
*
```

EDI replaces the null string at the beginning of the line with OLD STRING and prints the changed line.

[2] The ampersand character (&) should not be used as a delimiter because EDI treats it as a concatenation character. If you must use it as a delimiter, follow the special procedures presented in the EDI chapter in the *RSX-11M/M-PLUS Utilities Manual* for using the CONCATENATION CHARACTER command (CC).

To delete a string from the line, specify the null string as the new string, as follows:

```
* C /OLD STRING// [RET]
[TAB].TITLE [TAB] NUMA [TAB] COUNT NUMBER OF A'S
*
```

EDI replaces OLD STRING with the null string; that is, it deletes OLD STRING and prints the changed line.

2.3.3.2 APPEND Command

The APPEND command (AP) adds a string at the end of a line. The command does not need delimiting characters, since only one string can be specified; simply specify a space to separate the command and the string. For example:

```
*  [RET]
                        .IDENT /01/
* AP [TAB]  ; IDENTIFY MODULE VERSION [RET]
                        .IDENT /01/ [TAB] ; IDENTIFY MODULE VERSION
*
```

After adding the text at the end of the line, EDI displays the changed line.

2.3.3.3 DELETE & PRINT Command

Specify the DELETE & PRINT command (DP n) to remove a line or lines from the text in the buffer, where n is the number of lines to be deleted. You can use the TYPE n command with the DP n command to display the lines to be deleted, as follows:

```
* TY 3 [RET]
;
;
;AUTHOR:Z
* DP 2 [RET]
;AUTHOR:Z
*
```

The TY 3 command displays the current line and two succeeding lines (the pointer remains positioned at the current line). The DP 2 command deletes the current line and one succeeding line. EDI moves the pointer to the line after the last one deleted and prints that line.

2.3.3.4 EXIT Command

Use the EXIT command (EX), after changing text in the file, to close the editing session. For example:

```
* EX [RET]
[EXIT]

MCR>
```

The EXIT command without a file name creates a new version of the current file and copies the remainder of the file to the new version. Because exiting preserves the edits you have made to that point, you should exit fairly often from a lengthy editing session. If a system crash occurs, EDI retains the old version of your file (that is, it retains the edits up until you last exited) but does not retain the changes you are making. Frequent exits minimize the amount of editing that can be lost if a system crash occurs.

2.3.4 Inserting Code in the Source File

Use the INSERT command (I) to add multiple lines of text in the source file. To insert code in the source file FILE.MAC, use positioning commands to locate the line preceding where you want to place the new material. The INSERT command places new lines in the buffer after the current line. For example:

```
MCR> EDI FILE.MAC [RET]
[00052 LINES READ IN]
[PAGE 1]
* L FUNCTION: [RET]
; MODULE FUNCTION:
* I [RET]
;       THIS MODULE LOADS A BUFFER, [RET]
;       COUNTS THE NUMBER OF A'S (UPPER) [RET]
;       CASE ONLY  IN THE BUFFER, CONVERTS [RET])
;       THE NUMBER TO OCTAL, AND REPORTS [RET]
;       THE NUMBER OF A'S FOUND. [RET]
[RET]
*
```

The LOCATE command positions EDI to the line preceding where you want to place the new lines. Typing the I command followed by pressing the RETURN key places EDI in insert mode. After you type the lines, press the RETURN key twice in succession to leave insert mode.

Continue using positioning and editing commands to type in the remainder of the source program shown in Example 2–3.

Example 2–3 Source Code for FILE.MAC

```
        .TITLE  NUMA    COUNT NUMBER OF A'S
        .IDENT  /01/            ; IDENTIFY MODULE VERSION
;
;
; AUTHOR: Z
;

;
; CHANGES:
;

;
; MODULE FUNCTION GENERAL:
;       THIS MODULE LOADS BUFFER,
;       COUNTS THE NUMBER OF A'S (UPPER
;       CASE ONLY) IN THE BUFFER, CONVERTS
;       THE NUMBER TO OCTAL, AND REPORTS
;       THE NUMBER OF A'S FOUND.
;

        .PAGE                   ; BREAK PAGE FOR PREFACE
        .SBTTL  SYMBOL, MACRO, DATA DEFINITIONS
        .LIST   TTM             ; TERMINAL LISTING MODE
        .NLIST  BEX             ; SUPPRESS BIN EXTENSION
        .MCALL  EXIT$S          ; EXEC'S EXIT MACRO
```

Example 2–3 Cont'd on next page

Example 2–3 (Cont.) Source Code for FILE.MAC

```
;
; LOCAL SYMBOL DEFINITIONS:
        MSGLEN  = NUMEND-MSG
        SIZ     = 80.
        SIZA    = 6.
;

;
; LOCAL MACROS: NONE
;

;
; LOCAL DATA BLOCKS:
;
        .PSECT  DATA,D,RW

A:      .ASCII  /A/             ; DEFINE AN A
BUF1:   .BLKB   SIZ             ; DEFINE BUFFER
MSG:    .ASCII  /THE NUMBER OF A'S IS /
NUMA:   .BLKB   SIZA            ; DEFINE OCTAL COUNT
NUMEND  = .                     ; END OF MESSAGE
NUMC:   .BLKW   1               ; NUMBER OF CHARS TYPED
;

;
; MODULE FUNCTION DETAILED:

;       INPUTS:
;               BUF1 IS LOADED WITH CHARACTERS
;
;       OUTPUTS:
;               NUMA HOLDS THE NUMBER OF A'S
;
;       SIDE EFFECTS: NONE
;

; START CODE HERE

        .PAGE
        .SBTTL  ROUTINE TO COUNT A'S
        .PSECT
START:
        MOV     #BUF1,R0        ; LOAD BUFFER ADDR
        MOV     #SIZ,R1         ; LOAD BUFFER SIZE
        CALL    READ            ; READ FROM TTY
        TST     R2              ; ANY CHARS IN BUFFER?
        BEQ     END             ; IF NONE, FINISH UP
        CLR     R1              ; INIT # OF A'S COUNTER
        MOV     R2,NUMC         ; SAVE # OF CHARS TYPED
10$:
        CMPB    (R0)+,A         ; IS CHAR = A?
        BNE     20$             ; IF NO, GET NEXT CHAR
        INC     R1              ; COUNT AN A
20$:
        DEC     R2              ; ONE LESS CHAR
        BNE     10$             ; IF MORE, COMPARE NEXT
        .PAGE
        .SBTTL  TRANSLATE COUNT TO OCTAL
        MOV     #NUMA+6,R0      ; SET PTR TO OCTAL #
        MOV     #5,R2           ; SET COUNT OF DIGITS
```

Example 2–3 Cont'd on next page

Example 2–3 (Cont.) Source Code for FILE.MAC

```
30$:
        MOV     R1,-(SP)        ; STACK IS TEMP AREA
        BIC     #177770,@SP     ; STRIP LOW 3 BITS
        ADD     #60,@SP         ; MAKE OCTAL DIGIT
        MOVB    (SP)+,-(R0)     ; STORE OCTAL DIGIT
        ASR     R1              ; SHIFT TO
        ASR     R1              ;       NEXT
        ASR     R1              ;                3 BITS
        DEC     R2              ; ONE LESS DIGIT
        BNE     30$             ; IF MORE, REPEAT
        MOV     #MSG,R0         ; LOAD ADDR OF BUFFER
        MOV     #MSGLEN,R1      ; LOAD SIZ OF MESSAGE
        CALL    WRITE           ; REPORT THE RESULTS
END:    EXIT$S                  ; EXIT CLEANLY TO EXEC
        .END                    ; TELL ASSEMBLER END OF CODE
```

After you have typed in the code, use the techniques described previously to create two new source files, FILEA.MAC and FILEB.MAC, from the skeleton file. The code for these two files is shown in Examples 2–4 and 2–5. These two files and the file FILE.MAC are used in Chapter 4 to build and test a task. You might want to edit the skeleton file before you create the two new source files.

Example 2–4 Source Code for FILEA.MAC

```
        .TITLE  TTREAD  TERMINAL READ SUBROUTINE
        .IDENT  /01/

;
; AUTHOR: DEF   8-AUG-90
;

;
; CHANGES: NONE
;

;
; MODULE FUNCTION GENERAL:
;       THIS MODULE READS A LINE FROM A
;       TERMINAL INTO A BUFFER

;
        .PAGE                   ; BREAK PAGE FOR PREFACE
        .SBTTL  SYMBOL, MACRO, DATA DEFINITIONS
        .LIST   TTM             ; TERMINAL LISTING MODE
        .NLIST  BEX             ; SUPPRESS BIN EXTENSION
        .MCALL  QIO$S,WTSE$S

;
; LOCAL SYMBOL DEFINITIONS:
        EFN1    = 1
        LUN5    = 5
;

;
; LOCAL MACROS: NONE
;
```

Example 2–4 Cont'd on next page

Example 2–4 (Cont.) Source Code for FILEA.MAC

```
;
; LOCAL DATA BLOCKS:
;
        .PSECT  DATA,D,RW
IOST:   .BLKW   2               ; DEF IO STATUS WDS
;
;
; MODULE FUNCTION DETAILED:
;       INPUTS: R0 = ADDRESS OF BUFFER TO LOAD
;               R1 = LENGTH IN BYTES OF BUFFER
;
;       OUTPUTS:        R2 = NUMBER OF CHARS (BYTES) READ
;
;       SIDE EFFECTS:   IOT IF ERROR
;
; START CODE HERE:
        .PAGE
        .SBTTL  START OF CODE
        .PSECT
READ::                          ; DEFINE ENTRY POINT
        QIO$S   #IO.RLB,#LUN5,#EFN1,,#IOST,,<R0,R1>
                                ; QIO DIR PARAMETERS:
                                ;       RLB IS READ LOG BLOCK
                                ;       LUN5 = TKB DEFAULT
                                ;       EFN1 IS EVENT FLAG #1
                                ;       IOST = STATUS AREA
                                ;       <> = PARAMETER LIST
                                ;         R0 = START OF BUFFER
                                ;         R1 = SIZE OF BUFFER
        BCS     10$             ; IF SET, DIR ACCEPT ERROR
        WTSE$S  #EFN1           ; WAIT FOR IO COMPLETE,EF 1
        TSTB    IOST            ; CHECK IO STATUS
        BLT     10$             ; IF LT, IO ERROR
        MOV     IOST+2,R2       ; SAVE # OF BYTES READ
        RETURN                  ; GO BACK TO CALLER
10$:
        MOV     $DSW,R0         ; SAVE DIR STAT WD
        MOVB    IOST,R1         ; SAVE IO STAT BYTE
        IOT                     ; FORCE SST EXIT
        .END                    ; TELL ASSEMBLER END OF CODE
```

Example 2–5 Source Code for FILEB.MAC

```
        .TITLE  TTWRIT  TERMINAL WRITE SUBROUTINE
        .IDENT  /01/

;
; AUTHOR: DEF 8-AUG-90
;

;
; CHANGES: NONE
;

;
; MODULE FUNCTION GENERAL:
;
;       THIS MODULE WRITES A
;       LINE FROM A BUFFER TO
;       A TERMINAL

;
        .PAGE                   ; BREAK PAGE FOR PREFACE
        .SBTTL  SYMBOL, MACRO, DATA DEFINITIONS
        .LIST   TTM             ; TERMINAL LISTING MODE
        .NLIST  BEX             ; SUPPRESS BIN EXTENSION
        .MCALL  QIO$S,WTSE$S

;
; LOCAL SYMBOL DEFINITIONS:
        EFN1    = 1
        LUN5    = 5
;

;
; LOCAL MACROS: NONE
;

;
; LOCAL DATA BLOCKS:
;
        .PSECT  DATA,D,RW
IOST:   .BLKW   2               ; DEF IO STATUS WDS
;
; MODULE FUNCTION DETAILED:
;       INPUTS:
;
;               R0 = ADDR OF BUFFER TO WRITE
;               R1 = LENGTH IN BYTES OF BUFFER
;       OUTPUTS:
;
;               SUCCESS IN IOST
;       SIDE EFFECTS: IOT IF ERROR
;
; START CODE HERE:
```

Example 2–5 Cont'd on next page

Example 2–5 (Cont.) Source Code for FILEB.MAC

