Programming with RT-11

VOLUME 1

Program Development Facilities

Simon Clinch

Stephen Peters

The RT-11 Technical User's Series

Programming with RT–11

VOLUME 1

Program Development Facilities

Programming with RT–11

VOLUME 1

Program Development Facilities

Simon Clinch
Stephen Peters

digital
DECbooks

Designed by Virginia J. Mason
Printed in the United States of America

10 9 8 7 6 5 4 3 2

Documentation number EY-00022-DP
ISBN 0-932376-32-0

Library of Congress Cataloging in Publication Data

Clinch, Simon, 1959–
Programming with RT–11.

Includes index.
Contents: v. 1. Program development facilities.
1. RT–11 (Computer operating system) 2. MACRO–11 (Computer program language) 3. FORTRAN (Computer program language) 4. Basic (Computer program language) I. Peters, Stephen, 1951– . II. Title.
QA76.6.C55 1984 001.64'2 84-5031
ISBN 0-932376-32-0 (v. 1)

Contents

Acknowledgment

We would like to thank all those who contributed to this publication. In particular, we are grateful to Martin Gentry, who reviewed and updated the material in this book and Bernard Volz, who reviewed the material and revised practice exercises. The staff at Digital Press deserve special commendation for their invaluable assistance.

Introduction

Programming with RT–11 examines the RT–11 facilities that enable you to develop executable programs in MACRO–11, FORTRAN IV, or BASIC–11. *Programming with RT–11* comprises two volumes. Volume 1 covers the program development process, RT–11 debugging aids, libraries, overlays, and the FORTRAN IV and BASIC–11 subroutine conventions for MACRO–11 interfacing. Volume 2 discusses the use of programmed requests to perform file and terminal input/output, foreground/background communication, and synchronous and asynchronous input/output operations.

Volume 1 contains chapters 1 through 7. Chapter 1, "Developing Programs in MACRO–11 and FORTRAN IV," describes the program development cycle for MACRO–11 and FORTRAN IV, focusing on the conventions for assembling, compiling, and linking source programs. Chapter 2, "Executing Programs," examines the execution of foreground, background, and system jobs and discusses the EXECUTE command. Chapter 3, "Developing Programs in BASIC," gives an overview of the program development cycle for BASIC–11 and discusses the procedures for invoking the BASIC interpreter; creating, modifying, precompiling, and saving programs; and leaving the interpreter. Chapter 4, "Debugging Programs," explains the use of ODT and VDT in debugging programs and the process of failure analysis. Chapter 5, "Using Libraries," describes the creation, modification, and use of macro and object libraries. Chapter 6, "Designing and Implementing Overlay Structures," discusses the procedures for checking memory use and implementing program overlays. Chapter 7, "Using Language Interfaces," examines the functions of MACRO–11 subroutines in FORTRAN IV and BASIC–11 programs and describes the FORTRAN/MACRO and BASIC/MACRO interfaces. (The introductory chapter of volume 2 describes its thirteen chapters in detail.)

Equipment

In order to do the practice exercises, you will need access to a working RT–11 system with at least 500 blocks of disk space for your files. By a *working* system, we mean that:

- The RT–11 monitor program has been transferred from its storage disk to main memory (in other words, the system has been bootstrapped)
- The FORTRAN IV compiler or BASIC–11 interpreter has been installed and is available for use

Resources

Although every effort has been made to make *Programming with RT–11* self-contained volumes, you may need to refer to the following manuals from the RT–11 documentation set for additional information:

- *RT–11 Installation Guide*
- *RT–11 Programmer's Reference Manual*
- *RT–11 Software Support Manual*
- *RT–11 System Generation Guide*
- *RT–11 System Message Manual*

- *RT–11 System User's Guide*
- *RT–11 System Utilities Guide*

The documentation to which we refer throughout the text is written for RT–11 version 5.0. We also used a computer system equipped with RT–11 version 5.0 to generate the programs in our examples and practices. If you own a newer version of RT–11, you may also need a copy of the latest *System Release Notes* to determine the difference between your system and the one described here.

Programming with RT–11 is written under the assumption that you know how to program in MACRO–11, FORTRAN IV, or BASIC–11. The authors assume that you can manipulate files and get directory listings on an RT–11 system and are familiar with RT–11 conventions for device and file specifications, the operation of the Foreground/Background monitor, and monitor components and their functions. If you need additional information on RT–11 conventions and programming procedures, you may refer to some of the publications listed below:

- *Working with RT–11* (Digital Press, 1983)
- *Tailoring RT–11: System Management and Programming Facilities* (Digital Press, 1984)
- *MACRO–11 Language Reference Manual*
- *FORTRAN IV Language Reference*
- *RT–11/RSTS/E FORTRAN IV User's Guide*
- *BASIC–11 Language Reference Manual*
- *BASIC–11/RT–11 Installation Guide*
- *BASIC–11/RT–11 User's Guide*

For a directory of documentation products, write:
Digital Equipment Corporation
Circulation Department, MK01/W83
Continental Boulevard
Merrimack, NH 03054

Notations

The following symbols are used in the two volumes to represent specific elements:

〈KEY〉	indicates keyboard and keypad keys, their functions, or key combinations
`COMMANDS`	(uppercase) indicates input
`Prompts`	(upper and lowercase) indicates computer output
[ ]	indicates parts of a command that are optional (the brackets are not part of the command string)

Programming with RT–11

VOLUME 1

Program Development Facilities

1

1 Developing Programs in MACRO–11 and FORTRAN IV

RT–11 allows you to program in assembly, compiled, and interpreted languages. In this chapter, you will learn the basic command procedures and concepts needed to develop programs in MACRO–11, the language processed by the RT–11 assembler, and FORTRAN IV, one of the compiled languages you may use with RT–11. Options for the FORTRAN IV compiler are generally different from those for other compilers, but the phases of program development are the same for all compiled languages.

This chapter shows you how to use the commands, MACRO, FORTRAN, and LINK, in program development. When you have completed this chapter, you will be able to produce an executable file, using error-free FORTRAN IV or MACRO–11 source code. You will learn to control LINK, the linking program, by enabling or disabling the options: /BOTTOM, /FOREGROUND, /LDA, /MAP, and /STACK.

In addition, you will learn to control the assembly of MACRO–11 source programs by enabling or disabling the options: /ALLOCATE, /CROSSREFERENCE, /ENABLE, /LIST, /OBJECT, and /SHOW. You will learn to control the compilation of FORTRAN IV source programs by enabling or disabling the options: /CODE, /HEADER, /LIST, /OBJECT, /ONDEBUG, and /SHOW.

Program Development

The process of writing a program in source code, translating it into machine code, and producing an executable file is called program development. The process is made up of the following steps:

1. Create or Edit a Program. You use an editor program to create or modify a file containing your MACRO–11 or FORTRAN IV source code.
2. Assemble or Compile a Program. You use the MACRO–11 assembler or the FORTRAN IV compiler to check that the syntax of the source file is correct and to produce an object module if there are no syntax errors. An object module contains the machine code for your program. You can divide programs across more than one source file, so that they compile into more than one object module. Because object modules contain relocatable code and code that indicates how the modules can be linked together, object modules cannot be executed.
3. Link Object Modules. You link object modules to produce a file that can be loaded into memory and executed. Such a file is called a load module. You may specify whether the load module runs in foreground or in background.
4. Execute and Test the Load Module. You load the load module into memory and execute it. For foreground or system programs, you must provide parameters that control memory allocation.

Figures 1 and 2 show the sequence of steps used in developing MACRO–11 and FORTRAN IV programs.

Assembly and Compilation

To assemble a program written in MACRO–11 you type the command, MACRO. To compile a program written in FOR-

Figure 1.
Program Development in MACRO–11

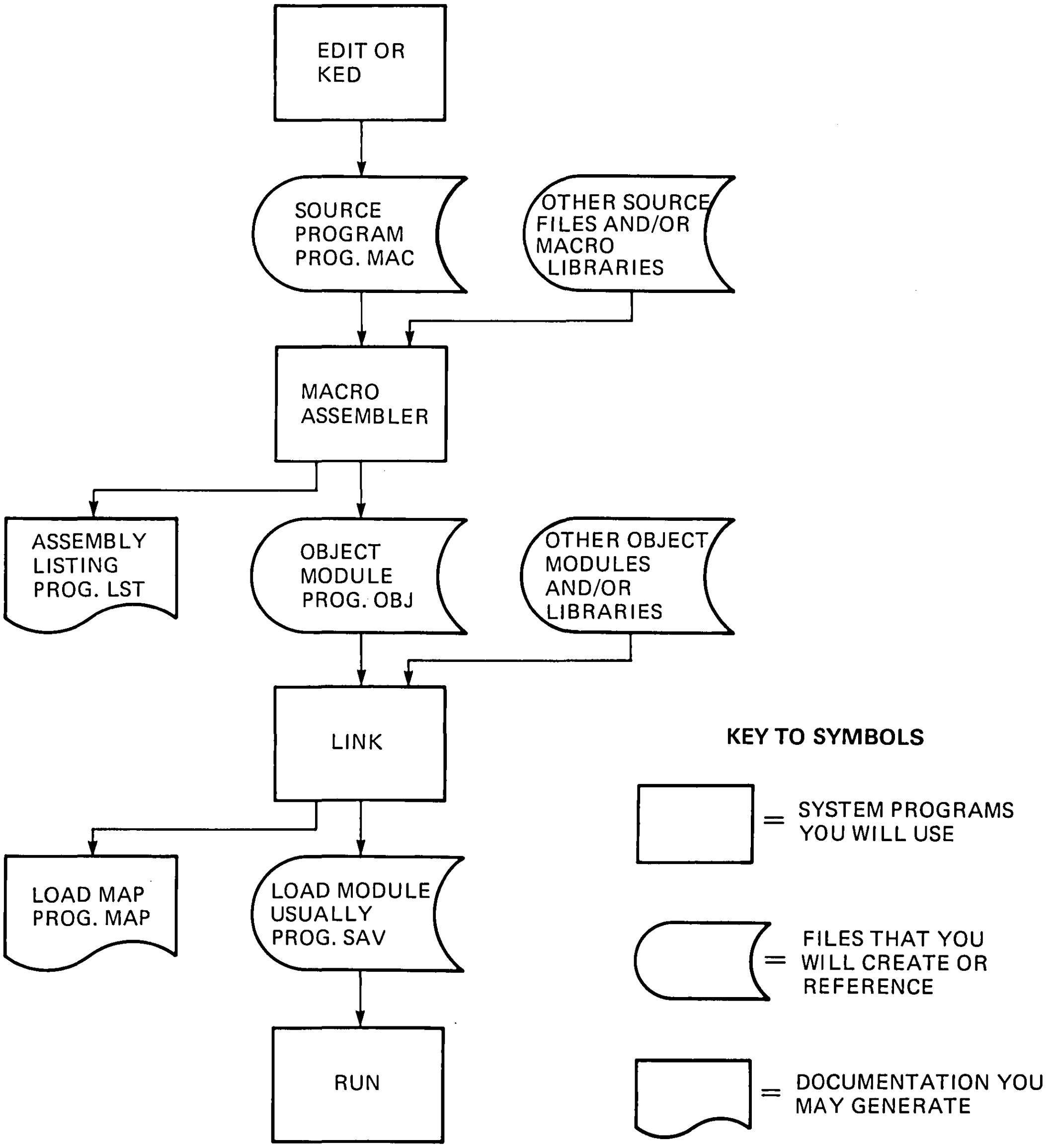
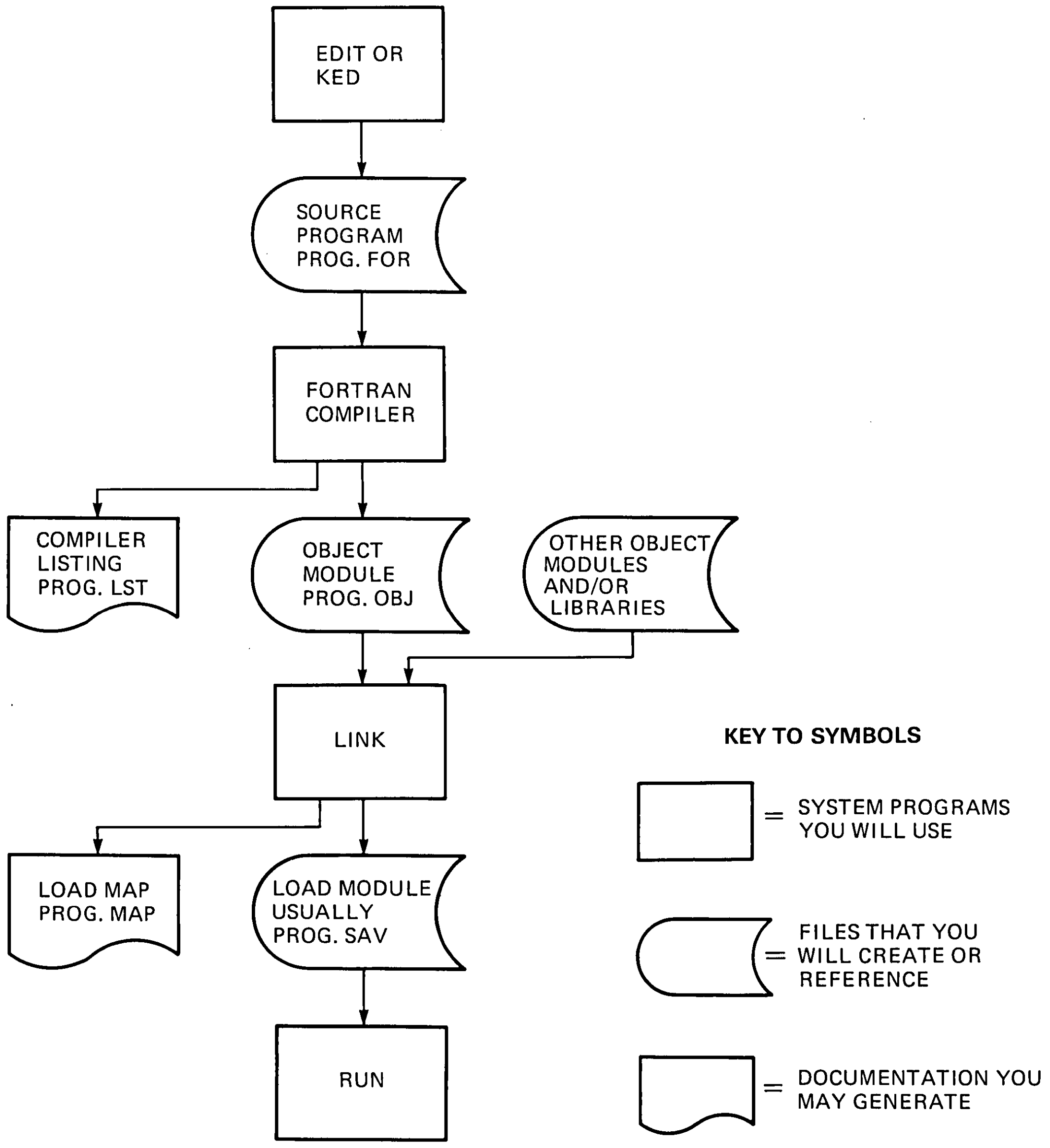

EDIT OR KED
SOURCE PROGRAM PROG. MAC
OTHER SOURCE FILES AND/OR MACRO LIBRARIES
MACRO ASSEMBLER
ASSEMBLY LISTING PROG. LST
OBJECT MODULE PROG. OBJ
OTHER OBJECT MODULES AND/OR LIBRARIES
LINK
LOAD MAP PROG. MAP
LOAD MODULE USUALLY PROG. SAV
RUN
KEY TO SYMBOLS
= SYSTEM PROGRAMS YOU WILL USE
= FILES THAT YOU WILL CREATE OR REFERENCE
= DOCUMENTATION YOU MAY GENERATE

Figure 2.
Program Development in FORTRAN IV


EDIT OR KED
SOURCE PROGRAM PROG. FOR
FORTRAN COMPILER
COMPILER LISTING PROG. LST
OBJECT MODULE PROG. OBJ
OTHER OBJECT MODULES AND/OR LIBRARIES
LINK
LOAD MAP PROG. MAP
LOAD MODULE USUALLY PROG. SAV
RUN
KEY TO SYMBOLS
= SYSTEM PROGRAMS YOU WILL USE
= FILES THAT YOU WILL CREATE OR REFERENCE
= DOCUMENTATION YOU MAY GENERATE

TRAN IV you type the command FORTRAN. These commands, combined with various options, allow you to:

- Produce zero, one or more object modules
- Produce listings
- Process multiple source files
- Control the allocation of storage space to output files

Controlling the Production of an Object Module

When the MACRO–11 assembler or FORTRAN IV compiler produces an object module, it automatically gives the file containing the module the file name of the source file with the file type .OBJ. If a file with the same file name and file type already exists, that file is automatically deleted. To prevent this, you can specify a unique file name and file type for the new object module produced by using the /OBJECT option in the following format:

MACRO/OBJECT:NEW-FILESPEC SOURCE-FILESPEC
or
FORTRAN/OBJECT:NEW-FILESPEC SOURCE-FILESPEC

EXAMPLE

When assembling the program PROG.MAC, you could specify that the object module file should be named NEWPRG.OBJ, by using the command:

```
.MACRO/OBJECT:NEWPRG PROG
```

so that the previous module PROG.OBJ would not be deleted.

You may want to see whether a source file is without syntax errors, but not need to produce an object module. You can use the /NOOBJECT option to prevent the production of object modules. The /NOOBJECT option is used in the following format:

```
MACRO/NOOBJECT  SOURCE-FILESPEC
or
FORTRAN/NOOBJECT  SOURCE-FILESPEC
```

A previous version of an object module can also be saved by making a copy of it in a file with a different name.

Generating Listings

When your assembler or compiler detects syntax errors in your program, you need to know where they occurred. MACRO–11 and FORTRAN IV listings can help because they:

- List the lines of code in the source file
- Indicate where any syntax errors were detected
- Give details about the program sections
- List assembler or compiler statistics

In addition, MACRO–11 assembler listings can:

- Include the table of symbol names that were used in the source code, together with their values
- Show the addresses that will be used by the load module and the data that will be loaded into those addresses
- Indicate which references are external or relocatable

FORTRAN IV compiler listings can:

- Include a table of variables, with their names, data types, and offsets
- Include a table of arrays, with their names, data types, section names, offsets, and dimensions

- Include a table of subroutines and functions referenced, with their names and types
- Provide each line of code with a sequence number

These listings will be discussed in detail later.

To get an assembler or compiler listing of a source file, you can use the /LIST option. The default file type for a listing is .LST.

EXAMPLE

The command:

```
.MACRO/LIST PROG
```

assembles the file PROG.MAC and tells the logical device LP: to print a listing. The command:

```
.FORTRAN/LIST PROG
```

compiles the file PROG.FOR and prints a listing.

You can also specify the file name and file type of a listing file. The command format is:

MACRO/LIST:FILESPEC SOURCE-FILENAME

or

FORTRAN/LIST:FILESPEC SOURCE-FILENAME

The /LIST option can store a listing in a file. The benefit of this is that the file need not be printed out immediately and more than one copy can be printed. To store a listing, you type the /LIST option after the file name.

EXAMPLE

```
.MACRO PROG/LIST
```

If you have the listing file PROG.LST and again process the source file PROG with the /LIST option, the previous listing file is automatically deleted. To prevent this, you can give the new listing a file name other than the default file name, using the /LIST option but with a file name qualifier. The command takes the form:

MACRO/LIST:LIST-FILESPEC SOURCE-FILENAME

or

FORTRAN/LIST:LIST-FILESPEC SOURCE-FILENAME

EXAMPLE

The following command to compile the FORTRAN IV program PROG.FOR creates the listing file AAA.BBB but does not delete PROG.LST:

```
.FORTRAN/LIST:AAA.BBB PROG
```

When you use a file name qualifier, listings are not automatically sent to LP:. To get a hard copy, you must use a separate PRINT command.

Practice 1–1

1. Type in either the MACRO–11 program PR0101.MAC or the FORTRAN IV program PR0101.FOR:

MACRO–11

```
        .TITLE  PR0101
        .MCALL  .PRINT,.EXIT
MESS:   .ASCIZ  /THIS PROGRAM SHOULD ASSEMBLE WITHOUT ERRORS/
        .EVEN
START:  .PRINT  #MESS
        .EXIT
        .END    START
```

FORTRAN IV

```
      PROGRAM PR0101
      TYPE 1000
1000  FORMAT (1H0, 'THIS PROGRAM SHOULD COMPILE WITHOUT ERRORS')
      END
```

2. Assemble or compile the program PR0101 to produce the object module PR0101.OBJ. Assemble or compile the program PR0101 again, to produce the listing MESS.LST and the object module MESS.OBJ without deleting the object module PR0101.OBJ.

Assembling or Compiling Multiple Source Files

You may want to process more than one source file using only one command if:

- You have a source program that is divided across more than one file
- You want to process more than one source program in order to produce more than one object module

Multiple MACRO-11 source files

Assume that your MACRO-11 source program is made up of the following five files:

COPRGT.MAC	Your own copyright text in the form of MACRO–11 comment lines
PRJCMT.MAC	Comments that apply to every module in the project for which this program is written
MACDEF.MAC	Common macro definitions
DATDEF.MAC	Common data definitions
PROG01.MAC	The code for the main part of your program

To assemble the program into one object module and produce a complete listing, you issue the following command.

EXAMPLE

```
.MAC/LIS:PROG01/OBJ:PROG01 COPRGT+PRJCMT+MACDEF+DATDEF+PROG01
```

Usually, the file name of output files defaults to the first file name in the list. Here it would be COPRGT. /LIST:PROG01 overrides the default so that the listing file PROG01.LST and the object module PROG01.OBJ are produced.

You can also use one command to assemble several MACRO–11 programs into separate object modules. The format of this command is:

MACRO FILENAME1,FILENAME2,...,FILENAMEn

This is the same as assembling each of the source files with a separate command. Figure 3 shows a MACRO–11 program in which the source code is divided across more than one file.

Multiple FORTRAN IV source files

Assume that you have a FORTRAN IV source program made up of three files:

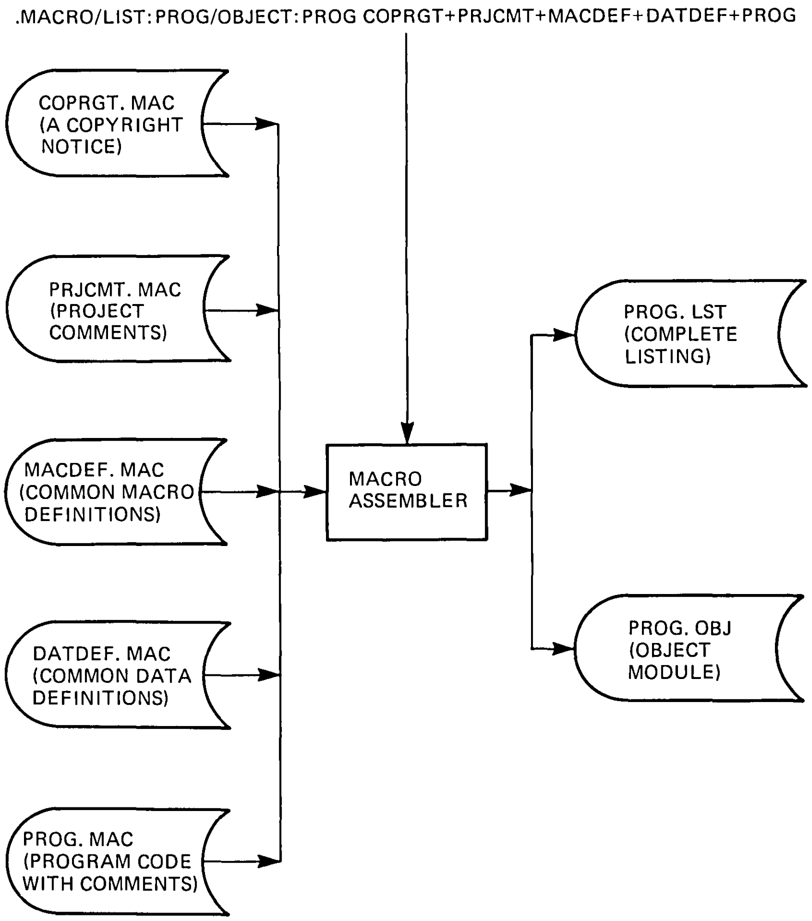
COPRGT.FOR	Your own copyright text in the form of FORTRAN IV comment lines
PRJCMT.FOR	Comments that apply to every module in the project for which this program is written
PROG01.FOR	The code for your program

To compile the program into one object module and produce a listing that includes the comment lines, you use the following command.

EXAMPLE

```
.FORTRAN/LIST:PROG01/OBJECT:PROG01 COPRGT+PRJCMT+PROG01
```

Figure 3.
Assembling Multiple MACRO–11 Source Files


.MACRO/LIST:PROG/OBJECT:PROG COPRGT+PRJCMT+MACDEF+DATDEF+PROG
COPRGT. MAC (A COPYRIGHT NOTICE)
PRJCMT. MAC (PROJECT COMMENTS)
MACDEF. MAC (COMMON MACRO DEFINITIONS)
DATDEF. MAC (COMMON DATA DEFINITIONS)
PROG. MAC (PROGRAM CODE WITH COMMENTS)
MACRO ASSEMBLER
PROG. LST (COMPLETE LISTING)
PROG. OBJ (OBJECT MODULE)

You can also use one command to separately process more than one program producing more than one object module. You can compile the three FORTRAN IV source programs, PROG01.FOR, PROG02.FOR, and PROG03.FOR, by using the following command.

EXAMPLE

```
.FORTRAN PROG01,PROG02,PROG03
```

The object modules, PROG01.OBJ, PROG02.OBJ, and PROG03.OBJ are produced. This single command is equivalent to the following three commands:

```
.FORTRAN PROG01
.FORTRAN PROG02
.FORTRAN PROG03
```

Figure 4 shows a FORTRAN IV program in which the source code is divided across more than one file.

General form

You can use both the plus sign (+) and the comma (,) in a command to assemble or compile more than one source file. In general, you can assemble or compile more than one set of source files so that each set produces one object module. Each set is separated from the next by a comma and each set is made up of one or more file names separated by a plus sign.

EXAMPLE

The command:

```
.MACRO A+B,C,D+E+F
```

produces A.OBJ from A.MAC and B.MAC; C.OBJ from C.MAC; and D.OBJ from D.MAC, E.MAC, and F.MAC.

Figure 5 illustrates a single MACRO–11 command causing more than one module to be assembled. Some of these programs are divided into more than one source file. A similar capability is available when you use FORTRAN IV.

