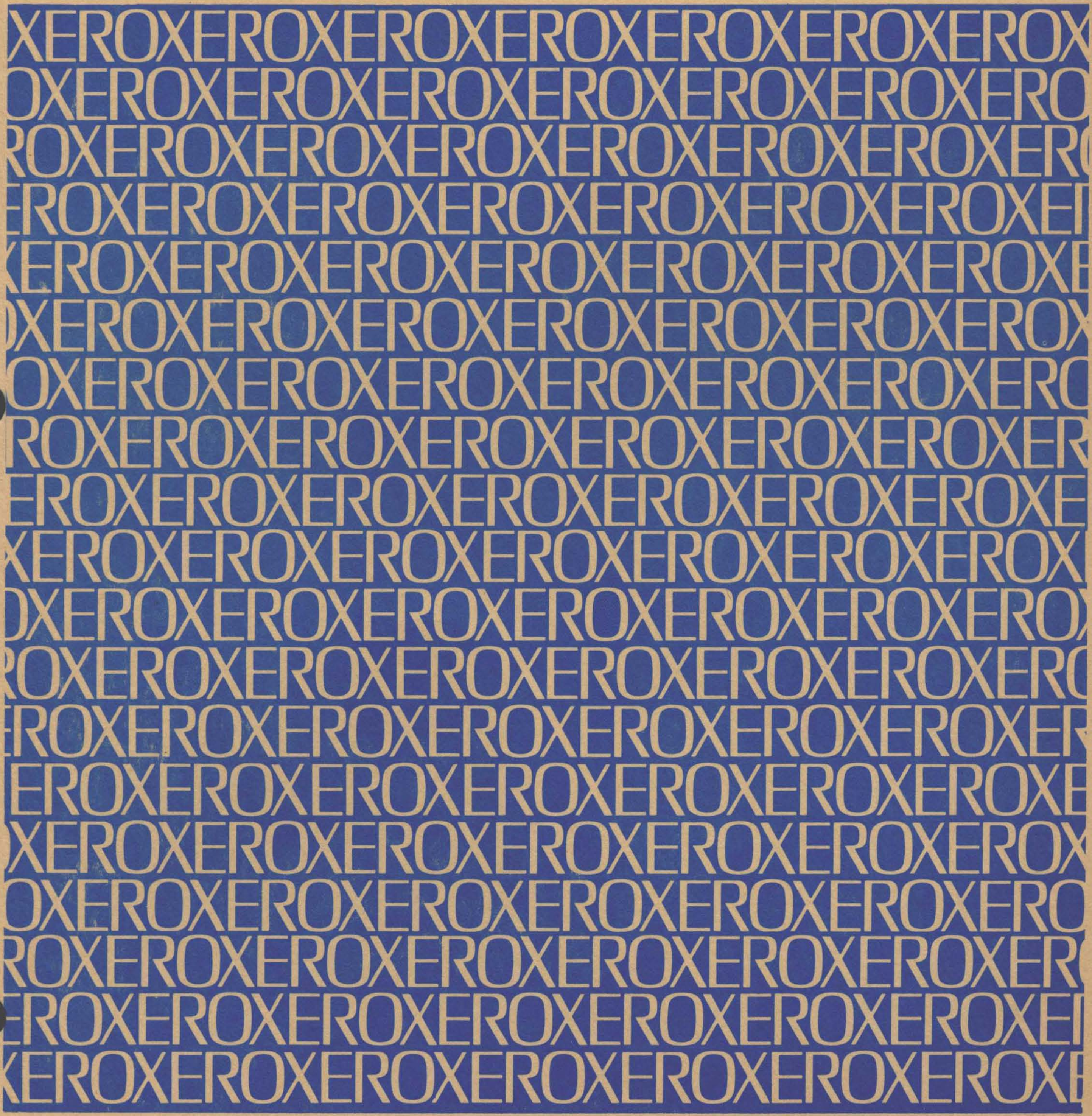
Xerox Real-Time Batch Monitor (RBM)

Sigma 2/3 Computers

User's Guide

Xerox Data Systems



90 17 85A

Xerox Data Systems

701 South Aviation Boulevard
El Segundo, California 90245
213 679-4511

XEROX

Xerox Real-Time Batch Monitor (RBM)

Sigma 2/3 Computers

User's Guide

FIRST EDITION

90 17 85A

January 1972

Price: $4.75

Printed in U.S.A.

