RSX–11M/M–PLUS Guide to Program Development

Order No. AA-H264A-TC

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

digital equipment corporation · maynard, massachusetts

First Printing, May 1979

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

Copyright © 1979 by Digital Equipment Corporation

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

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	TMS-11
ASSIST-11	RTS-8	ITPS-10
VAX	VMS	SBI
DECnet	IAS	PDT
DATATRIEVE	TRAX	

CONTENTS

CONTENTS

CONTENTS

FIGURES

TABLES

PREFACE

MANUAL OBJECTIVES

The RSX-11M/M-PLUS Guide to Program Development introduces the program development environment on the RSX-11M and RSX-11M-PLUS systems. It provides a synopsis of the information that has immediate usefulness in getting started in the program development process. In addition, the book 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 RSX-11 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 Introduction to RSX-11M-PLUS or the RSX-11M Beginner's Guide. Both 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 in which you can emulate them 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 as follows:

- Chapter 1, The Program Development Environment, introduces the software and hardware on which you develop programs.

- Chapter 2, Creating MACRO-11 Source Files, describes how to create an assembly language source program using a skeleton file and text editor.
- Chapter 3, Assembling and Correcting a Program Module, describes how to use the MACRO-11 assembler to generate an object module.
- Chapter 4, Building and Testing a Task, describes how to use the Task Builder to link object modules to create a loadable task image.
- Chapter 5, Using Debugging Aids, introduces debugging aids and discusses how to use them.
- Chapter 6, Creating and Using Program Libraries, describes how to create and maintain a library of macro source statements and a library of object module subroutines.
- Chapter 7, FORTRAN IV Procedures, 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 RSX-11M Documentation Directory and the RSX-11M-PLUS Documentation Directory list and describe all the documents in the documentation sets for each system.

CONVENTIONS USED IN THIS DOCUMENT

Throughout this book, symbols and other notation conventions are used to represent keyboard characters, to convey textual information, and to otherwise ease the presentation of material. The symbols and conventions used are explained below.

Convention	Meaning
(XXX)	A one- to three-character symbol indicates that you press a key on the terminal; for example, (RET) indicates the RETURN key and (LF) indicates the LINE FEED key.
(CTRL/X)	The symbol (CTRL/X) indicates that you must press the key labeled CTRL while you simultaneously press another key; for example, (CTRL/O) indicates the CTRL and O keys. In examples, this control key sequence is shown as ^x; for example, ^O indicates the result of typing (CTRL/O) because that is how the system echoes control key combinations.

Convention	Meaning
^	The circumflex character, appearing with another character, represents the system response to receiving a control character (CTRL/x). For example, when you type the CTRL/Z combination while running some system tasks, the system echoes ^Z. (On some terminals, the circumflex is replaced by the up-arrow (↑) character.)
"print" and "type"	As these words are used in the text, the system prints and the user types.
MCR>	The explicit prompt of Monitor Console Routine (MCR), the command interface used on RSX-11M and one interface available on RSX-11M-PLUS systems and the one used in this book.
>	A greater-than sign is the system command interface prompting character. Whenever control is returned to the user task terminal and you can type input, the prompt appears. RSX-11M systems have only Monitor Console Routine (MCR) but RSX-11M-PLUS systems may have both MCR and DIGITAL Command Language (DCL). To determine which command interface your terminal has, simply type the CTRL/C combination and the explicit prompt (either MCR> or DCL>) will appear.
red ink	Color-highlighted information in examples indicates information that you type. Information in examples not in the contrasting color constitutes computer output.
,	Commas in commands separate parameters. They also indicate positional entries on a command line.
.	A dot in a file specification separates the file name and the file type.
;	A semicolon in a file specification separates the file type and file version number.
/	A slash character in a file specification precedes a switch. Switches modify command action.

CHAPTER 1

THE PROGRAM DEVELOPMENT ENVIRONMENT

This chapter introduces the software and hardware that you typically need to develop programs on an RSX-11M or RSX-11M-PLUS 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

RSX-11M and RSX-11M-PLUS make software tools available to users as executable entities called system tasks. A system manager makes these tasks accessible by installing them on the system. To invoke an installed task, you need not know where the task resides. To request a task's services, you need only know the 3-character name of the task. The tools described in this guide should be installed on most systems.[1]

1.1.1 Text Editor

A text editor is the means by which you create source code. The examples in this book show the editor EDI. EDI is an interactive editing program that enables you to enter ASCII text at a terminal and store the text in a disk file. EDI also lets 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.

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 and read, from the beginning of the file, to the block of text.

---

[1] On systems with fewer resources, you may be required to invoke some system tasks that are not permanently installed. On such systems, you may need to use the RUN command and need to know in which UFD a task resides. This manual assumes that all tasks are installed.

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 is 0-length because EDI did not have time to preserve the edits from the temporary file.

1.1.2 **Assembly Language**

RSX-11M and RSX-11M-PLUS 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 of zero. The allocation of virtual addresses and relocation is left for the task building process.

Source input to MACRO-11 consists of free-format statements, each line of input containing a single statement. Input statements are either PDP-11 instructions, MACRO-11 assembler directives, macro calls, 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. A user-defined symbol is 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 classed 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. Using data storage directives, you can control the size and contents of data areas.

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 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 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, to be resolved by the Task Builder.

During the second pass, the assembler actually generates the object module and listing files, flagging with an error code in the listing file those source statements containing 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 1.1.5 in this chapter.)

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 vital 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 Task Builder map), allows you to access locations in the memory-resident task image during debugging.

1.1.3 Task Creation

The Task Builder (TKB) on RSX-11M and RSX-11M-PLUS systems is a multiple purpose 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 sharable 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.

To build a task image, TKB accepts, as basic input, the output of a language processor - an object module or multiple object modules. The Task Builder 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, the Task Builder'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 the creation of 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, the Task Builder 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 RSX-11M/M-PLUS Task Builder 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.

1.1.4 Debugging Aids

This section introduces the debugging aids described in this guide and provided with RSX-11M and RSX-11M-PLUS systems to assist in identifying faulty code.

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

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. ODT enables the BPT synchronous system trap (SST) entry point in the task. 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.

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.

1.1.4.2 **Postmortem Dump** - Postmortem Dump (PMD) is an installed task that is directed by the Executive to extract run-time related data about a terminated task, format it, and request a printed listing.[1] Normally, when a task generates a synchronous system trap (SST), such as caused by an improper reference to an odd address or a reference to a nonexistent memory location, the Executive tries to transfer control to an SST entry point defined by the task. If the task does not have an SST routine defined for the particular type of trap, the Executive begins abnormal task termination.

[1] PMD requires that the Executive option for abnormal task termination and device-not-ready messages be selected at system generation time.

To terminate the task, the Executive performs an abort operation and notifies the Task Termination Notification (TKTN) task. TKTN displays, on the user terminal, the reason for the termination and the contents of the task registers.[1] Without PMD, you can acquire no further information about the task.[2]

By enabling Postmortem Dumps for a task which itself does not handle synchronous system traps, you tell the Executive to supply more data at abnormal task termination. That is, the Executive follows the abort procedure and, in addition, creates a request for PMD to create the dump. PMD examines system and task structures to preserve status and run-time data, reads the task image from memory, and writes it to disk in a readable format. PMD then queues a request to print the file containing the dump data, after which the Executive completes the task abort procedure.

1.1.4.3 **Snapshot Dump** - The snapshot dump ($SNAP), also using PMD, generates an edited dump of a running task. Because the snapshot dump requires you to insert special code (for example, the $SNAP macro call) in a task, it is more difficult to use than PMD. However, by inserting the snapshot dump code in the task, you can choose the location at which the dump is created and select the extent and format of the dump. In addition, you can generate the dump from more than one location and, therefore, as many times as needed during task execution.

It is often useful to include debugging facilities such as $SNAP in your task based on defining a conditional variable. To include the facility while you are debugging, simply define the variable. You can then omit the facility merely by reassembling the code with the conditional variable undefined.

1.1.5 General Utilities

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

1.1.5.1 **Cross-Reference Processor** - The Cross-Reference Processor (CRF) is an installed task that receives requests from MACRO-11 and the Task Builder 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.

1.1.5.2 **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 User

[1] The TKTN task must be installed on the system to display the messages.