Figure 4.
Compiling Multiple FORTRAN IV Source Files

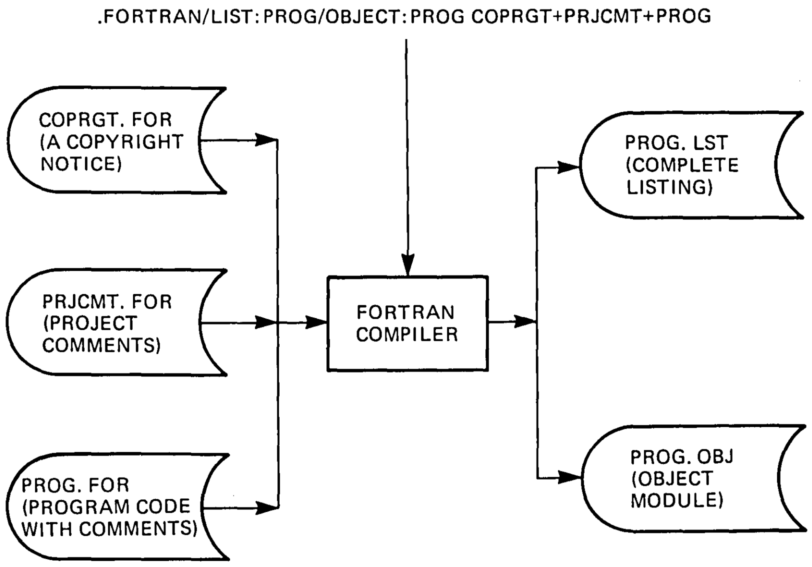
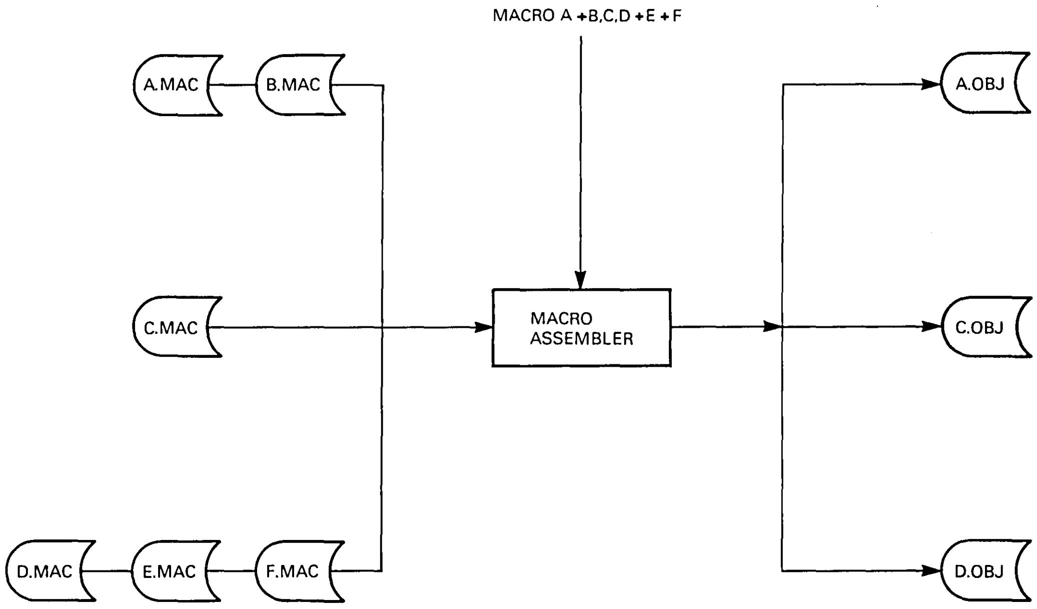

.FORTRAN/LIST:PROG/OBJECT:PROG COPRGT+PRJCMT+PROG
COPRGT. FOR (A COPYRIGHT NOTICE)
PRJCMT. FOR (PROJECT COMMENTS)
PROG. FOR (PROGRAM CODE WITH COMMENTS)
FORTRAN COMPILER
PROG. LST (COMPLETE LISTING)
PROG. OBJ (OBJECT MODULE)

Figure 5.
Producing Multiple Modules from Multiple MACRO–11 Source Files



Practice 1–2

Using MACRO–11

1. Type the following programs into three files. Name them PR0102.MAC, PR0103.MAC, and PR0104.MAC.

PR0102.MAC:

```
    .TITLE PR0102
;*
;* COPYRIGHT (c) 1984          A. N. Other
;*
;* This software was written by A. N. Other and may be
;* used by anyone. A. N. Other is not responsible for
;* any errors.
;*
;*
```

PR0103.MAC:

```
;***
; Project RTV5
; --------------
;
; This text describes the project RTV5, Program PR0104
; is a part of this project.
;*
```

PR0104.MAC:

```
        .MCALL  .PRINT,.EXIT
MESS:   .ASCIZ  /THIS PROGRAM IS PART OF PROJECT RTV5/
        .EVEN
START:  .PRINT  #MESS
        .EXIT
        .END    START
```

2. Assemble the source files PR0102, PR0103, and PR0104 to produce a single object module along with a single listing file. The listing file should contain the code from all three source files. Name the output files PR1234.OBJ and PR1234.LST.

Practice 1–3

Using FORTRAN IV

1. Type the following programs into three files. Name them PR0102.FOR, PR0103.FOR, and PR0104.FOR.

PR0102.FOR:

```
C *
C *
C *    Copyright (c) 1984       A.N. Other
C *
C *    This software was written by A. N. Other and
C *    may be used by anyone. A. N. Other is not
C *    responsible for any errors.
C *
C *
```

PR0103.FOR:

```
C *
C Project RTV5
C -------------
C
C This text describes the project RTV5, Program PR0104
C is a part of this project.
C *
```

PR0104.FOR:

```
      TYPE 1000
1000  FORMAT(1H0, 'THIS PROGRAM IS PART OF PROJECT RTV5')
      END
```

2. Compile the source files PR0102, PR0103, and PR0104 to produce a single object module, along with a single listing file. The listing file should contain the code from all three source files. Name the output files PR1234.OBJ and PR1234.LST.

Allocating Storage Space for Your Output Files

The size of listing files and object modules does not depend on the size of the source files from which they are produced. Listings can include or omit different types of information, as we will discuss later. Lines of source code generate any number of words of object code. For these reasons, the MACRO–11 assembler and the FORTRAN IV compiler are not able to determine the size of output files in advance.

If you know the approximate size of the output files, you can make sure that the assembler or compiler checks whether there is enough room by using the /ALLOCATE option. Assume that you know that your source file PROG will produce an object module of size 100 blocks, but you are not sure if there is enough room on the disk. You can check whether there will be enough room for this file.

EXAMPLE

```
.MACRO PROG/OBJECT/ALLOCATE:100
```

If there is not enough room for the allocation, the following error message appears:

```
?MACRO-F-Device full DK:PROG.LST
DK:PROG,DK:PROG[200]=DK:PROG
```

You would use the FORTRAN command in the same format to check if a disk has enough room for an object module. You can make a similar check for listing files.

More Macro–11 Assembler Options

In addition to the options already discussed, you can use other MACRO–11 assembler options to perform tasks. You can control the selection of information appearing in listings and you can generate a cross-reference listing to assist you in analyzing the source code. You can also control how the MACRO–11 assembler interprets your source code and generates your object code.

Optional Information in MACRO–11 Listings

Now that you know how to get a MACRO–11 listing, you need to know how to control the information it contains. You use the .LIST and .NLIST directives to select the information to be included in an assembler listing. For the lines of source code in which directives are not in operation, the assembler makes a default selection. These defaults are shown in table 1. To override the default selections for one listing without having to edit the source code, use the /SHOW and /NOSHOW options.

/SHOW option

You use the /SHOW option to include listing information that would otherwise be omitted by default. The format is:

```
MACRO/LIST/SHOW:ARGUMENT FILESPEC
```

EXAMPLE

The command:

```
.MACRO/LIST:PROG/SHOW:ME:LD PROG
```

causes the assembler to generate the object module PROG.OBJ and a listing file PROG.LST. This listing includes the macro expansions (ME) of any macros found in your program during assembly, and lists those directives (LD) that have no arguments.

Refer to table 6–2 in the *PDP–11/MACRO–11 Language Reference Manual* for a complete list of MACRO–11 listing control directives.

Table 1.
.LIST and .NLIST Directive Summary

Argument	Default	Controls
SEQ	List	Source line sequence numbers
LOC	List	Location counter
BIN	List	Generated binary code
BEX	List	Binary extensions
SRC	List	Source code
COM	List	Comments
MD	List	Macro definitions, repeat range expansions
MC	List	Macro calls, repeat range expansions
ME	List	Macro expansions
MEB	Nolist	Macro expansion binary code
CND	List	Unsatisfied conditionals, .IF and .ENDC statements
LD	Nolist	Listing directives with no arguments
TOC	List	Table of contents
TTM	Line printer mode	Output format
SYM	List	Symbol table

/NOSHOW option

The /NOSHOW option has the same general format as the /SHOW option, but it has the opposite effect. Use it to prevent listing information that would otherwise be included by default.

EXAMPLE

The command:

```
.MACRO/LIST:PROG/NOSHOW:BIN PROG
```

results in the generation of an object module PROG.OBJ and a listing file PROG.LST. The binary expansions (BIN) of source code are not included in the listing file.

Note that the /SHOW and /NOSHOW options may be used only in conjunction with the /LIST option. If you do not use the /SHOW or /NOSHOW option, the default selection of information appears in the listing.

Practice 1–4

Assemble the source file PR0101.MAC so that a listing file is produced. Using the /SHOW and /NOSHOW options, specify the following types of information to be included in the listing:

- The source code
- Comments
- Macro definitions
- Macro calls
- Macro expansions
- No other optional information

Study the listing and see where each type of information appears.

Cross-Reference Listings

If you have a very complex MACRO–11 program that you want to modify with only a normal listing, you may find it difficult to do the following:

- Check whether the source file contains user-defined symbols whose names are also Digital assembler mnemonics
- Find the position of the definition of a symbol in order to change that definition
- Identify all the places where a symbol is referenced
- Identify all the places where a symbol is modified
- Locate the specific positions at which errors are flagged

A cross-reference (CREF) listing can provide all of this information for you. To get a cross-reference listing use the /CROSSREFERENCE option. You can use the /CROSSREFERENCE option to include the following information in a listing:

- The name of each symbol referenced
- The type of symbol it is
- The position at which each symbol is defined (if the definition appears in the source files assembled)
- The positions at which each symbol is referenced

You give a code for each optional type of information to be included. This one-character code or argument indicates the sections of a cross-reference listing the assembler should include. Table 2 lists and describes these arguments. You must use the /CROSSREFERENCE option with the /LIST option and issue the command in the form:

```
MACRO/LIST/CROSSREFERENCE:ARGUMENT FILESPECS
```

The /CROSSREFERENCE option does not enable cross-references on lines of code that are disabled with the .NOCROSS directive.

Table 2.
Cross-Reference Sections

Argument	Section Type
S	User-defined symbols
R	Register symbols
M	Macro symbolic names
P	Permanent symbols (instructions, directives)
C	Control sections (.CSECT symbolic names)
E	Error codes
None	Equivalent to :S:M:E

Interpreting CREF Listings

A CREF listing includes the following information:

- A code letter with each page number for the argument represented in that section (for example, S–1)
- The name of each symbol, together with a list of one or more numbers specifying the page and lines on which the symbol occurs
- Additional information about each symbol indicated by a special character to the right of one of the line numbers. The special characters are:

#	refers to a symbol definition
*	refers to an operation that changes the contents of a location

EXAMPLE

The CREF listing:

```
VAR    1-20#    2-17    2-24*
LF     1-10#    2-20    2-23
```

informs you that the symbol VAR is defined on line 20 of page 1, is referenced on line 17 of page 2, and is modified on line 24 of the same page. LF is defined on line 10 of page 1 and is used on page 2, line 20, and page 2, line 23.

Controlling Code Recognition and Generation

The assembler interprets your source code and generates your object code according to programmed defaults. You can override these defaults in order to select from the options that then become available.

Table 3 lists the assembler features for controlling code recognition and generation by means of .ENABL and .DSABL directives in source files. Where these directives are not in operation, the assembler makes a default selection.

You can use the /ENABLE and /DISABLE options to override default selections without having to edit the source code. The /ENABLE option enables features that would otherwise be omitted by default. Use it as follows:

MACRO/ENABLE:ARGUMENT FILESPEC

EXAMPLE

The command:

```
.MACRO/ENABLE:LC PROG
```

causes the assembler to recognize lowercase character strings in the source code when building the object module PROG.OBJ.

If this option is not enabled, all characters are converted to uppercase. This is also true for any listing file that is produced when this option is selected.

More FORTRAN IV Compiler Options

In addition to the features we have discussed, you can use the FORTRAN IV compiler to select the types of information that appear in a listing and to select whether debugging lines (lines of code with a "D" in the first column) are treated as code or as comments. The compiler also tells you when errors occur. FORTRAN IV compiler optimization methods are discussed in chapter 6, "Designing and Implementing Overlay Structures."

Table 3.
.DSABL and .ENABL Directive Summary

Argument	Default	Enables or Disables
ABS	Disable	Absolute binary output
AMA	Disable	Assembly of all absolute addresses as relative addresses
CDR	Disable	Treating source columns 73 and greater as comments
DBG	Disable	Generation of internal symbol directory (ISD) records during assembly (See chapter 8 of the *RT-11 Software Support Manual* for more information on ISD records)
FPT	Disable	Floating-point truncation
GBL	Enable	Treating undefined symbols as globals
LC	Enable	Accepting lowercase ASCII input
LCM	Disable	Uppercase and lowercase sensitivity of MACRO-11 conditional assembly directives .IF IDN and .IF DIF
LSB	Disable	Local symbol block
PNC	Enable	Binary output
REG	Enable	Mnemonic definitions of registers

Optional Information in FORTRAN IV Listings

The types of information that you can select to appear in a FORTRAN IV listing are:

- Source program
- Diagnostic messages
- Storage map
- Generated code

By default, the first three types of information are included in the listing. You can override this default by using the /SHOW option in the following format:

FORTRAN/LIST/SHOW:code FILESPEC

The different selections you can make are shown in table 4.

EXAMPLE

Either of the commands below:

```
.FORTRAN/LIST:PROG/SHOW:2 PROG
```

or

```
.FORTRAN/LIST:PROG/SHOW:MAP PROG
```

instructs the compiler to generate an object module PROG.OBJ and a listing file PROG.LST. This listing includes only the storage map and diagnostic messages.

When you refer to a listing, it is often useful to know which compiler options were in effect. You can make this information appear in the listing by using the /HEADER option. The /HEADER option results in the generation of a listing that contains all the information generated by the command in the previous example, in addition to a list of compiler characteristics.

Table 4.
FORTRAN Listing Codes

Code	Listing Content
0	Diagnostics only
1 or SRC	Source program and diagnostics
2 or MAP	Storage map and diagnostics
3	Diagnostics, source programs, and storage map
4 or COD	Generated code and diagnostics
7 or ALL	Diagnostics, source program, storage map, and generated code

EXAMPLE

```
.FORTRAN/LIST/SHOW:2/HEADER PROGA
```

Debugging Lines

During the development of your programs, you may want to check run time and final data to determine whether routines are performing as planned. The most direct way to do this is to insert printing statements at carefully selected points so that the information you need can appear at the terminal. Because you may want to use these test statements regularly, the FORTRAN IV compiler has been designed to recognize all statement lines beginning with a "D" in column 1 as special debugging lines. The advantage of this is that you can choose to include debugging code without having to edit your source program. Use the /ONDEBUG option if you want the compiler to compile debugging lines.

EXAMPLE

The command:

```
.FORTRAN/ONDEBUG PROG
```

causes statements with "D" in column 1 of the FORTRAN IV program PROG to be compiled.

FORTRAN IV Error Messages

Despite careful creation and editing of your source code, errors do occur. During the first two phases of compilation, the compiler checks for syntax and definition errors. FORTRAN IV includes the appropriate error messages in the listing. There are different message formats for the errors detected in each of these two phases. Errors reported by the first phase of compilation have the format:

```
***** c
```

Here "c" is a code letter. The meanings of the various code letters are described in the table on page C-3 of the *RT–11/RSTS/E FORTRAN IV User's Guide*. For example, the code letter "S" refers to a syntax error. Errors reported from the second phase of compilation have the general format:

```
IN LINE nnnn, Error: description
```

Here "nnnn" is the internal sequence number of the statement in question, and "description" is a short description of the error.

You can find a list of compilation error messages in appendix C of the *RT–11/RSTS/E FORTRAN IV User's Guide*. This appendix includes an explanation of the probable causes of all types of errors recognized by the compiler.

Errors reported during the execution of your FORTRAN IV program are called OTS (Object Time System) errors. Appendix C of the *RT–11 RSTS/E FORTRAN IV User's Guide* also describes OTS errors in detail.

Linking

You use the RT–11 LINK utility to:

- Join object modules and resolve references across modules
- Relocate individual object modules as necessary, assign absolute (permanent) memory addresses, and, if necessary, define overlay structures (discussed later)
- Produce an executable form of your program called a load image and an optional load map

One major advantage of program linking is that it allows you to implement your program designs in a modular way. After you have assembled or compiled a number of individual modules, use the linker to join them into a single running program.

You can also use the linker to obtain modules from an object library and use them in building the program. An object library is a single file that contains more than one object module. You may join any combination of object modules and object library modules at link time. The linker produces two types of output files, load map files and load image files.

Load Map Files

As an option, the linker produces a load map. This is a listing file that describes how the save image file was put together. It indicates the base address within the save image of each module and named program section. It also lists the addresses of globally defined symbols. The generation of load maps is controlled by the /MAP option.

EXAMPLE

The command:

```
.LINK/MAP PROG
```

links the modules in file PROG.OBJ and generates a load map, which is directed towards the line printer. You can output the map to a file, using the :FILENAME qualifier:

```
.LINK/MAP:PROG.MAP PROG
```

This command additionally produces a file PROG.MAP, which contains the map listing for PROG.OBJ.

Practice 1–5

1. If you are using MACRO–11, type the following program into a file and name it PR0105.MAC.

```
; This is text
        .TITLE  PR0105
        .MCALL  .PRINT,.EXIT
MESS:   .ASCIZ  /THIS IS A MESSAGE/
        .EVEN
START:  .PRINT  #MESS
        .EXIT
        .END    START
```

   If you are using FORTRAN IV, type the following program into a file and call it PR0105.FOR.

```
C This is text
      TYPE 1000
1000  FORMAT (1H0, 'THIS IS A MESSAGE')
      END
```

2. Produce an object module, a save image, and a load map file for program PR0105. Print out the load map and on it, circle the sections corresponding to the following eight items and mark them with the letter indicated. Include the following:

   a. The load module file name and type

   b. The date of creation of the load module, provided that the date was entered using the DATE monitor command

   c. The time of creation of the load module

   d. The title of the load module

   e. The version number of the linker program

   f. All section names, together with the address where each section begins and the size of each section (octal bytes)

   g. The transfer address of the program (the starting address or entry point)

   h. The high limit of the program in octal bytes and decimal words

3. If any of the information listed above is missing, find the reason for its absence.

Load Image Files

The linker operates on the object module(s) that you include in your command line to produce a load image file. The three different types of load image files are save, relocatable, and absolute binary.

Save image

A save image is required to run your program under the Single Job (SJ) monitor, or as a background job under the Foreground/Background (FB) or Extended Memory (XM) monitors. The linker stores this image in a file which has a .SAV file type.

This file is an image of your program as it appears in memory immediately after you load it. Each word in the file is loaded into a location in memory. The first block of the file (block 0) contains the machine code that is loaded into locations 0 to 776 (octal). Block 1 is loaded into locations 1000 (octal) to 1776 (octal), and so on.

Figure 6 shows how a save image file is laid out. Locations 40 to 50 in block 0 of such a file contain the control parameters of your program. These are initialized by the linker and contain the information shown in table 5.

Locations 360 to 377 in block 0 of the file are reserved for use by RT–11. The linker stores the memory usage bits in the eight words of this block. The bit map is organized as follows: each bit of these words represents one 256-word block of memory and is set to 1 if your program occupies

Figure 6.
Save Image File Structure

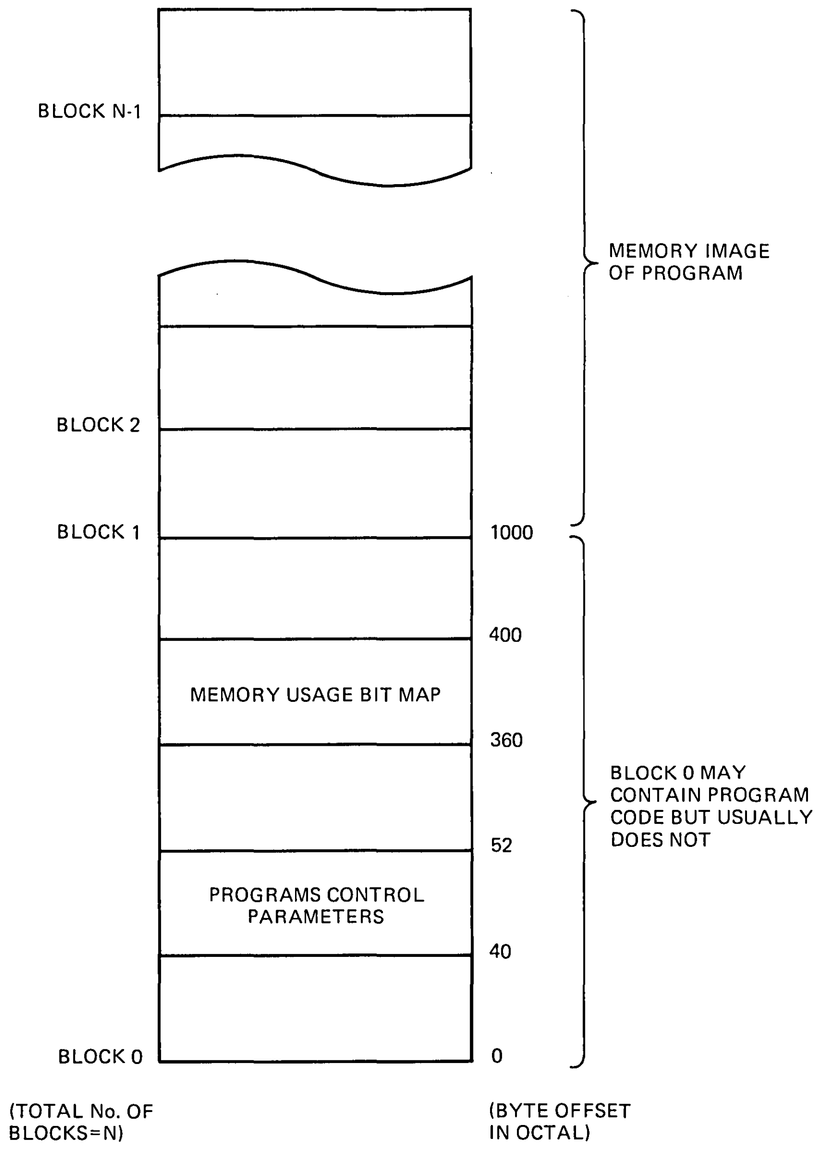


Table 5.
Information in Block 0

Location	Information
34	Trap vector (TRAP)
36	Trap vector (TRAP)
40	Program's relative start address
42	Initial location of stack pointer (changed by /M option)
44	Job Status Word
46	USR swap address
50	Program's high limit
52	Size of program's root segment, in bytes (used for .REL files only)
54	Stack size, in bytes (changed by /R:n option, used for .REL files only)
56	Size of overlay region, in bytes (0 if not overlaid, used for .REL files only)
60	.REL file ID (.REL in Radix-50, used for .REL files only)
62	Relative block number for start of relocation information (used for .REL files only)

that block of memory. Other locations in block 0 may contain program code, initial vector contents, or data, but under most conditions they are not used. The R, RUN, and GET commands use this information when loading your program. The information from block 1 to the last block of your file contains the image of your program.

Relocatable image

In order to run a program in foreground under the FB monitor, you must first use the linker to produce a relocatable image file. This allows the program to be loaded into higher memory, leaving the lower memory available for use by a normal save image. A relocatable image has the .REL file type. The structure of this file is shown in figure 7. Block

0 of the .REL file contains the program control parameters in locations 34 to 62 (see figure 8). Locations 40 to 50 have the same contents as the save image file.

The remainder of the file is divided into two parts. The first part begins in block 1 and occupies the number of blocks necessary to contain the memory image of your program, as in the .SAV file. Relocation information occupies the subsequent blocks, beginning with the block indicated in location 62 of block 0 of the file. The linker links your foreground program to start at location 1000 (octal) by default. However, when you load and run your program with the FRUN command, the FRUN processor uses this relocation information to load the program, not at location 1000, but rather, just below the resident monitor or loaded device handlers. During the relocation operation, the FRUN processor modifies certain locations in your program according to the relocation information in order to ensure that your program will run in available memory when started (described in chapter 2, "Executing Programs.")

To generate a relocatable image file, use the /FOREGROUND option of the linker. This option assigns the default file type .REL to the load module.

EXAMPLE

The command:

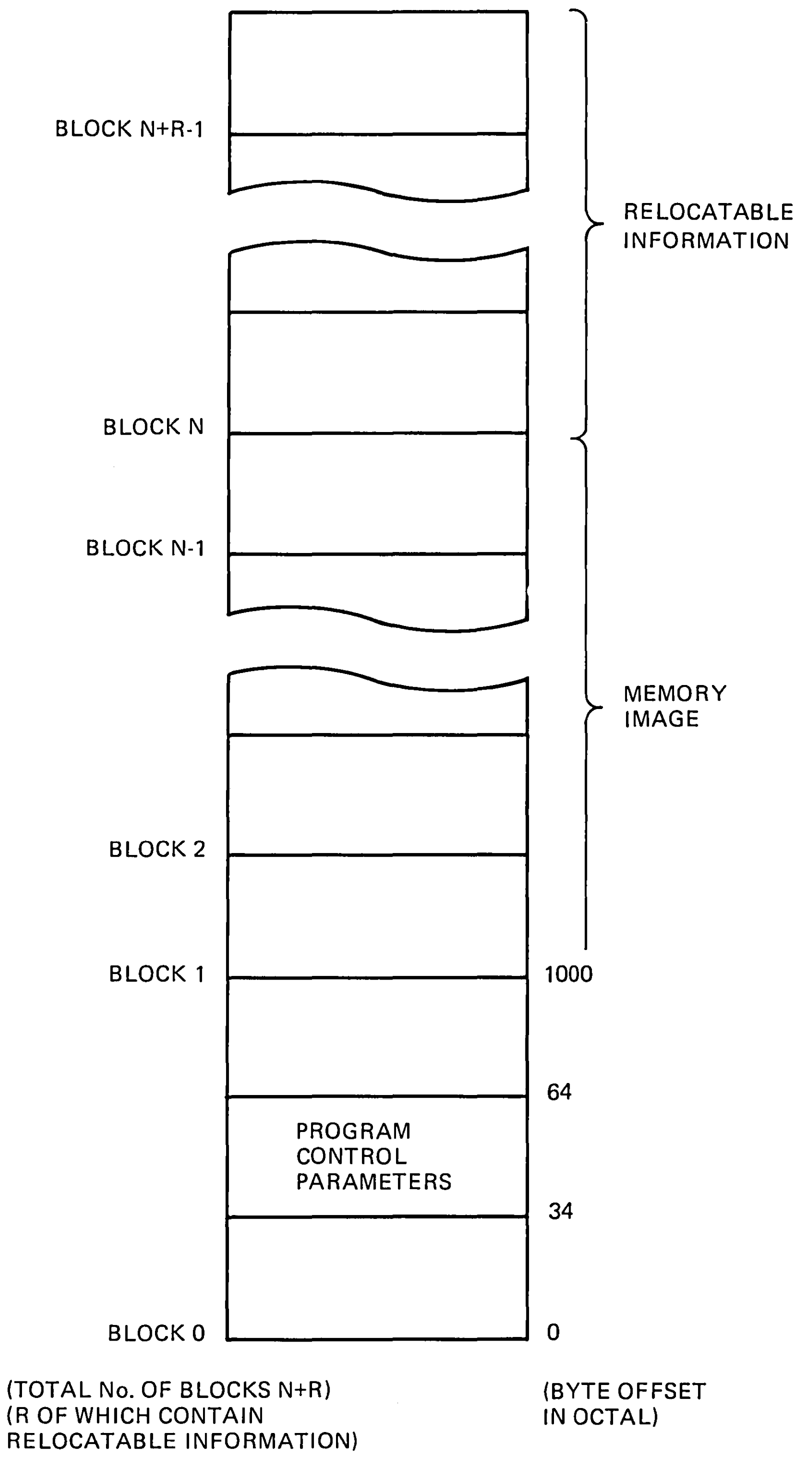
```
.LINK/FOREGROUND PROG
```

links the file PROG.OBJ and produces a relocatable image file PROG.REL.

Absolute binary image

Use an absolute binary image when you want a program to run without the operating system controlling the system resources. (Chapter 2, "Executing Programs," discusses how you can load this type of image using the absolute loader.) You must design this type of load module so that it can control any system resources it needs. Use the /LDA option

Figure 7.
Relocatable Image File Structure



in the linker command line to generate an absolute binary image.

EXAMPLE

The command:

```
.LINK/LDA PROG
```

links the modules in file PROG.OBJ and produces an absolute binary image in file PROG.LDA.

More Linker Options

Linker options which enable you to control certain features of a load image when it is loaded and executed include stack location and size, base address, and debugging aids.

Stack location and size

Unless you specify otherwise, the linker provides your load module with a default stack location and size. If your program requires a greater stack depth, you will need to allocate more stack space. If, on the other hand, your program does not need as great a stack depth, you can decrease the allocated depth to make more space available for use by program code and data.