NOTICE

This publication is the first edition of the Xerox Real-Time Batch Monitor (RBM)/RT,BP User's Guide for Sigma 2/3 computers, Publication Number 90 17 85A (dated February, 1972). This manual reflects the E00 version of the RBM system.

RELATED PUBLICATIONS

Title	Publication No.
Xerox Sigma 2 Computer/Reference Manual	90 09 64
Xerox Sigma 3 Computer/Reference Manual	90 15 92
Xerox Real-Time Batch Monitor (RBM)/RT,BP Reference Manual	90 10 37
Xerox Real-Time Batch Monitor (RBM)/OPS Reference Manual	90 15 55
Xerox Real-Time Batch Monitor (RBM)/System Technical Manual	90 11 53
Xerox Extended Symbol/LN,OPS Reference Manual	90 10 52
Xerox Symbol/LN,OPS Reference Manual	90 10 51
Xerox Basic FORTRAN and Basic FORTRAN IV/LN Reference Manual	90 09 67
Xerox Basic FORTRAN/OPS Reference Manual	90 10 61
Xerox Basic FORTRAN IV/OPS Reference Manual	90 15 25
Xerox FORTRAN/Library Technical Manual	90 10 36
Xerox ANS FORTRAN IV/LN Reference Manual	90 18 06
Xerox ANS FORTRAN IV/OPS Reference Manual	90 18 07

Manual Content Codes: BP - batch processing, LN - language, OPS - operations, RBP - remote batch processing, RT - real-time, SM - system management, TS - time-sharing, UT - utilities.

The specifications of the software system described in this publication are subject to change without notice. The availability or performance of some features may depend on a specific configuration of equipment such as additional tape units or larger memory. Customers should consult their XDS sales representative for details.

CONTENTS

PREFACE vi

1. THE RBM OPERATING SYSTEM 1

Real-Time Batch Monitor System 1
Real-Time Task 2
Foreground Programs 2
Background Jobs 3
The Monitor 3
Memory Allocation and Management 4
Concurrent Foreground/Background Operation 5
Checkpointing 5
Overlay Program 5
Response Time 6
Job Stream Summary 6

2. HOW TO COMPILE AND LOAD FORTRAN JOBS 9

Compilation with Source Listing 9
Compile Main Program with Subprograms 10
Execute Object Module from Card Input 10
Compile and Execute FORTRAN Program 11
Compile, Load, and Go from Permanent GO File 14
Compile, Load, and Go from Permanent OV File 15
Compile and Execute in Foreground Area 17
How to Use 026 Source and Data Decks 18

3. HOW TO ASSEMBLE AND LOAD EXTENDED SYMBOL JOBS 20

Extended Symbol Assemblies 20
Assemble, Load and Go 20
Load and Go from Permanent GO File 22
Modify an Assembly in a Permanent GO File 23
Assemble, Load, and Go from Permanent OV File 24
Assemble in Batch Mode 25
Assemble and Execute in Foreground Area 26

4. HOW TO CREATE AND MANIPULATE FILES 28

How to Create a File 29
How to Delete a File 31
How to Truncate a File 31
How to Squeeze a RAD Area 32
How to Access a File 33
How to Create a Program File 34
Sequential Access 34
Random Access 34
Creation Procedures 34
How to Create a New Library 36
How to Add a Library Routine 37
How to Delete a Library Routine 38
How to Recover Unused Library Space 39
How to Replace a Library Module 39

5. HOW TO BUILD AN OVERLAY PROGRAM 41

Communication between Segments 48
FORTRAN Segment Calls 48
Assembly Segment Calls 49
Segment Communication Using Common Areas 49
Blank COMMON 49
Labeled COMMON 49
How to Read a Load Map 50
Loader Process Summary 53

6. HOW TO USE MONITOR SERVICE ROUTINES 55

7. HOW TO USE UTILITY 59

How to Copy and Verify 59
How to Copy Card Reader Input to Line Printer 60
How to Copy and Verify from a RAD File to Paper Tape 61
How to Copy Magnetic Tape to Magnetic Tape 62
How to Copy a File to Line Printer 62
How to Prestore Control Commands 63
How to Use Utility Dump 64
How to Dump a Magnetic Tape 65
How to Dump a RAD File 65
How to Use the Object Module Editor 66
How to List Object Modules from GO File 66
How to Update Object Modules from Cards 67
How to Use the Record Editor 69
How to List a Specified File from Magnetic Tape 70
How to Modify a Source Module to a RAD File 70
How to Use the Sequence Editor 71
How to Generate and Sequence a File on Magnetic Tape 72
How to Update and Resequence Two Files on Magnetic Tape 73