[2] Commands exist that allow you to fix a task in memory and physically examine the contents of the task image. However, this is an involved procedure and beyond the scope of this book.

File Directory (UFD) to another, obtaining directory listings, renaming files, deleting files, and changing file protection codes. PIP handles all file-structured devices and is used for almost all file operations. The noteworthy exception to PIP capabilities is for certain PDP-11 Record Management Services (RMS-11) file operations, for which DIGITAL supplies special RMS-11 utilities.

1.1.5.3 **Queuing and Spooling** - RSX-11M and RSX-11M-PLUS systems differ in the manner in which each provides queuing and spooling facilities but both systems generally offer the same user functions. Almost all program development tasks automatically generate requests to the proper queuing tasks to print an ASCII output file on the system 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.

1.1.5.4 **Librarian Operations** - The Librarian program (LBR) can create and maintain specially formatted library files on disk: one for macro call definitions and one for object module subroutines.[1] The MACRO-11 assembler and the Task Builder 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 the Librarian are in the same format as those that DIGITAL supplies with the operating system.

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 object libraries as input to the Task Builder. The following two subsections describe these system libraries.

MACRO LIB → ASSEMBLER
Object libraries → Task Builder.

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 the User File Directory or UFD. The UFD is [1,1] on the system library device (referenced explicitly by the device-independent designation LB:). Table 1-1 summarizes the macro libraries DIGITAL supplies.

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.

---

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

Table 1-1
DIGITAL-Supplied Macro Libraries

File Name and Type	Description of Contents
RSXMAC.SML	System Macro Library. Contains the macro definitions for all RSX-11M and RSX-11M-PLUS 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 System (RMS-11). Contains the definitions for RMS-11 calls for sequential and relative file I/O. If your system has the optional RMS-11K software, this library will also contain calls for indexed file operations.

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. 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 enabling a task to issue system directives and to obtain 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 RSX-11M and RSX-11M-PLUS 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 building privileged tasks and for incorporating specially written device drivers in the system. (This topic is covered fully in the RSX-11M-PLUS Guide to Writing an I/O Driver, the RSX-11M Guide to Writing an I/O Driver, and the RSX-11M/M-PLUS Task Builder Manual and is not mentioned further in this guide.)

The Record Management System library RMSMAC.MLB is provided to support file and record access to RMS-11 data. RMS-11 is an upwards-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 RSX-11M and RSX-11M-PLUS systems, system object libraries provide general utility functions and special-purpose Executive features. These libraries, like the macro libraries, reside in UFD [1,1] on the system library device (LB:). Table 1-2 lists and describes the object libraries 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.
EXELIB.OLB	Executive Library. Contains the definitions of the Executive symbols.
ANSLIB.OLB	ANSI Magnetic Tape Library. On RSX-11M systems only, an alternate version of SYSLIB. Contains ANSI magnetic tape label handling routines and FCS big buffering support. (On RSX-11M-PLUS systems and systems with limited disk space, these routines are in SYSLIB.)
RMSLIB.OLB	Record Management System. Contains the routines for sequential and relative (RMS-11) and, optionally, indexed (RMS-11K) I/O.
FOROTS.OLB F4POTS.OLB	FORTRAN IV and FORTRAN IV-PLUS Library (optional). The Object Time System (OTS) and other routines for the PDP-11 FORTRAN IV language processors.

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 the Task Builder to resolve.

To ensure that subroutines are placed in the task image, the Task Builder, 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 defining 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. The Task Builder 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 local 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 the Peripheral Interchange Program (PIP).

1.3.1 Disks

Disks are the main storage media on RSX-11M and RSX-11M-PLUS 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 disk storage as a 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 hard-copy terminal. Video-display terminals are more convenient because they typically operate at faster rates than hard-copy devices. Hard-copy 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. If you are not familiar with using a terminal, you should read either the Introduction to RSX-11M-PLUS or the RSX-11M Beginner's Guide. Both of these documents explain how to access the system and use basic system commands.

1.3.3 Printers

Printers provide volume hard-copy output of data. On larger systems, you communicate with the printer through intermediate software called spooling programs. On smaller systems, you may have to specify explicitly that output is to go to a printer device in the absence of spooling programs.

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 through 7.

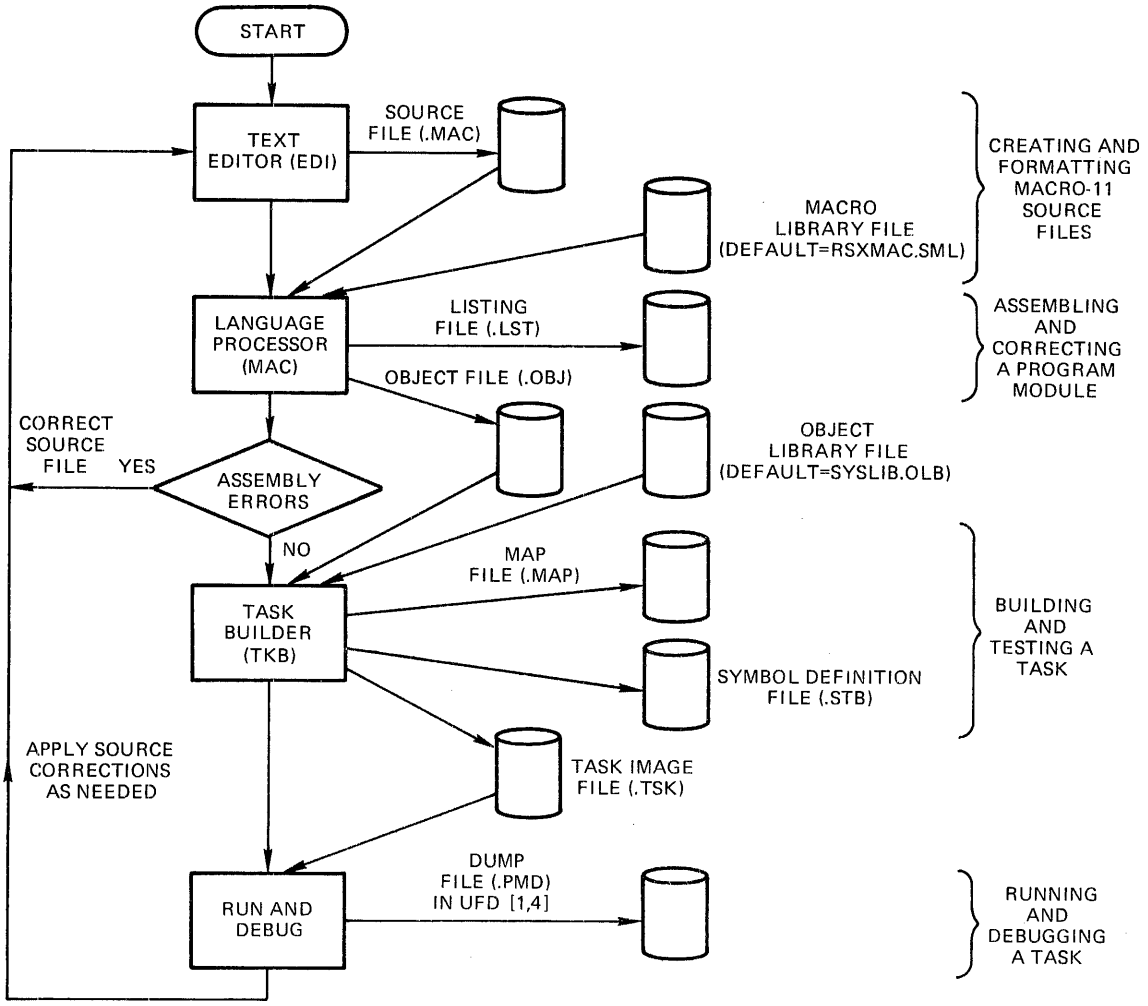

Figure 1-1 The Program Development Process

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 module to the Task Builder 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.

The Task Builder creates the file of loadable code, assuming certain default conditions about the run-time environment and building these characteristics into the task. The Task Builder 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 are encountered, you must edit the source file, reassemble or recompile, build a new task image file and try again.

1.5 GUIDE TO FURTHER READING

The sections or chapters in the following documents contain additional information on the subjects described in this chapter.