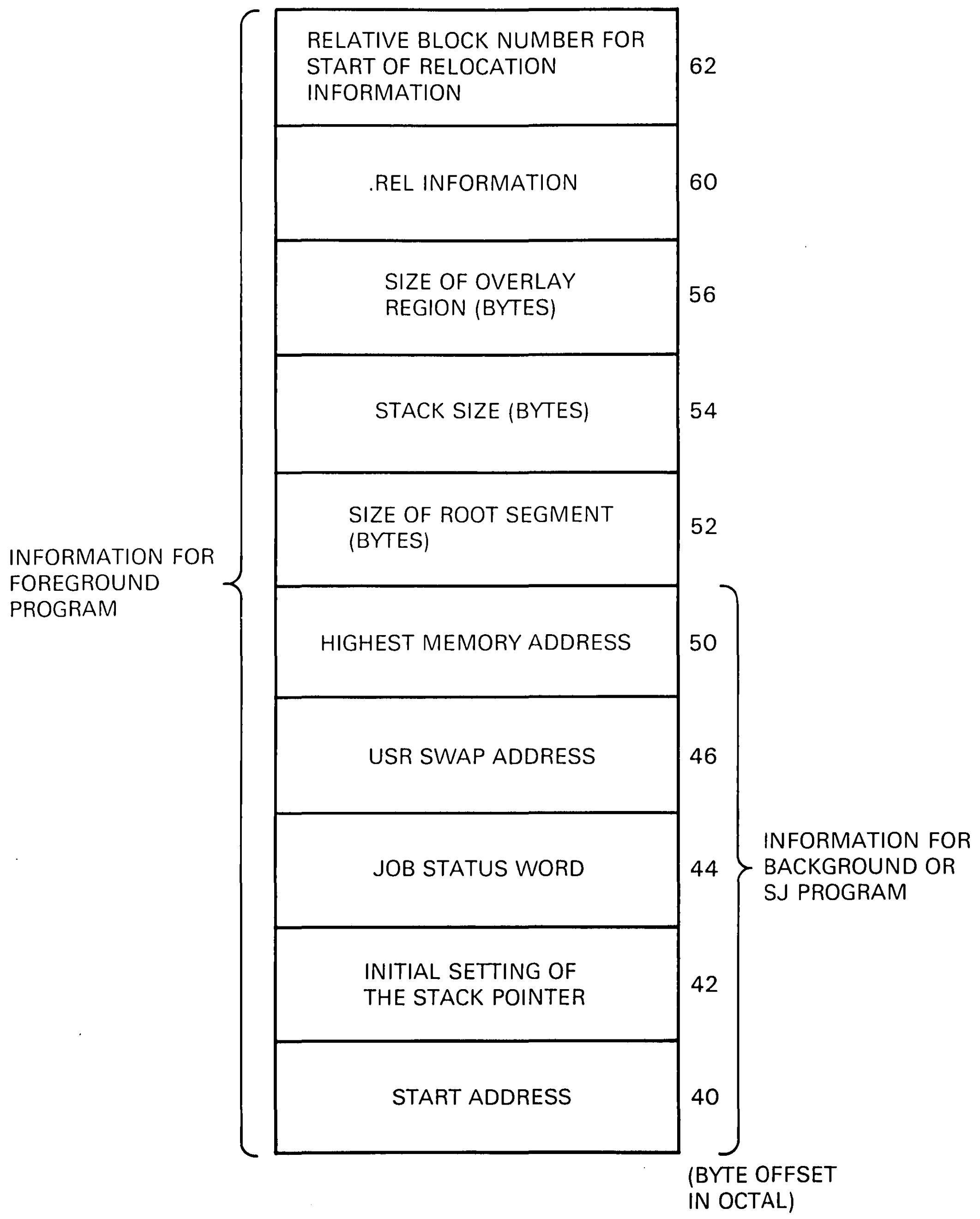
For save images, the stack location, which is determined by the initial value of the stack pointer (SP), determines the size of the stack. You can override the default location of 1000 (octal) by using the command:

```
LINK/STACK:location FILENAME
```

and giving your location in octal. If you omit the :location qualifier, the system will prompt you for a stack location. To make use of the space created, you must modify the base address.

For relocatable images, you cannot modify the actual

Figure 8.
Layout of Program Control Parameters for a .REL File


RELATIVE BLOCK NUMBER FOR START OF RELOCATION INFORMATION
62
.REL INFORMATION
60
SIZE OF OVERLAY REGION (BYTES)
56
STACK SIZE (BYTES)
54
SIZE OF ROOT SEGMENT (BYTES)
52
HIGHEST MEMORY ADDRESS
50
USR SWAP ADDRESS
46
JOB STATUS WORD
44
INITIAL SETTING OF THE STACK POINTER
42
START ADDRESS
40
INFORMATION FOR FOREGROUND PROGRAM
INFORMATION FOR BACKGROUND OR SJ PROGRAM
(BYTE OFFSET IN OCTAL)

location of the stack, since this is determined at run time, but you can override the default stack size of 128 bytes by using the command:

```
LINK/FOREGROUND:stack-size FILENAME
```

and giving the stack size in bytes (octal).

Base address

The base address of your program is located immediately above the stack. If you raise the stack location for a save image, you should raise the base address to prevent overlap between the stack and the code. If you lower the stack location, you also should lower the base address in order to move down the program code and data, making use of the unused space. To override the default base address of 1000 (octal), use the command:

```
LINK/BOTTOM:base-address FILENAME
```

and give the base address in octal. Relocatable images cannot be given a base address because the address is determined at run time (discussed in chapter 2, "Executing Programs").

Debugging aids

Debugging is the process of correcting run-time errors in a program. Two commonly used debugging aids are the Online Debugging Technique (ODT) and the FORTRAN IV Debugging Tool (FDT). ODT is supplied with all RT–11 systems as a standard system software item, whereas FDT is available only as part of the FORTRAN IV Real-time Extensions Package.

You use ODT by linking your object modules with ODT, using the command:

```
LINK/DEBUG  FILENAME
```

The resulting load module will be modified so that it includes the necessary code to use ODT. ODT is further discussed in chapter 4, "Debugging Programs." To use FDT, you specify /DEBUG:FDT, which overrides the default tool ODT. Instruction on the use of FDT is not given in this book.

Summary of Linker Options

LINK

/ALPHABETIZE	lists in the load map your program's global symbols in alphabetical order
/BITMAP	creates a memory usage bitmap
/BOTTOM	specifies the lowest address to be used by the relocatable code in the load module
/BOUNDARY	starts a specific program section in the root on a particular address boundary
/DEBUG	links ODT (on-line debugging technique) with your program
/DUPLICATE	places duplicate copies of a library module in each overlay segment that references the module
/EXECUTE	specifies a file name or device for the executable file
/EXECUTE/ALLOCATE	reserves space on a device for the executable file
/EXTEND	extends a program section to a specific octal value
/FILL	initializes unused locations in the load module and places a specific octal value in those locations
/FOREGROUND	produces an executable file in relocatable format for use as a foreground job under the FB or XM monitor
/GLOBAL	generates a global symbol cross-reference section in the load map

/INCLUDE	takes global symbols from any library and includes them in the linked memory image
/LDA	produces an executable file in LDA format
/LIBRARY	same as /LINKLIBRARY
/LINKLIBRARY	includes the library file you specify as an object module library in the linking operation
/MAP	produces a load map listing
/MAP/ALLOCATE	reserves space on a device for the load map listing file
/MAP/WIDE	produces a wide load map listing
/NOBITMAP	suppresses creation of a memory usage bitmap
/NOEXECUTE	suppresses creation of an executable file
/PROMPT	tells the system to accept lines of linker input until you enter two slashes (//)
/ROUND	rounds up the section you specify so that the size of the root segment is a whole-number multiple of the value you supply
/RUN	initiates execution of a background job which does not require responses from the terminal, produces a .SAV file
/SLOWLY	instructs the system to allow the largest possible memory area for the link symbol table
/STACK	allows you to modify location 42, the address containing the value for the stack pointer (SP)
/SYMBOLTABLE	creates a file that contains symbol definitions for all the global symbols in the load module

/TOP	specifies the highest address to be used by the relocatable code in the load module
/TRANSFER	allows you to specify the start address of the load module
/XM	enables special .SETTOP and .LIMIT programmed request features provided in the XM monitor
/XM/LIMIT	limits the amount of memory allocated by .SETTOP

References

RT–11 System User's Guide. Chapter 4 discusses options to the MACRO and FORTRAN commands.

RT–11/RSTS/E FORTRAN IV User's Guide. Appendix C contains material on error diagnostics.

RT–11 System Message Manual.

2

2 Executing Programs

After assembling or compiling your program source code and linking the resulting object modules to produce a load image file, you are now ready to execute the file. This chapter discusses the monitor commands: EXECUTE, R, RUN, FRUN, SRUN, and UNLOAD. When you have completed this chapter you will be able to run background, foreground, and system jobs, and send data from the terminal to one or more jobs running at the same time.

Program Execution

Having produced a load module, you are ready to execute it. To execute one program at a time, use the Single Job (SJ) monitor. To execute two programs at the same time, use the Foreground/Background (FB) monitor. If your machine has more than 32 Kwords of memory, you can use the Extended Memory (XM) monitor. Using the FB or XM monitor, you can run your programs and system jobs at the same time.

The commands that specify program execution directly are RUN, R, FRUN, and SRUN. When a save image is on the logical device SY:, you can run the image by entering its file name.

Using the Single Job Monitor

When you execute a job, you perform two functions, loading the code and data from the image file into memory and starting execution of the code. You can use a single command to perform both functions. For the Single Job monitor this command is:

RUN FILESPEC

EXAMPLE

To load and execute the file RK3:PROG.SAV, you type:

```
.RUN RK3:PROG
```

If the program is on the system device, you can shorten the command to R. To run the program SY:PROG.SAV, use the command:

R PROG

You can also run such a program just by typing its name:

```
PROG
```

You will probably need to debug a program before running it. To debug a program, load it into memory where you can examine the contents of locations and modify the data before starting execution. The commands you need in order to do this are discussed in chapter 4, "Debugging Programs."

Terminating Jobs

Normally a program exits via the .EXIT directive for MACRO–11, or CALL EXIT for FORTRAN IV; however, not all programs terminate in this way. Some programs cause a fatal monitor error before performing such an exit, while others "hang." A program hangs if it enters a permanent loop or if it waits for an event to occur that does not take place. You can abort a hung job by pressing ‹CTRL/C› twice.

Using the Foreground/Background Monitor

All the facilities of the SJ monitor are available in the FB monitor, plus additional features that enable you to load more than one program and schedule them for concurrent execution.

When a foreground program is running, it cannot be interrupted for the execution of background code. The background program runs only if the foreground program is waiting for an external event, such as the arrival of data from a peripheral device. A job is said to be blocked if it is waiting for an external event, for example, the performance of I/O operations.

Perhaps you have a foreground program that uses the CPU for long periods of time because no external events occur. If you need to allow time for a background job to execute, you should modify the foreground program so that this is possible. You can do this by including calls to system programmed requests (for example, I/O and timer requests) in the foreground program. RT–11 then blocks the program until the requested operations are complete. Programmed requests are discussed in volume 2.

Initiating Jobs

When you issue a command to KMON to run a user program in background, KMON is suspended until that program terminates. This means that during execution of a background job, you are not able to issue KMON commands, for example, to initiate a foreground job.

Thus, when you want to execute a foreground and background job at the same time, you must run the foreground job first. You do this by using the FRUN command.

EXAMPLE

```
.FRUN PROGF
.RUN PROGB
```

Special Considerations for Foreground Jobs

A number of conditions must be met when loading and running foreground jobs.

1. You must use the /FOREGROUND option to link the foreground job:

   ```
   .LINK/FOREGROUND PROG
   ```

2. You may need to alter the size of the foreground program stack. This is done by using the /FOREGROUND option with an optional numeric argument. (See chapter 1, "Developing Programs in MACRO–11 and FORTRAN IV.") The following command links PROG as a foreground job with a stack size of 300 (octal) bytes:

```
.LINK/FOREGROUND:300 PROG
```

3. You must load required device handlers (programs to control devices). For example, if your program uses an RX50 diskette, you must load its handler using the command:

   ```
   .LOAD DU:
   ```

4. You may need to create more space in memory than was allocated initially. For example, FORTRAN IV programs, running in the foreground under the FB monitor, need additional space for blocks of data that are created when files are opened. To reserve more space, you can use the /BUFFER option of the FRUN command. For example, the following command reserves 500 extra words of memory for the program PROG.REL:

   ```
   .FRUN PROG/BUFFER:500
   ```

   A formula provided in the section on the FRUN command in chapter 4 of the *RT–11 System User's Guide* helps determine the space needed to run a FORTRAN IV program as a foreground job. You need not reserve more space for a program running under the XM monitor because such programs can use more space as needed. Chapter 19, "Memory Use," has further discussion of how a program can reserve more space for itself at run time.

5. When you run a foreground or system job, any inactive jobs are removed from memory. If a foreground program has terminated, however, and you want to run a background job, then you must remove the inactive job yourself. You do this by using the command:

   ```
   .UNLOAD PROG
   ```

Foreground/Background Communication

You can run a foreground and a background program so that each communicates with the console terminal. Messages generated by jobs are indicated by the following prompts:

F> for foreground

B> for background

To type data to a foreground program, press ⟨CTRL/F⟩ followed by the data. For a background program, press ⟨CTRL/B⟩ followed by the data.

Practice 2–1 *Running a MACRO–11 Program in the Foreground and a FORTRAN IV Program in the Background*

1. Type in the MACRO–11 program PR0201.MAC listed below:

```
        .MCALL  .TWAIT,.GTLIN,.EXIT ;Declare macro calls
        .MCALL  .PRINT
        .ENABL  LC
START:  .GTLIN  #BUFF,#PROMPT   ;Input line with prompt
        TSTB    BUFF            ;Check for null line
        BEQ     1$              ;If null then exit
        JSR     PC,TWT          ;If not then perform a wait
        .PRINT  #NOTIFY         ;Print leader
        .PRINT  #BUFF           ;Print the buffer
        BR      START           ;Repeat process
1$:     .PRINT  #EXMES          ;Print exit message
        .EXIT                   ;Program exit
TWT:    .TWAIT  #AREA,#TIME     ;Perform first wait using
                                ;EMT

;       Compute bound wait

        CLR     R0              ;Init R0
        MOV     #2,R1           ;Init R1=2
5$:     DEC     R1
10$:    DEC     R0
        TST     R0              ;Has R0 reached 0 yet?
        BNE     10$             ;If not go back and decr.
        TST     R1              ;Has R1 reached 0 yet?
        BNE     5$              ;If not go back and decr.
        RTS     PC              ;Iteration complete so return

AREA:   .WORD   0,0
TIME:   .WORD   0,600.
BUFF:   .BLKW   41.
PROMPT: .ASCII  /PR0201-I,TEXT: /<200>
NOTIFY: .ASCII  /PR0201-I,Finished processing text: /<200>
```

```
EXMES:  .ASCIZ  /PR0201-I,Normal successful completion/
        .EVEN
        .END    START
```

2. Now type in the FORTRAN IV program PR0202.FOR listed below:

```
C INPUT A LOAD OF DATA
C
100   TYPE 6000
      READ (5,*)REALNO
      IF (REALNO.EQ.-1.0) GOTO 9999
      TYPE 6001
      TYPE *,REALNO
      GOTO 100
6000  FORMAT(1H0,'PR0202-I,Enter your data (-1 to finish): '$)
6001  FORMAT(1H0,'PR0202-I,Accepted data as: '$)
9999  END
```

3. Assemble PR0201.MAC by typing the command:

   ```
   MACRO PR0201
   ```

   Link PR0201 to run in the foreground (PR0201.REL).

4. Compile PR0202.FOR by typing the command:

   ```
   FORTRAN PR0202
   ```

   Link PR0202 to run in the background (PR0202.SAV).

5. Type in the command:

   ```
   SET USR NOSWAP
   ```

   An error will occur if you do not issue this command.

6. Run the program PR0201.REL in the foreground. It gives you the prompt:

   ```
   PR0201-I,TEXT:
   ```

   Press ⟨CTRL/F⟩ to communicate with this program and reply to the prompt by typing in a text string. PR0201 then waits, allowing you to perform background operations.

7. Using ⟨CTRL/B⟩ to communicate with KMON, run the pro-

gram PR0202.SAV in the background. PR0202 gives you the prompt:

```
PR0202-I, Enter your data (-1 to finish):
```

Enter a number (56, for instance). Each time you enter a number, PR0202 will accept and acknowledge the data with a response. For example:

```
PR0202-I-Data accepted as: 56.00000
```

8. After some time, the foreground job finishes waiting and tells you:

```
PR0201-I-Finished processing text: TEXT STRING
```

"TEXT STRING" is the text string you entered. The foreground job then prompts you with:

```
PR0201-I, TEXT:
```

At this point you can no longer enter data in either the background or the foreground without pressing ⟨CTRL/B⟩ or ⟨CTRL/F⟩.

9. Continue to enter strings in the foreground and data in the background for as long as you like. The foreground job terminates when you enter a null string. The background job terminates when you enter −1.

10. Unload the foreground job after its termination.

Terminating Jobs

Jobs running under the FB monitor terminate in the same way as jobs running under the SJ monitor, but to abort jobs under FB, use the following key sequences:

⟨CTRL/B⟩⟨CTRL/C⟩⟨CTRL/C⟩	for background jobs
⟨CTRL/F⟩⟨CTRL/C⟩⟨CTRL/C⟩	for foreground jobs

Using the Extended Memory Monitor

The commands used when running jobs under the Extended Memory monitor are the same as those used under the FB monitor, but job execution under the XM monitor can use more memory.

Under the SJ and FB monitors, system and user jobs share the 32 Kwords of addressable memory. By using special system services which manipulate windows into memory, each job under the XM monitor may use its own 32 Kwords of memory or expand its usable address space to 128 Kwords. In this way, the total amount of memory available becomes 32 to 128 Kwords for UNIBUS processors. RT–11 version 5 supports Q-bus processors with 22-bit addressing such as the PDP–11/23–PLUS, so that a maximum of 2048 Kwords is available for use.

Executing with System Jobs

Through the system generation process you can create an FB or XM monitor capable of simultaneously running up to six system jobs plus a foreground and a background job. This feature was built into the RT–11 specifically to support system programs supplied by Digital.

Digital now supplies two of these system jobs: an error logger (ERRLOG) and a device queue program (QUEUE). Digital does not encourage you to write your own system jobs; the four remaining system job slots are reserved for future use.

Scheduling

A monitor that supports system jobs provides the same type of scheduler that ordinary FB and XM systems use. The monitor services jobs according to their priority: the background job always has the lowest priority (0); the foreground job always has the highest priority (7). You cannot change these assignments. At any given time, the job that runs is the highest priority job that is not blocked.

You assign a priority to a system job by using the SRUN command in the format:

```
SRUN PROGRAM
```

This causes the monitor to assign to that job the highest unassigned priority. In order to give the job a specific priority, you use the /LEVEL:priority qualifier.

EXAMPLE

```
.SRUN PROG/LEVEL:3
```

You can assign priority values from 1 to 6. You cannot assign a priority to a job if another system job is running at the same priority. For example, if you run QUEUE as a system job, you should assign it the lowest priority so that more important jobs, such as the error logger, will not be blocked. You can assign a priority only when you start a system job with the SRUN command. The priority levels do not change dynamically; that is, you cannot change the priority of a job while it is running.

Starting System Jobs

Use the SRUN command to start system jobs. You reference a system job by its logical name, which is, by default, its file name. You may, however, assign a new logical name when you start the job, using the SRUN command with the /NAME:logical-name option. This is of specific benefit when you want to run multiple copies of a system or foreground job. The following commands show how you can run two system jobs, a foreground job, and a background job:

EXAMPLE

```
.SRUN SYS1/LEVEL:5/NAME:TEST
```

runs the system job SYS1.REL at priority 5 with the logical name TEST.

```
.SRUN SYS2/LEVEL:6
```

runs the system job SYS2.REL at priority 6.

```
.FRUN FROG
```

starts execution of foreground job FROG.REL (foreground job always executes at highest priority, 7).

```
.RUN PROG
```

starts execution of background job PROG.SAV (background job always executes at lowest priority, 0).

Communication

In a system job environment, you use ⟨CTRL/X⟩ to communicate with a system job in much the same way that you use ⟨CTRL/F⟩ for a foreground job and ⟨CTRL/B⟩ for a background job. This facility allows two or more jobs to share one terminal. You can communicate with system jobs in the following ways:

1. The system answers ⟨CTRL/X⟩ with the prompt:

   ```
   Job?
   ```

   Respond to the prompt by typing the job's logical name, followed by ⟨RETURN⟩. For example:

   ```
   ⟨CTRL/X⟩
   Job? SYS1
   ```

   If the job you specify is not running or cannot be found, the monitor prints a question mark immediately after the name of the job:

   ```
   ⟨CTRL/X⟩
   Job? SYS1?
   .
   ```

2. To abort ⟨CTRL/X⟩ before you have completed typing the job name, press ⟨CTRL/C⟩. This does not abort any job; it

only returns to the state at which the terminal was before you pressed ⟨CTRL/X⟩, for example:

```
.SRUN J1
Welcome to J1, please enter your data:
.⟨CTRL/X⟩
Job? ⟨CTRL/C⟩
.
```

3. To actually abort a system job, press ⟨CTRL/X⟩, then type in the job name, press ⟨RETURN⟩, and then press ⟨CTRL/C⟩ twice:

```
.⟨CTRL/X⟩
Job? SYS1⟨RETURN⟩
⟨CTRL/C⟩⟨CTRL/C⟩
.
```

While terminal input is routed to one system job, another may send data to the terminal. Thus, the monitor prints out an identification label every time the output source changes.

Executing Programs on Systems with Multiple Terminals

If your system supports multiple terminals, you can execute different jobs on different terminals. (The system generation option allows RT–11 to support multiple terminals.) To direct a foreground or system job to a specific terminal, use the /TERMINAL:n option with the FRUN or SRUN command. In this case, "n" is the logical unit number of the terminal.

EXAMPLE

The command:

```
.FRUN PROG/TERMINAL:2
```

executes the program PROC.REL in the foreground, using the terminal, whose logical unit number is 2, as its "private" console. Input and output to and from PROG.REL is then performed at that terminal.

The /TERMINAL option is discussed in the sections on FRUN and SRUN in chapter 4 of the *RT–11 System User's Guide.*

Executing MACRO–11 and FORTRAN IV Source Files

At times you will keep only the source files of error-free programs in order to save space on a storage device. When you want to execute these programs, you can do so with the single command EXECUTE. This command assembles/compiles, links, and then runs your program in the background. It takes the form:

```
EXECUTE FILESPEC
```

You specify the language processor you wish to use by one of two methods: either by giving the file type, for example .FOR, or, if the file type does not specify the language properly, by using the language option, for example EXECUTE/FORTRAN.

If the file name uniquely specifies your program (that is, there is only one type, whether in MACRO–11, FORTRAN IV, or the DIBOL language) you can omit the file type. EXECUTE searches for files of type .MAC, then .DBL, then .FOR.

If a fatal error occurs during assembly/compilation or linking, EXECUTE does not attempt to continue past that phase, but exists in the normal way. Many of the options available with the assembly and compilation commands and with LINK and RUN are also available as options to EXECUTE. However, if you want to use these options it is probably better to perform each step separately. The EXECUTE command does not work properly if the assembler or compiler and the linker are not on the system device.

Debugging

You can often isolate program problems by examining selected memory locations before, during, and after program execution. Under the RT–11 operating system this debugging method is supported by a number of software tools. In addition, tools are available that allow the more precise control of program execution that is necessary during debugging. This phase of program development is discussed in detail in chapter 4, "Debugging Programs."

Reference

RT–11 System User's Guide. Chapter 4 contains detailed explanation of the RUN command, the options of the FRUN command, and the facilities available from the EXECUTE command.

3

3 Developing Programs in BASIC

RT–11 allows you to create, edit, run, load, and save a BASIC program without exiting from the interpreter.

In addition to these program development operations, you can type some commands directly to the interpreter for immediate execution. Commands discussed in this chapter include the monitor command, BASIC, and the BASIC language commands, BYE, COMPILE, DEL, LIST, NEW, OLD, REPLACE, RESEQ, RUN, SAVE, and SUB.

In this chapter, you will learn to create or modify a BASIC program using the BASIC interpreter, run a BASIC program and save the standard or preprocessed form of a BASIC program in a file, execute BASIC statements in immediate mode, and exit from the interpreter.

Entering the BASIC Environment

You can use BASIC with any of the RT–11 monitors (SJ, FB, or XM). When using either the FB or XM monitors, you can run BASIC as either a foreground or background job. To load and start the BASIC interpreter, simply type the command BASIC. If there is not enough memory available to contain BASIC, an error message is displayed.

EXAMPLE

```
NOT ENOUGH MEMORY FOR BASIC
```

or

```
?KMON-F-Not enough memory
```

This situation often results when a large foreground job has been loaded.

BASIC is made up of a set of fixed language elements and a set of optional elements. When you enter the BASIC environment, you select which optional elements you wish to use.

EXAMPLE

```
.BASIC<RETURN>
```

BASIC asks you:

```
OPTIONAL FUNCTIONS (ALL, NONE, OR INDIVIDUAL)?
```

You type:

```
ALL<RETURN>
```

The response ALL allows you to use all of the optional functions available with the RT–11 BASIC interpreter. If you type NONE, the interpreter performs without any optional function. You may choose the NONE option when you want a program to contain only standard BASIC functions. To select the optional functions you want to use, you type INDIVIDUAL. After this response, the interpreter displays each function and requests a YES or NO reply. YES includes the function; NO excludes it.

The interpreter tells you that it is ready to accept BASIC commands and program lines by issuing the message:

```
READY
```

At this point, you can create new programs, retrieve old ones, edit and insert new material, save, run, or delete programs by issuing the appropriate BASIC commands to the interpreter. After typing each complete command, press ⟨RETURN⟩.

Creating a Program

Program lines, the program name, and any variables and their values are stored by the interpreter in its own memory. When you create a new program, you first initialize that memory and give the new program a name. You do this using the NEW command, which takes the form:

NEW PROGRAM-NAME

The program name may contain no more than six alphanumeric characters. Examples of valid program names include: 024680, STAR, or PROG01. Examples of invalid program names are: MAINPROG (too long) or PROG/1 (contains a nonalphanumeric character).

If you type the command NEW without giving a program name, BASIC asks you for the program name with the prompt:

```
NEW FILE NAME--
```

You should then supply the program name. If you press ⟨RETURN⟩ without giving a program name, the program assumes the default name NONAME.

Entering New Lines of Basic Program Code

Each line of BASIC program code begin with a line number. Line numbers must be in the range 1 to 32767. To insert a line of program code into a program, you type the line number, the program code, and press ⟨RETURN⟩.

EXAMPLE

```
10 PRINT "HELLO"⟨RETURN⟩
```

becomes line 10 of the program.

Retrieving a Saved Program

To avoid retyping, you can retrieve existing programs and copy them into the interpreter's memory with the OLD command, which takes the form:

OLD FILESPEC

The two types of saved programs which can be retrieved are normal BASIC programs and preprocessed BASIC programs. Normal BASIC programs are usually stored in a file with the file type .BAS. Preprocessed BASIC programs are usually stored in a file with the file type .BAC (or .BAX if you are using double-precision BASIC).

The default file type for the OLD command is .BAC. That is, if no file type is specified with a program name, the interpreter will search first for a program file with the file type .BAC.

EXAMPLE

If you have a program named PROG stored in the file PROG.BAS and you type:

```
OLD PROG
```

the interpreter will search first for a program stored in the file DK:PROG.BAC. If no .BAC file exists on DK:, the interpreter searches for and loads your file DK:PROG.BAS. If the file you specified does not exist, the interpreter displays the following message at the terminal:

```
?FILE NOT FOUND
```

When a program is retrieved, the interpreter initializes its memory as with the NEW command. It then loads each line from the file into its memory, using the file name as the name of the program.

A line of program code can contain no more than 129 characters, in addition to the characters that make up the line number. If a line is too long, the interpreter does not load the line but displays a message:

```
?LINE TOO LONG
```

If a line being loaded does not have a valid line number, the interpreter does not load the line but displays the message:

```
?SYNTAX ERROR
```

Executing a BASIC Program

After a program is loaded into the interpreter's memory, you can run it by typing the command RUN. The interpreter executes the program, starting with the lowest numbered line of code. You can also load a program from a file and execute it by issuing the RUN command in the format:

```
RUN FILENAME
```

EXAMPLE

```
RUN PROG(RETURN)
```

has the same effect as the commands:

```
OLD PROG
RUN
```

If no such program exists, the interpreter displays:

```
?FILE NOT FOUND
```

When a program is executed, the interpreter normally prints a header containing the program name and the system date and time. You can prevent this header from appearing by using the command RUNNH in place of RUN. To abort execution you press ⟨CTRL/C⟩ twice.

Editing a BASIC Program

When you have loaded an old program or have typed in a new program, you may wish to modify the program. The BASIC interpreter has several commands that allow you to edit a program in different ways.

Printing a Listing of a BASIC Program

To get a listing of the program at your terminal, use the LIST command. When you use the LIST command the interpreter prints a header for the program, followed by all the lines of code in ascending order according to line number. The header takes the same form as with the RUN command. Specifying LISTNH prevents this header from appearing. To list only a selection of lines from the program, specify a range with the LIST command.

EXAMPLE

```
LIST 300-400(RETURN)
```

This command causes all those lines between line numbers 300 and 400 to be listed at the terminal. You can also list more than one range of lines by using a comma as a separator.

EXAMPLE

```
LIST 200-250,500-550(RETURN)
```

This command causes the ranges of lines from 200 to 250 and 500 to 550 to be printed at the terminal. When you specify more than one range, the interpreter prints a blank line between ranges.

Inserting New Lines of Program Code

To insert a new line of code between two consecutive lines, type a line with a line number that falls between the line numbers of the consecutive lines.

EXAMPLE

To insert a line between these two lines:

```
100 PRINT "WELCOME TO THE PROGRAM"
200 INPUT A$
```

assign a line number between 100 and 200:

```
150 PRINT "WHAT IS YOUR INPUT";
```

Resequencing

You should increment your line numbers by at least five when writing a program (for example 120, 125, 130). This allows you to insert four new lines of program code if you need to. However, you may run out of space between two line numbers if you have to make many insertions. To remedy this situation, you can renumber lines of a program by using the RESEQ command in the format:

```
RESEQ start-line-no,range,increment
```

In this command format, "start-line-no" is the new lowest line number for the range; "range" is the range of lines that is to be renumbered; and "increment" is the designated increment between lines. The range of lines is renumbered in the form nnn–mmm; "nnn" is the lower limits and "mmm" is the upper limts of the resequencing.