8. HOW TO INTERFACE ANS FORTRAN IV AND EXTENDED SYMBOL SUBROUTINES 75

General Concepts and Conventions 75
- External References 75
- Temporary Stack 75
- Floating Accumulator 75
- Complex Accumulator 76
- Blank COMMON Storage 76
  - FORTRAN 76
  - Extended Symbol 76
- Complex Accumulator 76
- Blank COMMON Storage 76
  - FORTRAN 76
  - Extended Symbol 76
- Named COMMON Storage 76
  - FORTRAN 76
  - Extended Symbol 76

Coding Extended Symbol Routines for Calls from FORTRAN 77
- Standard Calling Sequence 77
- Argument Transfer Routines 78
- Subprogram Exit 78
- Temp Stack Definition 78
- Argument Checking 79

Calling Math Library Routines from Extended Symbol 79
- Math Routines 80
- Arithmetic Routines 80

9. HOW TO USE STANDARD PROCEDURE (S2) FILES 81

What May Be Stored in an S2 File 81
How to Create an S2 File 81

10. HOW TO REDUCE ASSEMBLY LANGUAGE HARDWARE REQUIREMENTS 85

Coping with Existing Resources 85
- GO File Elimination 85
- X1 File Elimination 86
- X3 File Elimination 86
- S2 File Elimination 86
- Redundant File Assignments 87

Coding for Existing Resources 87

11. HOW TO USE HARDWARE INTERRUPTS 88

Purpose of Hardware Interrupts 88
Summary of Hardware Interrupt Features 89
Interrupt Task Scheduling 90
Software Scheduling of Subtasks 90
RBM Organization 91

12. HOW TO CREATE A TASK CONTROL BLOCK 93

13. HOW TO CONNECT TASKS TO INTERRUPTS 95

14. HOW TO ATTAIN REENTRANCY IN ASSEMBLY LANGUAGE SUBROUTINES 96

15. HOW TO WRITE AN ASSEMBLY LANGUAGE INTERRUPT HANDLER 99

16. HOW TO WRITE AND EXECUTE A REAL-TIME PROGRAM 103

17. HOW TO CREATE A FORTRAN REAL-TIME SYSTEM 107

Reentrancy 107
Tasks 107
Basic Structure 107
Initialization 108
Subroutine Sharing 109
Public Library FORTRAN Routines 109

18. HOW TO DEBUG ASSEMBLY LANGUAGE PROGRAMS 110

How to Define an Insertion Block 110
How to Insert Snapshots and Code 110
How to Debug a Foreground Program 112
How to Use $NAME and @NAME 113
- Requirements for $NAME 113
- Requirements for @NAME 115

19. HOW TO ASSIGN AND USE DEVICE OPERATIONAL LABELS 117

20. HOW TO PATCH RBM 119

21. HOW TO SAVE AND RESTORE AN RBM SYSTEM 121

How to Create a Rebootable Save Tape 121
- Booting an RBM Save Tape 121
- Selectively Restoring Areas from a Rebootable Save Tape 122