Document	Location
RSX-11M-PLUS Operating System Manual	Chapter 5, Program Development Facilities Section 6.2, Editing Facilities Section 6.3, File Utilities Section 6.5, Cross Reference Processor (CRF)
Introduction to RSX-11M-PLUS	Chapter 1, M-PLUS How-To Chapter 2, Learning the System
RSX-11M Beginner's Guide	Chapter 1, The Terminal Chapter 3, The Files
Introduction to RSX-11M	Chapter 5, Program Development

CHAPTER 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 which you can use as a framework for all your source programs. This chapter 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, and explains some elementary editing commands 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 IAS/RSX-11 MACRO-11 Reference Manual, the reference for which is given in the list of further reading at the end of this chapter.

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.

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, appearing toward the front 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.

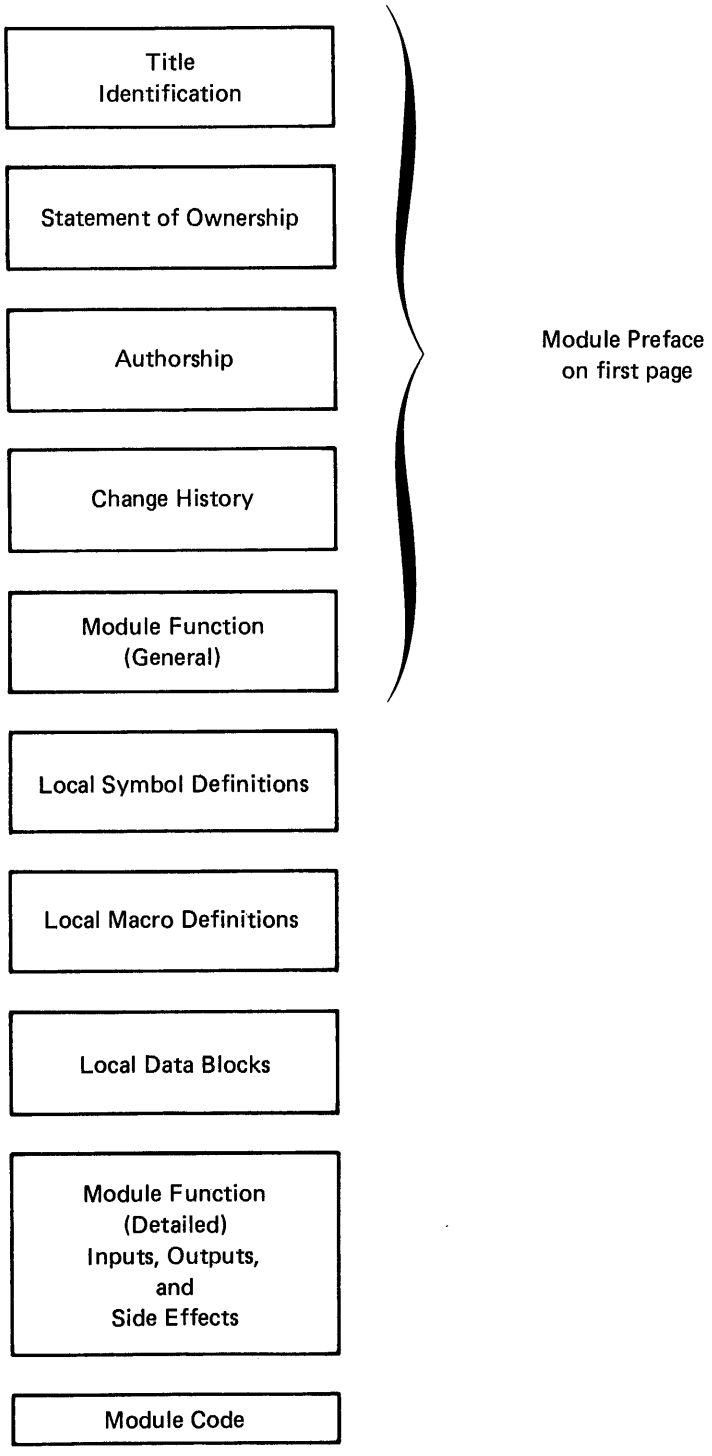

Figure 2-1 MACRO-11 Source File Format

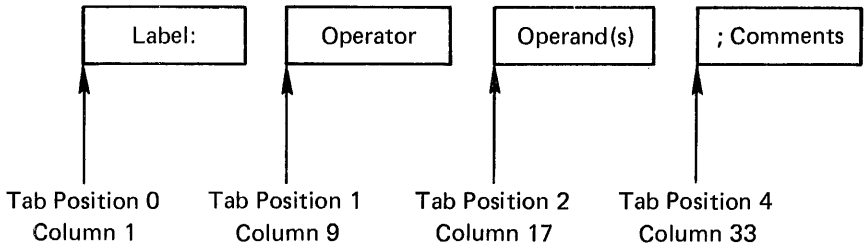

Figure 2-2 MACRO-11 Source Statement Format

Although the assembler allows 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. Figure 2-3 shows a sample skeleton.

Sections 2.1.1 through 2.1.12 describe the parts of the source file skeleton in detail.

2.1.1 .TITLE Directive

The .TITLE directive allows you to name the module. The assembler takes the first six nonblank 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 generally 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
```

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 from which the module is created. (Note that the sample code and commands shown in this guide use different names to help you distinguish their usages.)

```
        .TITLE  SKELTN  SOURCE FILE SKELETON
        .IDENT  /01/
;
;
; AUTHOR: Z
;

;
; CHANGES:
;

;
; MODULE FUNCTION:
;

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

;
; LOCAL SYMBOL DEFINITIONS:
;

;
; LOCAL MACROS:
;

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

;
; FUNCTION DETAILS:
;
;
;       INPUTS:
;
;       OUTPUTS:
;
;       SIDE EFFECTS:
;

; START CODE HERE

        .PAGE
        .SBTTL
        .PSECT
START:
END:    EXIT$S                          ; EXIT CLEANLY TO EXEC
        .END                            ; TELL ASSEMBLER END OF CODE
```

Figure 2-3 Sample Source File Skeleton

Moreover, the name the assembler extracts from the .TITLE directive is important in subsequent steps of program development. The Task Builder 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 object module is inserted in an object library, the Librarian program keeps this name in the directory of the library to refer to the object module.

2.1.2 .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.

2.1.3 Author Line

The author line identifies the originator of the code.

2.1.4 Changes Section

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 tag a change. The author of the change can identify the change in this section and flag each line of code with an additional comment such as the following.

```
; TOM JONES     2-APR-78        ;TJ001
;       ADD STATE TAX TO TOTAL
```

Then, in the code a changed or added line can be flagged with the notation TJ001.

```
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.[1]

2.1.5 Module Function

In the module function part of the source file, you can describe the general processing operations 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.

---

[1] A utility called the Source Language Input Program (SLP) is supplied with the system and can be used for source file maintenance. SLP provides the means to update lines in an existing source file and to apply an audit trail to identify lines deleted, replaced, and added.

2.1.6 Some Useful Directives

Between the module function description and the local symbol definitions is a convenient place to insert some general purpose directives. The following subsections describe these directives.

2.1.6.1 **.PAGE Directive** - The .PAGE directive causes a page break in the assembly listing. It appears as shown in Figure 2-3 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.

2.1.6.2 **.SBTTL Directive** - The .SBTTL 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 ought to 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 probably will not find the table of contents feature very useful.

2.1.6.3 **.LIST TTM Directive** - The .LIST TTM directive creates a listing formatted more conveniently for ouput on a terminal. (Section 3.4 of this guide shows how to display a listing at a terminal.) The directive can be included during the early stages of program development and removed from the stabilized code.

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

2.1.6.5 **.MCALL Directive** - The .MCALL directive is the means by which you 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$S directive (shown in the .MCALL statement) should be in every user program that is designed to exit gracefully. It should be the last statement the program (task) executes before it returns control to the Executive. (The EXIT$S directive performs important system housekeeping operations for the task.) The related definition

---

[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.

for EXIT$S resides in the file RSXMAC.SML in UFD [1,1] on the library device (LB:). DIGITAL recommends that user tasks exit by using the EXIT$S directive. (An alternative form of exiting allows 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 an .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 may generate an error because of illegal syntax for a .WORD directive.) The object code is not assembled in the object module. Later, when you build the task with the related object module and the macro name is not a valid symbol, the Task Builder 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.