If you omit any of these parameters, the interpreter uses default values which are as follows:

start-line-no:	10
range:	1 to 32767
increment:	10

EXAMPLE

If you have loaded the program:

```
10 PRINT "HELLO"
11 GOSUB 15000
12 PRINT G$
15 GOTO 32767
15000 G$="THIS IS A MESSAGE"
15001 RETURN
32767 END
```

and you type the command:

```
RESEQ(RETURN)
```

then the program in the interpreter's memory becomes:

```
10 PRINT "HELLO"
20 GOSUB 50
30 PRINT G$
40 GOTO 70
50 G$="THIS IS A MESSAGE"
60 RETURN
70 END
```

Deleting Lines of Program Code

In some cases you may want to remove a line of program code. To delete a line of code, use the command:

DEL line-no

EXAMPLE

In the program:

```
10 A$="HELLO"
20 B$="WELCOME"
30 PRINT A$
```

you can delete the second line by using the command:

```
DEL 20<RETURN>
```

Then the program in the interpreter's memory becomes:

```
10 A$="HELLO"
30 PRINT A$
```

You can also specify one or more ranges of lines to be deleted, separating the ranges with commas.

EXAMPLE

```
DEL 1-250,20000-32766
```

Changing Lines of Program Code

To change a line of code, you can use the interpreter in two ways: by retyping the line, or by making a substitution using the SUB command. When you want to change a line of code completely, simply type the new line with the same line number.

EXAMPLE

The line

```
10 PRINT "HELLO"
```

can be changed simply by typing:

```
10 GOSUB 15000
```

When you make a minor error and need to change only a few characters in the line, you may prefer to use the SUB command, which takes the form:

SUB line-no delimiter old-string delimiter new-string

In this command format:

line-no	is the line number of the line of code to be changed
delimiter	is any single character that appears neither in the old string nor in the new string
old-string	is the string whose first occurrence in the line of code is to be substituted for new string
new-string	is the new string that will appear in place of the old

EXAMPLE

When a program in the interpreter's memory contains the line:

```
100 PRINT "ENTER YOUR DATA (-1 TO FINNISH)";
```

The command:

```
SUB 100@NN@N
```

corrects the misspelled word and changes this line to:

```
100 PRINT "ENTER YOUR DATA (-1 TO FINISH)";
```

The SUB command may have an additional argument, a number. This number indicates that a certain occurrence of the old string should be replaced. For example, if the number is "5," the fifth occurrence of the old string would be replaced by the new string. The number goes at the very end of the command line, preceded by a delimeter. The default value is 1.

Saving a BASIC Program

When you have finished entering or editing the program in the interpreter's memory, you may want to save the program so that it can be retrieved and executed later. You can do this with one of three BASIC commands: SAVE, REPLACE, or COMPILE.

If you want to store the program in the interpreter's memory in a file that does not yet exist, use the SAVE command. When you issue the SAVE command by itself, the program will be saved as a file with the specification:

```
DK:PROGRAM-NAME.BAS
```

"PROGRAM-NAME" is the program name in the interpreter's memory.

You can choose the device name, file name, and file type of the same file by using the command format:

SAVE FILESPEC

For example, if your program's name is NONAME, the SAVE command would use the file specification DK:NONAME.BAS by default. To override the default file specification, you can supply a file specification with SAVE.

EXAMPLE

```
SAVE RK3:STAR.BAS
```

If you use the SAVE command and the file you specify already exists, then the interpreter prints the message:

```
?USE REPLACE
```

To delete an existing file and save a new file with the same file name, you use the command:

REPLACE FILESPEC

If the file you wish to replace is DK:PROGRAM-NAME.BAS, then you need only type REPLACE.

The SAVE and REPLACE commands copy the lines of a program from the interpreter's memory in the same format as for a listing. As an alternative, you can create a preprocessed file, which stores the lines of a program in a format that loads more quickly into the interpreter's memory. To save the program in the interpreter's memory in a preprocessed file use the command COMPILE.

The default file-specification for a preprocessed file is: DK:PROGRAM-NAME.BAC. To override this default, use the command:

COMPILE FILESPEC

If you are using double precision BASIC, then preprocessed files assume the file type .BAX.

Using Immediate Mode

If you want to execute BASIC language statements, without creating and running a program, you can do so by typing the statement without a line number.

EXAMPLE

The command:

```
PRINT "HELLO"
```

causes the interpreter to execute that command immediately.

This facility has a number of uses. You can use BASIC as a calculator by issuing the command:

PRINT arithmetic-expression

For instance, you can instruct the computer to multiply two values, divide by a third value, then print the result at the terminal.

EXAMPLE

```
PRINT 327*128/61
```

You can also use immediate mode in lieu of some monitor commands.

EXAMPLE

The BASIC command:

```
KILL "NONAME.BAS"
```

has the same effect as the monitor command:

```
.DELETE NONAME.BAS
```

Leaving the Interpreter

To return control from the BASIC interpreter to the RT–11 keyboard monitor use the command BYE. If you want to exit from a program to the monitor instead of the BASIC interpreter, you can use the BASIC language statement:

numeric-variable = SYS(4)

In the following example, the program prints "HELLO" at the terminal and then exits directly to the monitor.

EXAMPLE

```
10 PRINT "HELLO"
20 A=SYS(4)
30 END
```

Practice 3–1

Enter the BASIC interpreter selecting all optional functions. Create the program PR0301.BAS and save it:

```
10 PRINT "WHAT IS YOUR GAME?";
20 INPUT #0,A$
30 A=SYS(4)
```

Retrieve the program PR0301.BAS and run it. It will print the message:

```
WHAT IS YOUR GAME?
```

and accept input. Abort the program and modify it so that it gives the message:

```
WHAT IS YOUR NAME?
```

Save the program as a preprocessed file and leave the interpreter. Now reenter the interpreter as before and run PR0301 without using the OLD command.

Type your name and press ⟨RETURN⟩. The program should exit to the monitor.

Reference

BASIC–11/RT–11 User's Guide contains examples of commands discussed in this chapter.

4

4 Debugging Programs

Program errors (bugs) can be difficult to find. Although there are different methods for identifying these errors, all methods include checking program code and data at different points—either before, during, or after execution. RT–11 provides tools to help identify errors and make corrections in MACRO–11, FORTRAN IV, and BASIC–11 programs. These include, ODT (On-line Debugging Technique) and VDT (Virtual Debugging Technique).

This chapter discusses the testing of MACRO–11, FORTRAN IV, and BASIC programs to find errors. It also covers the use of utility programs, together with certain monitor commands and BASIC commands for debugging your programs. The monitor commands discussed in this chapter are: D, E, FRUN/PAUSE, GET, RESUME, and START.

You will learn how to stop a BASIC program after the execution of different statements and then check data or use ODT and VDT to check data at selected points during the execution of a MACRO–11 program. You will also learn to use GET, START, EXAMINE, and DEPOSIT to check data at selected points during the execution of FORTRAN IV and MACRO–11 programs.

Testing Programs

After you have removed all compilation and linking errors, you are ready to test your program. It is unlikely that you will detect all errors immediately. To be sure that a program has as few errors as possible, you must test it thoroughly. Here are some methods which you can use to do this:

1. For a given set of test data, determine what action the program should take.
2. Run the program with the test data and verify that the program performs as expected. Change this data again and again, and rerun the program so that every conditional branch that depends on this data is executed.
3. Examine any data output to the terminal and use the DUMP utility to check the contents of any output files.
4. Include printing statements to trace the path of execution through the code and to check values of data at key points. (FORTRAN IV programmers can use the TYPE statement in a debugging line.)

If your program is designed in modules, you can test each module in isolation. First, test the main module with dummy modules for each subroutine referenced; then include and test each subroutine referenced in turn until all of the program is tested. This is called top-down testing.

Your dummy modules should be written so that they accept and return only the arguments with which they are called, and so they identify themselves. Assume that a subroutine is designed to accept a single character from the terminal without echo. Figure 9 shows dummy versions of such a routine in FORTRAN IV and MACRO–11. In this case, the subroutine GETCHA has two arguments—the input channel and the byte value of the character. The dummy routine sets the value of the input character to 64, which is the ASCII code for a capital "A." The real routine would

Figure 9.
FORTRAN IV and MACRO–11 Dummy Subroutines

```
C ******************************
C
C Accept single character input
C
C DUMMY VERSION (FORTRAN IV)
C
C ******************************
        SUBROUTINE GETCHA(CHANNL,CHRCTR)
        INTEGER CHANNL,CHRCTR
        TYPE 8000
8000    FORMAT(' %DUMMY: GETCHA')
        CHRCTR=64
        RETURN
        END
; ******************************
;
; Accept Single Character Input
;
; DUMMY VERSION (MACRO-11)
;
; ******************************
        .TITLE  GETCHA
        .MCALL  .PRINT
GETCHA::                        ;Entry point
        .PRINT  #TEST           ;Print id message
        MOV     4(R5),R1        ;Address of return param.
        MOVB    #'A,(R1)        ;Put "A" at that address
        RTS     PC              ;Return from subroutine
TEST:   .ASCIZ  /%DUMMY: GETCHA/
        .END
```

read a character from the channel specified. Using the dummy subroutine, you can identify and correct errors in the main program before testing the real subroutine GETCHA. You can also write dummy subroutines to replace any system subroutines that are referenced.

Finding the Cause of an Error

Your test program should be made up of modules that have been tested and corrected, the module you want to test, and dummy modules for those that have not yet been tested. System subroutines should be treated as modules also. Each time you test a module, one of the following situations results:

1. There are no errors in the module.
2. The module does not produce the correct data.
3. The program fails and an error message is printed at the terminal.
4. The program fails to continue executing at some point, but no error message is printed.

When there are no errors found in the module, test another module by selecting a dummy routine, replacing it with the real one, and executing the program again. If a module's intermediate or final data does not have the expected values, first check the code to see why it produces incorrect data. At which line do data first go wrong? If a line contains a wrong calculation, correct the line. If some of the data is not structured correctly, restructure the data.

If an error message appears at the terminal, you may refer to the *RT–11 System Message Manual* for further information. If the program fails, with or without a message, then find out at which line the error occurred. If you still cannot find the cause of an error in a MACRO–11 or FORTRAN IV program, you should use the debugging aids discussed later. These aids help you make a detailed examination of the code and data of a program, but you must first understand how your code behaves before you proceed to debugging.

Locating an Error

You may be able to identify the line at which an error occurred by looking at the source code. For example, if you know what type of error occurred, then you can determine which lines of code may contain the error.

If this fails, an effective way of finding an error is by inserting PRINT statements at checkpoints in your code. The checkpoints could be before and after conditional branches, inside loops, and in other critical places in the program. When the program runs, the printed text will provide a trace of the program's execution, indicating the path taken before the error.

Gaining Access to Background Program Code

The RT–11 system supplies monitor commands for examining and modifying program machine code before and after execution. These methods are especially helpful if you are programming in MACRO–11.

Loading Programs without Execution

The first step in the process of examining the machine code of a program is to load the program into memory without executing it. You do this with the GET command, which takes the form:

```
GET FILENAME
```

"FILENAME" is the file in which the load image is stored.

Locating Values in a Loaded Program

The base address of each module is shown in the load map, which can be produced using the LINK/MAP command. The offset of a symbol in a module is shown in the assembler listing. Thus, the absolute address of a symbol is:

```
module-base-address + symbol-offset
```

"module-base-address" is the base address of the module and "symbol-offset" is the offset of the symbol, from the start of that module.

Figure 10.
Load Map for a Main Program Using Subroutine GETCHA

```
RT-11 LINK  V08.00      Load Map         Page 1
MAIN   .SAV       Title:  MAIN     Ident:
Section  Addr   Size     Global   Value    Global  Value
. ABS.   000000 001000 = 256.     words  (RW,I,GBL,ABS,OVR)
         001000 000140 = 48.      words  (RW,I,LCL,REL,CON)
                         GETCHA   001100
Transfer addr = 001000, High limit = 001136 = 303.  words
```

Figure 10 shows the load map for a program comprising a main routine and a subroutine. The base address of the module is 1100 (octal). Figure 11 shows a listing of the subroutine GETCHA. You can see that the offset of the symbol TEST is 000020 (octal), so the absolute address of TEST is 1120.

Examining Locations

When the program exits, or is aborted, you can examine the contents of an address by using the E (Examine) command.

EXAMPLE

To examine the contents of location 1000 (octal) you type:

```
.E 1000
```

and the monitor prints the value, in octal, stored at that address.

Modifying Loaded Programs

Having used the GET command to load a program into memory, you can modify the program code. Refer first to a loap map (produced by the linker) to check where values are stored, then use the E command to verify that you have the right address before changing the value. Use the D (Deposit) command to modify a value. The format is:

D address = value

Figure 11. Listing of Dummy Subroutine GETCHA (Produced with the Command MACRO/SHOW:MEB/LIST)

```
GETCHA  MACRO V05.00dd  05:19  Page 1
     1                                      ;******************************
     2                                      ;
     3                                      ;ACCEPT SINGLE CHARACTER INPUT
     4                                      ;
     5                                      ;DUMMY VERSION (MACRO)
     6                                      ;
     7                                      ;******************************
     8                                              .TITLE  GETCHA
     9                                              .MCALL  .PRINT
    10                                              .GLOBL  GETCHA
    11 000000                               GETCHA: .PRINT  #TEST
       000000  012700  000020'                      MOV     #TEST,%0
       000004  104351                               EMT     ^O351
    12 000006  016501  000004                       MOV     4(R5),R1
    13 000012  112711  000101                       MOVB    #'A,(R1)
    14 000016  000207                               RTS     PC
    15 000020     045     104     125   TEST:       .ASCIZ  /%DUMMY:
GETCHA/
       000023     115     115     131
       000026     072     040     107
       000031     105     124     103
       000034     110     101     000
    16         000001                               .END
GETCHA  MACRO V05.00d  05:19  Page 1-1
Symbol table
GETCHA  000000RG        TEST    000020R
. ABS.  000000     000  (RW,I,GBL,ABS,OVR)
        000037     001  (RW,I,LCL,REL,CON)
Errors detected:   0
*** Assembler statistics
Work  file  reads: 159
Work  file writes: 42
Size of work file: 154 Words   ( 1 Pages)
Size of core pool: 3328 Words   (13 Pages)
Operating  system: RT-11
Elapsed time: 00:00:07.00
DK:GETCHA,DK:GETCHA/L:MEB=DK:DUM
```

"address" is the absolute address in octal of the location to be modified, and "value" is the new value in octal that the address is to hold.

Executing the Code in Memory

After modifying the code, you can execute it by using the START command. The program then begins execution at the program's transfer address as shown in the map. If you want, you can specify a different start address.

EXAMPLE

```
.START 1400
```

Gaining Access to Foreground Program Code

To load a foreground program into memory without executing it, use the FRUN/PAUSE command. This has the same effect as the GET command for background programs, except that the base address of the program is printed out. To start execution of a paused foreground program, use the RESUME command. You cannot change the start address of a foreground program once it has been loaded.

System On-line Debugging Aids

Another way of detecting program errors is to use an on-line debugging aid. RT–11 supplies two similar on-line debugging aids, ODT and VDT, to help you debug MACRO–11 programs. ODT is for single-terminal systems; VDT for multiterminal systems.

Many high-level languages have their own debugging aids. For example, FORTRAN IV programs can be debugged using FDT. High-level language debugging aids are not discussed in this course.

Enabling On-line Debugging

If your main program references subroutines that are assembled in separate object modules, always make sure that the subroutine names are declared as globals (by using the .GLOBAL directive) before continuing with the debugging procedures.

Enabling debugging aids for background programs

To enable a debugging aid to be used with your background program, you must first get an assembler listing of your program, including all the addresses of the symbols used, and the binary expansions of the instructions and data. You then link your program modules with the debugging module DK:ODT.OBJ, using the LINK/DEBUG command:

```
LINK/DEBUG OBJECT-MODULES
```

"OBJECT-MODULES" is the list of object modules that you would normally use to produce your save image.

If the debugging module you want is not DK:ODT.OBJ, qualify the /DEBUG option with the file specification of the debugging module you want.

EXAMPLE

```
.LINK/DEBUG:SY:VDT.OBJ MAIN,SUBA,SUBB
```

Here the debugging module SY:VDT.OBJ has been specified.

You need a load map when you are linking with debug, so that you can see which modules are included and at what addresses in the load image. This information is essential during debugging.

EXAMPLE

Assume that you want to use VDT, which is contained in the file SY:VDT.OBJ. Assume also that the save image you want to produce from the object modules DK:PROG.OBJ, DK:SUBA.OBJ, and DK:SUBB.OBJ is the program DK:PROG.SAV. To produce a debugging version of PROG.SAV, you would type:

```
.LINK/DEBUG:SY:VDT/MAP:PROG.MAP PROG,SUBA,SUBB
```

When you run the load image that you have created, control is initially passed to the debugging aid.

Enabling debugging aids for foreground programs

To enable a debugging aid to be used in your foreground program, you first get an assembler listing that includes all the addresses of the symbols used and the binary expansions of the instructions and data. You then get a relocatable image and load map for your program by issuing the command:

```
LINK/FORE/MAP:MAP-FILESPEC OBJECT-MODULES
```

"MAP-FILESPEC" is the file that is to contain the load map, and "OBJECT-MODULES" is the list of object modules that will be used to produce your relocatable image file. You can then get a load image file from the debugging module.

EXAMPLE

To obtain a load image file from SY:ODT.OBJ, type:

```
.LINK SY:ODT
```

You can then run your program in foreground using the command:

```
FRUN/PAUSE FILESPEC
```

Make a record of the base address of your program and press ⟨CTRL/B⟩ to direct terminal input back to the monitor. Using the command ODT, you can run the debugging aid load image in the background.

Using ODT and VDT

ODT and VDT issue an asterisk (*) prompt and receive commands from the terminal. ODT always receives input from the system console terminal because it is designed for single-terminal systems. VDT receives input from the terminal to which the console is set. On systems that have been generated with multiterminal support, you must use VDT, even on the console terminal TT0:.

ODT and VDT read characters as they are typed. You do not need to terminate a command with ⟨RETURN⟩, because ⟨RETURN⟩ has a special function in ODT and VDT. You cannot simply correct input; ⟨DELETE⟩ cancels a command and you must retype it.

Gaining access to addresses

We have discussed how the base addresses of the modules of your program are shown in the load map. The addresses of all your symbols are shown in the assembler listing. The absolute address of a symbol can be calculated as the value of the expression:

```
base-address + symbol-address
```

To avoid calculating this value each time you want to access a symbol, use a relocation register. There are eight such registers, numbered from 0 to 7. You load the base address of a module into a relocation register using the command:

```
*base-address;register-numberR
```

"base-address" is in octal and "register-number" must be in the range 0 to 7.

EXAMPLE

To load the base address 1024 (octal) into the relocation register 0R, type:

```
*1024;0R
```

If you are debugging a foreground program, you should load the base address of the program into a relocation register. You should have a record of this address from when you loaded the program using the FRUN/PAUSE command. You can redefine any of the relocation registers at any time. In addition, you can use the following commands:

*;nR	clears relocation register n
*;R	clears all relocation registers

The notation register, offset may be used anywhere in ODT or VDT instead of using an absolute address.

Gaining access to registers

The following commands allow you to examine the values stored in the different types of registers:

*$Rn	displays the value of relocation register n
*$Bn	displays the value of breakpoint register n
*$n	displays the value of the program's general register Rn

Setting a breakpoint

If the program you are debugging crashes, link it with ODT and execute it again, allowing it to crash. You can then study the values stored at different locations at the time of the crash. If you want to execute only part of a program before examining locations, select a point at which the program must stop and return control to ODT. Such a point is called a breakpoint. Good places to put a breakpoint include:

- Subroutine calls
- First instruction within a subroutine
- Branches and jumps
- Locations to which branching and jumping is carried out
- The first instruction in a sequence of suspect code

ODT has eight breakpoint registers, numbered 0 to 7. This means that you can have as many as eight breakpoints at one time. You set a breakpoint using the command:

```
*address;register-numberB
```

It does not matter which breakpoint register is used for which breakpoint address. You can use any breakpoint register that has not been used or one that contains a breakpoint that you no longer need.

EXAMPLE

To set a breakpoint at location 1666, select a breakpoint register not yet used or one that contains a breakpoint that is no longer needed. If this applies to breakpoint register 3, then you would type:

```
*1666;3B
```

When the program is about to execute at location 1666, control is returned to ODT.

When control of the program is returned to ODT, you may examine the registers and data before allowing the program to continue executing.

You clear breakpoint registers in the same way that you clear relocation registers:

;nB	clears breakpoint register n
;B	clears all breakpoint registers

Starting execution

When your breakpoints are set, you can start execution with the command:

```
*address;G
```

If no breakpoints are set, execution continues until the program exits in the usual way or aborts. If you have set breakpoints, the program will execute until it reaches a breakpoint and then ODT prints the message:

```
Bn;address
*
```

In this message "n" is the number of the breakpoint register that caused the break, and "address" is the address at which execution stopped. At this point you can examine and modify values stored at addresses.

Examining and modifying locations

With breakpoints set to permit partial program execution, you should examine data before and after execution. By modifying data before execution, you can test the effect of that part of the program more thoroughly. To modify a value or merely examine it, first open its location, by using the / (Slash) command, which takes the form:

```
*address/
```

EXAMPLE

To open location 1026 for examination and modification, type:

```
*1026/
```

If you have loaded the base address of a module into a relocation register, you can also open an offset within the module by using the command:

```
*register,offset/
```

ODT accepts characters immediately, so it recognizes the / command without waiting for a ⟨RETURN⟩.

EXAMPLE

If you type the command to open the location at an offset of 2 from the relocation register 1, and the value there is 20 (octal), the characters that appear at the terminal are:

```
*1,2/ 000020 _
```

Note that the print head or cursor, (indicated by the underline character "_") stays on the same line. If you press ⟨RETURN⟩, the value stored at the address is not changed. To modify the value, type in its octal value and press ⟨RETURN⟩.

```
*0,12/000020 40⟨RETURN⟩
```

The / command causes ODT to access the word starting at the given location. If you want to access a byte, you use \ (Backslash) instead. You cannot modify a location without opening it first. A location is opened when you use the / or \ command. It is also opened when you perform ASCII and RADIX–50 input and output. These modes are discussed in chapter 18, "On-line Debugging Technique (ODT)," of the *RT–11 System Utilities Guide.*

Proceeding from a breakpoint

After you have investigated the conditions at a breakpoint, you can continue execution using the P (Proceed) command, which takes the form:

```
*;P
```

Execution then continues to the next breakpoint.

When you set a breakpoint in a loop, you can allow the program to execute the loop a specified number of times by setting a proceed count using the n;P command. The count n is the number of times that ODT can reach the current breakpoint before it suspends the program. It will suspend if it meets any other breakpoint before the loop count is exhausted.

Using the single-step mode

To perform a detailed examination of part of a program, you can use ODT's single-step mode instead of setting a number of breakpoints close together. This allows you to execute single instructions or a specific number of instructions. To enter single-step mode, you give the command:

```
*;1s
```

You can then execute a number of instructions by using the command:

```
*n;P
```

Here "n" is the number of instructions to be executed.

EXAMPLE

The command:

```
*6;P
```

executes the next six instructions.

To exit single-step mode, you give the command:

```
*;S
```

Exiting from ODT or VDT

To exit from ODT or VDT, press ⟨CTRL/C⟩ in response to the asterisk prompt. Control is then returned to the keyboard monitor.

Using VDT to Debug

Figure 12 shows a terminal session using VDT. The commands used are the same as for ODT. The comments describe the action occurring. The operator first loads relocation registers with the base address of the modules to be examined, then sets breakpoints before starting execution. When the first breakpoint is reached, the operator switches to single-step mode and steps until the message is printed. At the end of the session a location is opened, and the character A is changed to B. You will see that VDT prints single-step messages as if they resulted from a breakpoint at breakpoint register 8. You can set breakpoint registers only from 0 to 7.

Debugging BASIC Programs

The procedure for testing BASIC programs is almost the same as the procedure for testing MACRO–11 and FORTRAN IV programs. In BASIC, you remove all syntax errors detected by the interpreter. You may produce a preprocessed (.BAC) version of the program before testing it. Other than this, the test procedures listed in the section "Testing Programs" at the beginning of this chapter apply the same to BASIC as to FORTRAN IV and MACRO–11.

Dummy Routines

Good BASIC programs are written in a modular way so that each module in the program design is coded as a subrou-

Figure 12.
Using ODT/VDT

```
.MACRO/LIST:MAIN/SHOW:MEB MAIN,GETCHA
.LINK/DEBUG:SY:VDT MAIN,GETCHA/MAP:MAIN.MAP
.RUN MAIN
 VDT V05.01
*1000;0R               (Set relocation register 0 to
                       base address of module MAIN)
*1100;1R               (Set relocation register 1 to
                       base address of module GETCHA)
*0,30;0B               (Set breakpoint register 0 to
                       instruction  in  MAIN  that
                       calls GETCHA)
*1,0;1B                (Set breakpoint register 1 to
                       first instruction in module
                       GETCHA)
*0,0;G                 (Execute from start of MAIN)
B0;0,000030            (Message at first breakpoint)
*;P                    (Proceed to next breakpoint)
B1;1,000000            (Message at second breakpoint)
*;1S                   (Enter single step mode)
*;P                    (Execute single step)
B8;1,000004            (Single step message)
*;P                    (Execute single step)
%DUMMY: GETCHA         (Output resulting from
                       execution of GETCHA)
B8;1,000006            (Single step message)
*;S                    (Cancel single step mode)
*0,34;0B               (Set breakpoint to .PRINT
                       request in module MAIN)
*;P                    (Proceed to breakpoint)
B0;0,000034            (Breakpoint message)
*0,75/101 =A 102<RET>  (Examine value of CHAR. It is
                       ASCII 101  . Modify it to
                       ASCII 102.)
*/102 =B               (Verify the modified location)
*;P                    (Proceed--there are no more
                       breakpoints)
The character is: B    (Output resulting from
                       execution of MAIN)
```

tine. When performing top-down testing on a BASIC program, you test the main program logic by writing dummy subroutines to replace all the subroutines referenced.

EXAMPLE

For a module designed to display a file containing a list of employees, a dummy version of the subroutine could be:

```
10000 REM SUBROUTINE TO DISPLAY EMPLOYEE FILE
10010 REM DUMMY VERSION
10020 PRINT "%DUMMY - DISPLAY EMPLOYEE FILE"
10099 RETURN
```

When a BASIC error occurs at run time, BASIC prints at the terminal a message that includes the line at which the error was detected.

Setting Breakpoints

To set breakpoints in a BASIC program, insert STOP statements. You can then use immediate mode PRINT commands to analyze the contents of any open files or the values of all the variables in use.

EXAMPLE

If your program contains the lines:

```
10 DIM #1,M0$(100)=10
20 OPEN "MASTER.DAT" AS FILE #1
100 A%=VAL(SEG$(M0$(0),1,5))
110 B%=VAL(SEG$(M0$(0),6,10))
```

and you want to examine data in the virtual array M0$, then you can insert the line:

```
95 STOP
```

so that the program would open the virtual array file, and stop with the message:

```
STOP AT LINE 95
```

When your program has stopped you can access the file interactively. You can use immediate mode statements to view parts of your program and modify the data.

EXAMPLE

You can display the first record in the virtual array file with the command:

```
PRINT M0$(0)
```

You can now modify this file data interactively, for example with the command:

```
M0$(0)="00123"+SEG$(M0$(0),6,10)
```

Practice 4–1

MACRO–11, FORTRAN IV, and BASIC versions of a program are included in this exercise. The MACRO–11 (PR0403.MAC, PR0404.MAC, and PR0405.MAC) and FORTRAN IV (PR0403.FOR, PR0404.FOR, and PR0405.FOR) versions are modular. Each has a main program and two subroutines. The BASIC program (PR0403.BAS) contains equivalent subroutines and a function from lines 10000, 11000, and 15000.

The program is designed to accept twelve monthly values (in the range 0 to 100) and plot them as a histogram. One subroutine is designed to take a value and return the number of units of height that represent that value in the histogram. On the histogram 20 units of height represent the value 100, and other heights represent values in the same proportion. The other subroutine is designed to convert a string into the real number it represents.

The programs contain up to two errors each and will not print the histogram properly. The errors are different in each language. Your task is to use the testing procedures we have discussed to locate and correct the errors so that the program accepts a value in the range 0 to 100 for each month in the year and displays a histogram on the screen. You must do the following:

1. Select the program in the language you know best and create the files exactly as listed.
2. Assemble and run the program to see what happens. The MACRO–11 and FORTRAN IV programs are made up of three object modules each.
3. Write dummy subroutines (or, for BASIC, a dummy function) to replace the original ones. If you are programming in BASIC–11, make a copy of the program, calling it HISTO.BAS, instead of editing the original program. You will need to refer to the original later.
4. Use printing statements to display data at key points during the program. A list of the location of such points is shown earlier in this chapter. In order to print a message from a MACRO–11 program use the macro:

   ```
   .PRINT #string-address
   ```

   where "string-address" is the address of an .ASCIZ string. Then use ODT/VDT to carefully debug the program.