```
        .PAGE
        .SBTTL  START OF CODE
        .PSECT
WRITE::                             ; DEF ENTRY POINT
        QIO$S   #IO.WLB,#LUN5,#EFN1,,#IOST,,<R0,R1,#40>
                                    ; QIO$S PARAMETERS:
                                    ;  IO.WLB FUNCTION CODE
                                    ;  LUN5 (TKB DEFAULT)
                                    ;  EFN1 IS EVENT FLAG 1
                                    ;  STATUS AREA = IOST
                                    ;  PARAMETER LIST <>
                                    ;   R0 = START OF BUFFER
                                    ;   R1 = # OF CHARS TO WRITE
                                    ;   40 = OUTPUT <CR>,<LF>
        BCS     10$                 ; IF SET, DIR ACCEPT ERROR
        WTSE$S  #EFN1               ; WAIT FOR IO COMPLETE
        TSTB    IOST                ; CHECK IO STATUS
        BLT     10$                 ; IF LT, IO ERROR
        RETURN                      ; GO BACK TO CALLER
10$:
        MOV     $DSW,R0             ; SAVE DIR STAT WD
        MOVB    IOST,R1             ; SAVE IO STAT VALUE
        IOT                         ; SST DUMPS TASK REGS
        .END                        ; TELL ASSEMBLER END OF CODE
```

3 Assembling and Correcting a Program Module

This chapter describes the following functions:

1 Some uses of the MACRO-11 Assembler.

2 Some common types of coding errors.

3 Some ways to trouble-shoot and correct errors.

4 The way to generate a cross-reference listing.

The material in this chapter assumes that you have created the three source files described in Chapter 2.

3.1 Performing a Diagnostic Run on a Source File

Your first use of the MACRO-11 Assembler on a source file should be to perform a diagnostic run. You run the assembler only to check for general errors, not to produce an object module or a listing file.

To perform a diagnostic run from a program development system (PDS) terminal, type the following command line:

```
PDS> MACRO/NOOBJECT/DS:GBL FILE RET
     (any error messages appear)
PDS>
```

The source file is named FILE.MAC, but because the MACRO command defaults to a file type of MAC, the file type need not be specified. Normally, the MACRO command is used to create an object module, so that the /NOOBJECT qualifier is necessary to override this standard function. The MACRO command does not produce a listing file unless you request one with the /LIST qualifier. When you do not make a listing file, any errors that result from assembly are listed directly on your terminal.

The /DS:GBL qualifier causes MACRO-11 to disable the setting of undefined symbols to global and external. Ordinarily, when MACRO-11 finds a symbol that is not defined in the source file, it assumes that the reference is to a symbol defined externally (that is, in another module). By disabling this feature for your diagnostic run, you tell the assembler to flag any potential global references as an undefined symbol error. Since you already know which symbols in your source file are global, this disabling method is a convenient way to catch typographical errors in other symbol names.

To perform a diagnostic run from a monitor console routine (MCR) terminal, type the following command line:

```
MCR> MAC /DS:GBL=FILE RET
     (any error messages appear)
MCR>
```

The right side of the equal sign (=) gives the specification of the source file. The assembler searches for the file named FILE.MAC in your directory. The assembler applies the type MAC as a default. Because no file specifications occur on the left side of the equal sign, MACRO-11 does not produce any object module or listing file. When you do not specify a listing file in the command line, the assembler prints on the input terminal the lines that generated errors and reports the total number of errors found.

The left part of the command line (/DS:GBL) causes MACRO-11 to disable the setting of undefined symbols to global and external. Ordinarily, when MACRO-11 finds a symbol that is not defined in the source file, it assumes that the reference is to a symbol that is defined external to the module (in another module). (The notation GX in the listing symbol table denotes a global and externally defined symbol.) By disabling this feature in the diagnostic run, you tell the assembler to flag any potential global reference with an undefined symbol error. This disabling method is a convenient way to catch typographical errors in symbol names at assembly time rather than later when you link your object modules together.

The appearance of MACRO-11 messages at the terminal during the diagnostic run indicates that your module contains errors. If the assembler does not find any errors, it simply returns control to the Executive and MCR prints its prompt. Errors in the assembly are denoted by single-letter codes printed at the beginning of the faulty statement. These errors are summarized in an appendix of the *PDP-11 MACRO-11 Language Reference Manual*.

Only the following errors should appear from the diagnostic run following:

```
U     71 000010   004767          CALL    READ     ; READ FROM TTY
U     99 000110   004767          CALL    WRITE    ; REPORT THE RESULTS
ERRORS DETECTED:  2
/DS:GBL=FILE
```

The two undefined symbols READ and WRITE are the entry points defined in the source files FILEA.MAC and FILEB.MAC. These symbols are to be resolved by the task builder (TKB). (Note that this example was generated by the MCR command MAC/DS:GBL=FILE.)

3.2 Typical Errors Encountered During Assembly

Four error codes cover the majority of errors made in an assembly language source file. The following sections describe some of the most common conditions under which these error codes are generated.

3.2.1 The MACRO-11 Error Code A

Error code A indicates a general assembly error. Most of these errors are caused by typing mistakes such as the following:

- Omitting the semicolon (;) from a comment

  The semicolon separates your comment from the portion of the statement that the assembler evaluates. If you omit the semicolon, MACRO-11 attempts to evaluate your comment as part of the rest of the statement line.

- Omitting the period (.) from a MACRO-11 directive

  The leading period in the operator field tells the assembler that the statement contains a MACRO-11 directive. If you forget to include the period on a directive, the assembler cannot evaluate the operator as a directive. As a result, error code A is generated, the directive and its arguments are given a value of 0, and they are designated as global symbols.

- Misspelling a PDP-11 instruction mnemonic

  If you misspelled a PDP-11 instruction mnemonic (for example, MOVE instead of MOV), the assembler can evaluate the operands but not the operator. The *PDP-11 MACRO-11 Language Reference Manual* lists all the mnemonics alphabetically. (These mnemonics make up the permanent symbol table (PST)). The PDP-11 Programming Card also contains all the instruction mnemonics.

- Forming an illegal symbol

  The first character of a symbol must not be a numeral.

- Delimiting a directive argument improperly

  Many MACRO-11 directives require a character or argument string to begin and end with a certain delimiting character. If you use the wrong character or omit one of the delimiters, the assembler cannot properly match the delimiters and therefore cannot evaluate the directive. For example, the .ASCII directive requires the character string to begin and end with the same delimiting character.

Another type of general assembly error involves general addressing errors. The typical addressing error is to exceed the range of a branch instruction (that is, branching more than 128 words backwards or 127 words forwards). To correct this type of error, replace the branch instruction with code to test the proper condition, and with the JMP instruction to transfer control.

Also common as general assembly errors are illegal forward references. If you define a symbol based on another symbol defined by a forward reference, the assembler cannot evaluate the reference. For example:

```
A = B + 10.
C = A + 10.
```

The assembler cannot evaluate the symbol A because B is not yet defined.

3.2.2 The MACRO-11 Error Code U

Error code U signals an undefined symbol error. This error usually occurs because (1) a symbol name on the .MCALL directive was misspelled or (2) reference was made to a local label that does not exist in the current local symbol block.

3.2.3 The MACRO-11 Error Code Q

Error code Q indicates questionable syntax. This error usually results from either including too many (or too few) arguments in a directive or specifying an incorrect number of operands on an instruction. In addition, this error occurs when you omit the semicolon from a comment and the assembler attempts to evaluate the comment as part of the statement.

3.2.4 The MACRO-11 Error Code E

Error code E means that you have omitted the .END directive from the assembly language source file. If the assembler does not find the .END directive, it generates error code E with a line number of 0 after the last statement in the listing file.

Error code E also might also indicate an expression overflow. If the assembler encounters a nested expression that is too complex, it generates error code E and denotes the point of the overflow with a question mark (?). To clear the error condition, either simplify the expression or ask your system manager to build MACRO-11 with a larger stack.

3.3 Generating a Program Module and a Listing

After you correct the errors uncovered in the diagnostic run, you are ready to produce an object module and a listing file. The following PDS command line produces an object module and a listing file:

```
PDS> MACRO FILE/LIST [RET]
     (error summary printed)
```

This command line, like the command line for the diagnostic run, assumes default file types for the object file and the listing file. The assembler creates an object module called FILE.OBJ. The /LIST qualifier causes the assembler to create a file called FILE.LST. It is good programming practice to use the assembler defaults for file types and file names. Using the defaults helps you differentiate file types, and groups associated files under the same name. If you want to use other names and file types, you can override the defaults by supplying complete file specifications as arguments to the /LIST and /OBJECT qualifiers.

NOTE: The /LIST qualifier is added to the file specification rather than to the MACRO command in the example. This placement of the qualifier causes a listing file to be created in your directory, but the file is not printed on the line printer. A MACRO command line in the following format still causes the listing file FILE.LST to be created in your directory, but the file is also printed on your system's line printer:

```
PDS> MACRO/LIST FILE [RET]
     (error summary printed)
```

For the time being, you should use the /LIST qualifier as a file specification qualifier, to keep from printing too many copies. During the program development cycle, you create many files for which you do not need a permanent copy. It is easier and less wasteful to examine a listing file at your terminal than to generate numerous printed copies of listing files that must be discarded because of minor errors. After you attain an error-free assembly, you can print a copy of the latest version of the listing file.

When you request a listing file, the errors are printed in the file, not on your terminal. All you see on your terminal is a message giving the total number of errors found. If no message appears, there are no errors. Note, however, that freedom from assembly errors does not guarantee that the program will run properly.

The following command line performs the same functions from an MCR terminal:

```
MCR> MAC FILE,FILE/-SP=FILE [RET]
```

This command line, like the command line for the diagnostic run, depends on default file types that MACRO-11 automatically assigns. The leftmost file specification creates an object module called FILE.OBJ. The file type OBJ denotes that the file is an object module.

The comma (,) following the object file specification in the command line is a separating character that is required to distinguish different file specifications in command lines.

Following the comma in the command line is the listing file specification that creates the file called FILE.LST. The file type LST denotes that the file is a listing of source code produced by an assembler or compiler.

It is good programming practice to use the assembler defaults for file types and to apply the name of the source file to both the object and listing files. Using the defaults helps you to differentiate types of files, and keeping the same name helps relate different types of files to the proper source file.

The /-SP qualifier following the listing file specification in the command line inhibits automatic spooling of the listing to the line printer. During the program development cycle, you create many files for which you do not need a permanent copy. It is easier and less wasteful to examine a listing file at your terminal than to generate numerous copies of listing files that must be discarded because of minor errors. After you attain an error-free assembly, you can spool a copy of the latest version of the listing file retained on your disk.

When you request a listing file in the assembly, MACRO-11 does not print error lines on the terminal. Instead, if the assembler detects any errors, it prints a message giving the total number found. If the assembler finds no errors, it simply exits. The absence of a summary of error messages from the assembler means an error-free assembly. If errors occur, you can examine the listing file at the terminal. However, an error-free assembly does not guarantee that the program will run properly.

You can issue the following command lines to assemble the two other source files, FILEA.MAC and FILEB.MAC, which you created by using the procedures described in Chapter 2:

```
PDS> MACRO  FILEA/LIST [RET]
PDS> MACRO  FILEB/LIST [RET]
```

or:

```
MCR> MAC FILEA,FILEA/-SP=FILEA [RET]
MCR> MAC FILEB,FILEB/-SP=FILEB [RET]
```

The two command lines create the object modules FILEA.OBJ and FILEB.OBJ that you need to link into your task in Chapter 4.

3.4 Examining a Listing at the Terminal

Use the TYPE command to display the listing file at your terminal, as follows:

```
PDS> TYPE FILE.LST [RET]

     (file appears on screen)
```

From an MCR terminal, you can run the peripheral interchange program (PIP) to transfer a copy of your listing from your disk to your terminal. The following command line starts the transfer:

```
MCR> PIP TI:=FILE.LST [RET]

     (file appears on screen)
```

In the part of the command line to the left of the equal sign, the designation TI specifies your terminal (that is, the terminal initiating the request) as the output device.

NOTE: If you omit the colon (:) from TI, PIP creates a new file called TI in your directory and copies the input file to it.

To the right of the equal sign is the input file specification with both a name and type. For PIP, you must specify a file type because it does not apply a default file type for you. (Without a file type, PIP looks for a file with no type (that is, a file with a null type).)

You can use control commands to stop and restart the display temporarily, and alternately to suppress and resume the output request.

Table 3–1 Terminal Output Control Commands

Command	Effect
Ctrl/S	Temporarily stops the display
HOLD SCREEN	Alternately stops and restarts the display (VT200-series terminals only)
NO SCROLL Ctrl/S	Alternately stops and restarts the display (VT100-series terminals only)
Ctrl/O	Alternately suppresses and resumes the output to the terminal

Use the Ctrl/S and Ctrl/Q commands together to freeze the display on the screen and to request more lines to be displayed. While the Ctrl/S command is in effect, you can read what is on the screen. The Ctrl/Q command tells the system to restart the display where it left off when it sensed the Ctrl/S command.

The Ctrl/O command is used to suppress unwanted output. The command tells the system to stop sending characters to the terminal. The program, however, continues processing but simply omits displaying the output. (While the Ctrl/O command is in effect, the system disables keyboard input and does not echo any characters typed at the terminal.) By typing Ctrl/O again, you tell the system to resume output to the terminal. By typing successive Ctrl/Os, you can skip unnecessary portions of the output until the program reaches the correct part. If the program finishes processing the output request while the Ctrl/O command is in effect, the system automatically reenables keyboard input and a prompt appears on the terminal.

3.5 Generating a Cross-Reference Listing

Worthwhile additions to the assembly listing are the symbol and macro cross-reference listings. These listings give, in alphabetical order, each symbol and macro name defined or referred to and the number of the page and line in the listing where the definition or reference occurs. From a PDS terminal, type the following command line:

```
PDS> MACRO/NOOBJECT FILE/CROSS_REFERENCE RET
     (any errors cause total number to be printed)
```

NOTE: The command line does not include the /LIST qualifier. The /CROSS_REFERENCE qualifier implies the /LIST qualifier since the cross-reference listing is attached to the assembly listing. If you want to print the listing, use /CROSS_REFERENCE as a command qualifier rather than a file qualifier.

Remember, you can abbreviate the command unless you are keeping a record of terminal activity. The following command line has the same effect as that of the one above:

```
PDS> MAC/NOOB FILE/CROSS_REFERENCE RET
```

From an MCR terminal, you generate the cross-reference listing by typing the following:

```
MCR> MAC ,FILE/CR/-SP=FILE RET
     (any errors cause total number to be printed)
```

Because no file specification precedes the comma in the command, MACRO-11 omits creating the object module and produces only a listing file. The /CR switch tells the assembler to generate a request for the Cross-Reference Processor (CRF) task to produce a cross-reference listing. (Omitting the comma from the command causes an error because the command then requests an object module only. With an object module specification, the /CR and /-SP switches are illegal.)

NOTE: If, after you request a cross-reference listing, you discover that the information is missing from your listing, the CRF task is either not installed on your system or it is still processing the request. Ask your system manager to install the CRF task.

The CRF task appends the cross-reference listing to the end of the listing file, denoting the cross-references by the titles SYMBOL CROSS REFERENCE and MACRO CROSS REFERENCE.

3.6 Spooling a Copy of Listings

Once you have developed an error-free assembly, you can obtain a hard copy of the listing for reference. From a PDS terminal, type one of the following command lines:

```
PDS> PRINT FILE.LST [RET]
```

or:

```
PDS> MCR PIP FILE.LST/SP [RET]
```

The command lines create a request to the spooling task to print the file you specify. (You can specify more than one file at a time by listing more than one file specification in the command line, separating each with a comma.) Your request is placed in a queue of requests.

If your system does not have spooling, you can list the file directly on the printer, as follows:

```
PDS> MCR PIP LP:=FILE.LST [RET]
```

If the printer is not busy or allocated by another user, PIP outputs the file to LP0.

The process for obtaining a hardcopy listing is very similar from an MCR terminal.

From an MCR terminal, type one of the following command lines:

```
MCR> PRINT FILE.LST [RET]
```

or:

```
MCR> PIP FILE.LST/SP [RET]
```

The command lines create a request to the spooling task to print the file you specify. (You can specify more than one file at a time by listing more than one file specification in the command, separating each with a comma.) Your request is placed in a queue of requests.

If your system does not have spooling, you can list the file directly on the printer, as follows:

```
MCR> PIP LP:=FILE.LST [RET]
```

If the printer is not busy and is not allocated by another user, PIP outputs the file to LP0.

3.7 Cleaning Up the Disk Directory

After you edit and reassemble the source files several times, your directory becomes cluttered with multiple versions of the same files. PDS includes a DIRECTORY command for listing information about files. The following command lists all the files in your directory:

```
PDS> DIRECTORY [RET]

     (the directory listing appears)
```

The directory is going to contain a number of files with the same name and type, but different version numbers. Use the following command to purge all but the most recent version of these files:

```
PDS> PURGE *.MAC,*.LST,*.OBJ [RET]
```

From an MCR terminal, you can list the name, types, version numbers, and sizes of the files stored in your directory by typing the following command line:

```
MCR> PIP /LI [RET]

     (the directory listing appears)
```

The /LI switch causes PIP to list the directory information at your terminal. By default, the command line requests all names, types, and versions of files in your directory.

By examining the directory information, you notice that files with the same name and type have multiple versions. Use the following command line to the PIP program to purge all but the most recent version of the files:

```
MCR> PIP *.MAC,*.LST,*.OBJ/PU [RET]
```

The /PU switch purges all but the latest version of the files specified. The asterisk character (*) in the command denotes all file names.

4 Building and Testing a Task

This chapter describes ways to use the task builder (TKB) program to create a task image from program object modules. The procedures described in this chapter assume that you have created three error-free object modules, as described in Chapter 3.

4.1 Creating a Task Image

The TKB program creates a task image file that can be loaded into memory. You can supply as input to TKB either a single object module or multiple object modules. In most cases, however, your programs will consist of multiple object modules. The following sections describe the procedures and the way TKB reports error conditions.

4.1.1 Supplying a Single Object Module

Use the program development system (PDS) command LINK to create a task image file from a single object module. For example:

```
PDS> LINK FILE [RET]

     (any error messages appear)
```

Once again, all defaults are applied automatically. The LINK command defaults to an object module in a file named FILE.OBJ and causes TKB to produce a file named FILE.TSK that contains the task image.

To perform the same function from a monitor console routine (MCR) terminal, enter the following command line:

```
MCR> TKB FILE=FILE [RET]

     (any error messages appear)
MCR>
```

The right side of the equal sign (=) specifies the file that contains the object module. TKB assumes that the type in the file specification is OBJ. The left side of the equal sign gives the specification of the task image file to which TKB assigns the file type TSK. Again, as with the assembler, it is convenient to apply the same name to both the output file and the input file and to let TKB apply the default type specifications.

TKB tries to resolve all global references in the object module. If undefined references remain after the module has been processed, TKB searches the system object library SYSLIB.OLB in directory [1,1] on the library device (LB). If no errors are encountered in the process, TKB exits and the command prompt (>) appears.

If TKB detects an error during processing, it prints a message in one of the following forms for PDS users:

```
LINK -- *DIAG* - error message
```

or:

```
LINK -- *FATAL* - error message
```

If TKB detects an error during processing, it prints a message at the terminal in one of the following forms for MCR users:

```
TKB -- *DIAG* - error message
```

or

```
TKB -- *FATAL* - error message
```

If an error message appears and the error condition described is not operational (for example, lack of space for the task image file) or is not a fatal error, TKB creates the task image file anyway. Depending on the error condition, you might have to remove the cause of the error from the source file, reassemble the source file, and repeat the TKB procedure. In some instances, the diagnostic condition is merely a warning and has no ill effect when the task runs. (For guidelines on typical error conditions, see Section 4.4.)

When you create the task image from the single object module FILE.OBJ (refer to Section 3.3), TKB prints the following error message:

```
TKB -- *DIAG-*2 Undefined symbols segment FILE

       READ
       WRITE
```

The undefined symbols READ and WRITE are the entry points of the two routines defined by the object modules FILEA.OBJ and FILEB.OBJ. TKB searches the system object library to resolve global references left undefined in your input. Because TKB failed to find modules that defined these symbols, it reported the error condition. You can eliminate the error condition by following the procedures described in the following section.

4.1.2 Supplying Multiple Object Modules

TKB accepts multiple object modules as input to the LINK command. At a PDS terminal, type the names of the object files, separated by commas, as follows:

```
PDS> LINK FILE,FILEA,FILEB [RET]

     (any error messages appear)
```

The LINK command defaults to the file type OBJ for the three input files. The resulting task image file is named FILE.TSK. The LINK command defaults to the name of the first object file named to derive the name of the TSK file.

From an MCR terminal, specify the names of the multiple object files, separated by commas, on the right side of the equal sign in your TKB command line, as follows:

```
MCR> TKB FILE=FILE,FILEA,FILEB [RET]

     (any error messages appear)
MCR>
```

TKB performs the same actions as those described in Section 4.1.1 for one object module from a PDS or MCR terminal. Only one of the object modules specified must have been assembled with an .END directive giving the starting address of the task. If one of the modules does not contain the starting address, TKB assigns the default transfer address of 1, which causes an error when you run the task. See Section 4.4.

TKB can also accept as input a concatenated object module, which is merely a file containing multiple object modules. To create a concatenated file, use the PDS command COPY, as follows:

```
PDS> COPY [RET]
From?  FILE.OBJ,FILEA,FILEB [RET]
To?  FILCON.OBJ [RET]
```

or

```
PDS> COPY FILE.OBJ,FILEA,FILEB FILCON.OBJ [RET]
```

The response to the "From?" prompt lists the files to be concatenated. Note that you specify the file type only on the first file listed. This type becomes the default file type for subsequent files. The COPY command automatically concatenates these files into a single output file.

The single concatenated object file can then be the sole input to the LINK command, as shown in the following command line:

```
PDS> LINK/TASK:FILE FILCON [RET]
     (any error messages appear)
```

This operation saves file-processing overhead for TKB. As a result, building a task from a concatenated file is faster than listing the object modules separately.

Use the following peripheral interchange program (PIP) command line to create a concatenated file from an MCR terminal:

```
MCR> PIP FILCON.OBJ=FILE.OBJ,FILEA,FILEB/ME [RET]
MCR>
```

The right side of the command specifies the files to be concatenated. Specify the file type (OBJ) only on the first file because PIP applies it as the default filetype for subsequent names.

The /ME switch tells PIP to merge (concatenate) all the files into the one file specified on the left side of the equal sign. When you supply multiple file specifications on the right side of the command line, PIP uses the /ME switch as a default condition. The command line includes the /ME switch merely to emphasize the concatenate, or merge, operation.

The single concatenated object file can then be the sole input to TKB, as shown in the following command line:

```
MCR> TKB FILE=FILCON [RET]
     (any error messages appear)
MCR>
```

This operation saves file-processing overhead for the TKB program and is faster than supplying the object modules separately.

4.1.3 Using the Fast Task Builder

Often you are performing repetitive, straightforward task-building functions where speed is preferable to versatility. In such circumstances, you can use the fast task builder (FTB). From a PDS terminal, use the LINK command with the /FAST qualifier to specify the FTB. For example:

```
PDS> LINK/FAST/MAP FILE,FILEA,FILEB [RET]
```

To invoke the FTB from an MCR terminal, use the following command:

```
MCR> FTB FILE,FILE/-SP=FILE,FILEA,FILEB [RET]
MCR>
```

FTB runs three to four times faster than TKB but is less versatile than TKB. For example, FTB does not create a global cross-reference listing or a symbol definition file. In addition, the FTB map has less information than the TKB map.

4.2 Task Builder Defaults

When you build a task image, TKB applies certain default conditions to your program, including the partition where your task runs, the host system memory management characteristics, the task's checkpointability, and the number of logical units your task can access. If your program does not use the default conditions, the process of building a task becomes more complex.

TKB assigns your program to be run in the default partition, called GEN. If you are building a task to run in another partition, you can either supply the correct partition name at run time or rebuild the task and specify the correct partition name then.

TKB applies memory management characteristics depending on the system on which you build the task. If your system has memory management hardware, TKB allocates memory starting at virtual address 0 and assumes that the task will be relocated by memory management hardware. Therefore, the task can be run in any partition large enough to contain the image. If your system does not have memory management hardware, TKB assumes that the task runs at a fixed physical address that the system must supply.

TKB establishes the maximum number of logical units (six) the task can access and supplies the assignments for these logical units. The default assignments are: logical units 1 to 4, which are assigned to the system device (SY); unit 5, which is the task-initiating terminal (TI); and unit 6, which is the console listing device (CL). These defaults mean that the task can simultaneously refer to at most four files on the system device, one file on the task-initiating terminal, and one file on the system console listing device.

4.3 Generating a Map and a Global Cross-Reference Listing

Before you run the task and correct simple errors, you can produce a memory allocation file (called a map) and a cross-reference listing of global symbols. The map and global cross-reference file are useful in later stages of program development and for program documentation.

4.3.1 Requesting a Map and a Global Cross-Reference Listing

In most situations, you need a standard map and a global cross-reference listing for debugging a task. To create a map with a global cross-reference listing from a PDS terminal, type the following command line:

```
PDS> LINK/CROSS_REFERENCE/NOWIDE/NOTASK FILE,FILEA,FILEB RET
```

The /NOTASK qualifier suppresses the creation of a task image file. You request a cross-reference listing with the /CROSS_REFERENCE qualifier. The /NOWIDE qualifier reduces the width of the listing from 132 columns to 80 columns for display on a terminal. Since /CROSS_REFERENCE implies a map, you do not have to specify the /MAP qualifier.

If you want to create both the task image file and the map with the cross-reference listing at the same time, use the following command line:

```
PDS> LINK/CROSS_REFERENCE/NOWIDE FILE,FILEA,FILEB RET
```

TKB creates both FILE.TSK and FILE.MAP. The map includes a cross-reference listing.

The following command line performs the same procedure from an MCR terminal:

```
MCR> TKB ,FILE/CR/-SP/-WI=FILE,FILEA,FILEB RET
MCR>
```

The right side of the equal sign is the input object module (or concatenated object module or multiple object modules). The left side of the equal sign in the command line specifies the map file name, to which TKB appends the file type MAP. The comma preceding the map file name suppresses the creation of the task image file.[1]

To create a new version of the task image file when you request the map and global cross-reference listing, type the command as follows:

```
MCR> TKB FILE,FILE/CR/-SP/-WI=FILE,FILEA,FILEB RET
MCR>
```

TKB creates both files.

The /CR switch tells TKB to generate a request for the cross-reference processor (CRF) task to produce a global cross-reference listing. The /-WI switch reduces the width of the listing from 132 columns to 80 columns for display on a terminal. The CRF task executes the request from TKB and appends the global symbol cross-reference listing file to the end of the map file. The global cross-reference in the map listing is denoted by the title GLOBAL CROSS REFERENCE.

NOTE: If, after you request a global cross-reference listing, you discover that the map does not have one, the CRF task is either not installed on the system or is still processing the request. Consult the system manager to have the CRF task installed.

4.3.2 Examining the Map at the Terminal

Use the PDS command TYPE, as described in Section 3.4, to examine the map at your terminal. The command line is as follows:

```
PDS> TYPE FILE.MAP RET
     (file appears on screen)
```

From an MCR terminal, use the following PIP command line, as described in Section 3.4, to examine the map at your terminal:

```
MCR> PIP TI:=FILE.MAP RET
     (file appears on screen)
MCR>
```

Use the control commands Ctrl/S, Ctrl/Q, and Ctrl/O, summarized in Table 3–1, to control the terminal output.

[1] The task image specification is null when a comma appears first in the command line. If you omit the comma, TKB treats the file name for the map as a task image and generates a syntax error because the /CR/-SP switch is illegal with a task image file.

4.3.3 Requesting a Full Map

The map file produced, as described in Section 4.3.1, is a short form of the map that contains most of the information needed for debugging tasks. To generate a full form of the map, use the following command line from a PDS terminal:

```
        PDS>
LINK/LONG/MAP:FULL/CROSS_REFERENCE/SYSTEM_LIBRARY_DISPLAY/NOTASK FILE,FILEA,FILEB RET
```

The /LONG qualifier indicates that you want the LONG form of the map, and it causes TKB to add a "File Contents" section to the map. The /LONG qualifier implies the /MAP qualifier, but /MAP is used here to give the map file the name FULL.MAP so that you can distinguish the maps you have made for this demonstration session. The /SYSTEM_LIBRARY_DISPLAY qualifier (usually abbreviated to /SYS) tells TKB to include system library contributions to the task in the file contents section of the map. System symbols are also included in the global cross-reference listing.

To generate a full form of the map from an MCR terminal, specify the command line to TKB as follows:

```
MCR> TKB ,FULL/-SP/-SH/MA/CR=FILE,FILEA,FILEB RET
MCR>
```

The comma without a task image file name indicates you do not want TKB to build a task. The name FULL for the map file is used here to give that file the name FULL.MAP so that you can distinguish it from other maps you have created as part of these demonstrations. The /-SH switch indicates that you do not want the short form of the standard map. TKB therefore includes the file contents information in the map. The /MA switch tells TKB to include system library contributions to the task in the file contents section of the map. System symbols are also included in the global cross-reference listing.

4.4 Running the Task and Correcting Typical Errors

To execute your task, use the RUN command and the name of the task image file.[2] The form shown in the example is perhaps the most widely used form for program development. This form, which is the same in MCR and PDS, runs a task from a task image file in your directory. For example:

```
PDS> RUN FILE RET
```

Because the task FILE is not installed on the system, the RUN command searches your directory on device SY for a file named FILE.TSK. RUN installs it temporarily and runs it immediately. The task will be automatically removed when the task exits.

To run the task FILE, the Executive transfers control to the task starting (or transfer) address. If your task encounters an error condition, the Executive must decide whether to abort the task.

Errors that can cause the Executive to abort a task are either hardware-related or software-related. If the error is hardware related, such as a memory parity error or a load failure, the Executive begins aborting the task. In contrast, a synchronous system trap (SST) error condition, such as an illegal instruction, causes the Executive to attempt to transfer control to an SST routine. An SST routine is a routine within a task that services a particular type of SST condition. If your task defines a routine to service the type of trap, the Executive transfers control to it. If your task does not have the routine defined, the Executive aborts the task.

[2] Both MCR and PDS include a RUN command, each of which has many formats.

Aborting a task forces an orderly termination of the task. Included in the termination is a request for the task termination and notification (TKTN) program to display a message on your terminal. The program display includes the cause of the abort and a list of the task registers and the processor status word (PSW). For example:

```
14:16:26  Task   "TT30  " terminated
    Odd address or other trap four
    R0=000000
    R1=100103
    R2=147100
    R3=140130
    R4=000000
    R5=000000
    SP=001172
    PC=000003
    PS=170017
    MCR>
```

The information can help you ascertain the cause of the abort. If the cause of the error is hardware related, report the occurrence to your system manager, who can consult the error-logging data to find where the problem originated. If the cause of the error was an SST condition, you can use the data displayed by TKTN to find the problem.

The value of the program counter (PC) (minus 2) shown in the display tells you the address of the instruction that was being executed when the error was encountered. In the example shown above, the PC is at an odd address (000003). By examining the task map, you can ascertain that the PC address is not within the task code. This condition demonstrates one of the more common error conditions. The main module NUMA source file FILE.MAC does not define a task transfer address. The .END directive in a source file, used to define the starting address of a task, does not have the address symbol of the first instruction. If you omit the starting address definition, TKB supplies a default transfer address of 1. When you run the task, it causes an odd address trap and terminates. (Note that the PC has been incremented to 000003 because it is pointing to the next instruction in the code.) Therefore, you should ensure that the source file defines a starting address and that the address is even (on a word boundary).

To correct any errors in your task, you must edit the source file(s) concerned, reassemble the corrected file(s), and rebuild the task. For example:

```
MCR> EDI FILE.MAC [RET]
[00103 LINES READ IN]
[PAGE    1]
* L [TAB] .END [RET]
    .END                ; TELL ASSEMBLER END OF CODE
* C /D [TAB] /D [TAB] START/ [RET]
    .END    START       ; TELL ASSEMBLER END OF CODE
* EX [RET]
[EXIT]

MCR> MAC FILE,FILE/-SP=FILE [RET]
MCR> TKB FILE,FILE/-SP=FILE,FILEA,FILEB [RET]
MCR> RUN FILE [RET]
ABCABCABAB [RET]
THE NUMBER OF A'S IS 0004
MCR>
```

After you correct the errors and rebuild the task, you can run the task again. The task reads the line of text that you type, counts the number of As, displays the result, and exits.

The typical errors made in programming result in an SST condition. The common conditions are either an odd address or a memory-protection trap. Most of these errors occur when you use relative mode addressing instead of immediate mode. For example:

```
MOV     #BUF1,R0        ;
MOV     OFFSET(R0),R1   ;
```

The immediate mode reference #BUF1 moves the address of BUF1 into register 0. If you omit the number sign (#), however, you incorrectly specify relative mode addressing, as follows:

```
MOV     BUF1,R0
MOV     OFFSET(R0),R1
```

This instruction moves the contents of BUF1 and not the address of BUF1 into R0. The subsequent indexed mode reference generates either an odd address or memory-protection trap. (Your task is attempting illegally either to reference an odd address or to reference a location outside task memory). This type of error occurs often when you are using system directives that require parameters as immediate mode references, and when you omit the number sign from a parameter that makes the reference relative.

5 Using Debugging Aids

This chapter introduces three debugging aids that are helpful in the program development process: the on-line debugging tool (ODT), the postmortem dump (PMD), and the snapshot dump ($SNAP).

5.1 Using the On-Line Debugging Tool

The on-line debugging tool (ODT) is a special code that you include in your task image to assist you during debugging. ODT gives you interactive control of task execution, and it enables you to set breakpoints and to examine and change data and instructions within the memory-resident task. The ODT module links into your task image and thereby increases the size of the task image. Therefore, you remove ODT from your task when you finish debugging by rebuilding the task and omitting the ODT module. For more information, refer to the *IAS ODT Reference Manual*.

ODT commands differ from commands in other utility programs. Most programs have multicharacter commands that require a line terminator before they are executed. ODT commands, however, are single characters and require no line terminator. That is, ODT interprets input on a character-per-character basis rather than on a line-by-line basis. Therefore, as soon as you type a character that ODT recognizes as a command, ODT interprets it and performs the specified function. This difference in commands means that you must be careful when you are debugging your task with ODT.

5.1.1 Including ODT in a Task

Use the /DEBUG qualifier to the LINK command to include ODT in a task. (Refer to Section 3.3 for information on generating an object module.) For example:

```
PDS> LINK/DEBUG/TASK:BUG/MAP:BUG/CROSS_REFERENCE FILE,FILEA,FILEB RET
```

The /DEBUG qualifier specifies that you want to include ODT in the task. The /TASK:BUG qualifier specifies that you want the task image file to be named BUG.TSK. The /MAP:BUG qualifier specifies that you want the map to be named BUG.MAP. In this way, you can tell the difference between the versions of the task-built file with and without ODT. The task builder (TKB) accesses the file LB:[1,1]ODT.OBJ and links it into the task. The /CROSS_REFERENCE qualifier implies a /MAP qualifier. An accurate map of the task is necessary for use with ODT.

When using the separate instruction and data space capabilities found in some IAS operating systems, TKB inserts the module LB:[1,1]ODTID.OBJ into the task.

To include ODT in a task from a monitor console routine (MCR) terminal, type a command line similar to the following:

```
MCR> TKB BUG/DA,BUG/CR=FILE,FILEA,FILEB RET
MCR>
```

The /DA switch accompanying the task image file specification tells TKB to include ODT. TKB accesses the file ODT.OBJ in directory [1,1] on the library device and links it into the task BUG. You should request a map of the task because an accurate map is necessary for use with ODT.

5.1.2 Preparing to Use ODT

Before you run a task containing ODT, ensure that accurate listings of the assembled source files are available. These listings show the offsets into the modules in your task. The map of the task and the assembled source listings provide the data you need to set breakpoints and examine locations within the task.

5.1.3 Setting Up the Task

When you run a task containing ODT, ODT gains control, identifies itself (and the task it controls), and prints its command prompt. The following lines show the sequence:

```
> RUN BUG RET
ODT:TT30
_
```

The notation TT30 is the name that the system dispatcher assigned to the task. Such a name consists of the letters TT followed by the unit number of the terminal that requested the task. The task shown here was run from terminal number 30_8.

The underline character (_) is ODT's prompt. It indicates that ODT is ready to accept commands.

5.1.4 Relocation Registers

To access locations within the task, you should establish one or more relocation registers. This set of eight registers, numbered $R0 to $R7, enables you to specify locations within the task in terms of offsets from the start of modules in the task image.

To establish the proper addressing using offsets, you must first consult the location information in the task map. On the map listing, the portion titled memory allocation synopsis contains the location information for each program section and for each contribution to the program sections from different modules. A sample of the relevant portion of the map for the program BUG is shown in Example 5–1.

Example 5–1 Memory Allocation Synopsis from Task BUG Map

```
Memory allocation synopsis:
Section                                          Title   Ident   File
-------                                          -----   -----   ----
. BLK.:(RW,I,LCL,REL,CON)       001202 000340 00224.
                                001202 000122 00082.  NUMA    01      FILCON.OBJ;1
                                001324 000110 00072.  TTREAD  01      FILCON.OBJ;1
                                001434 000106 00070.  TTWRIT  01      FILCON.OBJ;1
DATA  :(RW,D,LCL,REL,CON)       001542 000166 00118.
                                001542 000156 00110.  NUMA    01      FILCON.OBJ;1
                                001720 000004 00004.  TTREAD  01      FILCON.OBJ;1
                                001724 000004 00004.  TTWRIT  01      FILCON.OBJ;1
$$$ODT:(RW,I,GBL,REL,OVR)       001730 005654 02988.
                                001730 005654 02988.  ODTRSX  M06     ODT.OBJ;121
```

The location information for a program section is the octal starting address of the program section and its extent in bytes (both octal and decimal values). For example, for the blank program section, the starting location is 1202_8 and the extent is 340_8, or 224_{10}, bytes. Under the program section location information are the octal starting addresses and extents in bytes for the contributions from each object module. For example, the contribution from TTREAD in the blank program section starts at location 1324_8 and extends for 110_8, or 72_{10}, bytes.

The following example shows how to place the starting addresses of the modules in relocation registers:

```
_1202;0R
_1324;1R
_1434;2R
_1542;3R
_1720;4R
_1724;5R
```

The R commands place the addresses in relocation registers 0 to 5. The addresses are octal; ODT accepts only octal numbers. As soon as you type the R in the command line, ODT generates a line feed and carriage return and prints another prompt. This action indicates that ODT has executed the command as soon as it was typed. Therefore, before typing the R (or any command), ensure that the command line is correct.

If you notice a typographical error in the line before you type the command itself, simply type Ctrl/U, enter the number 8 or 9, or press the DELETE key, as shown in the following example:

```
_1272;08 ?
_
```

ODT considers the decimal number 8 an illegal character. It discards the input line, displays a question mark (?) to signal an error, and prints the prompt on a new line. You must retype the entire line. If you do enter an incorrect address in the relocation register, simply retype the command, as follows:

```
_1272;0R
_1202;0R
```

ODT stores the most recently entered value in the register.

To access a location within a task most conveniently, you must create an address made up of the values stored in the relocation register and a value showing the distance of the location from the relocation register value.

The relocation register provides the base address of a module; the location counter value supplies an offset to the location within the program section for the module. The command 1202;0R places the starting address of the NUMA contribution to the blank program section in relocation register 0. Location counter value 20 in the assembly listing for NUMA is 20 bytes from the start of the address in relocation register 0. You use the two values to form the address of the location. The address is formed by typing the number of the relocation register, a comma (,), and the octal offset value. For example:

```
0,20
```

ODT adds the base value in relocation register 0 (1202 in this case) and the offset typed after the comma (20). This creates an effective address of 1222_8. You use this syntax with various ODT commands to access locations within the task address space.

Example 5–2 Portion of Assembly Listing for NUMA

```
NUMA   COUNT NUMBER OF A'S     MACRO M1200  8-AUG-86 12:39  PAGE 3
ROUTINE TO COUNT A'S

     66                          .SBTTL  ROUTINE TO COUNT A'S
     67 000000                   .PSECT
     68 000000           START:
     69 000000  012700           MOV     #BUF1,R0        ; LOAD BUFFER ADDR
     70 000004  012701           MOV     #SIZ,R1         ; LOAD BUFFER SIZE
     71 000010  004767           CALL    READ            ; READ FROM TTY
     72 000014  005702           TST     R2              ; ANY CHARS IN BUFFER?
     73 000016  001436           BEQ     END             ; IF NONE, FINISH UP
     74 000020  005001           CLR     R1              ; INIT # OF A'S COUNTER
     75 000022  010267           MOV     R2,NUMC         ; SAVE # OF CHARS TYPED
     76 000026           10$:
     77 000026  122067           CMPB    (R0)+,A         ; IS CHAR = A?
     78 000032  001001           BNE     20$             ; IF NO, BET NEXT CHAR
     79 000034  005201           INC     R1              ; COUNT AN A
     80 000036           20$:
     81 000036  005302           DEC     R2              ; ONE LESS CHAR
     82 000040  001372           BNE     10$             ; IF MORE, COMPARE NEXT
```

Example 5–2 shows a portion of the assembly listing for the blank program section in the module NUMA.

5.1.5 Examining Locations

To examine words within a module, type the address followed by the slash character (/), as follows:

```
_0,20/ 005001
```

The slash character causes ODT to open the designated location as a word and display its contents.

To close the currently open location, press either the RETURN key or the LINE FEED key. Pressing the RETURN key closes the location, as shown in the following example:

```
_0,20/005001  RET
_
```

ODT closes the location and prints its prompt on a new line.

Once you have opened a location, pressing the LINE FEED key enables you to examine successive words in the task image. The following example shows the procedure:

```
_0,32/ 001001  LF
0,000034/005201  RET
_
```

In response to the LINE FEED key, ODT closes the current location; opens the next sequential location in the task image; and displays the address of the location, a space, the slash character, and the contents of the location. The slash character signals that the location is open as a word.

NOTE: You can change the contents of the currently open location to n by typing the octal number n before pressing the RETURN or LINE FEED key. See Section 5.1.7.

To examine bytes instead of words within a task, type the address followed by the backslash character (\), as follows:

```
_0,32\ 001
```

The backslash character causes ODT to open the designated location as a byte and display its contents. You can examine successive bytes by pressing the LINE FEED key, after which ODT closes the currently open byte location, opens the next sequential byte location, and displays its contents. For example:

```
_32\001  LF
0,000033\002  RET
_
```

The backslash character preceding the contents signals that the location is open as a byte.

Before you proceed in the debugging session, you should verify the relocation register values by examining a location in each module and by comparing its contents with the values shown in the assembly listing. The following sequence shows the procedure:

```
_1,66/ 002403  RET
_2,72/ 000207  RET
_3,121\ 124  RET
_4,0/ 000000  RET
_5,0/ 000000  RET
_
```

As you examine each location, compare the contents ODT displays with the assembly listing. If the values do not match, either you have an incorrect listing or the relocation register value is wrong.

5.1.6 Setting Breakpoints Within the Task

To enable you to stop (or break) task execution, ODT provides eight registers called breakpoint registers. These registers, numbered $0B to $7B, enable you to specify locations of instructions where execution should stop.

To establish breakpoints in the task, specify the location of the instruction with the B command in the format a;nB, as shown in the following example:

```
_0,10;0B  RET
_1,74;1B  RET
_
```

The command places the designated addresses in breakpoint registers 0 and 1.

NOTE: In specifying the address of an instruction, ensure that the location is the first word of the instruction.

As soon as you type the B in the command line, ODT generates a carriage return and line feed and prints a prompt. Changing a breakpoint register is similar to changing a relocation register; simply retype the command line and give the altered contents.

After setting up the breakpoint registers, you can issue the G (Go) command to begin task execution. For example:

```
_G  RET
0B:0,000010
_
```

When you type the G command, ODT swaps a BPT instruction into each breakpoint location.[1] ODT passes control to the starting address of the task. The task executes until it reaches a BPT instruction, at which point ODT regains control. When ODT regains control, the task has not yet

[1] The eight breakpoint instruction registers, with register names $0I to $7I, contain the actual instructions during task execution.

executed the instruction at the location where the breakpoint is set. ODT swaps the instructions back into the locations at which breakpoints are set, and prints a message with the following information:

- The breakpoint register designation
- The relocation address at which execution stopped

In the example above, the message shows breakpoint register 0 and its contents (offset 10 from the base address in relocation register 0).

5.1.7 Changing the Contents of Locations with ODT

When execution stops at a breakpoint, you can examine and change data within the task image address space. When execution stops at a breakpoint location, the task's general registers are stored in ODT locations accessed by the names $0 to $7. The following sequence shows a way to display general registers 0, 1, and 2:

```
_$0/ 001543  [LF]
$1/000120  [LF]
$2/135600  [RET]
_
```

The slash character opens the general register as a word location and prints its contents. Pressing the [LINE FEED] key closes the current location and opens the next sequential location.

To change data, simply type a new value while the current location is open. The following sequence shows a way you can change register 2:

```
_$2/135600  100 [LF]
$3/140130  [RET]
_
```

While the location (register 2) is open, you can type the new value to replace the current contents. ODT writes the new value 100_8 into the currently open location before closing it and opening the next sequential location.

Any locations within the task can be examined and changed. The following sequence shows a way to open a location as a byte and change its contents:

```
_3,0\ 101  102 [RET]
_3,0\ 102  101 [RET]
_
```

The backslash character opens the specified address as a byte location. The new value 102_8 is written to the open location as a byte value. Pressing the [RETURN] key closes the location. The next command line examines offset 0 to verify that it contains 102_8, then changes the contents back to 101.

After you examine and change locations, resume execution with the P (Proceed) command, as follows:

```
_PABCABCABAB [RET]
1B:1,000074
_
```

The P command causes ODT to swap in the BPT instructions, restore the task general registers, and continue with the instruction where the break occurred.

NOTE: ODT does not supply a carriage return and line feed after you type the P. Therefore, the data that you type in response to the READ routine will follow the P on the same line.

Execution stops at the location contained in breakpoint register 1.

Use the G command to transfer control to another address and to continue execution. For example:

```
_1,76G
```

ODT transfers control to offset 76 and continues execution there. This command purposely transfers control to the error routine to show what occurs when an error is encountered. See Section 5.1.8.

5.1.8 Error Conditions and Terminating Task Execution

If the task generates an error condition, the Executive handles the processing as a synchronous system trap (SST). Control is passed to ODT, which prints a message similar to the following one:

```
IO:2,000000
_
```

This message gives a code that describes the reasons for the trap and tells the address following the location that generated the trap. In the message above, IO means the IOT instruction. If you can discover the cause of the trap, make the appropriate changes in the task and proceed. If you cannot isolate the cause of the trap, you should exit from ODT and start a new debugging session.

To help ascertain the cause of the trap, you can examine the task registers and stack before you start a new debugging session. Use the register name—the dollar sign ($) followed by the register number—to access the task registers as described in Section 5.1.7. To examine the stack, examine register 6 (the stack pointer) and use the at sign (@) character to open the location pointed to by the stack pointer. For example:

```
_$6/ 001200  @
001200/001216  RET
_
```

The slash character opens the stack pointer as a word and displays the address of the top of the stack. The at sign character takes the contents of the currently open location (that is, the stack pointer) as the address of the next location to be opened, opens it, and displays its contents, which is the top word on the stack.

To examine the stack, press the LINE FEED key to open and display each successive word on the stack. You can ascertain the highest address the stack can have by consulting the line labeled stack limits in the task attributes section of the map. The line gives four numbers: the low address of the stack area, the high address of the stack area, and the octal and decimal extent of the stack area. The high address tells you the last available location (that is, the bottom) of the stack. After you have examined the highest address, you have looked at all the items on the stack and can press the RETURN key to close the last available location.

To exit from the task by means of ODT, use the X command as follows:

```
_X
```

ODT performs the exit task directive and returns control to the Executive.

6 Creating and Using Program Libraries

This chapter describes the procedures to create and maintain a library of macro source statements and a library of object module subroutines. It also shows how to include in your task image the macro call definitions and the object subroutines from user-created libraries.

The decision about whether to implement specific code as a macro call or as an object module subroutine is left to the designer. In general, the difference between implementations is a tradeoff of assembly time versus linking time and, secondarily, convenience versus size. Each time your source file invokes a specific macro call, the assembler must include the macro expansion in the object module. However, when your program calls an external subroutine, the resolution of the call is done during linking. Moreover, using the macro call to generate in-line code is convenient, but each invocation of the call increases the size of the resulting task image. However, if your program calls a specific external subroutine more than once, the subsequent invocations do not include that code in the task.

6.1 Creating and Using a Macro Source Library

The librarian utility program (LBR) creates and maintains library files that can contain macro definitions, object modules, or other elements. Program development system (PDS) users can invoke LBR functions through the LIBRARY command. Monitor console routine (MCR) users can invoke LBR functions through the LBR command. This section discusses creating a library file of macro definitions. Such a file has the default type MLB and contains only macro definitions.

6.1.1 Creating the Macro Library

The following example shows how you create a macro library from one input file of source definitions by using the PDS command LIBRARY:

```
PDS> LIBRARY/CREATE:(BLOCKS:25,MODULES:128)/MACRO [RET]
Library?  USRMAC [RET]
Module(s)?  USRMAC [RET]
```

or:

```
PDS> LIBRARY/CREATE:(BLOCKS:25,MODULES:128)/MACRO USRMAC USRMAC [RET]
```

The /CREATE qualifier identifies the LBR function you want to invoke. The arguments to the /CREATE qualifier specify features of the library you are creating. Because there is more than one argument, they are enclosed in parentheses and are separated by commas. The argument BLOCKS:25 gives the length in blocks for the library file. (PDS uses the decimal value automatically for all LIBRARY command arguments.) If you omit this argument, LBR creates a file 100 blocks long by default. The argument MODULES:128 indicates the number of module name table entries to allocate for this library. (Each macro definition in the library requires an entry in the module name table.) The /MACRO qualifier identifies the type of library you wish to create. The default qualifier is /OBJECT (to create an object module library).

The "Library?" prompt requests that you name the library to be created. For macro libraries, the default file type is MLB. The "Module(s)?" prompt requests you to name the file or files containing the macro definitions. The default file type for this parameter is MAC. (If you do not name a file here, LBR creates an empty file.)

The MCR command description contains more detailed information on how LBR creates a library. The following MCR command line creates a macro library:

```
MCR> LBR USRMAC/CR:25.::128.:MAC=USRMAC [RET]
MCR>
```

The /CR switch tells LBR to create a library file. LBR creates the library file USRMAC.MLB. For input to the library file, LBR uses the file or files specified to the right of the equal sign (=). In Example 6–1, the input file is USRMAC.MAC.

Example 6–1 MACRO-11 Library Source Definitions