How to Save RBM System Files 123
- Restoring RBM System Files 123

FIGURES

1. Compile with Source Listing and Suppressed Binary Output 9
2. FORTRAN Compilation with Subprograms, Binary Output to BO Device, and Source and Object Listing to LO Device 10
3. Object Program Input from Card Reader for Execution 11
4. FORTRAN Load and Go Job 12
5. Load and Go Sequence 13
6. Compile, Load, and Go from User-Defined GO File 14
7. Compile, Load, and Go from Permanent OV File 16
8. Compile, Load, and Execute in Foreground Area 17
9. Job Setup with 026 Source and Data 19
10. Assemble Single Program with Listed Output to LO Device 20
11. Assemble, Load, and Go 21
12. Load and Go from User-Defined GO File 22
13. Assembly Update from Permanent GO File 23
14. Assemble, Load, and Go from Permanent OV File 24
15. Assemble in Batch Mode 25
16. Load and Execute in Foreground Area 27
17. Create a RAD File 30
18. Deleting a File 31
19. Truncating a File 32
20. Squeezing a RAD Area 32
21. Output a RAD Map 33
22. RAD Area Map Example 33
23. File Creation with Specific GranuleAllotment 35
24. File Creation with Granule Over-Allotment 35
25. Input Library Files 37
26. Add a Library Routine 38
27. Delete a Library Routine 38
28. Library Space Recovery 39
29. Replace Library Object Module 40
30. Load Map Example 51
31. Core Layout During Loading 54
32. Monitor Service Routine Examples 56
33. Copy Card Input to Line Printer 60
34. Copy and Verify File from RAD Area to Paper Tape 61
35. Copy and Verify Magnetic Tape to Magnetic Tape 62
36. Copy LO File to Line Printer 63
37. Verify Card Deck and Paper Tape with Forced Prestore 64
38. Dump a Magnetic Tape 65
39. Dump Sequential Access RAD File 65
40. List Object Module from RAD File 67
41. Update Object Modules from Card Reader to Magnetic Tape 67
42. OMEDIT Update Example 69
43. List Specific File for Magnetic Tape 70
44. Update Source Records in Source Module 71
45. Generate and Sequence a File on Magnetic Tape 72
46. Update and Resequence Two Magnetic Tape Files 73
47. Save Instruction Procedures During SYSLOAD 82
48. Create an S2 File 82
49. Assemblies Using S2 File 83
50. RBM Hardware Priority Interupt Levels 92
51. Reentrant Subroutine Calling Example 97
52. Temp Stack Usage Example 98
53. Interrupt Handler Source Listing 99
54. Interrupt Handler Assembly Listing 100
55. Real-Time Task Example, Checkpoint Call and Exit 103
56. Real-Time Task Example, Restart Background 103
57. Deck Example for Loading and Executing Real-Time Tasks 104
58. Sample Real-Time FORTRAN System Schematic 107
59. Overlay Loader Controls for Sample Real-Time FORTRAN System 108
60. Example, Background Conditional Snapshot with Two Code Insertions 110
61. Foreground Conditional Snapshot with Two Code Insertions 112
62. Foreground Debug Example Using $NAME 114
63. Background Debug Example Using @NAME 115
64. Hex Correction Input Example 120

PREFACE

The RBM User's Guide explains how to use some of the more basic and commonly required features of the RBM operating system. It is to be used with, but does not supplant, the RBM/RT, BP Reference Manual, 90 10 37.[†] The writing style is informal, and technical terminology is avoided wherever possible for the benefit of the new RBM user.

The primary intent of the User's Guide is to assist new users in getting their first programs on the RBM system with a minimum of study. No attempt has been made to provide comprehensive examples for utilizing every feature in the RBM system. Your understanding of this manual will be improved if you are already partly familiar with the contents of the RBM Reference Manual, although this is not strictly necessary. One assumption that has been made however, is that you have at least a basic understanding of one of the programming languages that operate under RBM (i.e., ANS FORTRAN IV or Extended Symbol).

The User's Guide illustrates the necessary interface between your program and the operating system's services through a series of job examples and short discussions of specific applications. The examples generally present the simplest case for each application. Once these are thoroughly understood, the use of more sophisticated options, their relationships to one another, and the techniques for implementing them will be more readily apparent.

An effort has been made to organize the text so that experienced real-time programmers can directly access topics of immediate interest. A study of the Table of Contents should enable you to go directly to such relevant items and skip through other material already familiar to you.

In reading the User's Guide, you will probably notice that some definitions and several other items of information are repeated in several places. This repetition is intentional. Its purpose is to reenforce your understanding of certain basic terms and concepts that will make the learning of more complex facets of the system an easier task.

A full understanding of the material in this manual will not, in itself, make you an expert in utilizing all the capabilities of the RBM system. Further study of the RBM Reference Manual and some experience in the writing, checkout, and running of programs will be necessary. When going on to an in-depth study of the RBM Reference Manual and related manuals, there is a simple technique that may be used to help you learn new features of the system. The technique is not to study each unfamiliar feature with respect to its relationship to other parts of the system (whose significance may also be slightly hazy), but rather to key each item directly to some real or imaginary program. You can do this by asking yourself the following questions, as appropriate:

1. Why would I want it for my program and what will it do?
2. How do I use it?
3. If I can't use it, why not?
4. Is it a service I must request for myself, or does the system provide it automatically?
5. What are its possible side effects?
6. Is there another, perhaps better, way that I can accomplish the same thing?