Note: The MACRO–11 exercise requires that your PDP–11 processor have the extended instruction set. The exercise makes use of the DIV, MUL, and SOB instructions which are not available on all PDP–11 models.

PR0403.MAC

```
        .TITLE  PR0403  Debugging Exercise
        .MACRO  MONTH,NAME     ;Macro to set up month table
        .PSECT  MOVNAM
        .$$.=.                 ;Each entry points to string
        .ASCII  /NAME/<200>    ;This is the string
        .PSECT
        .WORD   .$$.           ;This is the space for entry
        .ENDM
        .MCALL  .PRINT,.EXIT,.GTLIN
        .GLOBL  CNVSTR,HISPRT  ;Declare subroutines
MTAB::  MONTH   JAN            ;Build months table
        MONTH   FEB
        MONTH   MAR
        MONTH   APR
        MONTH   MAY
        MONTH   JUN
        MONTH   JUL
        MONTH   AUG
        MONTH   SEP
        MONTH   OCT
        MONTH   NOV
        MONTH   DEC
START:  MOV     #MTAB,R2       ;Get address of months table
        .PRINT  #INTRO         ;Print introduction
        MOV     #12.,R3        ;Initialize month loop
        MOV     #HEIGHT,R4     ;Get address of heights table
LOOP:   .PRINT  (R2)+          ;Mth part of prompt for month
        .GTLIN  #INB,#PROMPT   ;Get decimal number string
        MOV     #INB,R5        ;Get address of input buffer
        JSR     PC,CNVSTR      ;Convert string to binary
        CMP     #-1.,R0        ;Check returned value for -1
        BEQ     BADVAL         ;If so branch past height calc
        DIV     #5,R0          ;Convert value to height
BADVAL: MOVB    R0,(R4)+       ;Place height in height table
        SOB     R3,LOOP        ;Branch for next month
        MOV     #HEIGHT,R5     ;Pass address of height table
        JSR     PC,HISPRT      ;Output the histogram
        .EXIT
INTRO:  .ASCII  /THIS PROGRAM PRINTS A HISTOGRAM FROM 12 /
        .ASCII  /MONTHLY VALUES./<15><12>
        .ASCII  /THE MONTHS ARE JANUARY TO DECEMBER./<15><12>
        .ASCII  /PLEASE ENTER YOUR TWELVE VALUES:/
        .ASCIZ  /THEY MUST BE IN THE RANGE 0 TO 100/
PROMPT: .ASCIZ  /: /<200>
VALUE:  .BLKB   12.*4.
HEIGHT: .BLKB   12.
INB:    .BLKB   81.
VAL:    .BYTE   -1.
        .EVEN
        .END    START
```

PR0404.MAC

```
                .TITLE  PR0404  Debugging Exercise
                .MCALL  .PRINT

CNVSTR::
        MOV     R1,-(SP)    ;Save caller's registers
        MOV     R2,-(SP)
        MOV     R3,-(SP)
        MOV     R4,-(SP)
        CLR     R1          ;Init value accumulator
        MOV     R5,R2       ;Save address of string
NEXINT: MOVB    (R2)+,R3    ;Get next character
        BEQ     ENDSTR      ;Check for end of string
        CMPB    R3,#'9      ;Is character > '9?
        BGT     ENDINT      ;Branch if so
        CMPB    R3,#'0      ;Is character < '0?
        BLT     ENDINT      ;Branch if so
        BIC     #'0,R3      ;Now make digit binary
        MUL     #10.,R1     ;Multiply accumulator by 10
        BCS     BADVAL      ;Branch if overflow
        ADD     R3,R1       ;Add number to accumulator
        BCS     BADVAL      ;Branch if overflow
        BR      NEXINT      ;Process next character

ENDINT: CMPB    R3,#'.      ;Is it a decimal point?
        BNE     BADVAL      ;If not it isn't valid!
NEXDVL: MOVB    (R2)+,R3    ;Get next character
        BEQ     ENDSTR      ;Check for end of string
        CMPB    R3,#'9      ;Is character > '9?
        BGT     BADVAL      ;Branch if so, invalid
        CMPB    R3,#'0      ;Is character < '0?
        BLT     BADVAL      ;Branch if so, invalid
        BR      NEXDVL      ;Get next fractional char

ENDSTR: MOV     R1,R0       ;End of string: return val
        BR      RESTOR      ;Goto finale of subroutine

BADVAL: MOV     #-1.,R0     ;Not valid, return -1

RESTOR: MOV     (SP)+,R4    ;Restore saved registers
        MOV     (SP)+,R3
        MOV     (SP)+,R2
        MOV     (SP)+,R1
        RTS     PC          ;Return to caller

        .END
```

PR0405.MAC

```
                .TITLE  PR0405  Debugging Exercise
                .MCALL  .PRINT,.TTYOUT
                .GLOBL  MTAB
HISPRT::
                MOV     R1,-(SP)        ;Save caller's registers
                MOV     R2,-(SP)
                MOV     R3,-(SP)
                MOV     R4,-(SP)
                .PRINT  #HEADER         ;Print three blank lines
                MOV     #20.,R1         ;Init height loop counter
                MOV     #E100,R3        ;First height level = 20 (100)
                BR      ODD             ;Branch to output value

ILOOP:          MOV     #OIDSTR,R3      ;Point to ruler section
                MOV     R1,R2           ;Copy height loop counter
                BIT     #1,R2           ;Is height level odd?
                BNE     ODD             ;Branch if so to print ruler
                ASR     R2              ;Divide loop counter by 2
                BISB    #'0,R2          ;Make this value a character
                MOVB    R2,DIGIT        ;Insert into ruler section
                MOV     #EIDSTR,R3      ;Point to ruler section
ODD:            .PRINT  R3              ;Print ruler section
                MOV     R5,R4           ;Get pointer to monthly height
                MOV     #12.,R2         ;Init months loop counter
JLOOP:          MOVB    (R4)+,R3        ;Get next monthly height
                MOV     #BLANK,STR      ;Default section is blank
                TSTB    R3              ;Test monthly height
                BPL     GO1             ;If positive go past BAD handler
                CMP     R1,#1           ;BAD: check for height level=1
                BNE     LOWER           ;If not go print blank anyway
                MOV     #BAD,STR        ;If BAD+hgt lvl=1 section is BAD
GO1:            CMPB    R3,R1           ;Compare height lvl with month's
                BLT     LOWER           ;If below, go print blank
                MOV     #BLOCK,STR      ;Otherwise print shaded block
LOWER:          .PRINT  STR             ;Print section
                SOB     R2,JLOOP        ;End of loop for months
                SOB     R1,ILOOP        ;End of loop for height level

                .PRINT  #BASE           ;Print base of histogram
                MOV     #MTAB,R1        ;Get address of months table
                MOV     #12.,R2         ;Init months loop counter
MLOOP:          .PRINT  (R1)+           ;Print month pointed to
                MOVB    #32.,R0         ;Create a space character
                .TTYOUT                 ;Print space character
                SOB     R2,MLOOP        ;End of months loop
                .PRINT  #CRLF           ;Finish mths ruler with CR/LF
                MOV     (SP)+,R4        ;Restore saved registers
                MOV     (SP)+,R3
                MOV     (SP)+,R2
                MOV     (SP)+,R1
                RTS     PC              ;Return to caller
```

PR0405.MAC (continued)

```
HEADER: .ASCIZ  <15><12><15><12><15><12>
EIDSTR: .ASCII  <15><12>/ /
DIGIT:  .ASCII  /*0!/<200>
OIDSTR: .ASCII  <15><12>/  -!/<200>
E100:   .ASCII  /100!/<200>
BLOCK:  .ASCII  /### /<200>
BLANK:  .ASCII  /    /<200>
BAD:    .ASCII  /BAD /<200>
BASE:   .ASCII  <15><12>/  0+---------------------------/
        .ASCII  /-----------------------/
        .ASCII  <15><12>/    /<200>
CRLF:   .ASCIZ  / /
        .EVEN
STR:    .WORD   0
        .END
```

PR0403.FOR

```
C ****************************************************************
C
C DEBUGGING AND FAILURE ANALYSIS
C
C PRACTICE 4-1, PR0403.FOR
C
C ****************************************************************
C
C INITIALIZE VARIABLES AND ARRAYS
C ==============================
      REAL VALUE(12)
      BYTE VALSTR(8)
      REAL*4 MONTH(12)
      INTEGER HEIGHT(12)
      DATA MONTH /' JAN',' FEB',' MAR',' APR',' MAY',' JUN',
     1            ' JUL',' AUG',' SEP',' OCT',' NOV',' DEC'/
C
C MAIN PROGRAM LOGIC
C *****************
C
C PRINT INSTRUCTIONS
C =================
      TYPE 6000
6000  FORMAT (' THIS PROGRAM PRINTS A HISTOGRAM FROM 12 MONTHLY ',
     1        'VALUES.'/' THE MONTHS ARE JANUARY TO DECEMBER.'/
     2        ' PLEASE ENTER YOUR TWELVE VALUES: ',
     3        'THEY MUST BE IN THE RANGE 0 TO 100')
C
C ACCEPT VALUES AS STRINGS, PROMPTED BY THE MONTH AND PROCESS THEM
C ================================================================
      DO 100 I=1,12
      TYPE 6010,MONTH(I)
6010  FORMAT (A4,':',1X$)
      DO 50 J=1,8
      VALSTR(J)=' '
50    CONTINUE
      READ (5,5010,END=90) (VALSTR(K),K=1,8)
      HEIGHT(I)=-1
5010  FORMAT (8A1)
C
C CONVERT STRING TO REAL VALUE
C ----------------------------
      VALUE(I)=CNVSTR(VALSTR)
C
C CONVERT REAL VALUE TO HEIGHT ON CHART
C -------------------------------------
      IF (VALUE(I) .NE. -1.0) HEIGHT(I)=VALUE(I)*20/100
      GO TO 100
90    TYPE *,' '
100   CONTINUE
C
C PRINT HISTOGRAM
C ===============
      CALL HISPRT(HEIGHT,MONTH)
      CALL EXIT
      END
```

PR0404.FOR

```
C ***************************************************************
C
C DEBUGGING AND FAILURE ANALYSIS
C
C PRACTICE 4-1, PR0404.FOR
C
C ***************************************************************
C
      FUNCTION CNVSTR(STRING)
      BYTE STRING(8),CURCHA
      INTEGER POINTA,DIGIT
      REAL CNVSTR,DIV
C
C BAD VALUES ARE SET TO -1.0
C VALUES OUT OF RANGE ARE TREATED AS BAD
C ======================================
C
C INITIALIZE RETURN VALUE AND POINTER INTO STRING
C -----------------------------------------------
      CNVSTR=0.0
      POINTA=1
C
C PROCESS EACH CHARACTER, STRING IS TERMINATED BY SPACE OR LENGTH=8
C -----------------------------------------------------------------
10    IF (POINTA .GT. 8) GO TO 100
      CURCHA=STRING(POINTA)
      IF (CURCHA .GT. ' ') GO TO 100
      IF (CURCHA .GT. '9') GO TO 50
      IF (CURCHA .LT. '0') GO TO 50
      DIGIT=CURCHA-'0'
      CNVSTR=(10.0*CNVSTR)+DIGIT
      POINTA=POINTA+1
      GO TO 10
50    IF (CURCHA .NE. '.') GO TO 200
      DIV=1.0
75    POINTA=POINTA+1
      IF (POINTA .GT. 8) GO TO 100
      CURCHA=STRING(POINTA)
      IF (CURCHA .GT. ' ') GO TO 100
      IF (CURCHA .GT. '9') GO TO 200
      IF (CURCHA .LT. '0') GO TO 200
      DIV=DIV*10.0
      DIGIT=CURCHA-'0'
      CNVSTR=CNVSTR+DIGIT/DIV
      GO TO 75
C
C BRANCH TO HERE AT END OF STRING PROCESSING
C ----------------------------------------
100   IF (CNVSTR .GT. 100.0) GO TO 200
      RETURN
C
C BRANCH TO HERE IF VALUE IS BAD
C ----------------------------
200   CNVSTR=-1.0
      RETURN
      END
```

PR0405.FOR

```
C ********************************************************************
C
C DEBUGGING AND FAILURE ANALYSIS
C
C PRACTICE 4-1, PR0405.FOR
C
C ********************************************************************
C
      SUBROUTINE HISPRT(HEIGHT,MONTH)
      INTEGER HEIGHT(12)
      REAL*4 MONTH(12)
      TYPE 6100
6100  FORMAT (' '///)
      DO 100 I=20,1,-1
      IF (I .EQ. 20) GO TO 10
      IF (2*(I/2) .EQ. I) TYPE 6000,I*5
      IF (2*(I/2) .NE. I) TYPE 6010
      GO TO 20
10    TYPE 6020
6000  FORMAT ('  ',I2,'!'$)
6010  FORMAT ('   -!'$)
6020  FORMAT (' 100!'$)
20    DO 90 J=1,12
      IF (HEIGHT(J) .NE. I) GO TO 30
      TYPE 6030
6030  FORMAT (' ### '$)
      HEIGHT(J)=HEIGHT(J)-1
      GO TO 90
30    IF (I .NE. 1) GO TO 35
      IF (HEIGHT(J) .NE. -1) GO TO 35
      TYPE 6040
6040  FORMAT (' BAD '$)
      GO TO 90
35    TYPE 6050
6050  FORMAT (5X$)
90    CONTINUE
      TYPE 6060
6060  FORMAT (' ')
100   CONTINUE
      TYPE 6070
6070  FORMAT ('   0+'$)
      TYPE 6075
6075  FORMAT ('------------------------------------------------')
      TYPE 6080
6080  FORMAT ('    '$)
      TYPE 6090,(MONTH(K),K=1,12)
6090  FORMAT (' ',12A4)
      RETURN
      END
```

PR0403.BAS

```
10 REM ***************************************************************
20 REM
30 REM DEBUGGING AND FAILURE ANALYSIS
40 REM
50 REM PRACTICE 4-1,  PR0403.BAS
60 REM
70 REM ***************************************************************
80 REM
90 REM  INITIALIZE VARIABLES AND ARRAYS
100 REM ==============================
110 DIM V(12%)
120 DIM M$(12%)
130 DIM H%(12%)
140 REM
150 REM MAIN PROGRAM LOGIC
160 REM *****************
170 REM
180 REM READ MONTH STRINGS INTO ARRAY
190 REM ============================
200 FOR I%=1% TO 12% \ READ M$(I%) \ NEXT I%
210 REM
220 REM PRINT INSTRUCTIONS
230 REM ==================
240 PRINT "THIS PROGRAM PRINTS A HISTOGRAM FROM 12 MONTHLY VALUES."
250 PRINT "THE MONTHS ARE JANUARY TO DECEMBER."
260 PRINT "PLEASE ENTER YOUR TWELVE VALUES: THEY MUST BE IN THE RANGE 0 TO 100"
270 REM
280 REM ACCEPT VALUES AS STRINGS, PROMPTED BY THE MONTH
290 REM ==============================================
300 FOR I%=1% TO 12% \ PRINT M$(I%)": "; \ LINPUT #0%,V$ \ GOSUB 10000
310 REM
320 REM CONVERT EACH VALUE INTO A HEIGHT INTEGER
330 REM ======================================
340 H%(I)=FNA%(V(I%)) \ NEXT I%
350 REM
360 REM DISPLAY HISTOGRAM
370 REM =================
380 GOSUB 11000
390 REM
400 REM END OF MAIN PROGRAM LOGIC
410 REM ************************
420 GO TO 32767
10000 REM
10010 REM SUBROUTINE TO CONVERT STRING INTO A REAL NUMBER
10020 REM ==============================================
10030 REM BAD VALUES ARE SET TO -1.0
10040 REM VALUES OUT OF RANGE ARE TREATED AS BAD
10050 V(I%)=0%
10060 L%=LEN(V$) \ F$=SEG$(V$,1%,1%) \ IF F$<>"" THEN V$=SEG$(V$,2%,L%)
```

PR0403.BAS (continued)

```
10070 IF F$="" THEN 10180
10080 IF F$>"9" THEN 10110
10090 IF F$<"0" THEN 10110
10100 V(I%)=10*V(I%)+VAL(F$) \ GO TO 10060
10110 IF F$<>"." THEN V(I%)=-1 \ GO TO 10180
10120 D=1
10130 D=D*10 \L%=LEN(V$) \ F$=SEG$(V$,1%,1%) \ IF F$<>"" THEN V$=SEG$(V$,2%,L%)
10140 IF F$="" THEN 10180
10150 IF F$>"9" THEN V(I%)=-1% \ GO TO 10180
10160 IF F$<"0" THEN V(I%)=-1% \ GO TO 10180
10170 V(I%)=V(I%)+VAL(F$)/D \ GO TO 10130
10180 IF V(I%)>100 THEN V(I%)=-1
10190 RETURN
11000 REM
11010 REM PRINT HISTOGRAM
11020 REM ===============
11030 PRINT  \ PRINT  \ PRINT
11040 FOR I%=20% TO 1% STEP -1%
11050 I$=STR$(I%*5) \ IF I%<20% THEN I$=" "+I$
11060 IF 2%*(I%/2%)=I% THEN PRINT I$; \ GO TO 11080
11070 PRINT "  -";
11080 PRINT "!";
11090 FOR J%=1% TO 12%
11100 IF H%(J%)=I% THEN PRINT " ###"; \ H%(J%)=H%(J%)-1% \ GO TO 11140
11110 IF I%<>1% THEN 11130
11120 IF H%(J%)=-1% THEN PRINT " BAD"; \ GO TO 11140
11130 PRINT "    ";
11140 NEXT J% \ PRINT  \ NEXT I%
11150 PRINT "  0+"; \ FOR I%=1% TO 12% \ PRINT "----"; \ NEXT I% \ PRINT
11160 PRINT "    "; \ FOR I%=1% TO 12% \ PRINT " ";M$(I%); \ NEXT I% \ PRINT
11170 RETURN
15000 REM
15010 REM FUNCTION TO CALCULATE HEIGHT
15020 REM ===========================
15030 DEF FNA%(X)=INT(X*20/100)
20000 REM
20010 REM DATA DECLARATION FOR MONTH STRING ARRAY
20020 REM ======================================
20030 DATA JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC
32000 REM
32010 REM END OF PROGRAM
32020 REM ==============
32767 END
```

References

RT–11 System User's Guide. Chapter 4 discusses the FRUN/PAUSE and RESUME commands in detail.

RT–11 System Utilities Guide. Chapter 18 lists and explains additional commands that will enable you to use ODT and VDT more effectively.

RT–11 System Message Manual.

5

5 Using Libraries

This chapter deals with the creation and maintenance of your own object and macro libraries. You will learn to carry out operations using the RT–11 monitor command LIBRARY, which enables you to maintain such libraries and their contents. The options used with the LIBRARY command to create and maintain libraries are /CREATE, /DELETE, /EXTRACT, /INSERT, /LIST, /MACRO, and /REMOVE.

When you have completed this chapter you will be able to create a new object library; insert, replace, and delete modules from libraries; list the contents of a library; and link a program using object modules contained within an object library of your own creation. If you are using MACRO–11 you will learn to create a new macro source library and assemble a program using macros within a macro library of your creation.

When you have a number of routines that are used for one type of application, you may want to group them into a library. The linker uses only those subroutines that are referenced, and the MACRO–11 assembler uses only those macros that are referenced. If these subroutines or macros are contained in a library, you do not have to type in a whole list of selected files.

Two libraries are supplied with your system, the system object library SYSLIB.OBJ which contains a set of subroutines in object code form, and the system macro library SYSMAC.SML which contains a set of system macros.

Using Object Libraries

You can store object modules either in an object file or as a subroutine in an object library. When you want to link a set of object modules, some of them may be contained in an object library. There are two ways of including modules from an object library when using the linker:

- Specifying each library that contains subroutines that are called by using the /LINKLIBRARY option (which can be abbreviated to /LIBRARY.)
- Specifying the library in the same way that you specify object modules; the linker can detect that the file is an object library and will access any routines that are called.

EXAMPLE

Assume that you have compiled or assembled a main program PROG.OBJ and three subroutines, SUBA, SUBB, and SUBC. Assume also that SUBA is contained in the object library LIBA.OBJ, that SUBB is contained in the object library LIBB.OBJ, and that SUBC is contained in the object file SUBC.OBJ. Then, to produce the load image PROG.SAV, you can link these files using either the command:

```
.LINK/LIBRARY:LIBA/LIBRARY:LIBB PROG,SUBC
```

or the command:

```
.LINK PROG,LIBA,LIBB,SUBC
```

Searching Sequence for Object Code Subroutines

When you link a main program and your code contains references to subroutines, the linker first looks for each subroutine in any object modules you specify, taking them from left to right in the command line. If it cannot find the subroutine there, it looks in any libraries you specify, taking them from left to right in the command line. If that fails, it looks in the system object library (SY:SYSLIB.OBJ). You do not need to specify this library. If that also fails, it prints warnings at the terminal. Once a subroutine has been found, any subroutines with the same name that come later are ignored. Only subroutines that are referenced are used by the linker.

Using Object Libraries with EXECUTE

To specify that object libraries are to be used during linking, you can use the /LINKLIBRARY option with the EXECUTE command. This command takes the same format as the LINK/LIBRARY command:

```
EXECUTE/LINKLIBRARY:LIBRSPEC  PROGSPEC
```

Using Macro Libraries

You can store macro definitions either:

- In the modules which use them or
- In a macro library

By storing the macro definitions in a macro library, modules which use them need not define the macros. These modules can call the macros which are defined in a library by using the .MCALL directive. The .MCALL directive is fully described in chapter 7 of the *MACRO–11 Language Reference Manual.*

When you assemble your modules, you must also specify the macro libraries containing the macro definitions needed by your module(s). The macro libraries must contain the definitions for all of the macros specified by the .MCALL directive. You can specify macro libraries by using the /LIBRARY option with the MACRO command after each macro library file specification.

EXAMPLE

```
.MACRO MYMACS/LIBRARY+PROG
```

assembles PROG.MAC with the macro library MYMACS.MLB.

You can also specify macro libraries by including the .LIBRARY directive and specifying the macro library file specifications in the modules themselves. The .LIBRARY directive is discussed in chapter 6 of the *MACRO–11 Language Reference Manual.*

Searching Sequence for Macro Libraries

The MACRO assembler searches the macro libraries for the macros specified by the .MCALL directive. The search begins with the user specified libraries (either from the /LIBRARY option or the .LIBRARY directive) and continues onto the system library (SY:SYSMAC.SML) if the macro is not found. When found, the definition is extracted from the macro library for use in the modules.

Using Macro Libraries with EXECUTE

You may also specify macro libraries when you use the EXECUTE command to assemble modules. The EXECUTE command also has a /LIBRARY option which is used in the same way as with the MACRO command.

EXAMPLE

```
.EXECUTE/MACRO MYMACS/LIBRARY+PROG
```

assembles the PROG.MAC module using the macro library MYMACS.MLB. After the module is assembled, it is linked and executed.

Creating Libraries

You can use the RT–11 librarian to create library files so that they contain the contents of one or more files. We will discuss only the creation of object libraries and macro libraries, although it is possible to use the library structure to group together other types of files. You create libraries by using the LIBR/CREATE command.

Creating a New Object Library

You use an object library to group together a set of object modules that have been assembled or compiled from subroutines written in source code. You may optionally include one main program object module in the library. To create an object library that initially includes the contents of one or more object modules, use the command:

LIBR/CREATE LIBRARY-FILE OBJECT-MODULES

In this command "LIBRARY-FILE" is the file specification of the object library you want to create, and "OBJECT-MODULES" is a list of file specifications (separated by commas) of the object modules to be inserted in the library. If you omit the file types of the object modules, the librarian assumes that they have the .OBJ file type. Object libraries will also be given the default file type .OBJ.

EXAMPLE

If you have two object modules, DK:SUBA.OBJ and DK:SUBB.OBJ, and want to group them to form the object library DK:MYLIB.OBJ, then you type:

```
.LIBR/CREATE MYLIB SUBA,SUBB
```

If you try to include more than one main program in such a command line, no library is created.

Stored in an object library is an index to the modules it contains. The name of each module is taken from the name of the subroutine. The library also contains a table of all the entry points (globals) used in the modules. For example, assume that you have created a FORTRAN IV source file PROG.FOR containing the subroutine CALC. If you compile this subroutine to produce the object module PROG.OBJ and then include that module in an object library, it is entered in the library under the name of CALC.

Only one object module can be stored in the library under a given name. If you create a library including two object modules whose subroutine names are the same, the librarian prints a warning message at the terminal and only the first such module is used in the library.

Creating a New Macro Library

When you have a number of macros that are all used in one application, it is useful to group them in a macro library. The MACRO–11 assembler then accesses only those macros that are referenced when you assemble a source file using the library. You place macro definitions extracted from various modules into one or more files. Such macros are identified by the .MACRO directive. To create a macro library that contains the macros found in one or more source files, use the command:

```
LIBR/MACRO/CREATE LIBRARY-FILE MACRO-FILES
```

In this command "LIBRARY-FILE" is the file-specification of the macro library you want to create and "MACRO-FILES" is a list (separated by commas) of source files containing the macros that are to be included in the library.

If you omit the file type of the source files you use, the librarian assumes that they have the default file type of .MAC. Macro library files are assigned the default file type .MLB.

EXAMPLE

If you have two files, DK:MACA.MAC and DK:MACB.MAC, containing only macros, and you want to group them to form the macro library DK:MYMACS.MLB, you type:

```
.LIBR/MACRO/CREATE MYMACS MACA,MACB
```

The name of each macro in the source files you specify must be unique. If there are two macros with the same name in these source files, only the first one encountered is included, and the librarian prints warnings at the terminal.

Creating an Object Module from an Object Library

When you have successfully included an object module in a library, you no longer need to keep the original object module for program development and you can delete it. You may want to recreate an object module, for example, to produce a version of it on a different storage volume. You create object modules from object libraries using the /EXTRACT option. To use this utility, type the command:

```
LIBR/EXTRACT
```

The librarian then asks you to supply the following:

1. The library file containing the subroutines you want to recreate. For this the librarian prompts you with:

   ```
   Library?
   ```

2. The file specification of each object module you want to create. For this the prompt is:

   ```
   File    ?
   ```

3. The name of each subroutine whose object code is to be used to create this module. These names are often referred to as globals. For each of these the prompt is:

   ```
   Global ?
   ```

EXAMPLE

Assume that you want to recreate the object module DK:PRCALC.OBJ from the subroutine CALC that is contained in the object library DK:CALCNS.OBJ. Then you would use the /EXTRACT option and respond to the prompts as follows:

```
.LIBR/EXTRACT<RETURN>
Library? CALCNS
File    ? PRCALC
Global ? CALC
Global ?<RETURN>
```

Maintaining Libraries

When you develop programs in a modular way, you may want to add new modules, delete obsolete ones, replace old ones with new ones, and also list the contents of your library. If you are a MACRO–11 programmer, you will also want to perform similar operations on the macros in your macro library.

Maintaining Object Libraries

The RT–11 librarian allows you to maintain your object libraries using the following options:

/DELETE	removes an object module from the library, deleting from the symbol table globals that no longer apply
/INSERT	includes a new object module in the library, updating the symbol table with any new global symbols
/LIST	gets a directory listing of all the modules in the library

/REMOVE	deletes global symbol(s) from the library index without deleting the routines they represent

Maintaining Macro Libraries

You cannot modify or list a macro library using the LIBR command alone. To modify a macro library, edit the original source file(s) and recreate the library, using the LIBR/MACRO/CREATE command as discussed previously.