```
;
; SAVE - STORES REGISTER ON STACK
;
        .MACRO   SAVE,REG                ; PUSH REG ONTO STACK
        MOV      REG,-(SP)
        .ENDM
;
; RESTOR - POPS REGISTER VALUE OFF STACK
;
        .MACRO RESTOR,REG
        MOV      (SP)+,REG               ; POP REG OFF STACK
        .ENDM
        .END
```

Following the /CR switch in the command line are parameters, separated by colons, that LBR uses to create the library.[1] The first parameter, 25_{10}, gives the length in blocks for the library file. If you omit this parameter, LBR uses 100_{10} blocks as the default length. When creating the library file, you can allow for some future additions to the library by making the size larger than necessary. (LBR expands a library file as needed if you add modules that cause the file to exceed its original size. However, the library is no longer contiguous.) The second parameter is blank because it applies only to object libraries. The third parameter, 128_{10}, is the number of module name table entries to allocate for this library. (An entry in the module name table is required for each macro definition.) Following the third parameter is the type of library to create (MAC for macro definition). You must specify this parameter because the default is an object library.

In creating the macro library, LBR allocates the requested amount of contiguous file space. If sufficient contiguous space is not available, LBR generates the OPEN FAILURE error and terminates. To have the library created, you must either free up some space on the volume or try a smaller library size.

When the library file is created, LBR attempts to insert into the library the macro definitions from the input file. LBR searches the input file for .MACRO directives and .ENDM directives. If the macro definitions are nested, only the outermost directives are directly callable from the library. From each macro definition, LBR extracts the name and creates an entry in the module name table. The entry in the module name table is the means by which the assembler finds the associated macro definition in the library. Any code or comments outside the directives are discarded and all trailing blank and tab characters, blank lines, and comments are eliminated from the macro text itself. (This action, called squeezing, conserves memory for the assembler and reduces the space required to hold the macro definitions.) Errors that occur during the insertion of definitions usually indicate improper definitions, such as a missing .ENDM directive.

[1] The numeric parameters are followed by decimal points to force LBR to interpret them as decimal numbers. If you omit the decimal points, LBR treats the numbers as octal.

6.1.2 Using the Macro Definitions from the Library

Once the macro definitions are in the library, you need perform only three actions to have the assembler include the macro expansions in your code:

1 Include the name of the macro in a .MCALL directive in your program source file.

2 Invoke the macro call within the source file.

3 Specify the name of the library file in the command line to the assembler.

Thus, to invoke the two macro library definitions SAVE and RESTOR in your program, precede the macro calls themselves with a statement such as the following:

```
.MCALL        SAVE,RESTOR        ; CALL DEFINITIONS FROM USRMAC
```

This statement should preferably occur at the start of the source file. When you assemble a source file that refers to a library file, you must name both files using the PDS command MACRO. For example:

```
PDS> MACRO USRMAC/LIBRARY, USRTST/LIST [RET]
```

The name of the macro library can appear anywhere but last in the list of input files and must be marked with the /LIBRARY qualifier. The next file named is the first source file.

There is further discussion of how LBR creates a library. Use the following MCR command line to include a library in an assembly:

```
MCR> MAC USRTST,USRTST/-SP=USRMAC/ML,USRTST [RET]
MCR>
```

To the right of the equal sign in the command line, specify the name of the macro library and the /ML switch. The comma (,) separates the macro library file name and the source file name. The /ML switch indicates to the assembler that the file is a macro library. The name of the macro library must precede the source file that refers to the macro definitions.

NOTE: If the library specification follows the source file name in the command and the corresponding definitions are not in the System Macro Library RSXMAC, MACRO-11 does not recognize the library file and generates assembly errors in the lines that contain calls to library definitions.

To process the macro calls in the source file, the assembler uses the names given in the .MCALL directive to generate symbols for the macro symbol table.[2] To expand the macro calls not defined in the source file, the assembler searches the library you specified before it searches the system default macro library. MACRO-11 does not search the system macro library for definitions that are found in the user library file.

6.2 Creating and Using an Object Module Library

You can use LBR to create a library file containing object modules. Such a file has the file type OLB (object library) as a default and can contain only object modules.

[2] If you omit the name of the macro call from the .MCALL directive, the assembler cannot recognize the call itself in the code. (A corresponding entry is not in its macro symbol table.) It treats an unrecognized macro call as an implicit .WORD directive. If the macro name is not a valid symbol, its usage is flagged as an undefined reference by the task builder (TKB).

6.2.1 Creating the Object Module Library

To create an object module library, you must have a file or files that contain the object modules to be inserted into the library. The following command line creates the object library and inserts the modules FILEA.OBJ and FILEB.OBJ. This PDS command line creates an object module library consisting of the object modules in FILEA.OBJ and FILEB.OBJ:

```
PDS> LIBRARY/CREATE:(BLOCKS:25,GLOBALS:128,MODULES:64)/OBJECT [RET]
Library?  USROBJ [RET]
Module(s)?  FILEA,FILEB [RET]
```

or:

```
PDS> LIBR/CREATE:(BLO:25,GLOB:128,MOD:64)/OBJ USROBJ FILEA,FILEB [RET]
```

Because it is the default, the /OBJECT qualifier is not required, but it is a good idea to include it. The default file type for an object module library is OLB. The default file type for the object module files is OBJ. The arguments to the /CREATE qualifier are the same as those used in creating a macro library with the addition of the GLOBALS argument, which applies to object libraries only. The GLOBALS argument specifies the number of entry point table slots to reserve. (An entry point is any global symbol in a module by which your program refers to the associated module.) If you do not supply a value, LBR defaults to GLOBALS:512.

A good estimate for the number of GLOBALS is twice the number of modules the library is to contain. The value should be a multiple of 64. If not, LBR raises the number to the next multiple of 64. Again, all these numbers are automatically decimal numbers in PDS.

If you supply a value of 0, you must access the module by its name. You can then maintain modules with duplicate entry points in the same library. The names of the modules must still be unique. When building a library with GLOBALS:0, you must specify the correct module names to TKB when you build your task. See Section 6.2.2.

The following MCR command line creates an object module library:

```
MCR> LBR USROBJ/CR:25.:128.:64.=FILEA,FILEB [RET]
MCR>
```

There is further discussion of how LBR creates an object module library in the *RSX Utilities Manual*.

The /CR switch tells LBR to create a library file. LBR uses the name preceding the /CR switch as the name of the library and applies the default file type OLB. Following the /CR switch in the command line are parameters, separated by colons, used in creating the file.[3]

The first parameter, 25_{10}, gives the size in blocks at which to create the library file. If you omit the parameter, LBR supplies 100_{10} blocks as the default size. When creating the library, you can allow for future additions by making the size larger than necessary. (LBR will expand a library file as needed if you add modules that will cause the file to exceed its original size. However, the library will no longer be contiguous.)

The second parameter, 128_{10}, in the command gives the number of entry point table slots to reserve.[4] A good estimate for the number of entry points is twice the number of modules the library will contain (that is, two entry points per module). If you omit this parameter, LBR

[3] The numeric parameters are followed by decimal points to force LBR to interpret them as decimal numbers. If you omit the decimal points, LBR treats the numbers as octal.

[4] LBR allows you to build an object library having zero entry points. This feature allows you to maintain modules with duplicate entry points in the same library. (The names of the modules must still be unique.) When using such a library, you must specify the correct module name(s) to TKB when you build your task. See Section 6.2.2.

supplies 512_{10} as the default number. If the value you supply is not an integral multiple of 64_{10}, LBR raises the number to the next highest multiple of 64_{10}.

The third parameter, 64_{10}, is the number of module name table entries to create for the library. (The module name is the means by which LBR refers to the module code in the library.) If you omit this parameter from the command line, LBR supplies 256_{10} as the default number. If the value you specify is not an integral multiple of 64_{10}, LBR raises the number to the next highest multiple of 64_{10}.

The last parameter (omitted from the command line above) specifies the type of library to build. LBR supplies OBJ as the default type.

In creating the object library file, LBR allocates the requested amount of contiguous space. You can estimate the number of contiguous blocks that LBR requires by using the Peripheral Interchange Program (PIP). Request a directory listing of all the files to be inserted in the library and use the total number of blocks PIP calculates. If sufficient contiguous space is not available, LBR generates the OPEN FAILURE error and terminates. To have the library created, you must either free up some space on the volume or try to build a smaller object library.

When the object library is created, LBR attempts to insert into the library the object modules from the input file(s). It arranges the entries in the module name table in alphabetical order by module name. The module name that LBR uses is the one you specified in the .TITLE directive when you assembled the object module. The module names and entry points must be unique.[5] LBR finds the global symbols in each object module and enters them in the entry point table. If LBR finds a module name or an entry point that duplicates one already used, it prints an error message and stops processing.

If LBR finds an error, it does not insert any modules in the library from the file containing the error. You must eliminate the error condition and insert the modules from the corrected file again. If LBR does not find any errors, it enters all the modules in the library. To ascertain what modules were inserted, obtain a listing of the library, as described in Section 6.2.3.

6.2.2 Using the Object Modules from the Library

When the object modules are in the library, you need perform only two actions to have TKB include the routines in your task:

1. Include the CALL x statement in the calling module (where x is an entry point to the called module). (It is assumed that the called module has a global statement to define the entry point.)[6]
2. Specify the name of the library file and the names of the called modules in the command line to TKB.

Thus, to invoke subroutines from the library, ensure that the CALL statements are in your program.

[5] If you suppress including entry points in the library entry point table, LBR allows you to insert, in the library, object modules having duplicate entry points. This feature enables you to maintain slightly different modules of the same general type in the same library. You select the correct module by specifying the unique module name to TKB when you build your task. See Section 6.2.2.

[6] CALL is a macro statement that is a permanent symbol in the MACRO-11 Assembler. It standardizes subroutine calling conventions. CALL x translates to JSR PC,x (Jump to Subroutine program counter, where x is the subroutine entry point).

When you build a task, use the PDS command LINK, as follows:

```
PDS> LINK/TASK:SUPLIB/MAP:SUPLIB [RET]
File(s)?  FILE,USROBJ/INCLUDE:(TTREAD,TTWRIT) [RET]
```

or

```
PDS> LINK/TA:SUPLIB/MAP:SUPLIB FILE, USROBJ/INCLUDE:(TTREAD,TTWRIT) [RET]
```

By including file specifications as arguments to the /TASK and /MAP qualifiers, you cause the outfiles from the LINK command to be named SUPLIB.TSK and SUPLIB.MAP, respectively. The /INCLUDE qualifier identifies the file USROBJ.OBJ as an object library. The names appearing in parentheses, after /INCLUDE, are the names of the modules to be extracted from the library and placed in the task. (Remember that these module names are derived from the names given in the .TITLE directive in the MACRO source files, and not from the file from which these modules were assembled.)

This method of specifying an object library search is more direct and faster than the method described in Section 6.2.3. If you are using a large library, TKB need only search the module name table for those object modules you specify. The disadvantage is that you have the responsibility to specify the names of all the modules that your task requires. If, however, you are using a library with zero entry points, the /INCLUDE qualifier is the only method of telling TKB which modules to include from that library.

The following command line is the MCR equivalent for including specific object modules from a library in your task:

```
MCR> TKB SUPLIB,SUPLIB/-SP=FILE,USROBJ/LB:TTREAD:TTWRIT [RET]
MCR>
```

The /LB switch after a name in the command line indicates to TKB that the file is an object library. TKB accesses the file USROBJ.OLB in the directory that is the same as the current User Identification Code (UIC). The names appearing after the /LB switch in the command line are the names of the modules to be extracted from the library and placed in the task. TKB searches the module name table of the library for these modules. (Remember that these module names are derived from the name given in the .TITLE directive, and not from the file names from which the modules were created.)

Note that the module names in the command line are preceded by colons, which are necessary to distinguish the names as library module names. Placing a comma before a name tells TKB to treat the name as an object module and to search your directory for a file with that name and a type of OBJ. That is, the colon tells TKB to process what follows as an argument of the /LB switch, and the comma tells TKB to treat what follows as a file name.

This method of specifying an object library search is more direct and faster than the method described in Section 6.2.3. If you are using a large library, TKB need search only the module name table for those object modules you specify. The disadvantage is that the responsibility is yours to specify the names of all the modules that your task requires. In one situation, this is the only method by which to use a library: If you are using a library with zero entry points, this is the sole method of telling TKB which modules to include from that library.

6.2.3 Using the Library to Resolve Undefined Global Symbols

Often the modules in a task refer to global symbols that are defined in other modules. If the modules that define the global symbols reside in a library, you can have TKB search the library. In PDS, the /LIBRARY qualifier on an input file specification for a LINK command indicates that the entire library is to be searched. The /LIBRARY qualifier replaces the /INCLUDE qualifier. The following example shows the form of the command line:

```
PDS> LINK/TASK:LB/MAP:LB FILE, USROBJ/LIBRARY [RET]
```

The /LIBRARY qualifier tells TKB to search the library entry point table for symbols that are referred to but not defined. When TKB finds a symbol in the table that is unresolved in the task, TKB extracts the defining module and places it in the task. If any symbols remain unresolved after the user library search, TKB searches the system library.

This method requires less effort on your part than using the /INCLUDE qualifier, but if you are using a large library, the task build may take considerable time.

The following command line is the MCR equivalent for using the TKB switch /LB to search an entire library:

```
MCR> TKB LB,LB/-SP=FILE,USROBJ/LB [RET]
MCR>
```

The /LB switch with no module names tells TKB to search the library entry point table for symbols that are referred to but not defined. When TKB finds a symbol in the table that is unresolved in the task, TKB extracts the defining module and places it in the task. If any symbols remain unresolved after the user library search, TKB searches the system library.

This method of specifying an object library search requires less effort on your part than the method described in Section 6.2.2, because TKB searches the entry point table to resolve any global references undefined at that point in the processing. If you are using a large library, TKB may take longer in searching the entry point table than if you had specified the names of the modules to include in your task.

6.2.4 Dual Use of the Library

In certain circumstances, you might want TKB to include specific modules from the library and also to search the same library to resolve any undefined references. For example, you might have conditional code in the main part of a task and not know what global symbols are referenced. TKB enables you to specify the two forms of the library search. In PDS, you can do this by combining the /INCLUDE and /LIBRARY qualifiers in the same command line:

```
PDS> LINK/TASK:LBOPT/MAP:LBOPT [RET]
File(s)?  FILE, USROBJ/INCLUDE:TTREAD, USROBJ/LIBRARY [RET]
```

or:

```
PDS> LINK/TASK:LBOPT/MAP:LBOPT FILE, USROBJ/INC:TTREAD, USROBJ/LIBRARY [RET]
```

Once again, the arguments to the /TASK and /MAP qualifiers change the names of the associated output files. The /INCLUDE qualifier on the file specification for USROBJ.OLB instructs TKB to extract the named module. Notice that since only one module is named, the parentheses are unnecessary. The /LIBRARY qualifier on the file specification for USROBJ.OLB tells TKB to search that library for any unresolved global symbols. TKB includes in the task any modules from the library that are unresolved at that point in the building of the task. If any unresolved symbols remain after the search of the user library, TKB searches the system library.

The following command line shows the MCR procedure for specifying two forms of library search to TKB:

```
MCR> TKB LBOPT,LBOPT/-SP=FILE,USROBJ/LB:TTREAD,USROBJ/LB [RET]
MCR>
```

The first appearance of the /LB switch tells TKB to extract the named module. The second occurrence tells TKB to search the library for any unresolved global symbols. TKB includes in the task any modules from the library that define the global symbols that are unresolved at that point in the building of the task. If any unresolved symbols remain after the user library search, TKB searches the system library.

6.3 Maintaining User Libraries

This section decribes three simple operations used to maintain a user library: adding modules to, replacing a module in, and obtaining information about the library. In MCR, you can accomplish this with various commands to the library utility program (LBR). From PDS, use the LIBRARY/INSERT, LIBRARY/REPLACE, and LIBRARY/LIST commands.

6.3.1 Adding Modules to a Library

Add modules to a library with the LIBRARY/INSERT command. For example:

```
PDS> LIBRARY/INSERT [RET]
Library?  USRMAC.MLB [RET]
Module(s)?  MAC1,MAC2 [RET]
```

or:

```
PDS> LIBRARY/INSERT USRMAC.MLB MAC1,MAC2 [RET]
```

Give the name and type of the library in response to the "Library?" prompt. Give the names of the files containing the library modules in response to the "Module(s)?" prompt. The default file type for files containing library modules is the same as for the type of library that you specify.

You cannot add modules to a library that has no remaining entries in the module name table. (If you are creating an object module library, sufficient entry point table slots must exist as well.) When LBR inserts a module in a library, it checks to be sure that a module of the same name does not currently reside in the library. If such a module is found, LBR reports an error and exits. (For inserting object modules, LBR also checks for duplicate entry point names.) To add modules with duplication, see the discussion of the PDS command LIBRARY/REPLACE in Section 6.3.2.

From an MCR terminal, modules can be added to a library with an LBR command line such as the following:

```
MCR> LBR USRMAC.MLB/IN=MAC1,MAC2 [RET]
MCR>
```

To add modules to a library, specify the name and type of the library file and the /IN switch to the left of the equal sign in the LBR command line. To the right of the equal sign, give the name of the modules, separated by a comma. You need not supply a file type because LBR applies the correct type as a default according to the type of the library you specify.

The library must have a sufficient number of name table entries available (and, for object modules, entry point table slots). Each global symbol in an object module requires an available entry point table slot. A module name table entry must be available for each object module and macro definition added. When inserting a module, LBR checks to ensure that a module of the same name does not currently reside in the library. If a duplicate name is found, the program reports the

duplicate name and terminates. For object modules being inserted, LBR also checks for duplicate entry point names. To add modules with duplication, you must either eliminate the duplicate names or change the /IN switch to the /RP switch. See Section 6.3.2.

6.3.2 Replacing a Module in a Library

After you create a library, a typical maintenance function is changing and updating modules in the library. Because a module of the same name (and, for object modules, the same entry points) already exists, you must perform a replace operation.

In PDS, use the following LIBRARY/REPLACE command to accomplish the replacement operation:

```
PDS> LIBRARY/REPLACE [RET]
Library?  USROBJ [RET]
Module(s)?  FILEA [RET]
Module "TTREAD" replaced
```

or:

```
PDS> LIBRARY/REPLACE USROBJ FILEA [RET]
Module "TTREAD" replaced
```

This command line causes LBR logically to delete the module TTREAD and all associated entry points for that name from USROBJ.OLB. LBR then inserts the new version of module TTREAD from FILEA.OBJ and prints a message. If a module to be replaced is not found in the library, LBR performs an insertion but prints no message.

Note that LIBRARY/REPLACE causes a logical deletion and does not reclaim the space occupied by the module you replace. To reclaim this lost space, you should occasionally use the LIBRARY/COMPRESS command.

The following command line is the MCR equivalent for performing the replace operation:

```
MCR> LBR USROBJ/RP=FILEA [RET]
Module "TTREAD" replaced

MCR>
```

LBR accesses the library file USROBJ.OLB, logically deletes the module TTREAD and all of the entry points for that name, and inserts the new version of module TTREAD from the file FILEA.OBJ. LBR prints a message telling you the name of each module it replaced. If a module to be replaced does not exist in the library file, LBR assumes that the module is to be inserted, automatically inserts it, but does not print the message.

LBR does not automatically reclaim the space occupied by a module that you replaced. Therefore, to reclaim this lost space, you should occasionally run LBR and compress the library file.

6.3.3 Obtaining Information About a Library

To obtain information about a library from a PDS terminal, type a LIBRARY/LIST command in the following format:

```
PDS> LIBRARY/LIST:LBLIST/NAMES/FULL [RET]
Library?  DD.OLB [RET]
```

or:

```
PDS> LIB/LIS:LBLIST/NAMES/FULL DD.OLB [RET]
```

This command line causes LBR to access the library file DD.OLB. The list appears in the file in your directory called LBLIST.LST. The /FULL qualifier lists entry points and full information (size, date of creation, and, for object modules, identification).

To list the information on the terminal instead of a file, use the LIBRARY/LIST command without a file specification argument to the /LIST qualifier:

```
PDS> LIBRARY/LIST/FULL USRMAC.MLB [RET]
     (LBR lists information)
```

To obtain information about a library from an MCR terminal, type a command line to LBR similar to the following:

```
MCR> LBR DD.OLB,LBLIST/LE/FU/-SP [RET]
MCR>
```

This command line causes LBR to access the library file DD.OLB in the default directory. The comma separates the library file name from the listing file specification. The /LE and /FU switches cause LBR to list entry points and full information (size, date of creation, and, for object modules, identification) in the file LBLIST.LST in the default directory. The /-SP switch inhibits automatic spooling of the listing to the line printer.

To list information at the terminal, simply omit the file name from the command line, as follows:

```
MCR> LBR USRMAC.MLB/FU [RET]
     (LBR lists information)
MCR>
```

Because a macro library does not have entry points, you can omit the /LE switch from the command line.

```
MCR> LBR [1,1]USROBJ.OLB,[303,10]LBLIST/LE/FU [RET]
MCR>
```

This command line causes LBR to access the library file USROBJ.OLB in directory [1,1]. The comma separates the library file name from the listing file specification. The /LE and /FU switches tell LBR to list entry points and full information (size, date of creation, and, for object modules, identification) in the file LBLIST.LST in directory [303,10]. If you omit the directory specification from the listing file, LBR creates the listing file in the directory of the library.

To list information at the terminal, simply omit the file name from the command line, as follows:

```
MCR> LBR [1,1]USRMAC.MLB/FU [RET]
MCR>
```

Because a macro library does not have entry points, you can omit the /LE switch from the command line.

6.4 Guide to Further Reading

The following manuals and chapters contain additional information on the subjects described in this chapter:

IAS Task Builder Reference Manual

Chapter 4, "Qualifiers, Switches, and Options"

PDP-11/MACRO-11 Language Reference Manual

Chapter 7, Macro Directives

Chapter 8, Operating Procedures

IAS Utilities Manual

Chapter 10, Librarian Utility Program (LBR)

7 FORTRAN IV Procedures

PDP-11 FORTRAN IV is one of several high-level languages optionally available on the IAS operating system. This chapter briefly introduces the product and summarizes its program development procedures.

7.1 Overview of PDP-11 FORTRAN IV

The FORTRAN IV language processor on IAS consists of the following elements:

- The compiler task FOR
- The PDS command FORTRAN
- The MCR command FOR
- An Object Time System (OTS) library
- An optional resident library

The FORTRAN IV compiler accepts an American Standard Code for Information Interchange (ASCII) disk file containing source statements and generates a disk file in object module format and, optionally, a listing file suitable for printing. The user interface to the compiler is similar to that of the MACRO-11 Assembler. The program development procedures are like those for assembly language modules: you supply the object file to the Task Builder (TKB) to obtain an executable program.

The DIGITAL Command Language (PDS) command FORTRAN parallels in function the PDS command MACRO, and the Monitor Console Routine (MCR) command FOR parallels in function the MCR command MAC. The FORTRAN and FOR commands pass an ASCII source file to the FORTRAN compiler, and the compiler generates an object module and, if desired, a listing file. The program development procedures are very similar to those for assembly language modules: you pass the object module to TKB using a LINK command to obtain a file containing an executable task image.

The FORTRAN IV Object Time System (OTS) is a collection of object module subroutines the system uses as needed in creating an executable program. On systems with more than one high-level language, the OTS routines for FORTRAN IV must be segregated from those of other languages. Sometimes, the OTS routines reside in the system object library SYSLIB. Regardless of their location, however, the OTS routines must be accessible to TKB. The difference to you is whether the library containing the OTS routines must be explicitly named. If the OTS routines are in SYSLIB, TKB can locate them without an explicit specification because, as a default condition, it automatically searches the system library.

The FORTRAN IV compiler does not generate all of the machine code required by a task at run time. Common sequences of code reside in the OTS library. During compilation, FORTRAN IV flags these common sequences as undefined global symbols. TKB must then resolve the undefined references by selecting from the OTS those modules that resolve the symbols in the object module.

In a narrow sense, the OTS contains the routines that the compiler has previously designated to be linked into your task. In practice, however, the OTS can contain various routines, callable by the user, in addition to the routines required by the compiler-assigned references.

As an option, a system installation can have a common area containing shareable FORTRAN IV OTS routines. This common area, called a resident library, contains the most frequently used routines, taken from the OTS and made available for user tasks to link to and share at run time. Thus, with a resident library, TKB generates references to the routines in the resident library that you specify when you build the task. TKB does not include those routines in your task image. The routines use virtual address space in the task but do not require additional physical memory in the task image. The resident library, tailored to the needs and requirements of a particular system, saves task-build time and memory by the amount of code that need not be repeated in each memory-resident FORTRAN IV task.

7.2 FORTRAN IV Program Development Procedures

The program development procedures for FORTRAN IV are quite similar to those for MACRO-11. Therefore, this chapter does not include the level of detail found in previous chapters of this manual. To edit a FORTRAN IV source file, use the same commands you used to edit the MACRO-11 files shown in Chapter 2. If you are using an editor other than the Line Text Editor (EDI), perform the same operations using your editor.

7.2.1 Creating the Source File

To create a sample FORTRAN IV source file, invoke the editor task EDI and use the following commands to insert the lines of code shown in Example 7-1:

```
MCR> EDI AVERAGE.FTN [RET]
[CREATING NEW FILE]
INPUT
                    insert the lines here and
                    type the RETURN key twice to exit from
                    insert mode
[RET]
[RET]
* EXIT [RET]
[EXIT]

MCR>
```

Because EDI cannot insert a blank line in the text (EDI requires at least one nonprinting character such as a space or tab character; see Section 2.2.1.1), be sure to use the C (comment line) in column 1 of the source file in place of the blank line, for readability. If you insert a line with a space or tab character in it, the FORTRAN IV compiler generates an error because it expects a valid label on a nonblank line.

You can use the TAB character to facilitate line formatting. The FORTRAN IV compiler positions the character following an initial TAB character to the proper column. That is, a digit following an initial TAB is considered a continuation character (column 6), and a nondigit is considered the beginning of the statement (column 7).

Example 7–1 FORTRAN IV Sample Source Code AVERAGE.FTN