2.1.6.6 **.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 at which 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 on which the .END directive appears.

2.1.7 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 and then use the symbol in the buffer definition.

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

This method has several advantages. First, 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. Second, if all the symbols are gathered in one place in alphabetical order, reading the code is much simplified. Third, 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.

2.1.8 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, then, to place all macro definitions in a standard place near the front of the source file.

2.1.9 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. See Section 2.1.11.

2.1.10 Module Function - Detailed

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

2.1.11 .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 may give a program section for data either the read-only (RO) or the read/write (RW) attribute.[1] (The assembler applies other, default attributes not relevant to this discussion.)

The three most important aspects of the .PSECT directive are: (1) contributions defined for a specific program section can be in separate places in a source file or in separate source files; (2) the attributes of the program section are passed to the Task Builder; and (3) contributions for a specific program section with the same attributes are collected in one continuous allocation of memory space by the Task Builder. 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.

[1] RSX-11M systems do not support hardware protection of program sections that have the RO attribute. RSX-11M-PLUS systems support hardware protection of program sections that have the RO attribute if they are in the pure code of a multiuser task. Consult the RSX-11M/M-PLUS Task Builder Manual for a discussion of program section allocation in multiuser tasks.

2.2 CREATING A SOURCE FILE FROM A SKELETON FILE

This section describes how to use an editor, EDI, to create a skeleton file and then to create a source file from the skeleton.

2.2.1 Performing the Initial Input

To create the skeleton file, run the editor by typing the command EDI and the specification of a new file (one that is not in your directory).

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

The editor runs, determines that the file does not exist, creates the file, and tells you to begin typing the input.

Type the input according to Figure 2-4. Leave any typographical errors until after you have become familiar with the editing commands described in Section 2.3. The notation conventions appearing in 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 Figure 2-4, type a space or tab on a new line followed by the RETURN key. If you type the RETURN key twice in succession (that is, type the RETURN key to enter a line of text and immediately type the RETURN key again on the new line), EDI terminates the input. Thus, to enter a blank line, you need type only one nonprinting character, such as TAB, on a new line.

2.2.1.2 **Terminating the Input and the EDI Program** - To terminate the input, type 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]

>
```

When EDI exits, it prints the message [EXIT] and returns control to MCR. The MCR prompt (>) indicates that the command interpreter is ready to accept a new command.

```
>EDI SKEL.MAC
[CREATING NEW FILE]
INPUT
(TAB)      .TITLE (TAB) SKELTN (TAB)  SOURCE FILE SKELETON
(TAB)      .IDENT (TAB) /01/
;
;
; AUTHOR: Z
;
(TAB) (RET)
;
; CHANGES:
;
(TAB) (RET)
;
; MODULE FUNCTION:
;
(TAB) (RET)
;
(TAB)      .PAGE (TAB) (TAB)            ; BREAK PAGE FOR PREFACE
(TAB)      .SBTTL (TAB) SYMBOL, MACRO, DATA DEFINITIONS
(TAB)      .LIST (TAB) TTM (TAB)   (TAB) ; TERMINAL LISTING MODE
(TAB)      .NLIST (TAB) BEX (TAB)  (TAB) ; SUPPRESS BIN EXTENSION
(TAB)      .MCALL (TAB) EXIT$S (TAB) (TAB) ; EXEC'S EXIT MACRO
(TAB) (RET)
;
; LOCAL SYMBOL DEFINITIONS:
;
(TAB) (RET)
;
; LOCAL MACROS:
;
(TAB) (RET)
;
; LOCAL DATA BLOCKS:
;
(TAB)      .PSECT<TAB>DATA,D,RW
(TAB) (RET)
;
; FUNCTION DETAILS:
(TAB) (RET)
;
; (TAB)   INPUTS:
;
; (TAB)   OUTPUTS:
;
; (TAB)   SIDE EFFECTS:
;
(TAB) (RET)
(TAB)      .PAGE
(TAB)      .SBTTL
(TAB)      .PSECT
; START CODE HERE
START:
(TAB) (RET)
END: (TAB) EXIT$S  (TAB)   (TAB)       ; EXIT CLEANLY TO EXEC
(TAB)      .END (TAB)  (TAB)    (TAB)  ; TELL ASSEMBLER END OF CODE
(RET)
*EXIT
[EXIT]

>
```

Figure 2-4 Creating the Skeleton File SKEL.MAC

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:

```
>EDI SKEL.MAC
[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 with that name and containing all of the text in your skeleton.

```
*EXIT FILE.MAC
[EXIT]

>
```

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 learned the editing commands, you may want to 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 will 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.

```
>EDI FILE.MAC
[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
*
```

Typing the RETURN key twice advances the pointer two times and displays each line. Typing the ESCAPE key moves the pointer back to the previous line and displays the line.

TYPE n (TY n)

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

```
*TYPE 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).

LIST (LI)

The LIST command 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).

```
*LIST
     (all lines are listed)
*TYPE

[*BOB*]
*EXIT
[EXIT]

>
```

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. This section describes several of the commands most commonly used in editing files.

BEGIN (B), END (E)

The BEGIN and END commands 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:

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

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

```
*BEGIN
*TYPE

[*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.

```
*(RET)
(TAB) .TITLE (TAB) SKELTN (TAB) SOURCE FILE SKELETON
*
```

Typing the RETURN key advances the pointer and displays the line.

LOCATE (L)

If the text you want to examine is within the buffer, you can type the LOCATE command with a string to be located.

```
*LOCATE MODULE
; MODULE FUNCTION:
*
```

A space should separate the command and the search string to be located. EDI displays the line on which 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.

```
*LOCATE NODULE
[*EOB*]
*
```

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

PLOCATE (PL)

If the string for which you are searching is not in the buffer, you can use the PLOCATE command to tell EDI to search successive buffers until it locates the string.

```
*BEGIN
*PLOCATE .END
    .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 on which the string occurs. If EDI does not find the string, it prints a message indicating that the end of file has been reached.

```
*PLOCATE .ENDR
[*EOF*]
*
```

At the end of file (signaled by [*EOF*]), EDI leaves an empty buffer in which 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 can not go back to it except by starting at the beginning of the file again.

```
*EXIT
[EXIT]

>
```

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.

RENEW (REN)

The RENEW command lets you read new lines from the input file.

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

>
```

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

CHANGE (C)

The CHANGE command alters text on the current line, allowing you to:

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.[1] A convenient character to use as a delimiter is the slash character (/) as shown in the following example.

```
>EDI FILE.MAC
[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 allows you to include an ellipsis (...) in the old string. For example:

```
*C /SO...ON/COUNT NUMBER OF A'S/
    .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 allow 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.

---

[1] 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 Chapter 3 of the RSX-11 Utilities Manual for using the Concatenation Character (CC) command.

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.

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.

AP

A special 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 as follows.

```
*(RET)
                    .IDENT /01/
*AP (TAB) ; IDENTIFY MODULE VERSION
                    .IDENT /01/ (TAB)  ; IDENTIFY MODULE VERSION
*
```

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

DP n

To remove a line or lines from the text in the buffer, specify the DP n command, where n is the number of lines to be deleted. The TYPE n command can be used with the DP n command to display the lines to be deleted.

```
*TYPE 3
;
;
;AUTHOR:Z
*DP 2
;AUTHOR:Z
*
```

The TYPE 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.

EXIT (EX)

After changing text in the file, close the editing session as follows.

```
*EXIT
[EXIT]

>
```

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

INSERT (I)

The INSERT, or I, command allows you 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 I command places new lines in the buffer after the current line. For example:

```
>EDI FILE.MAC
[00052 LINES READ IN]
[PAGE 1]
*L FUNCTION:
; MODULE FUNCTION:
*I (RET)
;       THIS MODULE LOADS A 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. (RET)
(RET)
*
```

The L command (for LOCATE) positions EDI to the line preceding where you want to place the new lines. Typing the I command followed by 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 Figure 2-5.