Practice 5–1

Using FORTRAN IV

1. Type the following programs into four files. Name them PR0501.FOR, PR0502.FOR, PR0503.FOR, and PR0504.FOR.

PR0501.FOR:

```
C PR0501.FOR
C
C Set up data for word processing-like subroutines
C and then call those subroutines
C
      INTEGER*2 NAME(10),DATE(10)
      REAL MONEY
      DATA NAME
     1/'Mr',' G','ri','ff','it','hs',', ',' ',' ',' '/
      DATA DATE
     1/'Ju','ly',' 1','Ot','h ','19','77','. ',' ',' '/
      DATA MONEY /16.27/
      CALL TEXT1(NAME,MONEY)
      CALL TEXT2(DATE)
      CALL TEXT3
      CALL EXIT
      END
```

PR0502.FOR:

```
C PR0502.FOR
C
      SUBROUTINE TEXT1 (NAME,MONEY)
      INTEGER*2 NAME(10),VFLAG
      REAL MONEY
      TYPE 1000,(NAME(I),I=1,10)
      VFLAG=4
      IF (MONEY .LT. 1000.0) VFLAG=3
      IF (MONEY .LT. 100.0) VFLAG=2
      IF (MONEY .LT. 10.0) VFLAG=1
      GOTO(100,200,300,400) VFLAG
100   TYPE 2000,MONEY
      GOTO 500
200   TYPE 2010,MONEY
      GOTO 500
300   TYPE 2020,MONEY
      GOTO 500
400   TYPE 2030,MONEY
500   TYPE 1010
      RETURN
1000  FORMAT(1H0,'Dear ',10A2//
     1' During our last quarter, our records showed that you ',
     2'owed us ')
1010  FORMAT(' .')
2000  FORMAT(' ',F4.2,$)
2010  FORMAT(' ',F5.2,$)
2020  FORMAT(' ',F6.2,$)
2030  FORMAT(' ',F7.2,$)
      END
```

PR0503.FOR:

```
C PR0503.FOR
C
      SUBROUTINE TEXT2 (DATE)
      INTEGER*2 DATE(10)
      TYPE 1000,(DATE(I),I=1,10)
      TYPE 1500
1500  FORMAT(' Accordingly we sent you a letter of invoice ',
     1'at that time.')
1000  FORMAT(' We have been expecting your payment since ',10A2)
      RETURN
      END
```

PR0504.FOR:

```
C PR0504.FOR
C
      SUBROUTINE TEXT3
      TYPE 500
      TYPE 1000
      TYPE 1500
      TYPE 2000
500   FORMAT(' Regrettably this was an error on our part.')
```

```
1000  FORMAT(' '/' PLEASE SEND THE MONEY STRAIGHT AWAY.')
1500  FORMAT(' '/' Yours sincerely,')
2000  FORMAT(' '///' A.N. Other (Manager)')
      RETURN
      END
```

2. Compile each of these FORTRAN IV files to produce four object modules.
3. Create an object library from the modules PR0502.OBJ and PR0503.OBJ, giving the library the name TEXLIB.OBJ.
4. Insert the module PR0504.OBJ and get a listing of the library.
5. Produce and run the save image PR0501.SAV using the main object module PR0501 and the object library TEXLIB.
6. The program will print two paragraphs of text at your terminal. One of the sentences printed by the program is:

   ```
   PLEASE SEND THE MONEY STRAIGHT AWAY.
   ```

   Change the sentence to:

   ```
   PLEASE ACCEPT OUR APOLOGIES.
   ```

   by editing the source program PR0504.FOR. Replace the FORMAT statement labelled 1000 with the line:

   ```
   1000  FORMAT(' '/' PLEASE ACCEPT OUR APOLOGIES.')
   ```

7. Update the object library; produce and run a new save image.

Practice 5–2

Using MACRO–11

1. Type the following programs into three files. Name the files PR0505.MAC, PR0506.MAC, and PR0507.MAC respectively.

PR0505.MAC:

```
          .TITLE   PR0505 Subroutine MCTST
          .MCALL   .EXIT,GOSUB
          .ENABL   LC
MCTST::   GOSUB    TEXT1,<#NAME,#MONEY>
          GOSUB    TEXT2,<#DATE>
          GOSUB    TEXT3
          RTS      PC
MONEY:    .FLT2    16.27
NAME:     .ASCIZ   /Mr. Griffiths,     /
DATE:     .ASCIZ   /July 10th 1977.    /
          .END
```

PR0506.MAC:

```
; *
; *     B O N
; *
; *     Branch if bit set on
; *
; *
        .MACRO  BON   MASK,TEST,LABEL
        BIT     MASK,TEST
        BNE     LABEL
        .ENDM   BON
```

PR0507.MAC:

```
; *
; *
; *     G O S U B
; *
; *      Macro to call a high-level language subroutine
; *
; *
         .MACRO GOSUB  SUBR,PARS
         .GLOBL SUBR
         Q$P=0
         .IRP   X,<PARS>
         Q$P=Q$P+1
         .ENDR
         Q$$P=Q$P+Q$P+2
         SUB    #Q$$P,SP
```

```
MOV     SP,R5
MOV     #Q$P,(R5)+
.IRP    XX,<PARS>
MOV     XX,(R5)+
.ENDR
MOV     SP,R5
CALL    SUBR
ADD     #Q$$P,SP
.ENDM   GOSUB
```

2. Type the following FORTRAN IV program into a file named PR0508.FOR:

```
CALL MCTST
CALL EXIT
END
```

3. Create the macro library PRMACS.MLB from the source files PR0506.MAC and PR0507.MAC.

4. Assemble the program PR0505 to create the object module PR0505.OBJ, using the macro library you have just created.

5. Compile the FORTRAN IV program PR0508, and link the programs PR0508.OBJ and PR0505.OBJ with the library TEXLIB.OBJ that you created in practice 5–1 to produce a save image and run that image.

The program should print at the terminal the letter you saw in practice 5–1.

Reference

RT–11 System Users Guide. Chapters 4 and 12 discuss the LIBRARY command in detail.

6

6

Designing and Implementing Overlay Structures

When you need to write a large program or modify an existing one so that it becomes larger, you may find that it takes up so much memory that other jobs, which need to run at the same time, are unable to run. Even with the XM monitor, it is possible for a program to be too large for the available memory. This chapter discusses ways of improving memory use and speed of execution.

You will learn to design and implement an overlay structure for a MACRO–11 or FORTRAN IV program, check the memory use of an overlaid program from the load map, and control when the User Service Routine is swapped in and out of memory during execution of a program. If you are programming with FORTRAN IV, you will also learn to use options of the compiler to generate more efficient machine code.

Limitations on Available Memory

The memory space available for running a program is less than the full addressing space of the system, some of which is taken up by system programs, such as device handlers and the monitor. In a foreground/background environment, often the remaining space has to be shared between a foreground and a background job. Therefore, the memory requirements of both of these jobs may have to be reduced. Some ways of reducing memory requirements are:

- Using overlay programs
- Swapping out the User Service Routine (USR)
- Using compiler optimization techniques

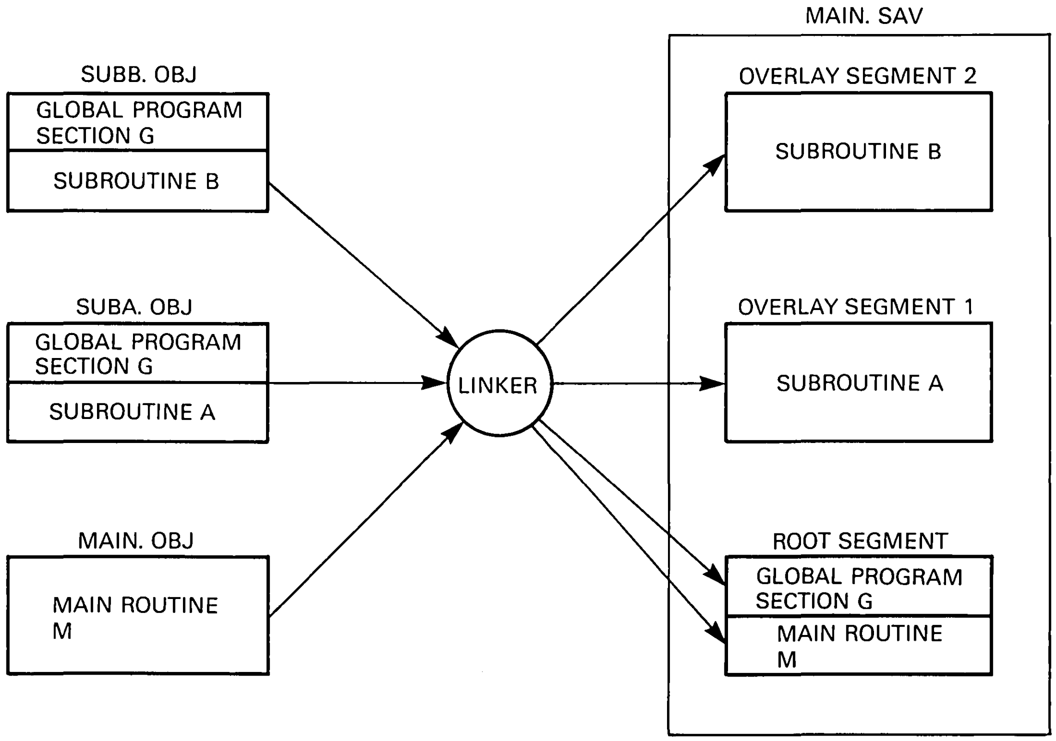
Overlays

If a program is too large to be entirely resident in memory at one time, you can reduce its memory requirements by using overlays. This means that parts of your program are resident in memory, while other parts are swapped out to a file. To do this you define an overlay structure for your program. An overlay structure is a system by which the program's memory is divided into a root region and a number of overlay regions.

When you overlay a program, the linker extracts those parts of each object module that must be resident in memory throughout the execution of the program and groups them into the root segment. These parts are global program sections and include global .PSECTs (MACRO–11) or COMMON blocks (FORTRAN IV). The root segment resides in the root region. The linker places the remaining code of an object module in an overlay segment. Figure 13 shows how this is done for a program that is made up of a main object module, MAIN, and two subprogram modules, SUBA and SUBB.

In an overlay structure, you assign a group of overlay segments to each overlay region. Only one overlay segment is resident in a region at one time and the remaining over-

Figure 13.
Using the Linker to Implement an Overlay Structure

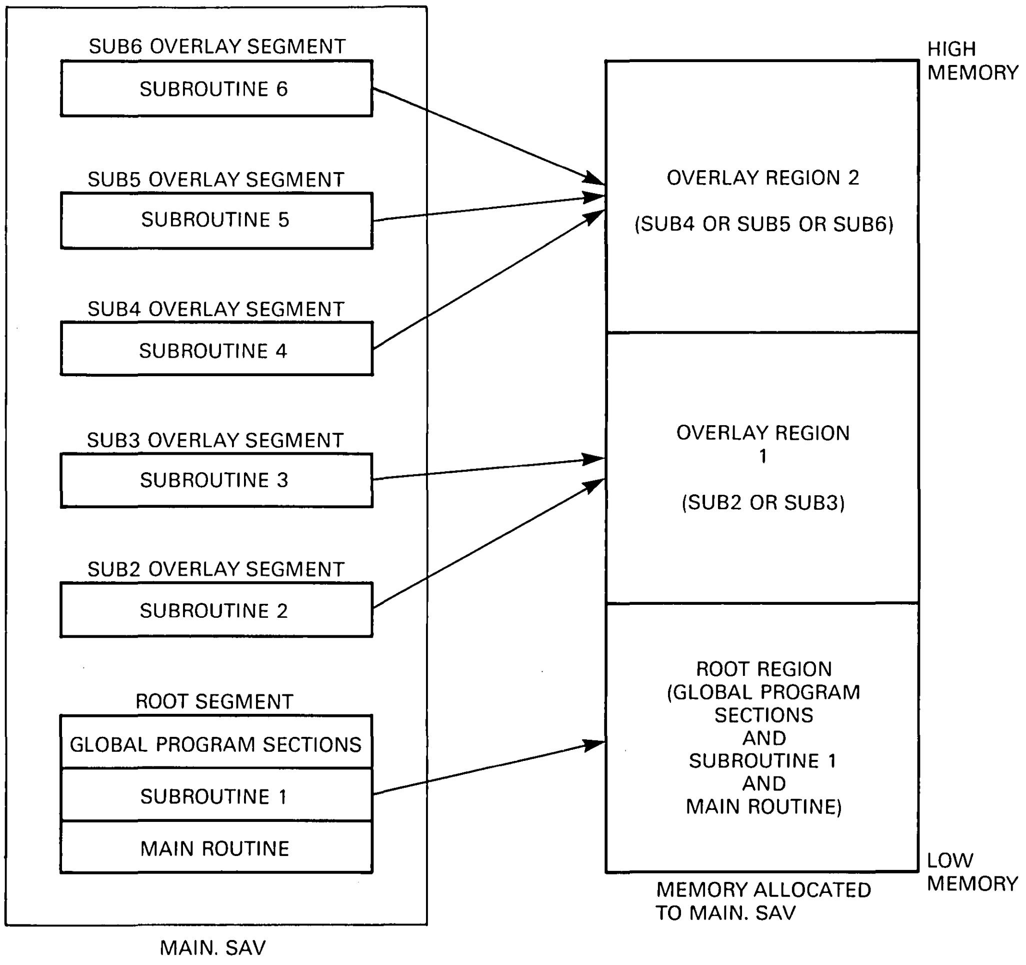


lay segments are swapped out to a file called an overlay file. The swapping of overlay segments is carried out at run time by the Run-time Overlay Handler. For example, assume that your program is made up of a main object module, MAIN, and six subroutine modules, SUB1 to SUB6. Assume also that you define an overlay structure so that:

- The root segment contains MAIN and SUB1
- A first overlay region is assigned the overlay segments produced from SUB2 and SUB3
- A second overlay region is assigned the overlay segments produced from SUB4, SUB5, and SUB6

Figure 14 shows how the program's memory is shared.

Figure 14.
Overlay Segments Sharing a Program's Memory



Specifying Overlay Structures

You specify an overlay structure for your program to the linker by using the /O option. This option is used in the following format:

```
.R LINK
*save-image,map-file=root-list//
*object-module-1/O:1
```

```
*object-module-2/O:1
*...
*object-module-n/O:r//
```

save-image	is the file specification in which the save image is to be built
map-file	is the file specification of the load map to be produced from the linker
root-list	is the list of those object modules, including the main routine, that contain code that must be placed entirely in the root segment; also, link libraries that contain code that is referenced elsewhere in the root segment
object-module-i (i=1,...,n)	is the object module used for the overlay segment
/O:j (j=1,...,r)	specifies to which region the overlay segment is assigned

Assume that you want to create an overlay structure in which references are made to code contained in the object libraries LIBA and LIBB. Your command would look like the following:

EXAMPLE

```
.R LINK
*MAIN,MAIN.MAP=MAIN,SUB1,LIBA,LIBB//
*SUB2/O:1
*SUB3/O:1
*SUB4/O:2
*SUB5/O:2
*SUB6/O:2//
```

Code linked directly from an object library is placed in the root segment. If you want to produce an overlay segment from an object library module, you must first extract it from the library, using the LIBR/EXTRACT option, thereby producing an object module in a separate file. You can then link your program and assign your module to an overlay region in the usual way.

Practice 6–1

Use either the FORTRAN IV or MACRO–11 program you created in the exercises in chapter 4, "Debugging Programs." The main program calls each of the subroutines in turn. The subroutines shown below do not reference each other.

PR0403	a main program that accepts monthly values and prints a histogram
PR0404	the routine CNVSTR for converting values to heights on the histogram
PR0405	the subroutine HISPRT for printing the histogram

Design and implement an overlay structure for this program by assembling the source files into separate object modules and using the linker to create the overlaid program HISTO.SAV. The program should use the minimum amount of memory without your changing the code of the routines. Get a load map of the overlaid program from the linker.

High-level Language Optimization

A compiled language program generally uses more memory space than an equivalent assembly language program. This is because high-level language compilers often include a number of general purpose subroutines in the code that they produce, such as the OTS for FORTRAN IV. Also, the code generator of a compiler is usually far less capable than the average MACRO–11 programmer in producing efficient machine code. For this reason, many compilers are equipped with optimization routines that reduce the code generated without affecting the action of the program.

Different high-level language compilers offer different optimization techniques for the code they generate. They usually have options that allow you to select the extent to which optimization is carried out. For example, BLISS–16 has a very large and complex code generator and, if you select full optimization, it generates code only 10% larger than that produced from an equivalent MACRO–11 program.

The two optimization options provided by the FORTRAN IV compiler are: selection of generated code and inclusion of line numbers or vectors. Additionally, some optimization techniques are automatically performed by the FORTRAN IV compiler.

Generated Code

Basically, there are two ways in which the FORTRAN IV compiler can generate code. These two methods are usually called in-line code and threaded code.

Types of code and their characteristics

In-line code is usually arranged so that it is executed from start to end with as few machine code subroutine calls as possible. Threaded code is made up of a list of small routines. The only purpose of each routine is to call a subroutine. Before doing this, the routine sets up a data section that will be active only during the execution of that subroutine. To determine which type of code to use you should consider the following:

- In-line code is always executed faster than threaded code because fewer subroutine calls are used
- Unless most of your data exceeds the storage requirements of INTEGER*2 variables (two bytes per variable), there is little difference in size between in-line code and threaded code
- If most of your data uses REAL*4, REAL*8, or COMPLEX*8 variables, then threaded code uses much less memory

Although the above relationships are usually true, they differ from program to program. You should compile and test production programs with both in-line and threaded code to determine the best match of speed and size for your applications.

Selection of generated code

By default, the FORTRAN compiler generates in-line code. In order to select threaded code, you use the /CODE option as follows:

```
FORTRAN/CODE:THR FILENAME
```

Connected with the /CODE option are three values that select specific types of in-line code appropriate for use in your machine's arithmetic operations. You should include the values that agree with your system configuration (check with your system manager).

EXAMPLE

This compilation command informs the compiler that it can use the extended instruction set (EIS):

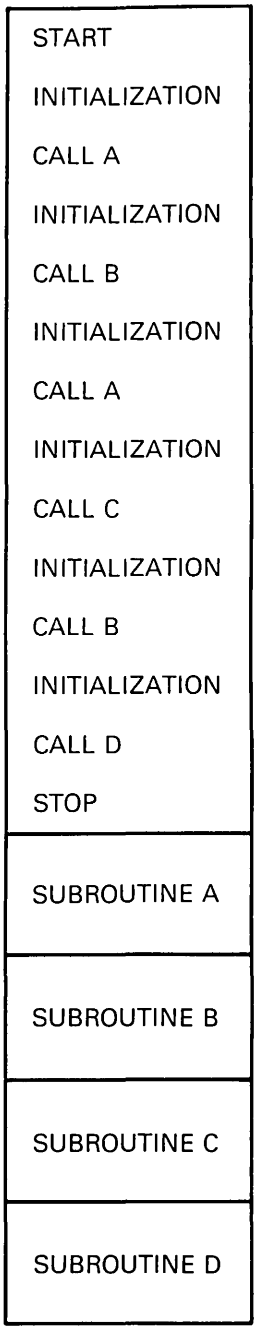
```
.FORTRAN/CODE:EIS PROG
```

Figure 15 illustrates the structure of the object module that has been produced from a FORTRAN IV source file using the /CODE:THR option.

Vectors

For the RT–11 FORTRAN IV compiler, the vector method decreases the time needed to compute the address of an element in an array of more than one dimension. It does so by computing in advance some of the multiplication operations and storing the resulting values in a table called a vector. If you use the /NOVECTORS option to prevent the creation of these vectors, you can reduce the memory space used, but you sacrifice some execution speed.

Figure 15.
Structure of Threaded Code



Sequence Numbers

When you are debugging code, it is useful to see the sequence numbers generated by the compiler, because error diagnostic messages include these numbers. Sequence numbers are also useful if you want to examine compiler-generated code. These sequence numbers, however, occupy memory. The /NOLINENUMBERS option allows you to disable the generation of internal sequence numbers, but it should be used only for production programs which you think are free of errors.

EXAMPLE

```
.FORTRAN/NOLINENUMBERS PROG
```

Additional Optimization Techniques

In addition to the options discussed above, there are many programming techniques that allow you to create more efficient FORTRAN IV programs. Other optimization techniques implemented automatically when your program is compiled include:

- Simplifying arithmetic expressions
- Performing often needed computations, such as loop iteration counts, in registers
- Using bit shifting operations to implement multiplication and division

Swapping the User Service Routine

The User Service Routine (USR), which processes your program's file I/O requests, does not reside permanently in memory. By default, it is swapped in and out of memory as needed, allowing use of more memory. Swapping the USR out of memory makes an additional 2 Kwords available for use. Swapping, however, may cause problems with FORTRAN IV program execution if your program is very large, because the USR may be swapped over some of your code. You can prevent USR swapping by using the /NOSWAP option.

EXAMPLE

There are two ways to use the /NOSWAP option:

1. Use the /NOSWAP option in your command line to the compiler.

```
.FORTRAN/NOSWAP PROG
```

2. Set USR to NOSWAP by using the monitor command:

```
.SET USR NOSWAP
```

To reset the system to swap USR, use the monitor command:

```
.SET USR SWAP
```

References

RT–11 System Utilities Manual. Chapter 11 describes in detail overlays and system utility options of the linker. It offers guidelines to help you select an overlay structure and discusses options that enable you to control the production of overlaid programs. The chapter also explains the different features that apply to extended memory (available with the XM monitor) and virtual overlays.

RT–11/RSTS/E FORTRAN IV User's Guide. Chapter 4 details how to structure programs that will make maximum use of FORTRAN IV execution capabilities.

RT–11 Software Support Manual. Chapter 2 explains USR considerations for foreground and background jobs and USR swapping considerations.

7

7 Using Language Interfaces

In the first six chapters of this book, the development of programs in MACRO–11, FORTRAN IV, and BASIC–11, independent of one another, is discussed. In this chapter, you will learn to write a FORTRAN IV program that calls a MACRO–11 subroutine. You will also learn to write a MACRO–11 program that calls a FORTRAN IV subroutine, and learn to formulate MACRO–11 subroutines that can be called from a FORTRAN IV or BASIC–11 program. If you are using BASIC, you will be able to write a BASIC program that calls a MACRO–11 subroutine.

Usefulness of Language Interfaces

When developing a program in FORTRAN IV or BASIC, you may find that these languages cannot perform some operations or that they perform them in a complex or inefficient way. To solve this problem, you can write MACRO–11 subroutines to be called by your high-level language program, or you can use existing MACRO–11 subroutines. For example, you are programming in BASIC and you want to manipulate a device. But BASIC–11 does not provide an interface to the hardware address of devices, so you would need to write a MACRO–11 subroutine to do the task.

If you are programming in FORTRAN IV, you may want to create a data structure that is not available using the FORTRAN IV language alone. You might use MACRO–11 subroutines to create such a structure and then perform operations on it.

If you are programming in MACRO–11, you may find user-written FORTRAN IV subroutines and FORTRAN IV library or SYSLIB routines useful. It is usually easier to create an interface between a MACRO–11 program and FORTRAN IV subroutines than to rewrite these routines from scratch.

Calling MACRO–11 Subroutines from a FORTRAN IV Program

To call a MACRO–11 subroutine from a FORTRAN IV program you use the FORTRAN IV CALL statement in the same way that you call FORTRAN IV subroutines. The sections which follow discuss the conventions used in this process:

- Transferring control
- Passing arguments
- Returning function values
- Using registers

- Maintaining the stack
- Structuring common blocks
- Receiving arguments

Transferring Control

When the FORTRAN IV compiler processes a subroutine, it generates the instruction that is created in MACRO–11 by the line:

```
JSR     PC, subname
```

Here "subname" is the global symbol for the entry point into the subroutine. If the subroutine is written in MACRO–11, you must ensure that it contains a global declaration of this entry point. You can do this using the .GLOBL directive.

Because the MACRO–11 subroutine is called with JSR PC,subname, it must return with the instruction:

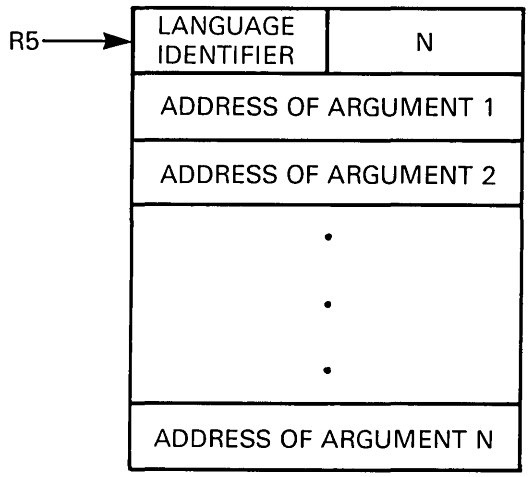
```
RTS     PC
```

Passing Arguments

When a FORTRAN IV program calls a subroutine, it passes the arguments in a contiguous block of memory called an argument block. This block contains n+1 words where "n" is the number of arguments to be passed.

The first word in the block is made up of two bytes. The high-order byte contains an identifier for the calling language. For FORTRAN IV calling routines, this identifier is zero. The low-order byte contains the number of arguments "n." Each word that follows contains the address of an argument. The structure of an argument block is shown in figure 16. Before calling the subroutine, the main program stores in register R5 the address of the first word of the argument block.

Figure 16.
Structure of an Argument Block
in FORTRAN IV Subroutine Calls



Returning Function Values

You can also write MACRO–11 subroutines that behave like FORTRAN IV FUNCTION subprograms. The argument block has the same structure as that of a normal subroutine, but additional values are returned. The calling FORTRAN IV routine looks for these return values in one or more of the registers R0 to R3. The registers examined depend upon the type of function called. These conventions are listed in table 6.

Using Registers

The FORTRAN IV compiler generates code that uses registers R0 to R4. Your MACRO–11 subroutine may still use these registers, because their values are saved whenever a subroutine is called. You may use R5 in your MACRO–11 subroutine after you have referred to it to access the argument block.

Table 6.
FORTRAN IV Function Return Conventions

Type of Function	Return Register(s)
INTEGER*2 LOGICAL*1 LOGICAL*2	R0
INTEGER*4 LOGICAL*4 REAL	R0 (low order) R1 (high order)
DOUBLE PRECISION	R0 (highest order) R1 : R2 : R3 (lowest order)
COMPLEX	R0 (high real) R1 (low real) R2 (high imaginary) R3 (low imaginary)

Maintaining the Stack

The value of the stack pointer (R6) must not change after a subroutine execution. If you use the stack to store arguments or make computations as part of the execution of your MACRO–11 subroutine, make sure that the number of stack PUSHes equals the number of stack POPs.

Creating Common Blocks

When you use the FORTRAN IV COMMON statement to create common blocks, the FORTRAN IV compiler generates these blocks as named .PSECTs, using the name of the common block as the .PSECT name.

EXAMPLE

The FORTRAN IV statement:

```
COMMON /BUF/IA(10), IB(20)
```

is equivalent to the MACRO–11 code:

```
        .PSECT  BUF,RW,D,GBL,REL,OVR
IA:     .BLKW   10.
IB:     .BLKW   20.
```

FORTRAN IV assigns a blank COMMON block the name:

```
.$$$$.
```

Receiving Arguments

Your MACRO–11 subroutine must be able to locate the argument block with which it was called. The address of the first word of the argument block is stored in R5 at the start of execution of the subroutine. If the subroutine is to be called by a variable number of arguments, then you must read that number from the low-order byte of the first word of the block.

For calling languages other than FORTRAN IV, the high-order byte of the first word in the argument block may not be zero. For example, when BASIC–11 calls a MACRO–11 routine, it places the value 202 (octal) into this high-order byte. Each word that follows in the block contains the address of an argument. Therefore, you can use the auto-increment mode to access each argument, incrementing R5 each time. Figure 17 shows a MACRO–11 subroutine that accesses a variable number of arguments that were passed by a FORTRAN IV routine. Each argument points to an integer. The subroutine then returns the highest integer that was received, using the convention of a FORTRAN IV FUNCTION.

Figure 17.
A MACRO–11 Subroutine that Can Be Called as a FORTRAN IV Function

```
        .TITLE  GETHST
GETHST::
        MOV     (R5)+,R1        ;Put number of args in R1
        BIC     #177400,R1      ;Clear high order byte
        BEQ     20$             ;Exit if no arguments
        MOV     (R5)+,R0        ;Read 1st argument value
10$:    DEC     R1              ;Decrement the argument
        BEQ     20$             ;Branch if no more args
        MOV     (R5)+,R2        ;Get address of next arg
        CMP     R0,(R2)         ;Is highest exceeded?
        BGE     15$             ;Branch if not
        MOV     (R2),R0         ;Move new highest into R0
15$:    BR      10$             ;Next argument
20$:    RTS     PC              ;Exit at end,
        .END                    ;Return highest in R0
```

Calling FORTRAN IV Routines from a MACRO–11 Program

If you are writing a MACRO–11 program, you may want to use existing FORTRAN IV subroutines and functions rather than rewrite them. You may also find that it is easier to write a subroutine in FORTRAN IV than to write it in MACRO–11. Two considerations apply:

1. You must initialize the Object Time System (OTS).
2. You must use the conventions for the passing of data and control.

You need not initialize OTS if you are using FORTRAN IV library routines written in MACRO–11, such as those provided in FORLIB and SYSLIB.

Initializing OTS

When you link a MACRO–11 main program with object modules that include FORTRAN IV routines, you must make sure that all the Object Time System (OTS) routines referenced by the FORTRAN IV code are linked to the program. The best way to do this is to write the MACRO–11 program as a subroutine and call it from a simple FORTRAN IV program. The FORTRAN IV program will contain only a call without arguments to the subroutine, followed by an END statement. This coding causes the FORTRAN IV compiler to reference the OTS routines automatically.

The following example shows a program MAIN.MAC with an entry point at the symbol "START" which will be linked to FORTRAN IV routines.

EXAMPLE

1. Insert a global declaration of the transfer symbol of the main program so that it can become a subroutine module, for example, using the directive:

```
.GLOBL   START
```

2. Make sure that the main program terminates with the instruction:

```
RTS    PC
```

3. Remove the transfer symbol from the .END directive so that the last line reads:

```
.END
```

and not:

```
.END START
```

4. Write a FORTRAN IV routine with the following code:

```
CALL START
CALL EXIT
END
```

You can now assemble and compile and link the modules in the usual way to produce your load image.