Such a program-oriented approach to learning new features is less confusing, and will eventually lead to a recognition of the relationships between system components (much of the material in the RBM User's Guide was prepared by using this same study technique).

[†] The User's Guide contains a number of references to appropriate sections of the RBM/RT, BP Reference Manual, 90 10 37. For purposes of brevity, the title of that manual has been shortened to "RBM Reference Manual" in all such references.

If you are unfamiliar with the RBM system and have not had an opportunity to attend formal or informal training classes, it may be useful to review the available RBM documentation. By placing each manual in its proper context in terms of relevance to your own programming efforts, such a review may suggest general guidelines for further study and the priorities of such study.

- RBM/RT,BP Reference Manual, 90 10 37: This manual is addressed to all personnel within an RBM installation. It is the primary reference for all resources and services provided by the RBM system and gives detailed descriptions of each feature with explicit instructions for utilizing them. Since FORTRAN users commonly call Monitor services indirectly through the FORTRAN library, the areas of immediate interest to the FORTRAN programmer will be the sections dealing with the background processors, with special emphasis on their control command structure: Monitor, RAD Editor, Overlay Loader, and Utility Processor. The sections on operator communication (unsolicited key-ins) and the Public Library will also be pertinent. Assembly language users, in addition to the items above, will require a knowledge of the Monitor service routines and I/O sections of the manual, since these users request Monitor services directly in their code. Real-time programmers, both assembly language and FORTRAN users, will need to study the real-time programming section (assembly language users should give some emphasis to the Task Control Block subsection). The section on System Generation will be of primary interest to the installation's system programmers. The Standard Object Language appendix will usually be of interest only to those engaged in writing their own language processors and to systems programmers. The RBM Reference Manual is not intended to be a tutorial text. It is a feature oriented and logically organized approach to the system, as opposed to the User Guide's job oriented approach.

- RBM/OPS Reference Manual, 90 15 55: This manual is addressed to programmers/operators working at the console. It is basically a digest of operations and diagnostic features abstracted from the RBM Reference Manual and is organized in alphabetical sequences, rather than logical, for fast reference when corrective action is required while running a job. The RBM/OPS Manual confines itself to control command formats, operator key-ins, various diagnostic and warning messages, and some peripheral device considerations and data switch settings.

- RBM/System Technical Manual, 90 11 53: This manual is addressed to system programmers and analysts who are concerned with the internal structure of the RBM system for maintenance purposes. It is to be used in conjunction with the system listings supplied to each RBM installation, and the manual is essentially a "road map" through the RBM listings. Generally speaking, this manual has limited value or interest to applications programmers.

- Sigma 2 or 3 Computer Reference Manuals, 90 09 64 and 90 15 92: These manuals (whichever is pertinent) are addressed to all RBM users, but have particular relevance to real-time users because of the descriptions of the hardware interrupt structure. Of special importance to assembly language users are the sections on the machine instruction set and I/O, since these users generally are "closer to the machine" than FORTRAN users.

The programming language manuals listed in "Related Publications" include information on the language processor interface with RBM. (FORTRAN/RBM interface will be found in the appropriate FORTRAN Operations Manual.)

System programmers, real-time programmers, or any others who may be involved in heavy file management or specialized hardware peripheral considerations should consult the XDS PAL Manual, supplied to each installation, for a description of the appropriate peripheral reference manuals. It should be noted that the PAL Manual also contains descriptions of available software packages that were generated by other members of the User's Group.

1. THE RBM OPERATING SYSTEM

Before discussing ways for your program to utilize basic RBM services it is necessary to come to a common understanding of certain concepts, terms, and processes within the RBM context. The first of these is the operating system itself.

An operating system consists of all the software used at a local facility to perform the services for which the installation was designed. The software includes the Real-Time Batch Monitor, language processors (assemblers and compilers), service processors, and user programs, all closely integrated for a given number of applications.