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

;
; CHANGES:
;

;
; MODULE FUNCTION:
;       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

;
; 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
;
```

Figure 2-5 Source Code for FILE.MAC

```
; FUNCTION DETAILS:

;       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, BET 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
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
```

Figure 2-5 (Cont.) Source Code for FILE.MAC

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 Figures 2-6 and 2-7. These two files and the file FILE.MAC will be used in Chapter 4 to build and test a task. You may want to edit the skeleton file before you create the two new source files.

2.4 GUIDE TO FURTHER READING

The sections or chapters in the following documents contain additional information on the subjects described in this chapter.

Document	Location
IAS/RSX-11 MACRO-11 Reference Manual	
	Chapter 2, Source Program Format
	Appendix E, Sample Coding Standard
	Section 6.1, Listing Control Directives
	Section 6.6, Terminating Directives
	Section 6.8, Program Sectioning Directives
	Section 7.8, MACRO Library Directive
RSX-11M/M-PLUS Task Builder Manual	
	Section 2.1, Linking Object Modules
	Section 6.1.26, SQ (Sequential)
RSX-11 Utilities Manual	
	Chapter 3, Line Text Editor (EDI)
RSX-11M/M-PLUS Executive Reference Manual	
	Section 1.4.1, Macro Name Conventions
	Section 4.3.20, Task Exit (EXIT$S)
	Section 4.3.33, Queue I/O Request and Wait

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

;
; AUTHOR: DEF   9-APR-79
;

;
; CHANGES: NONE
;

;
; MODULE FUNCTION:
;
;       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
;

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

IOST:   .BLKW   2               ; DEF IO STATUS WDS
;
```

Figure 2-6 Source Code for FILEA.MAC

```
; FUNCTION DETAILS:

;       INPUTS:
;
;               R0 = ADDR OF BUFFER TO WRITE
;               R1 = LENGTH IN BYTES OF BUFFER
;       OUTPUTS:
;
;               SUCCESS IN IOST
;       SIDE EFFECTS: IOT IF ERROR
;

        .PAGE
        .SBTTL  START OF CODE
        .PSECT
; START CODE HERE
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
```

Figure 2-6 (Cont.) Source Code for FILEB.MAC

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

;
; AUTHOR: DEF 9-APR-79
;

;
; CHANGES: NONE
;

;
; MODULE FUNCTION:
;
;       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
;
```

Figure 2-7 Source Code for FILEB.MAC

```
; FUNCTION DETAILS:

;       INPUTS:
;
;               R0 = ADDR OF BUFFER TO WRITE
;               R1 = LENGTH IN BYTES OF BUFFER
;       OUTPUTS:
;
;               SUCCESS IN IOST
;       SIDE EFFECTS: IOT IF ERROR
;


        .PAGE
        .SBTTL  START OF CODE
        .PSECT
; START CODE HERE
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
```

Figure 2-7 (Cont.) Source Code for FILEB.MAC

CHAPTER 3

ASSEMBLING AND CORRECTING A PROGRAM MODULE

This chapter describes a few uses of the MACRO-11 assembler, some of the common types of coding errors, some ways to uncover and correct errors, and the way to generate a cross-reference listing.

The material in this chapter assumes that you have created the three source files as 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 listing file. To perform a diagnostic run, type the following command.

```
>MAC /DS:GBL=FILE

     (any error lines appear)

>
```

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 UFD. The assembler applies the type .MAC as a default. Because there are no file specifications 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, 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 (/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 Appendix D of the IAS/RSX-11 MACRO-11 Reference Manual.

The only errors that should appear from the diagnostic run are the following:

```
U    67 000010  004767          CALL    READ     ; READ FROM TTY
U    95 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 TKB.

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 IAS/RSX-11 MACRO-11 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.

- Not properly delimiting a directive argument

  Many MACRO-11 directives require a character or argument string to begin with 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 a general assembly error 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 may 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 command produces both files.

```
>MAC FILE,FILE/-SP=FILE
     (error summary printed)
>
```

This command, like the command 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 is a separating character that is required to distinguish different file specifications in command lines.

Following the comma in the command 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 designation /-SP following the listing file specification in the command 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 there are errors, 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 commands to assemble the two other source files, FILEA.MAC and FILEB.MAC, which you created using the procedures described in Chapter 2.

```
>MAC FILEA,FILEA/-SP=FILEA
>MAC FILEB,FILEB/-SP=FILEB
>
```

These two commands create the object modules FILEA.OBJ and FILEB.OBJ that you will need to link into your task in Chapter 4.

3.4 EXAMINING A LISTING AT THE TERMINAL

You can run the Peripheral Interchange Program (PIP) to transfer a copy of your listing from disk to the terminal. The following command starts the transfer.

```
>PIP TI:=FILE.LST

     (file appears on screen)

>
```

In the command 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 UFD 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 temporarily stop and restart the display and to alternately suppress and resume the output request. The commands are summarized in Table 3-1.

Table 3-1
Terminal Output Control Commands

Command	Effect
CTRL/S	Temporarily stops the display
CTRL/Q	Restarts the display stopped by CTRL/S
CTRL/O	Alternately suppresses and resumes the output to the terminal

The CTRL/S and CTRL/Q commands are used 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 for suppressing 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 CTRL/O 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 CTRL/O 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. You generate the cross-reference listing by typing the following.

```
>MAC ,FILE/CR/-SP=FILE

     (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 designation tells the assembler to generate a request for the 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 designations /CR and /-SP are illegal.)

NOTE

If, after you request a cross-reference listing, you discover that the information is missing from your listing, the CRF task either is not installed on your system or 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 file by typing one of the following commands.

```
>PIP FILE.LST/SP
>
```

or

```
>PRINT FILE.LST
>
```

These commands create a request to the spooling task to print the file you specify. (You can request more than one file at a time by including the file specifications in the command and separating each specification with a comma.) Your request is placed in a queue of requests that is processed by a separate task.

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

```
>PIP LP:=FILE.LST
>
```

If the printer is not busy or is not allocated by another user, PIP outputs the file to printer unit 0.

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.

You can list the name, types, version numbers, and sizes of the files stored in your UFD by typing the following command.

```
>PIP /LI

     (the directory listing appears)

>
```

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

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

```
>PIP *.MAC,*.LST,*.OBJ/PU
>
```

The designation /PU purges all but the latest version of the files specified. The asterisk character in the command denotes all files having any name and the type specified.

3.8 GUIDE TO FURTHER READING

The sections or chapters in the following documents contain additional information on the subjects described in this chapter.

Document	Location
IAS/RSX-11 MACRO-11 Reference Manual	Chapter 8, Operating Procedures Section 8.1.3, RSX-11 File Spec Switches Section 8.4, MACRO-11 Error Messages Appendix D, Diagnostic Error Message Summary
RSX-11 Utilities Manual	Section 4.2.2, Performing File Control Functions Chapter 6, Print and Queue Utility Appendix D, Cross Reference Processor (CRF)
RSX-11M-PLUS Batch and Queue Operations Manual	Chapter 3, Queuing Jobs

CHAPTER 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

To create a task image file from a single module, supply the file name of the object module as in the following command.

```
>TKB FILE=FILE

     (any error messages appear)

>
```

The right side of the equal sign specifies the file containing 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 there are undefined references after the module has been processed, TKB searches the system object library SYSLIB.OLB in UFD [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 at the terminal in one of the following forms.

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

or

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

TKB error messages are summarized in an appendix of the RSX-11M/M-PLUS Task Builder Manual.

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 may 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, 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 Section 4.1.2.

4.1.2 Supplying Multiple Object Modules

TKB accepts multiple object modules as input. On the right side of the equal sign, type the names of the object files separated by commas, as in the following example.

```
>TKB FILE=FILE,FILEA,FILEB

     (any error messages appear)

>
```

TKB performs the same actions as described in Section 4.1.1 for one object module. Only one of the object modules specified must have been assembled with a .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 also processes a concatenated object module, which is merely a file containing multiple modules. To create a concatenated file, use PIP as follows:

```
>PIP FILCON.OBJ=FILE.OBJ,FILEA,FILEB/ME
>
```

The right side of the command specifies the files to be concatenated. You need specify the file type (.OBJ) only on the first file because PIP applies it as the default file type for subsequent names. The designation /ME 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, PIP uses /ME as a default condition. The command string includes /ME merely to emphasize the concatenate, or merge, operation.)

The single concatenated object file can then be the sole input to TKB as in the following command.