Conventions

Conventions similar to those discussed for the FORTRAN IV/MACRO–11 interface apply to MACRO–11 routines that call FORTRAN IV subroutines. Here is a summary of the conventions that apply to the MACRO–11/FORTRAN IV interface:

1. To call the FORTRAN IV subroutine, use the MACRO–11 instruction:

   ```
   JSR     PC,subname
   ```

2. The MACRO–11 program must create an argument block in the manner previously described and leave R5 pointing to the first word in the block.
3. If the FORTRAN IV routine is a function, it returns values in registers R0 to R3 as discussed previously.
4. If you want the values to be kept, the MACRO–11 routine should save the registers R0 to R4 before calling the FORTRAN IV routine.
5. If you want to access COMMON blocks that will be used by your FORTRAN IV routines, they must be declared as .PSECTs in the manner previously described.

Figure 18 shows a MACRO–11 routine that calls a FORTRAN IV function and subroutine, using these conventions.

Figure 18.
A MACRO–11 Routine that Calls a FORTRAN IV Function and FORTRAN IV Subroutines

```
    .TITLE  CALLER, MACRO routine calling FORTRAN subrout.
;
;   GOSUB   Macro to call a high level language subroutine
;
      .MACRO  GOSUB SUBR,PARS
      .GLOBL  SUBR                    ;Declare subroutine name
      Q$P=0                           ;Initialize arg count
      .IRP    X,<PARS>                ;Start of loop to count args
      Q$P=Q$P+1                       ;Increment counter
      .ENDR                           ;End of loop
      Q$$P=Q$P+Q$P+2                  ;Byte count for arg block
      SUB     #Q$$P,SP                ;Save space on stack
      MOV     SP,R5                   ;R5 points to start of block
      MOV     #Q$P,(R5)+              ;Push no. of arguments
      .IRP    XX,<PARS>               ;Start of loop to push args
      MOV     XX,(R5)+                ;Push next argument
      .ENDR                           ;End of loop
      MOV     SP,R5                   ;Restore R5 to beg. of block
      CALL    SUBR                    ;Call the subroutine
      ADD     #Q$$P,SP                ;Pop the arguments off stack
      .ENDM   GOSUB                   ;End of Macro definition

; Main Subroutine Code
START:: GOSUB   GETVAL,#RVAL          ;Call FOR func to input RVAL
        CMPB    #1,R0                 ;R0 = 1 if end of data
        BEQ     20$                   ;If end, print result, exit
        GOSUB   RADD,<#RVAL,#C>       ;Call FOR routine to accu-
        BR      START                 ;mulate total, and repeat
20$:    GOSUB   ROUT,#RVAL            ;At end, call FOR routine
        RTS     PC                    ;to print total, and return
RVAL:   .FLT2   0.0                   ;RVAL is floating point acc
C:      .FLT2   0.0                   ;C is floating point overflow
        .END
```

BASIC–11 Programs that Call MACRO–11 Subroutines

To call MACRO–11 subroutines from your BASIC–11 program, you must first assemble the subroutine with the MACRO–11 source code of the BASIC–11 interpreter provided in the distribution kit. You then link a new version of the BASIC–11 interpreter that allows you to call MACRO–11 subroutines, using the BASIC–11 CALL statement.

Using the BASIC–11 CALL Statement

In a BASIC–11 program, Assembly Language Routines (ALRs) are accessed using the CALL statement, which takes the form:

CALL string-expression (argument-list)

In this statement, "string-expression" is the name, enclosed in quotes or stored in a string variable, of the ALR. "Argument-list" is a list of variables that are to be passed to or from the ALR. For example, assume that the BASIC–11 interpreter has been modified so that it includes an ALR that performs single-character input from the terminal, without echo or carriage return. Assume that the routine has one argument—an integer containing the byte value of the character to be returned. You need to use the CALL statement in your BASIC program to access the ALR.

EXAMPLE

The statement:

```
1010 CALL "GETCHA" (C0%)
```

in your program places the byte value of the received character in the variable C0%.

Modifying the BASIC Interpreter

To include a tested MACRO–11 subroutine in the BASIC–11 interpreter, you must:

1. Insert the name of the routine in the user routine name table in the BSCLI.MAC file. This procedure is discussed in chapter 4 of the *BASIC–11/RT–11 User's Guide.*

2. Make sure that the argument list used in your MACRO–11 subroutine uses the same convention as

that described in the user's guide. For example, the MACRO–11 routine must take care not to include the high-order byte of the first word of the argument block, when reading the number of arguments. For BASIC–11, this high-order byte is given the value 202 (octal).

3. Assemble your routine with the MACRO–11 source files that make up the BASIC–11 kit. This procedure is discussed in chapter 4 of the *BASIC–11/RT–11 Installation Guide.*

4. Build a version of the BASIC–11 interpreter that includes support for the CALL statement. This procedure is discussed in chapter 4 of the *BASIC–11 /RT–11 Installation Guide.*

Practice 7–1 *Writing a FORTRAN IV Program that Calls a MACRO–11 Subroutine*

1. Write a FORTRAN IV program (PR0702.FOR) that does the following:

   a. Accepts a REAL*4 value from the terminal. If the value is −1, the program exits. If it is less than −32767 or more than 32767, it repeats the accept request.

   b. Calls the subroutine SUBA with the REAL*4 variable as the argument. This subroutine is written in MACRO–11 and is listed below. It performs an operation which may or may not divide the variable by 2.

2. Then print the returned value of the variable on the terminal using format F10.3 and go back to step 1. Compile your program. Create the MACRO–11 subroutine listed below, in file PR0701.MAC:

```
        .TITLE PR0701
SUBA::  MOVB    (R5),R1         ;Put number of arguments in R1
        BEQ     20$             ;Exit if no arguments
        TST     (R5)+          ;Point to 1st argument
10$:    BIC     #200,@(R5)+    ;Clear 56th bit of high order
                               ;and increment argument pointer
        DEC     R1             ;Repeat to end of argument list
        BNE     10$
20$     RTS     PC             ;Return to caller
        .END
```

3. Assemble this program using the command:

   ```
   .MAC PR0701
   ```

4. Link your program with the subroutine, and run your program.

Practice 7–2 *Writing a MACRO–11 Program that calls FORTRAN IV Subroutines*

1. Write the following FORTRAN IV INTEGER*2 function and FORTRAN IV subroutine:

   a. NINPUT. This INTEGER*2 function takes one REAL*4 argument. The subroutine accepts a REAL*4 number from the terminal, giving the prompt:

      ```
      ENTER NUMBER>
      ```

      It puts the number it reads into the REAL*4 function argument. If the number is −1.0, the function should return the value 1. Otherwise, the function value should be 0.

   b. NOUT. This subroutine receives a REAL*4 argument. It prints the value of the argument on the terminal and returns.

2. Create the following MACRO–11 program PR0704.MAC which calls both of these routines.

   ```
   .TITLE  PR0704
   .GLOBL  NINPUT,NOUT
   ```

```
PR0704::MOV     #ARG,R5    ;Point to argument block
        JSR     PC,NINPUT  ;Read a floating point number
        CMP     R0,#1      ;Is returned value = 1
        BEQ     QUIT       ;Exit if it is
        BIC     #200,RVAL  ;Clear 56th bit of high order
        MOV     #ARG,R5    ;Point to argument block
        JSR     PC,NOUT    ;Print result
        BR      PR0704     ;Repeat
QUIT:   RTS     PC         ;Return to OTSINI

RVAL:   .FLT4   0.0        ;Floating point variable
ARG:    .WORD   1,RVAL     ;Argument block
        .END
```

The logic is the same as for practice 7–1. The main program must be called by the following FORTRAN IV routine PR0703.FOR to initialize OTS.

```
      PROGRAM OTSINI
C...OTS INITIALIZATION PROGRAM FOR MACRO-11 PROGRAM
C...WHICH CALLS FORTRAN IV SUBROUTINES
      CALL PR0704
      CALL EXIT
      END
```

3. Compile your subroutines and PR0703. Assemble the MACRO–11 program using the command:

```
.MAC PR0704
```

4. Link all of the object modules together, naming the OTS initialization file first:

```
.LINK PR0703,PR0704,NINPUT,NOUT,SY:FORLIB
```

5. Run the program.

Practice 7–3

Writing a MACRO–11 Subroutine

1. Write a MACRO–11 subroutine called SUBA, to do the following:

a. Receive any number of arguments from a program. These arguments are .FLT4 format.

b. Clear the least significant bit of the exponent (bit 7 of the most significant word) of each argument, and then return. (Make sure that your subroutine will work if it is called from a program that was not written in FORTRAN IV.)

2. Create the FORTRAN IV program PR0702.FOR listed below.

```
      PROGRAM PR0702
      EXTERNAL SUBA
      REAL*4 RVAL
50    TYPE 100
100   FORMAT ('ENTER NUMBER)'$)
      READ (5,*,ERR=1110,END=1000) RVAL
      IF (RVAL .EQ. -1.0) GOTO 999
      CALL SUBA (RVAL)    !CALL MACRO-11 SUBROUTINE
      TYPE 150,RVAL
150   FORMAT('CHANGED TO)',F10.3)
      GOTO 50             !REPEAT TILL -1.0 ENTERED
999   STOP
C...NUMBER OUT OF RANGE
1000  TYPE 1010
1010  FORMAT (' ')
1100  TYPE 1110
1110  FORMAT (' ?VALUE BAD OR OUT OF RANGE')
      GOTO 50             !TRY AGAIN
      END
```

3. This program reads a REAL*4 number from the terminal and calls this subroutine. It then prints the value returned from SUBA and loops to get the next value. If the number received is -1, it exits.

4. Assemble your subroutine. Compile the FORTRAN IV program using the command:

```
.FORTRAN PR0702
```

5. Link PR0702.OBJ with your subroutine, and run the program.

Practice 7–4 *Writing a MACRO–11 Program that calls FORTRAN IV Subroutines*

1. Write a MACRO–11 program (PROG.MAC), which will call subroutines written in FORTRAN IV. Also write a FORTRAN IV routine (OTSINI.FOR) to initialize OTS for your program.

   a. The MACRO–11 program should call the subroutine NINPUT, giving it the address of a .FLT4 variable as an argument. If, on return from the subroutine, register R0 contains 1, then exit. The routine returns a floating point value in the argument specified.

   b. Clear the least significant bit of the exponent of the floating point variable (bit 7 in the most significant word).

   c. Call the subroutine NOUT, giving it the address of the floating point variable as an argument. This subroutine prints the value of the variable. On return from the subroutine go back to step (a).

2. Create the following subroutine NINPUT as file PR0705.FOR:

```
      FUNCTION NINPUT(RVAL)
C.FORTRAN FUNCTION TO ACCEPT A REAL*4 NUMBER
      INTEGER NINPUT
      REAL*4 RVAL
      DATA NINPUT/0/
50    TYPE 100
100   FORMAT(' ENTER NUMBER>'$)
      READ(5,*,ERR=1000,END=999) RVAL
      IF (RVAL .EQ. -1.0) NINPUT=1
      RETURN
999   TYPE 1099
1000  TYPE 1100
1099  FORMAT (' ')
1100  FORMAT ('?VALUE BAD OR OUT OF RANGE')
      GOTO 50
      END
```

3. Create the following subroutine NOUT as file PR0706.FOR:

```
      SUBROUTINE NOUT(RVAL)
C...SUBROUTINE TO OUTPUT A REAL*4 NUMBER
      REAL*4 RVAL
      TYPE 1000,RVAL
      RETURN
1000  FORMAT(' ',F10.3)
      END
```

4. Assemble your MACRO–11 program and compile the OTS initialization routines PR0705.FOR and PR0706.FOR. Link all of the object modules together, naming the OTS initialization file first, for example:

```
.LINK OTSINI,PROG,PR0705,PR0706
```

5. Run the program.

Practice 7–5

Using a BASIC–11 Program to Call a MACRO–11 Subroutine

1. Create the following BASIC–11 program PR0708.BAS, designed to call the assembly language routine SUBA which you wrote in practice 7–3.

```
300 REM PR0708.BAS
310 REM
1000 PRINT "ENTER NUMBER>";
1010 INPUT R
1020 IF R=-1 GOTO 32767
1030 CALL SUBA (R)
1040 PRINT R
1050 GOTO 1000
32767 END
```

2. Produce a new version of the BASIC–11 interpreter, calling it MYBAS.SAV, to include the subroutine

SUBA. You will have to create a new version of BSCLI.MAC. When you do this, make a copy of BSCLI.MAC and call it NEWBCL.MAC instead of editing the original source file.

3. Run the program PR0708.BAS. It asks you to enter a number. If the number is −1, then the program will exit. Otherwise it will call SUBA, passing it the number as an argument. On return it will output the number and then repeat the process.

References

RT–11 Programmer's Reference Manual. Chapter 1 discusses in detail the interfaces between MACRO–11 and FORTRAN IV programs and offers programming examples.

BASIC–11/RT–11 Language Reference Manual. Chapter 8 explains how to use assembly language routines with BASIC.

BASIC–11/RT–11 User's Guide. Chapter 4 describes the use of assembly language routines with BASIC.

BASIC–11/RT–11 Installation Guide. Chapter 4 details the process for installing different versions of the BASIC interpreter.

Solutions to Practices

CHAPTER 1

1-1. MACRO-11

```
.EDIT/CREATE PR0101.MAC

.MACRO PR0101

.MACRO/OBJECT:MESS/LIST:MESS PR0101
```

1-1. FORTRAN IV

```
.EDIT/CREATE PR0101.FOR

.FORTRAN PR0101

.FORTRAN/LIST:MESS/OBJECT:MESS PR0101
```

1-2. MACRO-11

```
.EDIT/CREATE PR0102.MAC

.EDIT/CREATE PR0103.MAC

.EDIT/CREATE PR0104.MAC

.MACRO/LIST:PR1234/OBJ:PR1234 PR0102+PR0103+PR0104
```

1–3. FORTRAN IV

```
.EDIT/CREATE PR0102.FOR

.EDIT/CREATE PR0103.FOR

.EDIT/CREATE PR0104.FOR

.FORT/LIST:PR1234/OBJ:PR1234 PR0102+PR0103+PR0104
```

1–4. MACRO–11

```
.MACRO PR0101/LIST/SHOW:SRC:COM:MD:MC:ME
```

1–5. MACRO–11

```
.EDIT/CREATE PR0105.MAC

.MACRO PR0105

.LINK PR0105/MAP
```

```
                 e
RT-11 LINK  V08.01      Load Map     Friday 10-Feb-84 11:05   Page 1
PR0105.SAV       Title:  PR0105  Ident:         b          c
     a                     d
Section  Addr   Size     Global  Value   Global  Value   Global  Value

 . ABS.  000000 001000 = 256.    words  (RW,I,GBL,ABS,OVR)
         001000 000032 = 13.     words  (RW,I,LCL,REL,CON)
              f
Transfer address = 001022, High limit = 001030 = 268.    words
                      g                        h
```

1–5. FORTRAN IV

```
.EDIT/CREATE PR0105.FOR

.FORTRAN PR0105

.LINK PR0105/MAP,SY:FORLIB/LIBRARY
```

```
                   e                                           b
RT-11 LINK  V08.01        Load Map       Friday 10-Feb-84 11:06   Page 1
PR0105.SAV       Title:  .MAIN.  Ident:  FORV02                c
     a                     d
Section  Addr    Size     Global   Value    Global   Value    Global   Value
             f
 . ABS.  000000 001000  = 256.     words  (RW,I,GBL,ABS,OVR)
                          $USRSW   000000   $RF2A1   000000   $HRDWR   000000
                          .VIR     000001   $NLCHN   000006   $WASIZ   000152
                          $LRECL   000210   $TRACE   004737
 OTS$I   001000 013744  = 3058.    words  (RW,I,LCL,REL,CON)
                          $$OTSI   001000   $OTI     001026   $$OTI    001030
                          $SETOP   001240   $$SET    002712   $OPNER   003206
                          $CHKER   003244   $IOEXI   003270   $EOL     003336
                          EOL$     003340   IFW$     003510   $IFW     003514
                          $$IFW    003520   IFW$$    003556   MOI$SS   003626
                          MOL$SS   003626   MOI$SM   003632   MOI$SA   003636
                          MOI$IS   003642   MOL$IS   003642   REL$     003642
                          MOI$IM   003646   MOI$IA   003652   MOI$MS   003656
                          MOI$MM   003662   MOI$MA   003666   MOI$0S   003672
                          MOI$0M   003676   MOI$0A   003702   MOI$1S   003706
                          MOI$1M   003714   MOI$1A   003722   ISN$     003730
                          $ISNTR   003734   LSN$     003750   $LSNTR   003754
                          RET$L    004110   RET$F    004114   RET$I    004122
                          RET$     004124   $INITI   004160   $CLOSE   004276
                          $ERRTB   005054   $ERRS    005161   $FCHNL   010722
                          $FIO     011564   $$FIO    011570   $PUTRE   012734
                          $PUTBL   013242   $GETBL   013452   $EOFIL   013636
                          $EOF2    013652   SAVRG$   013672   THRD$    014050
                          $STPS    014052   STP$     014060   $STP     014060
                          FOO$     014064   $EXIT    014104   $WAIT    014230
                          $VRINT   014272   $DUMPL   014574
 OTS$P   014744 000054  = 22.      words  (RW,D,GBL,REL,OVR)
 SYS$I   015020 000000  = 0.       words  (RW,I,LCL,REL,CON)
 USER$I  015020 000000  = 0.       words  (RW,I,LCL,REL,CON)
 $CODE   015020 000032  = 13.      words  (RW,I,LCL,REL,CON)
                          $$OTSC   015020
 OTS$O   015052 001140  = 304.     words  (RW,I,LCL,REL,CON)
                          $$OTSO   015052   $OPEN    015052
 SYS$D   016212 000000  = 0.       words  (RW,I,LCL,REL,CON)
 $DATAP  016212 000040  = 16.      words  (RW,D,LCL,REL,CON)
 OTS$D   016252 000010  = 4.       words  (RW,D,LCL,REL,CON)
                          NHCLN$   016256
 OTS$S   016262 000002  = 1.       words  (RW,D,LCL,REL,CON)
                          $AOTS    016262
 SYS$S   016264 000000  = 0.       words  (RW,D,LCL,REL,CON)
 $DATA   016264 000000  = 0.       words  (RW,D,LCL,REL,CON)
 USER$D  016264 000000  = 0.       words  (RW,D,LCL,REL,CON)
 .$$$$.  016264 000000  = 0.       words  (RW,D,GBL,REL,OVR)

Transfer address = 015020, High limit = 016262 = 3673.  words
                       g                        h
```

CHAPTER 2

2–1. Foreground/Background Jobs

```
.EDIT/CREATE PR0201.MAC

.EDIT/CREATE PR0202.FOR

.MACRO PR0201

.LINK PR0201/FOREGROUND

.FORTRAN PR0202

.LINK PR0202,SY:FORLIB/LIBRARY

.SET USR NOSWAP

.FRUN PR0201

F>
PR0201-I, TEXT: ^FDATA FOR FOREGROUND JOB.

B>

.^BRUN PR0202
PR0202-I, Enter your data (-1 to finish): 56
PR0202-I, Accepted data as:   56.00000
PR0202-I, Enter your data (-1 to finish): 34
PR0202-I, Accept
F>
PR0201-I, Finished processing text: DATA FOR FOREGROUND JOB.
PR0201-I, TEXT: ^FTHIS IS LINE TWO!

B>
ed data as:   34.00000
PR0202-I, Enter your data (-1 to finish): ^B-1

STOP --

.
F>
PR0201-I, Finished processing text: THIS IS LINE TWO!
PR0201-I, TEXT: ^F
B>

.
F>

PR0201-I, Normal successful completion

B>

.UNLOAD F
```

CHAPTER 3

3-1. BASIC-11

```
.R BASIC
BASIC-11/RT-11 V02-03
OPTIONAL FUNCTIONS (ALL, NONE, OR INDIVIDUAL)? A

READY
NEW PR0301

READY
10 PRINT "WHAT IS YOUR GAME?"
20 INPUT #0,A$
30 A=SYS(4)
SAVE PR0301

READY
OLD PR0301

READY
LIST

PR0301    10-FEB-84  10:02:33
10 PRINT "WHAT IS YOUR GAME?";
20 INPUT #0,A$
30 A=SYS(4)

READY
RUN

PR0301    10-FEB-84  10:02:40
WHAT IS YOUR GAME?^C

STOP AT LINE 20

READY
SUB 10!GAME!NAME
10 PRINT "WHAT IS YOUR NAME?"

READY
COMPILE PR0301

READY
BYE

.R BASIC
BASIC-11/RT-11 V02-03
OPTIONAL FUNCTIONS (ALL, NONE, OR INDIVIDUAL)? A

READY
RUN PR0301
WHAT IS YOUR NAME?A.N. OTHER
```

CHAPTER 4

4–1. MACRO–11

```
.EDIT PR0403.MAC
        .TITLE  PR0403  Debugging Exercise
        .MACRO  MONTH,NAME      ;Macro to set up month table
        .PSECT  MOVNAM
        .$$.=.                  ;Each entry points to string
        .ASCII  /NAME/<200>     ;This is the string
        .PSECT
        .WORD   .$$.            ;This is the space for entry
        .ENDM

        .MCALL  .PRINT,.EXIT,.GTLIN
        .GLOBL  CNVSTR,HISPRT   ;Declare subroutines

MTAB::  MONTH   JAN             ;Build months table
        MONTH   FEB
        MONTH   MAR
        MONTH   APR
        MONTH   MAY
        MONTH   JUN
        MONTH   JUL
        MONTH   AUG
        MONTH   SEP
        MONTH   OCT
        MONTH   NOV
        MONTH   DEC

START:  MOV     #MTAB,R2        ;Get address of months table
        .PRINT  #INTRO          ;Print introduction
        MOV     #12.,R3         ;Initialize month loop
        MOV     #HEIGHT,R4      ;Get address of heights table
LOOP:   .PRINT  (R2)+           ;Mth part of prompt for month
        .GTLIN  #INB,#PROMPT    ;Get decimal number string
        MOV     #INB,R5         ;Get address of input buffer
        JSR     PC,CNVSTR       ;Convert string to binary
        MOV     R0,R1           ; ** Copy returned value (DIV)
        CMP     #-1.,R0         ;Check returned value for -1
        BEQ     BADVAL          ;If so branch past height calc
        CLR     R0              ; ** Clear high order of value
        DIV     #5,R0           ;Convert value to height
BADVAL: MOVB    R0,(R4)+        ;Place height in height table
        SOB     R3,LOOP         ;Branch for next month
        MOV     #HEIGHT,R5      ;Pass address of height table
        JSR     PC,HISPRT       ;Output the histogram
        .EXIT

INTRO:  .ASCII  /THIS PROGRAM PRINTS A HISTOGRAM FROM 12 /
        .ASCII  /MONTHLY VALUES./<15><12>
        .ASCII  /THE MONTHS ARE JANUARY TO DECEMBER./<15><12>
        .ASCII  /PLEASE ENTER YOUR TWELVE VALUES:/
        .ASCIZ  /THEY MUST BE IN THE RANGE 0 TO 100/
PROMPT: .ASCIZ  /: /<200>
VALUE:  .BLKB   12.*4.
HEIGHT: .BLKB   12.
INB:    .BLKB   81.
VAL:    .BYTE   -1.
        .EVEN
        .END    START
```

```
.MACRO PR0403,PR0404,PR0405

.LINK/EXEC:HISTO PR0503,PR0404,PR0405

.RUN HISTO
THIS PROGRAM PRINTS A HISTOGRAM FROM 12 MONTHLY VALUES.
THE MONTHS ARE JANUARY TO DECEMBER.
PLEASE ENTER YOUR TWELVE VALUES: THEY MUST BE IN THE RANGE 0 TO 100
JAN: BAD
FEB: 100
MAR: 90
APR: 80
MAY: 70
JUN: 60
JUL: 50
AUG: 40
SEP: 30
OCT: 20
NOV: 10
DEC: 0


100!    ###
  -!    ###
 90!    ### ###
  -!    ### ###
 80!    ### ### ###
  -!    ### ### ###
 70!    ### ### ### ###
  -!    ### ### ### ###
 60!    ### ### ### ### ###
  -!    ### ### ### ### ###
 50!    ### ### ### ### ### ###
  -!    ### ### ### ### ### ###
 40!    ### ### ### ### ### ### ###
  -!    ### ### ### ### ### ### ###
 30!    ### ### ### ### ### ### ### ###
  -!    ### ### ### ### ### ### ### ###
 20!    ### ### ### ### ### ### ### ### ###
  -!    ### ### ### ### ### ### ### ### ###
 10!    ### ### ### ### ### ### ### ### ### ###
  -!    ### ### ### ### ### ### ### ### ### ###
  0+------------------------------------------------
    JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC
```

4–1. FORTRAN IV

```
.EDIT PR0404.FOR
C ************************************************************************
C
C DEBUGGING AND FAILURE ANALYSIS
C
C PRACTICE 4-1, PR0404.FOR
C
C ************************************************************************
C
      FUNCTION CNVSTR(STRING)
      BYTE STRING(8),CURCHA
      INTEGER POINTA,DIGIT
      REAL CNVSTR,DIV
C
C BAD VALUES ARE SET TO -1.0
C VALUES OUT OF RANGE ARE TREATED AS BAD
C ------------------------------------
C
C INITIALIZE RETURN VALUE AND POINTER INTO STRING
C ----------------------------------------------
      CNVSTR=0.0
      POINTA=1
C
C PROCESS EACH CHARACTER, STRING IS TERMINATED BY SPACE OR LENGTH=8
C ---------------------------------------------------------------
10    IF (POINTA .GT. 8) GO TO 100
      CURCHA=STRING(POINTA)
      IF (CURCHA .EQ. ' ') GO TO 100    ! ** CORRECTION (GT --> EQ)
      IF (CURCHA .GT. '9') GO TO 50
      IF (CURCHA .LT. '0') GO TO 50
      DIGIT=CURCHA-'0'
      CNVSTR=(10.0*CNVSTR)+DIGIT
      POINTA=POINTA+1
      GO TO 10
50    IF (CURCHA .NE. '.') GO TO 200
      DIV=1.0
75    POINTA=POINTA+1
      IF (POINTA .GT. 8) GO TO 100
      CURCHA=STRING(POINTA)
      IF (CURCHA .GT. ' ') GO TO 100
      IF (CURCHA .GT. '9') GO TO 200
      IF (CURCHA .LT. '0') GO TO 200
      DIV=DIV*10.0
      DIGIT=CURCHA-'0'
      CNVSTR=CNVSTR+DIGIT/DIV
      GO TO 75
C
C BRANCH TO HERE AT END OF STRING PROCESSING
C ----------------------------------------
100   IF (CNVSTR .GT. 100.0) GO TO 200
      RETURN
C
C BRANCH TO HERE IF VALUE IS BAD
C ----------------------------
200   CNVSTR=-1.0
      RETURN
      END
```

```
.FORTRAN PR0403,PR0404,PR0405

.LINK PR0403,PR0404,PR0405,SY:FORLIB/LIBRARY

.RUN PR0403
THIS PROGRAM PRINTS A HISTOGRAM FROM 12 MONTHLY VALUES.
THE MONTHS ARE JANUARY TO DECEMBER.
PLEASE ENTER YOUR TWELVE VALUES: THEY MUST BE IN THE RANGE 0 TO 100
JAN: BAD
FEB: 100
MAR: 90
APR: 80
MAY: 70
JUN: 60
JUL: 50
AUG: 40
SEP: 30
OCT: 20
NOV: 10
DEC: 0


100!    ###
  -!    ###
 90!    ### ###
  -!    ### ###
 80!    ### ### ###
  -!    ### ### ###
 70!    ### ### ### ###
  -!    ### ### ### ###
 60!    ### ### ### ### ###
  -!    ### ### ### ### ###
 50!    ### ### ### ### ### ###
  -!    ### ### ### ### ### ###
 40!    ### ### ### ### ### ### ###
  -!    ### ### ### ### ### ### ###
 30!    ### ### ### ### ### ### ### ###
  -!    ### ### ### ### ### ### ### ###
 20!    ### ### ### ### ### ### ### ### ###
  -!    ### ### ### ### ### ### ### ### ###
 10!    ### ### ### ### ### ### ### ### ### ###
  -!    ### ### ### ### ### ### ### ### ### ###
  0+------------------------------------------------
    JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC
```