```
      PROGRAM AVERAGE
C     PROGRAM TO COMPUTE AVERAGE OF NUMBERS ENTERED AT TERMINAL
C     THE NUMBER '0' INDICATES END OF INPUT
C
      TOTAL = 0                ! INITIALIZE ACCUMULATOR
      N = 0                    ! INITIALIZE COUNTER
5     N = N + 1
      WRITE (5,10)             ! PROMPT TO ENTER NUMBER
10    FORMAT (' ENTER NUMBER, END WITH 0')
      READ (5,20) K            ! READ NUMBER FROM TERMINAL
20    FORMAT I10
      IF (K .EQ. 0) GOTO 4     ! 0 MEANS NO MORE INPUT
      TOTAL = TOTAL + K        ! COMPUTE TOTAL WITH NUMBER
      GO TO 5
C
C NOW, COMPUTE TOTAL BY DIVIDING IT BY THE NUMBER OF TIMES
C THROUGH THE LOOP
C
40    TOTAL = TOTAL/N
      WRITE (5,50) TOTAL       ! DISPLAY THE RESULT
50    FORMAT (' AVERAGE IS ',F10.2)
      STOP
      END
```

7.2.2 Performing a Diagnostic Run

To determine whether there are any syntax or grammar errors in a source file, you can perform a diagnostic run using the PDS command FORTRAN, as follows:

```
PDS> FORTRAN/NOOBJECT    AVERAGE/LIST [RET]
AVERAG
FOR -- [AVERAG] ERRORS: 1, WARNINGS: 0
```

This command line requests FORTRAN IV to compile the file AVERAGE.FTN and create a listing file, AVERAGE.LST, but no object file. By default, the listing file contains source code and diagnostic messages only.

When you request a listing file in a compilation, FORTRAN IV reports at the terminal the name of the program unit being compiled and a summary of any errors. To discover what caused any errors, you must examine the section of the listing entitled FORTRAN IV DIAGNOSTICS. List the file on your terminal with the PDS command TYPE, as follows:

```
PDS> TYPE AVERAGE.LST [RET]

          (PIP displays listing)
```

See the following discussion of the MCR format for information on reading the listing file.

To perform the diagnostic run from an MCR terminal, issue the following command line:

```
MCR> FOR ,AVERAGE/-SP=AVERAGE [RET]
AVERAG
FOR -- [AVERAG] ERRORS:  1, WARNINGS:  0
MCR>
```

This command line requests FORTRAN IV to compile the file AVERAGE.FTN, which resides in your directory. The compiler creates a listing file, AVERAGE.LST, but no object module. (The leading comma in the command means a null file specification for the object file. If you omit the

comma, FORTRAN IV creates the object file but not the listing file.) As a default condition, the listing file contains source code and diagnostic messages only.

When you request a listing file in a compilation, FORTRAN IV reports at the terminal the name of the program unit being compiled and a summary of errors found. To discover what caused any errors, you must examine the section of the listing entitled FORTRAN IV DIAGNOSTICS. Display the listing file by typing the following MCR command line:

```
MCR> PIP TI:=AVERAGE.LST [RET]

     (PIP displays listing)

MCR>
```

On a video display terminal, use the CTRL/S and CTRL/Q commands to stop and resume output.

The following line appears in the diagnostic section of the listing:

```
IN LINE 0008,   ERROR:   SYNTAX ERROR
```

Line 0008 refers to the statement number 0008 assigned by the compiler. The error referred to is described in an appendix of the language user's guide. In the source code part of the listing, line 0008 is shown as follows:

```
0008    20    FORMAT I10
```

The compiler detected missing parentheses on the field descriptor in the FORMAT statement. To correct the error, you must edit the source file, as shown in the following example:

```
MCR> EDI AVERAGE.FTN [RET]
[00023 LINES READ IN]
[PAGE 1]
* L I10 [RET]
20      FORMAT I10
* C /I10/(I10)/ [RET]
20      FORMAT (I10)
* EXIT [RET]
[EXIT]

MCR>
```

The L command locates the line containing the string I10 and prints the entire line. The C command replaces the string I10 with (I10) and prints the line so that you can verify the change. The EXIT command terminates the editing session and creates the new, edited version of the file. Next, you can use the edited version to create an object module.

7.2.3 Creating an Object Module

To create an object module, simply omit the /NOOBJECT qualifier from the PDS command line you entered previously, as follows:

```
PDS> FORTRAN AVERAGE/LIST [RET]
AVERAG
```

This command line requests FORTRAN IV to compile file AVERAGE.FTN and to create listing file AVERAGE.LST and object file AVERAGE.OBJ. If FORTRAN IV detects any errors, it prints a summary at the terminal (see Section 7.2.1. If no errors occur, FORTRAN IV returns control to PDS, which prints the "PDS>" prompt.

The same procedure from an MCR terminal is as follows:

```
MCR> FOR AVERAGE,AVERAGE/-SP=AVERAGE [RET]
AVERAG
MCR>
```

This command line requests FORTRAN IV to compile file AVERAGE.FTN and to create object file AVERAGE.OBJ and listing file AVERAGE.LST. If FORTRAN IV detects any errors, it prints a summary at the terminal as described in Section 7.2.1. If no errors occur, FORTRAN IV returns control to MCR, which prints the ">" prompt.

7.2.4 Creating a Task Image

FORTRAN IV object modules do not contain all the object code required at run time. Therefore, when you run TKB, either from MCR or with the PDS command LINK, you must specify as input both the name of the object module and the name of the library containing the FORTRAN IV OTS routines. The commands for both PDS and MCR users follow:

```
PDS> LINK AVERAGE, LB:[1,1]FOROTS/LIBRARY [RET]
```

or

```
MCR> TKB AVERAGE=AVERAGE,LB:[1,1]FOROTS/LB [RET]
```

These command lines request TKB to resolve any undefined references by searching the library FOROTS.OLB in directory [1,1] on the system library device and by linking compiler-designated routines to module AVERAGE.OBJ.[1] You can add, as input to TKB, file names of any external object modules that the main module calls. As a result of the command line, TKB creates a task image file AVERAGE.TSK. (A memory allocation file is not needed.) If TKB detects any errors, it proceeds according to whether the error is fatal or diagnostic. Refer to the *IAS Task Builder Reference Manual* for guidelines on error processing.

When building a task image, the library FOROTS.OLB might contain routines that support the FP-11 Floating Point Processor. If this is the case, the commands for PDS and MCR users are as follows:

```
PDS> LINK/CODE:FPP AVERAGE,LB:[1,1]FOROTS/LIBRARY [RET]

MCR> TKB AVERAGE/FP=AVERAGE,LB:[1,1]FOROTS/LB [RET]
```

You also must use these commands when you rebuild your task after debugging it.

The task image created by TKB has certain default conditions. The task AVERAGE can be built to run successfully without having to override these default conditions. When you build a task from a FORTRAN IV module, you might have to specify special switches in the command line or supply options to TKB. Refer to the specific language's user's guide for information regarding TKB default FORTRAN IV conditions and FORTRAN-specific options and switches.

[1] In the command line, the name shown for the FORTRAN IV Object Time System (FOROTS) is only a convention recommended by DIGITAL. Consult the system manager at your installation because the FORTRAN IV OTS routines can reside in another library or in the system library SYSLIB. (If the OTS routines do reside in SYSLIB, you need not specify the name of the OTS in the command line to TKB because TKB automatically searches the system library.)

7.2.5 Running and Debugging a Task

To execute the task AVERAGE, type the following PDS or MCR command line:

```
> RUN AVERAGE [RET]
```

The program then displays the following lines on your terminal:

```
ENTER NUMBER, END WITH 0
66 [RET]
ENTER NUMBER, END WITH 0
66 [RET]
ENTER NUMBER, END WITH 0
0 [RET]
AVERAGE IS      44.00
TT30  --  STOP
>
```

Obviously, 44 is not the average of 66 and 66; therefore, the program must contain an error. If you cannot locate the error by looking at the program listing, you can attempt to locate the error by placing debugging statements in the code.

To add debugging statements to the program, simply edit the source file with lines of code beginning with D in column 1. For example, you can include statements to print values of variables before and after the loop, as follows:

```
MCR> EDI AVERAG.FTN [RET]
[00023 LINES READ IN]
[ PAGE    1]
* L 5 [RET]
5 N = N + 1
* I [RET]
 D [TAB]    WRITE (5,6) N,TOTAL [RET]
D6 [TAB]    FORMAT (' ***DEBUG LINE N = ',I10,', TOTAL = ',F10.0) [RET]
[RET]
* L 50 [TAB] [RET]
50 FORMAT (' AVERAGE IS ',F10.2)
* I [RET]
D [TAB]     WRITE (5,51) N [RET]
D51 [TAB]   FORMAT (' ***DEBUG LINE N = ',I10) [RET]
[RET]
* EXIT [RET]
[EXIT]
MCR>
```

The L commands locate and print the contents of the lines that precede where debugging statements are to be placed. The I commands insert the debugging statements. You terminate the insert operation by pressing the [RETURN] key twice. After the inserts are made, the EXIT command closes the file and terminates EDI.

Next, recompile the module and request FORTRAN IV to include the debugging statements, as shown in the following PDS and MCR command lines:

```
PDS> FORTRAN/OBJECT:DEBUG/D_LINES/LIST:DEBUG [RET]
File(s)?  AVERAGE [RET]
AVERAG
```

or:

```
MCR> FOR DEBUG,DEBUG=AVERAGE/DE [RET]
```

The compiler generates the files DEBUG.OBJ and DEBUG.LST. Because of the /D_LINES qualifier (/DE switch in the MCR line), the compiler includes statements beginning with D in column 1. If you omit this qualification, the debugging lines are treated simply as comment lines.

Now, rebuild the task with debugging lines, using the PDS or MCR command lines as follows:

```
PDS> LINK DEBUG,LB:[1,1]FOROTS/LIBRARY [RET]
```

or:

```
MCR> TKB DEBUG=DEBUG,LB:[1,1]FOROTS/LB [RET]
MCR>
```

Note that this operation has nothing to do with the on-line debugging tool (ODT), which is a MACRO-11 debugging tool.

Run the task with the following PDS or MCR command line:

```
> RUN DEBUG [RET]
***DEBUG LINE N =      1, TOTAL =     0.
ENTER NUMBER, END WITH 0
66 [RET]
***DEBUG LINE N =      2, TOTAL =     66.
ENTER NUMBER, END WITH 0
66 [RET]
***DEBUG LINE N =      3. TOTAL =     132.
ENTER NUMBER, END WITH 0
0 [RET]
AVERAGE IS      44.00
***DEBUG LINE N =      3
TT30 -- STOP
>
```

The debugging statements enable you to inspect the values of variables. As you can see, the loop counter N is incremented one extra time for the number 0. The value N must be decremented by 1.

To correct the error, edit the source file as follows:

```
MCR> EDI AVERAGE.FTN [RET]
[00027 LINES READ IN]
[PAGE    1]
* L TOTAL/ [RET]
40 TOTAL = TOTAL/N
* C ;N;(N-1); [RET]
40 TOTAL = TOTAL/(N-1)
* EXIT [RET]
[EXIT]
MCR>
```

Next, repeat the compile, link, and run operations. From a PDS terminal, use the following sequence of command lines:

```
PDS> FORTRAN AVERAGE/LIST [RET]
AVERAG
PDS> LINK AVERAGE,LB:[1,1]FOROTS/LIBRARY [RET]
PDS> RUN AVERAGE [RET]
ENTER NUMBER, END WITH 0
66 [RET]
ENTER NUMBER, END WITH 0
66 [RET]
ENTER NUMBER, END WITH 0
```

```
0 [RET]
AVERAGE IS 66.00
TT30 -- STOP
```

The program is compiled without the debugging statements. The output shows that the correction eliminated the error.

The same procedure from an MCR terminal is as follows:

```
MCR> FOR AVERAGE,AVERAGE/-SP=AVERAGE [RET]
AVERAG
MCR> TKB AVERAGE=AVERAGE,LB:[1,1]FOROTS/LB [RET]
MCR> RUN AVERAGE [RET]
ENTER NUMBER, END WITH 0
66 [RET]
ENTER NUMBER, END WITH 0
66 [RET]
ENTER NUMBER, END WITH 0
0 [RET]
AVERAGE IS   66.00
TT30 -- STOP
MCR>
```

The program is compiled without the debugging statements. The output shows that the correction eliminated the error.

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

X

Reader's Comments

This form is for document comments only. Digital will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify)_____

Name_____Date_____

Organization_____

Street_____

City_____State_____Zip Code_____
or Country

Do Not Tear - Fold Here and Tape

digital™

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO 33 MAYNARD MASS

POSTAGE WILL BE PAID BY ADDRESSEE

IAS Engineering/Documentation
Digital Equipment Corporation
5 Wentworth Drive GSF/L20
Hudson, NH 03051-4929

Do Not Tear - Fold Here