```
>TKB FILE=FILCON

     (any error messages appear)

>
```

This operation saves file processing overhead for the TKB program and is possibly 40 percent 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 should use the Fast Task Builder (FTB). Its interface is the same as that of TKB. For example:

```
>FTB FILE,FILE/-SP=FILE,FILEA,FILEB
>
```

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 has.

4.2 TASK BUILDER DEFAULTS

When you build a task image, TKB applies certain default conditions to your program including the partition in which 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. You can consult the RSX-11M/M-PLUS Task Builder Manual for the procedures to override the default conditions.

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.

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.

The Task Builder assumes that the task is not checkpointable and does not use the floating-point processor. 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 through 4 are assigned to the system device (SY:), unit 5 is the task initiating terminal (TI:), and unit 6 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 is 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 global cross-reference listing for debugging a task. To create a map with a global cross-reference listing, type the following command.

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

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 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.

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

TKB creates both files.

The designation /CR tells TKB to generate a request for the CRF task to produce a global cross-reference listing. The designation /-WI 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 either is not installed on the system or is still processing the request. Consult the system manager to have the CRF task installed.

---

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

4.3.2 Examining the Map at the Terminal

The same commands described in Section 3.4 can be used to examine a map at the terminal. The following command shows the procedure.

```
>PIP TI:=FILE.MAP

     (file appears on screen)

>
```

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

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 information needed for debugging tasks. To generate a full form of the map, specify the command to TKB as follows.

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

The designation /-SH indicates that you do not want the short form of the standard map. TKB therefore includes the file contents information in the map. The designation /MA tells TKB to include system library contributions to the task in the file contents section of the map. (System symbols also are included in the global cross-reference listing.)

4.4 RUNNING THE TASK AND CORRECTING TYPICAL ERRORS

You execute your task by using the RUN command and the name of the task image file.[1] For example:

```
>RUN FILE
```

Because the task FILE is not installed on the system, the Run processor searches your UFD on device SY: for a file named FILE.TSK. Run installs it temporarily and runs it immediately. (The task will be automatically removed on exit.)

To run 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.

---

[1] The RUN command has many formats for scheduling and rescheduling tasks. The format shown in the example is the most widely employed.

Aborting a task forces an orderly termination of the task. Included in the termination is a request for the Task Termination and Notification task (TKTN) to display a message on your terminal. The display includes the cause of the abort and a list of the task registers and Processor Status word (PS). For example:

```
TASK  "TT30  " TERMINATED
ODD ADDRESS OR OTHER TRAP FOUR
R0=000000
R1=100101
R2=135600
R3=000000
R4=000000
R5=000000
SP=001172
PC=000003
PS=170017
>
```

The information can help you ascertain the cause of the abort.[1] If the cause of the error is hardware-related, report the occurrence to your system manager who can consult the error logging data to ascertain the origin of the problem. 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 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.) Therefore, you should ensure that the source file defines a starting address and that the address is even (on a word boundary).

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

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

>MAC FILE,FILE/-SP=FILE
>TKB FILE,FILE/-SP=FILE,FILEA,FILEB
```

[1] The format of the information varies between RSX-11M and RSX-11M-PLUS (that is, one system may have the time of the abort and another system may report the processor on which the abort occurred). However, the basic data displayed is the same.

```
>RUN FILE
ABCABCABAB
THE NUMBER OF A'S IS 0004
>
```

After you correct the error 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 either to illegally 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 you omit the number sign from a parameter that makes the reference relative.

4.5 GUIDE TO FURTHER READING

The sections or chapters in the following documents contain additional information on the subjects described in this chapter.

Document	Location
RSX-11M/M-PLUS Task Builder Manual	Chapter 1, Commands Section 6.1, Switches Section 6.2, Options Appendix F, Error Messages Appendix E, The Fast Task Builder (FTB)
RSX-11 Utilities Manual	Section 4.2.2, Performing File Control Functions Appendix D, Cross Reference Processor (CRF)

CHAPTER 5

USING DEBUGGING AIDS

This chapter introduces a few debugging aids that are helpful in the program development process.

5.1 THE ON-LINE DEBUGGING TOOL

The On-Line Debugging Tool (ODT) is special code that you include in your task image to assist you during debugging. ODT gives you interactive control of task execution, and allows you to set breakpoints and examine and change data and instructions within the memory-resident task. The ODT module is linked into your task image, thereby increasing 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.

ODT commands differ from commands in other utility programs. Most programs have multiple-character 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 especially alert when you are debugging your task with ODT.

5.1.1 Including ODT in a Task

To include ODT in a task, type a command similar to the following one.

```
>TKB BUG/DA,BUG/CR/-SP=FILE,FILEA,FILEB
>
```

The designation /DA accompanying the task image file specification tells TKB to include ODT. The task builder accesses the file ODT.OBJ in UFD [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 command shows the sequence.

```
>RUN BUG
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 (octal).)

The underline character (_) indicates that ODT is ready to accept commands.

To access locations within the task, you should establish one or more relocation registers. This set of eight registers, numbered 0 through 7, allows 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 printout, 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 Figure 5-1.

```
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 005572 02938.
                            001730 005572 02938. ODTRSX M05.02 ODT.OBJ;36
```

Figure 5-1 Memory Allocation Synopsis from Task BUG Map

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 (octal) and the extent is 340 (octal), or 224 (decimal), 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 and extends for 110 (octal), or 72 (decimal), 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 through 5. (The addresses are octal; ODT accepts only octal numbers.) As soon as you type the R in the command line, ODT generates line feed and carriage return operations 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 either the CTRL/U combination; the number 8 or 9; or 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.

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

ODT stores the most recently entered value in the register.

To access a location within a task, you must create an address by combining the relocation register and a location counter value for the relevant program section shown in the assembly listing.

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

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. In the example, you placed in relocation register 0 the starting address of the NUMA contribution to the blank program section. Location counter value 20 in the assembly listing for NUMA is 20 bytes from the start of the address in relocation register 0. You combine the two values to form the address of the location. The combination is formed by typing the number of the relocation register, a comma (,), and the offset value (in octal). 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 (octal). You use this syntax with various ODT commands to access locations within the task address space.

```
NUMA    COUNT NUMBER OF A'S      MACRO M1113  10-APR-79 10:18  PAGE 3
ROUTINE TO COUNT A'S

     62                          .SBTTL  ROUTINE TO COUNT A'S
     63 000000                   .PSECT
     64 000000              START:
     65 000000  012700           MOV     #BUF1,R0        ; LOAD BUFFER ADDR
     66 000004  012701           MOV     #SIZ,R1         ; LOAD BUFFER SIZE
     67 000010  004767           CALL    READ            ; READ FROM TTY
     68 000014  005702           TST     R2              ; ANY CHARS IN BUFFER?
     69 000016  001436           BEQ     END             ; IF NONE, FINISH UP
     70 000020  005001           CLR     R1              ; INIT # OF A'S COUNTER
     71 000022  010267           MOV     R2,NUMC         ; SAVE # OF CHARS TYPED
     72 000026              10$:
     73 000026  122067           CMPB    (R0)+,A         ; IS CHAR = A?
     74 000032  001001           BNE     20$             ; IF NO, GET NEXT CHAR
     75 000034  005201           INC     R1              ; COUNT AN A
     76 000036              20$:
     77 000036  005302           DEC     R2              ; ONE LESS CHAR
     78 000040  001372           BNE     10$             ; IF MORE, COMPARE NEXT
```

Figure 5-2 Portion of Assembly Listing for NUMA

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. The contents ODT displays should agree with the value shown in the assembly listing.

To close the currently open location, type either the RETURN key or the LINE FEED key. 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, typing 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 typing the RETURN or LINE FEED key. See Section 5.1.5.

To examine bytes 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 typing the LINE FEED key, after which ODT closes the currently open byte location, opens the next sequential byte location, and displays its contents.

```
_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 comparing its contents with what shows 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 what appears in the assembly listing. If the values do not match, either you have an incorrect listing or the relocation register value is wrong.

5.1.4 Setting Breakpoints within the Task

To allow you to stop (or break) task execution, ODT provides eight registers called breakpoint registers. These registers, numbered 0 through 7, let you specify locations of instructions at which execution should stop. The registers are denoted by a number and the B command.

To establish breakpoints in the task, specify the location of the instruction and the B command as in the following example.

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

The command places the designated addresses in breakpoint registers 0 and 1. (Changing a breakpoint register is the same as changing a relocation register: simply retype the command and give the altered contents.)