4-1. BASIC-11

```
.R BASIC
BASIC-11/RT-11 V02-03
OPTIONAL FUNCTIONS (ALL, NONE, OR INDIVIDUAL)? A

READY
OLD PR0403

READY
SUB 340!I)!I%)
340 H%(I%)=FNA%(V%(I%)) \ NEXT I%

READY
LIST

PR0403    10-FEB-84  15:34:18
10 REM ******************************************************************
20 REM
30 REM DEBUGGING AND FAILURE ANALYSIS
40 REM
50 REM PRACTICE 4-1,  PR0403.BAS
60 REM
70 REM ******************************************************************
80 REM
90 REM  INITIALIZE VARIABLES AND ARRAYS
100 REM -----------------------------
110 DIM V(12%)
120 DIM M$(12%)
130 DIM H%(12%)
140 REM
150 REM MAIN PROGRAM LOGIC
160 REM *****************
170 REM
180 REM READ MONTH STRINGS INTO ARRAY
190 REM ---------------------------
200 FOR I%=1% TO 12% \ READ M$(I%) \ NEXT I%
210 REM
220 REM PRINT INSTRUCTIONS
230 REM -----------------
240 PRINT "THIS PROGRAM PRINTS A HISTOGRAM FROM 12 MONTHLY VALUES."
250 PRINT "THE MONTHS ARE JANUARY TO DECEMBER."
260 PRINT "PLEASE ENTER YOUR TWELVE VALUES: THEY MUST BE IN THE RANGE 0 TO 100"
270 REM
280 REM ACCEPT VALUES AS STRINGS, PROMPTED BY THE MONTH
290 REM ---------------------------------------------
300 FOR I%=1% TO 12% \ PRINT M$(I%)": "; \ LINPUT #0%,V$ \ GOSUB 10000
310 REM
320 REM CONVERT EACH VALUE INTO A HEIGHT INTEGER
330 REM --------------------------------------
340 H%(I%)=FNA%(V(I%)) \ NEXT I%
350 REM
360 REM DISPLAY HISTOGRAM
370 REM ----------------
380 GOSUB 11000
390 REM
400 REM END OF MAIN PROGRAM LOGIC
410 REM ************************
420 GO TO 32767
```

```
10000 REM
10010 REM SUBROUTINE TO CONVERT STRING INTO A REAL NUMBER
10020 REM ===============================================
10030 REM BAD VALUES ARE SET TO -1.0
10040 REM VALUES OUT OF RANGE ARE TREATED AS BAD
10050 V(I%)=0%
10060 L%=LEN(V$) \ F$=SEG$(V$,1%,1%) \ IF F$<>"" THEN V$=SEG$(V$,2%,L%)
10070 IF F$="" THEN 10180
10080 IF F$>"9" THEN 10110
10090 IF F$<"0" THEN 10110
10100 V(I%)=10*V(I%)+VAL(F$) \ GO TO 10060
10110 IF F$<>"." THEN V(I%)=-1 \ GO TO 10180
10120 D=1
10130 D=D*10 \ L%=LEN(V$) \ F$=SEG$(V$,1%,1%) \ IF F$<>"" THEN V$=SEG$(V$,2%,L%)
10140 IF F$="" THEN 10180
10150 IF F$>"9" THEN V(I%)=-1% \ GO TO 10180
10160 IF F$<"0" THEN V(I%)=-1% \ GO TO 10180
10170 V(I%)=V(I%)+VAL(F$)/D \ GO TO 10130
10180 IF V(I%)>100 THEN V(I%)=-1
10190 RETURN
11000 REM
11010 REM PRINT HISTOGRAM
11020 REM ===============
11030 PRINT  \ PRINT  \ PRINT
11040 FOR I%=20% TO 1% STEP -1%
11050 I$=STR$(I%*5) \ IF I%<20% THEN I$=" "+I$
11060 IF 2%*(I%/2%)=I% THEN PRINT I$; \ GO TO 11080
11070 PRINT "  -";
11080 PRINT "I";
11090 FOR J%=1% TO 12%
11100 IF H%(J%)=I% THEN PRINT " ###"; \ H%(J%)=H%(J%)-1% \ GO TO 11140
11110 IF I%<>1% THEN 11130
11120 IF H%(J%)=-1% THEN PRINT " BAD"; \ GO TO 11140
11130 PRINT "    ";
11140 NEXT J% \ PRINT  \ NEXT I%
11150 PRINT "  0+"; \ FOR I%=1% TO 12% \ PRINT "----"; \ NEXT I% \ PRINT
11160 PRINT "    "; \ FOR I%=1% TO 12% \ PRINT " ";M$(I%); \ NEXT I% \ PRINT
11170 RETURN
15000 REM
15010 REM FUNCTION TO CALCULATE HEIGHT
15020 REM ============================
15030 DEF FNA%(X)=INT(X*20/100)
20000 REM
20010 REM DATA DECLARATION FOR MONTH STRING ARRAY
20020 REM =======================================
20030 DATA JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC
32000 REM
32010 REM END OF PROGRAM
32020 REM ==============
32767 END
```

```
READY
RUN

PR0403    10-FEB-84  15:34:47
THIS PROGRAM PRINTS A HISTOGRAM FROM 12 MONTHLY VALUES.
THE MONTHS ARE JANUARY TO DECEMBER.
PLEASE ENTER YOUR TWELVE VALUES: THEY MUST BE IN THE RANGE 0 TO 100
JAN: BAD
FEB: 100
MAR: 90
APR: 80
MAY: 70
JUN: 60
JUL: 50
AUG: 40
SEP: 30
OCT: 20
NOV: 10
DEC: 0


100!    ###
  -!    ###
 90!    ### ###
  -!    ### ###
 80!    ### ### ###
  -!    ### ### ###
 70!    ### ### ### ###
  -!    ### ### ### ###
 60!    ### ### ### ### ###
  -!    ### ### ### ### ###
 50!    ### ### ### ### ### ###
  -!    ### ### ### ### ### ###
 40!    ### ### ### ### ### ### ###
  -!    ### ### ### ### ### ### ###
 30!    ### ### ### ### ### ### ### ###
  -!    ### ### ### ### ### ### ### ###
 20!    ### ### ### ### ### ### ### ### ###
  -!    ### ### ### ### ### ### ### ### ###
 10!    ### ### ### ### ### ### ### ### ### ###
  -!    ### ### ### ### ### ### ### ### ### ###
  0+-----------------------------------------------
    JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC

READY
REPLACE PR0403

READY
```

CHAPTER 5

5-1. FORTRAN IV

```
.EDIT/CREATE PR0501.FOR

.EDIT/CREATE PR0502.FOR

.EDIT/CREATE PR0503.FOR

.EDIT/CREATE PR0504.FOR

.FORTRAN PR0501,PR0502,PR0503,PR0504

.LIBRARY/CREATE TEXLIB PR0502,PR0503

.LIBRARY TEXLIB PR0504/INSERT

.LIBRARY/LIST:TT: TEXLIB

.LINK PR0501,TEXLIB/LIB,SY:FORLIB/LIB

.RUN PR0501
Dear Mr Griffiths,


During our last quarter, our records showed that you owed us
16.27.

We have been expecting your payment since July 10th 1977.
Accordingly we sent you a letter of invoice at that time.
Regrettably this was an error on our part.

PLEASE SEND THE MONEY STRAIGHT AWAY.

Yours sincerely,



A.N. Other (Manager)

.EDIT PR0504.FOR

.FORTRAN PR0504

.LIBRARY TEXLIB PR0504/REPLACE

.LINK PR0501,TEXLIB/LIB,SY:FORLIB/LIB

.RUN PR0501
Dear Mr Griffiths,


During our last quarter, our records showed that you owed us
```

```
16.27.

We have been expecting your payment since July 10th 1977.
Accordingly we sent you a letter of invoice at that time.
Regrettably this was an error on our part.

PLEASE ACCEPT OUR APOLOGIES.

Yours sincerely,



A.N. Other (Manager)
```

5–2. MACRO–11

```
.EDIT/CREATE PR0505.MAC

.EDIT/CREATE PR0506.MAC

.EDIT/CREATE PR0507.MAC

.EDIT/CREATE PR0508.FOR

.LIBRARY/MACRO/CREATE PRMACS PR0506,PR0507

.MACRO/OBJECT:PR0505 PRMACS/LIBRARY+PR0505

.FORTRAN PR0508

.LINK PR0508,PR0505,TEXLIB,SY:FORLIB

.RUN PR0508
Dear Mr Griffiths,


During our last quarter, our records showed that you owed us
16.27.

We have been expecting your payment since July 10th 1977.
Accordingly we sent you a letter of invoice at that time.
Regrettably this was an error on our part.

PLEASE ACCEPT OUR APOLOGIES.

Yours sincerely,



A.N. Other (Manager)
```

CHAPTER 6

6-1. MACRO-11

```
.MACRO PR0403,PR0404,PR0405

.R LINK
*HISTO,HISTO=PR0403//
*PR0404/O:1
*PR0405/O:1//
*^C
```

```
RT-11 LINK  V08.01      Load Map       Friday 10-Feb-84 11:10  Page 1
HISTO .SAV      Title:  PR0403  Ident:

Section  Addr   Size    Global  Value   Global  Value   Global  Value

 . ABS.  000000 001000 = 256.   words  (RW,I,GBL,ABS,OVR)
                        .OHAND  000000
 $OHAND  001000 000106 = 35.    words  (RW,I,GBL,REL,CON)
                        $OVRH   001002  O$READ  001024  O$DONE  001036
                        $ODF1   001102  $ODF2   001104
 $OTABL  001106 000034 = 14.    words  (RW,D,GBL,REL,OVR)
         001142 000626 = 203.   words  (RW,I,LCL,REL,CON)
                        MTAB    001142
 MOVNAM  001770 000060 = 24.    words  (RW,I,LCL,REL,CON)
 Segment size = 002050 = 532.   words

Overlay region  000001  Segment 000001
         002052 000126 = 43.    words  (RW,I,LCL,REL,CON)
                        CNVSTR 002052
 Segment size = 000126 = 43.    words

Overlay region  000001  Segment 000002
         002052 000376 = 127.   words  (RW,I,LCL,REL,CON)
                        HISPRT 002052
 Segment size = 000376 = 127.   words

Transfer address = 001172, High limit = 002446 = 659.   words
```

6-1. FORTRAN IV

```
.FORTRAN PR0403,PR0404,PR0405

.R LINK
*HISTO,HISTO=PR0403,SY:FORLIB//
*PR0404/O:1
*PR0405/O:1//
*^C
```

```
RT-11 LINK  V08.01      Load Map        Friday 10-Feb-84 11:13  Page 1
HISTO .SAV      Title:  .MAIN.  Ident:  FORV02

Section  Addr   Size    Global  Value   Global  Value   Global  Value

 . ABS.  000000 001000 = 256.   words  (RW,I,GBL,ABS,OVR)
                        $USRSW  000000  $RF2A1  000000  $HRDWR  000000
                        .OHAND  000000  .VIR    000001  $NLCHN  000006
                        $WASIZ  000152  $LRECL  000210  $TRACE  004737
 $OHAND  001000 000106 = 35.    words  (RW,I,GBL,REL,CON)
                        $OVRH   001002  O$READ  001024  O$DONE  001036
                        $ODF1   001102  $ODF2   001104
 $OTABL  001106 000034 = 14.    words  (RW,D,GBL,REL,OVR)
 OTS$I   001142 017752 = 4085.  words  (RW,I,LCL,REL,CON)
                        $$OTSI  001142  $CVTFB  001142  $CVTFI  001142
                        $CVTCB  001156  $CVTCI  001156  $CVTDB  001156
                        $CVTDI  001156  CIC$    001170  CID$    001170
                        CLC$    001170  CLD$    001170  $DI     001170
                        CIF$    001200  CLF$    001200  $RI     001200
                        CIL$    001312  CLI$    001316  DIF$PS  001320
                        DIF$MS  001324  DIF$IS  001334  $DIVF   001342
                        DIF$SS  001354  $DVR    001354  MUF$PS  001642
                        MUF$MS  001646  MUF$IS  001656  $MULF   001664
                        MUF$SS  001676  $MLR    001676  $OTI    002234
                        $$OTI   002236  $SETOP  002446  $$SET   004120
                        $INITI  004414  NMI$1M  004532  NMI$1I  004544
                        BLE$    004554  BEQ$    004556  BGT$    004564
                        BGE$    004566  BRA$    004570  BNE$    004574
                        BLT$    004576  CAI$    004606  CAL$    004614
                        $CLOSE  004644  CMF$PS  005422  CMF$MS  005426
                        CMF$IS  005442  $CMPF   005450  CMF$SS  005462
                        $CMR    005462  CMF$PI  005474  CMF$MI  005500
                        CMF$II  005510  CMF$SI  005514  CMF$PP  005526
                        CMF$MP  005532  CMF$IP  005542  CMF$SP  005546
                        CMF$PM  005556  CMF$MM  005562  CMF$IM  005572
                        CMF$SM  005576  $DUMPL  005630  END$    006000
                        ERR$    006012  $END    006024  $ERR    006042
                        $OPNER  006064  $CHKER  006122  $IOEXI  006146
                        $EOL    006214  EOL$    006216  $ERRTB  006366
                        $ERRS   006473  EXIT    012234  $FCHNL  012240
                        $FIO    013102  $$FIO   013106  MOF$MS  014252
                        MOF$PS  014264  MOF$RS  014270  MOF$RM  014276
                        MOF$RA  014306  MOF$RP  014312  $GETRE  014316
                        $TTYIN  014372  ADI$SS  014552  ADI$SA  014556
                        ADI$SM  014562  ADI$IS  014566  ADI$IA  014572
                        ADI$IM  014576  ADI$MS  014602  ADI$MA  014606
                        ADI$MM  014612  CMI$SS  014616  CMI$SI  014622
                        CMI$SM  014626  CMI$IS  014632  CMI$II  014636
                        CMI$IM  014642  CMI$MS  014646  CMI$MI  014652
                        CMI$MM  014656  IFR$    014662  $IFR    014666
                        $$IFR   014672  IFR$$   014724  IFW$    014746
                        $IFW    014752  $$IFW   014756  IFW$$   015014
                        ILW$    015064  $ILW    015070  TVS$    015210
                        $TVS    015212  MOI$SS  016024  MOL$SS  016024
                        MOI$SM  016030  MOI$SA  016034  MOI$IS  016040
                        MOL$IS  016040  REL$    016040  MOI$IM  016044
                        MOI$IA  016050  MOI$MS  016054  MOI$MM  016060
                        MOI$MA  016064  MOI$0S  016070  MOI$0M  016074
                        MOI$0A  016100  MOI$1S  016104  MOI$1M  016112
                        MOI$1A  016120  ICI$S   016126  ICI$M   016132
```

```
RT-11 LINK  V08.01     Load Map        Friday 10-Feb-84 11:13  Page 2
HISTO .SAV      Title:  .MAIN.  Ident:  FORV02

                        ICI$P   016136  ICI$A   016140  DCI$S   016144
                        DCI$M   016150  DCI$P   016154  DCI$A   016156
                        MOI$IP  016162  MOI$SP  016164  MOI$PP  016172
                        MOI$MP  016176  MOI$PS  016206  MOI$PM  016214
                        MOI$PA  016222  MOI$0P  016230  MOI$1P  016236
                        ISN$    016246  $ISNTR  016252  LSN$    016266
                        $LSNTR  016272  MOL$SM  016426  MOL$SA  016432
                        MOL$MS  016436  MOL$MM  016446  MOL$MA  016452
                        MOL$SP  016456  MOL$PP  016464  MOL$MP  016470
                        MOL$PM  016500  MOL$PS  016506  MOL$PA  016512
                        MOL$IM  016520  MOL$IA  016526  MOL$IP  016534
                        $PUTRE  016544  RET$L   017052  RET$F   017056
                        RET$I   017064  RET$    017066  $PUTBL  017122
                        $GETBL  017332  $EOFIL  017516  $EOF2   017532
                        SAVRG$  017552  THRD$   017730  $STPS   017732
                        STP$    017740  $STP    017740  FOO$    017744
                        $EXIT   017764  $OTIS   020110  $$OTIS  020112
                        TVL$    020232  $TVL    020232  TVF$    020240
                        $TVF    020240  TVD$    020246  $TVD    020246
                        TVQ$    020254  $TVQ    020254  TVP$    020262
                        $TVP    020262  TVI$    020270  $TVI    020270
                        $WAIT   020424  $VRINT  020466  SAF$IM  020770
                        SAF$SM  020772  SVF$IM  021002  SVF$SM  021004
                        SAF$MM  021024  SVF$MM  021030  SAI$IM  021034
                        SAI$SM  021036  SVI$IM  021044  SVI$SM  021046
                        SAI$MM  021056  SVI$MM  021062  SAL$IM  021066
                        SAL$SM  021070  SVL$IM  021074  SVL$SM  021076
                        SAL$MM  021104  SVL$MM  021110
 OTS$P    021114 000054 = 22.   words  (RW,D,GBL,REL,OVR)
 SYS$I    021170 000000 = 0.    words  (RW,I,LCL,REL,CON)
 USER$I   021170 000000 = 0.    words  (RW,I,LCL,REL,CON)
 $CODE    021170 000506 = 163.  words  (RW,I,LCL,REL,CON)
                        $$OTSC  021170
 OTS$O    021676 001140 = 304.  words  (RW,I,LCL,REL,CON)
                        $$OTSO  021676  $OPEN   021676
 SYS$O    023036 000000 = 0.    words  (RW,I,LCL,REL,CON)
 $DATAP   023036 000310 = 100.  words  (RW,D,LCL,REL,CON)
 OTS$D    023346 000010 = 4.    words  (RW,D,LCL,REL,CON)
                        NHCLN$  023352
 OTS$S    023356 000002 = 1.    words  (RW,D,LCL,REL,CON)
                        $AOTS   023356
 SYS$S    023360 000000 = 0.    words  (RW,D,LCL,REL,CON)
 $DATA    023360 000220 = 72.   words  (RW,D,LCL,REL,CON)
 USER$D   023600 000000 = 0.    words  (RW,D,LCL,REL,CON)
 .$$$$.   023600 000000 = 0.    words  (RW,D,GBL,REL,OVR)
 Segment size = 023600 = 5056.  words

Overlay region  000001  Segment 000001
 OTS$I    023602 001134 = 302.  words  (RW,I,LCL,REL,CON)
                        ADF$IM  023602  ADF$PM  023610  SUF$PM  023614
                        SUF$MM  023620  ADF$MM  023632  SUF$IM  023642
                        SUF$SM  023646  ADF$SM  023652  $CVTIF  023672
                        $CVTIC  023706  $CVTID  023706  CCI$    023720
                        CDI$    023720  $IC     023720  $ID     023720
                        CFI$    023734  $IR     023734  ADF$IS  024020
                        ADF$PS  024026  SUF$PS  024032  SUF$MS  024036
                        ADF$MS  024050  SUF$IS  024060  $ADDF   024066
```

```
RT-11 LINK  V08.01      Load Map        Friday 10-Feb-84 11:13  Page 3

HISTO .SAV      Title:  .MAIN.  Ident:  FORV02

                        $SUBF     024102  SUF$SS    024114  $SBR      024114
                        ADF$SS    024120  $ADR      024120  ADD$      024134
                        MOF$SM    024560  MOF$SP    024570  MOF$IM    024574
                        MOF$IP    024606  MOF$OM    024614  MOF$OA    024624
                        MOF$OP    024630  SUI$SS    024634  SUI$SA    024640
                        SUI$SM    024644  SUI$IS    024650  SUI$IA    024654
                        SUI$IM    024660  SUI$MS    024664  SUI$MA    024670
                        SUI$MM    024674  CML$MI    024700  CML$SI    024702
                        SAL$IP    024710  SAL$SP    024712  SVL$IP    024716
                        SVL$SP    024720  SAL$MP    024726  SVL$MP    024732
 SYS$I     024736 000000 = 0.     words  (RW,I,LCL,REL,CON)
 USER$I    024736 000000 = 0.     words  (RW,I,LCL,REL,CON)
 $CODE     024736 000522 = 169.   words  (RW,I,LCL,REL,CON)
                        CNVSTR 024736
 OTS$O     025460 000000 = 0.     words  (RW,I,LCL,REL,CON)
 SYS$O     025460 000000 = 0.     words  (RW,I,LCL,REL,CON)
 $DATAP    025460 000032 = 13.    words  (RW,D,LCL,REL,CON)
 OTS$D     025512 000000 = 0.     words  (RW,D,LCL,REL,CON)
 OTS$S     025512 000000 = 0.     words  (RW,D,LCL,REL,CON)
 SYS$S     025512 000000 = 0.     words  (RW,D,LCL,REL,CON)
 $DATA     025512 000020 = 8.     words  (RW,D,LCL,REL,CON)
 USER$D    025532 000000 = 0.     words  (RW,D,LCL,REL,CON)
 Segment size = 001730 = 492.     words

Overlay region  000001  Segment 000002
 OTS$I     023602 000766 = 251.   words  (RW,I,LCL,REL,CON)
                        DII$PS    023602  DII$MS    023610  DII$IS    023614
                        DII$SS    023616  $DVI      023616  OCI$      023726
                        ICI$      023734  $ECI      023750  OCO$      024130
                        ICO$      024136  CMI$IP    024334  CMI$SP    024336
                        CMI$MP    024344  CMI$PP    024354  CMI$PS    024360
                        CMI$PI    024366  CMI$PM    024374  NMI$II    024402
                        NMI$MI    024440  NMI$PI    024446  NPI$II    024456
                        NPI$MI    024462  NPI$PI    024466  SAF$IP    024472
                        SAF$SP    024474  SVF$IP    024504  SVF$SP    024506
                        SAF$MP    024526  SVF$MP    024532  SAI$IP    024536
                        SAI$SP    024540  SVI$IP    024546  SVI$SP    024550
                        SAI$MP    024560  SVI$MP    024564
 SYS$I     024570 000000 = 0.     words  (RW,I,LCL,REL,CON)
 USER$I    024570 000000 = 0.     words  (RW,I,LCL,REL,CON)
 $CODE     024570 000700 = 224.   words  (RW,I,LCL,REL,CON)
                        HISPRT 024570
 OTS$O     025470 000000 = 0.     words  (RW,I,LCL,REL,CON)
 SYS$O     025470 000000 = 0.     words  (RW,I,LCL,REL,CON)
 $DATAP    025470 000220 = 72.    words  (RW,D,LCL,REL,CON)
 OTS$D     025710 000000 = 0.     words  (RW,D,LCL,REL,CON)
 OTS$S     025710 000000 = 0.     words  (RW,D,LCL,REL,CON)
 SYS$S     025710 000000 = 0.     words  (RW,D,LCL,REL,CON)
 $DATA     025710 000022 = 9.     words  (RW,D,LCL,REL,CON)
 USER$D    025732 000000 = 0.     words  (RW,D,LCL,REL,CON)
 Segment size = 002130 = 556.     words

Transfer address = 021170, High limit = 025730 = 5612.  words
```

CHAPTER 7

7-1. FORTRAN IV

```
.EDIT/CREATE PR0702.FOR
      PROGRAM PR0702
C...READ A REAL*4 NUMBER, CALL SUBA AND PRINT RESULT
C...SUBA IS WRITTEN IN MACRO-11
      EXTERNAL SUBA
      REAL*4 RVAL
50    TYPE 100
100   FORMAT (' ENTER NUMBER>',$)
      READ (5,*,ERR=1100,END=1000) RVAL
      IF (RVAL .EQ. -1.0) GOTO 999
      CALL SUBA(RVAL)   !CALL MACRO-11 SUBROUTINE
      TYPE 150,RVAL
150   FORMAT (' CHANGED TO>',F10.3)
      GOTO 50           !REPEAT TILL -1.0 ENTERED
999   STOP
C...NUMBER OUT OF RANGE
1000  TYPE 1010
1010  FORMAT (' ')
1100  TYPE 1110
1110  FORMAT (' ?VALUE BAD OR OUT OF RANGE')
      GOTO 50           !TRY AGAIN
      END

.FORTRAN PR0702

.EDIT/CREATE PR0701.MAC

.MACRO PR0701

.LINK PR0702,PR0701,SY:FORLIB/LIB

.RUN PR0702
ENTER NUMBER>123
CHANGED TO>    61.500
ENTER NUMBER>1
CHANGED TO>     0.500
ENTER NUMBER>2
CHANGED TO>     2.000
ENTER NUMBER>-1

STOP --
```

7–2. FORTRAN IV

```
.EDIT/CREATE NINPUT.FOR
      FUNCTION NINPUT(RVAL)
C...FORTRAN FUNCTION TO ACCEPT A REAL*4 NUMBER
      INTEGER NINPUT
      REAL*4 RVAL
      DATA NINPUT/0/
50    TYPE 100
100   FORMAT (' ENTER NUMBER>'$)
      READ (5,*,ERR=1000,END=999) RVAL
      IF (RVAL .EQ. -1.0) NINPUT=1
      RETURN
999   TYPE 1099
1000  TYPE 1100
1099  FORMAT (' ')
1100  FORMAT (' ?VALUE BAD OR OUT OF RANGE')
      GOTO 50
      END

.EDIT/CREATE NOUT.FOR
      SUBROUTINE NOUT(RVAL)
C...SUBROUTINE TO OUTPUT A REAL*4 NUMBER
      REAL*4 RVAL
      TYPE 1000,RVAL
      RETURN
1000  FORMAT (' ',F10.3)
      END

.EDIT/CREATE PR0704.MAC

.EDIT/CREATE OTSINI

.FORTRAN NINPUT,NOUT,OTSINI

.MAC PR0704

.LINK PR0703,PR0704,NINPUT,NOUT,SY:FORLIB

.RUN PR0703
ENTER NUMBER>123
    61.500
ENTER NUMBER>1
     0.500
ENTER NUMBER>2
     2.000
ENTER NUMBER>-1
```

7-3. MACRO-11

```
.EDIT/CREATE SUBA.MAC
        .TITLE  SUBA
SUBA::  MOVB    (R5),R1         ; Get number of arguments
        BEQ     20$             ; Branch if no arguments
        TST     (R5)+           ; Point to 1st argument
10$:    BIC     #200,@(R5)+     ; Clear 56th bit of high order
        DEC     R1              ; Repeat to end of list
        BNE     10$
20$:    RTS     PC              ; Return to caller
        END

.EDIT/CREATE PR0702.FOR

.MAC SUBA

.FORTRAN PR0702

.LINK PR0702,SUBA,SY:FORLIB/LIB

.RUN PR0702
ENTER NUMBER>123
CHANGED TO>    61.500
ENTER NUMBER>1
CHANGED TO>     0.500
ENTER NUMBER>2
CHANGED TO>     2.000
ENTER NUMBER>-1

STOP --
```

7-4. MACRO-11

```
.EDIT/CREATE PROG.MAC
        .TITLE  PROG
        .GLOBL  NINPUT,NOUT
PROG::  MOV     #ARG,R5         ;Point to argument block
        JSR     PC,NINPUT       ;Read a real number
        CMP     R0,#1           ;Is returned value = 1
        BEQ     QUIT            ;Exit if it is
        BIC     #200,RVAL       ;Clear 56th bit of high order
        MOV     #ARG,R5         ;Point to argument block
        JSR     PC,NOUT         ;Print result
        BR      PROG            ;Repeat
QUIT:   RTS     PC              ;Return to OTSINI
RVAL:   .FLT4   0.0             ;Floating point variable
ARG:    .WORD   1,RVAL          ;Argument block
        .END

.EDIT/CREATE OTSINI.FOR
      PROGRAM OTSINI
      CALL PROG
      CALL EXIT
      END
```

```
.EDIT/CREATE PR0705

.EDIT/CREATE PR0706

.MACRO PROG

.FORTRAN OTSINI,PR0705,PR0706

.LINK/EXEC:PROG OTSINI,PROG,PR0705,PR0706,SY:FORLIB/LIB

.RUN PROG
ENTER NUMBER>123
    61.500
ENTER NUMBER>1
     0.500
ENTER NUMBER>2
     2.000
ENTER NUMBER>-1
```

7-5. NEWBCL.MAC

```
        .TITLE BSCLI
        .IDENT /000008/         ;Copyright (c) 1974, 1975, 1976
                                ;by Digital Equipment Corporation
             .
             .
             .

        ROOT
        .GLOBL FTABI,BKGI
        .GLOBL SUBA             ; ** Modification for practice
FTABI:  .WORD FTBL
FTBL:   .WORD SUBNM             ; ** Modification for practice
        .WORD 0
SUBNM:  .BYTE 4                 ; ** Modifications for practice
        .ASCII "SUBA"           ;       .
        .EVEN                   ;       .
        .WORD SUBA              ; ** End of modifications
BKGI:   .WORD 0
```

Command to Reassemble Root of Interpreter

```
.MACRO/OBJECT:NEWBCL BSMAC+BSASM+NEWBCL
```

Replies to SUCNFG Program

```
MYBAS

Y

NEWBCL

SUBA
```

Index

Programming with RT-11, Volume 1 provides an overview of the RT-11 tools that facilitate program development. Included are detailed discussions of the MACRO assembler, FORTRAN compiler, BASIC interpreter, as well as, the RT-11 linker. It also examines the execution of foreground, background, and system jobs and discusses the use of ODT and VDT in debugging programs. The last chapters of this volume focus on the RT-11 facilities which increase programming efficiency. They describe the use of libraries, overlays, and language interfaces.

The authors develop courses for Educational Services Division of Digital Equipment Corporation in Reading, England.

digital

For a complete list of DECbooks, write to:

Digital Press
Digital Equipment Corporation
30 North Avenue, Burlington, MA 01803

ISBN 0-932376-32-0