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, ODT generates the carriage return and line feed operations and prints a prompt.

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

```
_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 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 giving:

- 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).

5.1.5 Examining and Changing Locations with ODT

When execution stops at a breakpoint, you can examine and change data within the task image address space. (You can also do these operations before you start execution. Instructions, as well as data, can be altered.) When a task stops at a breakpoint location, its general registers are stored in ODT locations accessed by the dollar sign ($) character. The following sequence shows a way to display general registers 0, 1, and 2.

```
_$0/ 001543 (LF)
$1 /000120 (LF)
$2 /135600 (RET)
_
```

The dollar sign followed by a number refers to a particular task general register. The slash (/) character opens the general register as a word location and prints its contents. Typing the LINE FEED key closes the current location and opens the next sequential location.

---

[1] Eight breakpoint instruction registers, referred to by the letter I, contain the actual instructions during task execution.

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 (octal) 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 (octal) is written to the open location as a byte value. Typing the RETURN key closes the location. The next commands examine offset 0 to verify that it indeed contains 102 (octal) and change 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 at which the break occurred. The task executes the READ routine which prompts for input at the terminal.

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.

You can type a line of input, after which execution stops at the location contained in breakpoint register 1.

The G command is used to transfer control to another address and continue execution. For example:

```
_1,76G
```

ODT transfers control to offset 76 and continues execution there. This command purposely tranfers control to the error routine to show what occurs when an error is encountered. See Section 5.1.6.

5.1.6 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 (similar in format to the breakpoint halt message) gives a code describing 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 dollar sign ($) followed by the register number to access the task registers as described in Section 5.1.5. To examine the stack, examine register 6 (the Stack Pointer) and use the at sign (@) to open the location pointed at by R6. For example:

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

The slash (/) character opens R6 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, R6) 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, type 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 type 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 simply performs the exit task directive and returns control to the Executive.

5.2 POSTMORTEM DUMP

Another debugging aid is the Postmortem Dump (PMD). It requires no special code in your program. You simply request TKB to enable PMD for your task as follows.

```
>TKB FILE/PM,FILE/-SP=FILE,FILEA,FILEB
>
```

The designation /PM in the command after the task image file name tells TKB to set a bit in the task flag word.[1] (You can tell whether a task includes PMD by inspecting the task attributes section of the map. A line item called TASK ATTRIBUTES will have the designation PM.)

When PMD is in effect for a task, the occurrence of an error that generates a synchronous system trap (SST) causes the Executive to handle the termination of your task in a special manner.[2] Instead of simply aborting the task, the Executive generates a request for PMD to create a formatted disk file showing the task image context. When a task generates a synchronous system trap, the Executive initiates the normal task termination procedure (the printing of an error message and general register contents at the terminal) and additionally generates the request for PMD. To inform you that a dump is in effect, the Executive causes the following message to appear at the terminal.

```
POST MORTEM DUMP WILL BE GENERATED
```

PMD receives the request, creates a file in UFD [1,4] on the library device, and generates a request to the spooler to print the file. The file has the name of the task and a type of .PMD. The print spooler automatically deletes a file with the type .PMD after it is printed.

5.3 THE SNAPSHOT DUMP

The snapshot dump capability is a subset of the Postmortem Dump but requires special code in the task. Whereas PMD generates a dump of an entire task, the snapshot dump can produce a dump of only a portion of the task. Also, PMD generates a dump only when the task terminates abnormally, but the snapshot code can produce a dump at any place in the task execution.

You include the necessary snapshot code in the task by editing the source file and inserting the snapshot macro calls where you want to produce a dump.[3] After you reassemble the modules containing the snapshot calls, rebuild the task and substitute the reassembled modules. When you use snapshot macro calls, you do not need any special switches or options for TKB.

When you run the task and that section containing the special code is executed, a snapshot dump is taken. The special code generates a request for the PMD task. (No special messages are printed at the terminal.) To hold the dump, PMD creates a file with the name of the task and a type of .PMD in the UFD the same as the UIC under which the task is running. PMD then generates a request for the spooling task to print and delete the file.

---

[1] The keyword and option PMD=YES on the RUN command and the keyword PMD on the ABORT command also allow you to enable Postmortem Dumps for your task. See the RSX-11M/M-PLUS MCR Operations Manual.

[2] This discussion assumes that the task does not handle synchronous system traps through the SVTK$ directive and specially coded routines.

[3] The snapshot macro calls the PMD task as described in Chapter 8 of the RSX-11M/M-PLUS Task Builder Manual.

5.4 GUIDE TO FURTHER READING

The sections or chapters in the following documents contain additional information on the subjects described in this chapter.

Document	Location
RSX-11M/M-PLUS Task Builder Manual	Chapter 8, Memory Dumps Section 8.1, Postmortem Dumps Section 8.2, Snapshot Dumps
IAS/RSX-11 ODT Reference Manual	Section 4.3, Linking and Initiating ODT Section 3.11, Relocation Register Commands Section 3.12, Relocation Calculation Commands Section 3.5, Task Breakpoint Commands Section 3.2, Commands for Opening, Changing and Closing Locations Section 3.14, Reprinting Open Locations Section 3.6, Program Execution Commands Section 4.5, Returning Control to the Host System

CHAPTER 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 program (LBR) creates a library file which can contain macro definitions. Such a file has a default type .MLB (macro library) and contains only macro definitions.

6.1.1 Creating the Macro Library

To create a user library of macro definitions, you must have a file or files which have the macro source definitions. The Librarian program can accept as input either one file containing multiple definitions or multiple files, each of which has one or more definitions. Figure 6-1 shows one file with two macro definitions.

The following command creates a macro library file from one input file of source definitions.

```
>LBR USRMAC/CR:25.::128.:MAC=USRMAC
>
```

The designation /CR 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 the example, the input file is USRMAC.MAC.

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

Figure 6-1 MACRO-11 Library Source Definitions

Following the designation /CR in the command are parameters, separated by colons, that LBR uses to create the library.[1] The first parameter, 25 (decimal), gives the length in blocks for the library file. If you omit this parameter, LBR uses 100 (decimal) 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 will expand a library file as needed if you add modules which will cause the file to exceed its original size. However, the library will no longer be contiguous.) The second parameter is blank because it applies only to object libraries. The third parameter, 128 (decimal), 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 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 occurring 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 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 the macro definitions in the library file, use a command similar to the following.

```
>MAC USRTST,USRTST/-SP=USRMAC/ML,USRTST
>
```

To the right of the equal sign in the command, specify the name of the macro library and the designation /ML. The comma separates the macro library file name and the source file name. The designation /ML 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.[1] 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.

---

[1] 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 TKB.

6.2 CREATING AND USING AN OBJECT MODULE LIBRARY

LBR may be used 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.

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 creates the object library and inserts the modules FILEA.OBJ and FILEB.OBJ.

```
>LBR USROBJ/CR:25.:128.:64.=FILEA,FILEB
>
```

The designation /CR tells LBR to create a library file. LBR uses the name preceding /CR as the name of the library and applies the default file type .OLB. Following /CR in the command are parameters, separated by colons, used in creating the file.[1]

The first parameter, 25 (decimal), gives the size in blocks at which to create the library file. If you omit the parameter, LBR supplies 100 (decimal) 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 which will cause the file to exceed its original size. However, the library will no longer be contiguous.)

The second parameter, 128 (decimal), in the command gives the number of entry point table slots to reserve.[2] (An entry point is any global symbol in a module by which your program refers to the associated module.) 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 supplies 512 (decimal) as the default number. If the value you supply is not an integral multiple of 64 (decimal), LBR raises the number to the next highest multiple of 64 (decimal).

The third parameter, 64 (decimal), 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, LBR supplies 256 (decimal) as the default number. If the value you specify is not an integral multiple of 64 (decimal), LBR raises the number to the next highest multiple of 64 (decimal).

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

---

[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.

[2] 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.

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 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.[1] 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 6.3.3.

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.)[2]

2. Specify the name of the library file and the names of the called modules in the command to TKB.

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

When you build the task, use a command similar to the following.

```
>TKB SUBLIB,SUBLIB/-SP=FILE,USROBJ/LB:TTREAD:TTWRIT
>
```

The designation /LB after a name in the command indicates to TKB that the file is an object library. TKB accesses the file USROBJ.OLB in the UFD that is the same as the current UIC. The names appearing

---

[1] 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.

[2] CALL is a macro statement which is a permanent symbol in the MACRO-11 assembler. It standardizes subroutine calling conventions. CALL X translates to JSR PC,x (where x is the subroutine entry point).

after /LB in the command 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 are preceded by colons. The colons 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 UFD 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 /LB 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 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. The following example shows the usage of /LB with no module names to request the search.

```
>TKB LB,LB/-SP=FILE,USROBJ/LB
>
```

The designation /LB 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, it 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 to 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 may want TKB to definitely include specific modules from the library and also to search the same library to resolve any undefined references that may occur. For example, you may have conditional code in the main part of a task and do not know what global symbols are referenced. TKB allows you to specify the two forms of the library search as in the following command.

```
>TKB LBOPT,LBOPT/-SP=FILE,USROBJ/LB:TTREAD,USROBJ/LB
>
```

The first appearance of the /LB designation 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 to maintain a user library - adding modules to, replacing a module in, and obtaining information about the library.

6.3.1 Adding Modules to a Library

Modules can be added to a library with an LBR command such as the following.

```
>LBR USRMAC.MLB/IN=MAC1,MAC2
>
```

To add modules to a library, specify the name and type of the library file and the /IN designation (insert) to the left of the equal sign in the LBR command. 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 designation to /RP (replace). See Section 6.3.2.

6.3.2 Replacing a Module in a Library

After you create a library, a typical maintenance function you will perform 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. For example:

```
>LBR USROBJ/RP=FILEA
MODULE "TTREAD" REPLACED

>
```

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, type a command to LBR similar to the following.

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

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

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

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

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

6.4 GUIDE TO FURTHER READING

The sections or chapters in the following documents contain additional information on the subjects described in this chapter.

Document	Location
RSX-11M/M-PLUS Task Builder Manual	Section 6.1.13, LB (Library File)
IAS/RSX-11 MACRO-11 Reference Manual	Section 8.1.3, RSX-11 File Specification Switches Section 7.8, Macro Library Directive: .MCALL
RSX-11 Utilities Manual	Chapter 14, Librarian Utility Program (LBR)

CHAPTER 7

FORTRAN IV PROCEDURES

PDP-11 FORTRAN IV is one of several high-level languages optionally available on RSX-11M and RSX-11M-PLUS systems. 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 RSX-11M and RSX-11M-PLUS consists of the following elements:

- Compiler task FOR
- Object Time System library
- An optional shareable library

The FORTRAN IV compiler accepts an ASCII disk file containing source statements. It can generate a disk file in object module format and 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 TKB to obtain an executable program.

The FORTRAN IV Object Time System (OTS) is a collection of object module subroutines required to create 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 Object Time System contains the routines that the compiler designates to be linked into your task. In practice, however, the OTS can be an ordinary library file containing various routines in addition to the routines required by the compiler-assigned references. In a wider sense, the OTS can contain user-callable

routines as well as routines for which the compiler generates 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 shareable 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 the assembler. Therefore, this chapter does not present the detail found in Chapters 2 through 6. For example, to edit a FORTRAN IV source file, you use the same commands as you used to edit an assembly language source file as described in Chapter 2.

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 Figure 7-1.

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

>
```

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), use the C (comment line) in column 1 for readability in the source file in place of the blank line. If you insert a line with a space or tab character on it, the FORTRAN IV compiler generates an error because it expects a valid label on a nonblank line.

To format the source statements and avoid counting spaces, you can use the TAB character. The FORTRAN IV compiler will position the character following an initial TAB character to the proper column. That is, a digit following an initial TAB will be considered a continuation character (column 6) and a nondigit will be considered the beginning of the statement (column 7).

```
      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 40      ! 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
```

Figure 7-1 FORTRAN IV Sample Source Code AVERAGE.FTN

7.2.2 **Performing a Diagnostic Run**

To see whether there are any syntax or grammar errors in a source file, you can perform a diagnostic run. For example:

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

This command requests FORTRAN IV to compile the file AVERAGE.FTN, which resides in your UFD. 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 program 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 the errors, you must examine the section of the listing entitled FORTRAN IV DIAGNOSTICS. Display the listing file by typing the following command.

```
>PIP TI:=AVERAGE.LST

     (PIP displays listing)

>
```

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

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

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

Line 8 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 8 is shown as follows.

```
0008     20     FORMAT I10
```

The compiler detected the missing parentheses on the field descriptor in the FORMAT statement. You must edit the source file, as in the following example.

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

>
```

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 add the file name to the command string you used to perform the diagnostic run.

```
>FOR AVERAGE,AVERAGE/-SP=AVERAGE
AVERAG
>
```

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

7.2.4 Creating a Task Image

The object module created by the FORTRAN IV compiler does not contain all the code required at run time. Therefore, when you run TKB, you must specify as input both the name of the object module and the name of the library containing the FORTRAN IV Object Time System routines.[1] The following command shows the procedure.

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

This command requests TKB to link the module AVERAGE.OBJ and resolve any undefined references by searching the library FOROTS.OLB in UFD [1,1] on the system library device.[1] You can add, as input to TKB, file names of any external object modules which the main module calls. As a result of the command, 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 an appendix in the RSX-11M/M-PLUS Task Builder Manual for guidelines on error processing.

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 may have to specify special switches in the command or supply options to TKB. Refer to the language user's guide for information regarding Task Builder default FORTRAN IV conditions and FORTRAN-specific options and switches.

7.2.5 Running and Debugging a Task

To execute the task AVERAGE, type the following command.

```
>RUN AVERAGE
ENTER NUMBER, END WITH 0
66
ENTER NUMBER, END WITH 0
66
ENTER NUMBER, END WITH 0
0
AVERAGE IS     44.00
TT30  --  STOP
>
```

The program is not computing the average correctly. If you cannot locate the error by looking at the program listing, you can place debugging statements in the code and assemble the module with them.

---

[1] In the command, 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 may 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 to TKB because TKB automatically searches the system library.)

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.

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

>
```

The L commands locate and print the contents of the lines that precede where the debugging statements are to be placed. The I commands insert the debugging statements. The insert operation is terminated by typing two successive RETURN keys. After the inserts are made, the EXIT command closes the files and terminates EDI.

Next, recompile the module and request FORTRAN IV to include the debugging statements as shown in the following command.

```
>FOR DEBUG,DEBUG/-SP=AVERAGE/DE
AVERAG
>
```

The compiler generates the files DEBUG.OBJ and DEBUG.LST. Because of the designation /DE in the command, the compiler includes statements beginning with D in column 1. If you omit /DE, the debugging lines are treated as comment lines.

Next, build and run the task with the debugging lines as follows.

```
>TKB DEBUG=DEBUG,LB:[1,1]FOROTS/LB
>RUN DEBUG
***DEBUG LINE N =       1, TOTAL =      0.
ENTER NUMBER, END WITH 0
66
***DEBUG LINE N =       2, TOTAL =      66.
ENTER NUMBER, END WITH 0
66
***DEBUG LINE N =       3.  TOTAL =      132.
ENTER NUMBER, END WITH 0
0
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 again as follows.

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

Next, repeat the compilation, linking, and running as follows.

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

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

7.3 GUIDE TO FURTHER READING

The section or chapters in the following documents contain additional information on the subjects described in this chapter.

Document	Location
IAS/RSX-11 FORTRAN IV User's Guide	
	Section 2.1, FORTRAN IV Object Time System
	Section 2.2, Object Code
	Section 2.6, OTS and Shareable Libraries
	Section 1.3, Using the Task Builder to Link FORTRAN IV Programs
	Section 1.5, Operating Procedures
	Section 1.6, Debugging a FORTRAN IV Program
	Appendix C, FORTRAN IV Error Diagnostics
RSX-11M/M-PLUS Task Builder Manual	
	Appendix F, Error Messages

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

READER'S COMMENTS

NOTE: 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 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

Please cut along this line.

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

RT/C SOFTWARE PUBLICATIONS TW/A1
DIGITAL EQUIPMENT CORPORATION
1925 ANDOVER STREET
TEWKSBURY, MASSACHUSETTS 01876

Do Not Tear - Fold Here

Cut Along Dotted Line