No two operating systems are completely alike, because the user-created programs that form the user's extension of the operating system and the processors chosen from the manufacturers list of software modules are selected to fit unique requirements of the local facility. These requirements frequently change, and the system is correspondingly enlarged or otherwise modified.

These factors are generally found in all operating systems, but there are other characteristics that make RBM unique.

REAL-TIME BATCH MONITOR SYSTEM

A real-time system such as RBM has event-driven scheduling for all real-time operations. That is, certain external events occurring outside the immediate environment of the installation, such as various processes in an automated factory or biomedical diagnostic lab, determine which real-time computer operation is taking place at any given moment in time. The sequence of these real-time operations is not under the immediate control of a computer operator or programmer.

Furthermore, although a number of outside events can be taking place simultaneously, each event has a priority of importance in relation to all others, and this priority level determines which event gets service from the system first. Each real-time event produces a signal that is connected to a hardware interrupt. The interrupt is the connecting link between the external event and the real-time task that responds to it.

Under control of a real-time program, the computer can be set to respond to external events in many ways that do not involve scanning or decision making by the CPU. For example, when one interrupt level becomes ACTIVE, two or three higher, related levels might be DISABLED (postponed) long enough for the active level to complete a portion of its work before the higher levels could become active. Yet any incoming signals could be remembered. Further, a higher level that was ARMED and ENABLED could become ACTIVE immediately upon receiving an external signal.

This leads into one of the most important and distinguishing features of a true real-time operating system such as RBM: unlike other forms of operating systems that may offer some real-time capability as a secondary feature, RBM guarantees that its ability to provide extremely fast responses to interrupts will not be degraded by outside interference. That is, within the predictable limits of the hardware interrupt priority system and certain structural considerations of the real-time software (program segmentation), RBM absolutely protects a currently active real-time event from any slowing of response time (within 100 microseconds) by the actions of a lower-priority real-time process, a background process, or even inadvertent human interference such as hitting a Control Panel INTERRUPT switch. Further, the Monitor never interjects itself between real-time processes and their interrupts, but remains passively ready to respond immediately to calls for service from real-time external events within the self-imposed time limit stated above.

Some body of user-designed code (called a task) is associated with each interrupt level, and the RBM system is organized around the concept of tasks and programs. The most fundemental unit of software associated with an interrupt is the real-time task.

REAL-TIME TASK

A task is a body of code (and data) associated with one and only one hardware priority interrupt. This task is executed only if its corresponding interrupt level becomes ACTIVE. The task executes at the priority of its level and may be interrupted by a higher priority task. When completed, the higher level task will exit to the next lower waiting task. It does this by restoring any registers that have been modified, by restoring system pointers and status values to their previous conditions and by executing an instruction to exit from the current active level. These functions are performed either by a Monitor Service Routine or by the task itself.

Tasks may be triggered by other tasks or by external events. For example, a very high priority task may be connected (via an interrupt level) to an external signal. When the signal triggers the interrupt, the high priority task may be used to collect real-time data. The task may then terminate itself, triggering (under program control) a lower priority level task to process the data. The lower level task can continue processing at a less critical time when all intermediate levels are inactive. This procedure prevents loss of valuable data due to conflicting demands on computer time, yet enables the computer to be used as fully as possible for the most critical operations. It should be noted that it is never necessary for a task to know specifically which task it is interrupting or which task interrupted it. Note also that there is a distinction between real-time tasks and foreground programs.

FOREGROUND PROGRAMS

A real-time program is a collection of one or more related tasks and common data loaded and controlled as a unit. This collection of tasks may have (but need not have) contiguous interrupt levels in hardware priority sequence. Such a program is identifiable by name, so that it may be loaded into core memory or released from memory on request. A foreground program could consist of a single task, and the triggering of its interrupt would set off a series of processes in serial sequence.

It is possible to have foreground programs that are not controlled by external events but which nevertheless execute in foreground memory (and are connected to hardware interrupt levels) with all the protection privileges and use of dedicated I/O devices that real-time tasks have. Such programs may be loaded and released by the computer operator by other foreground tasks, or may be loaded from the background job stack. They may be operated periodically from a real-time clock, I/O end-actions, etc.

A foreground program may be called in and initialized by any of the following:

Another foreground task

Computer operator (unsolicited key-in)

Background job stream

Execution of the tasks connected to interrupts may be caused by any of the following processes:

External interrupt

Real-time clock

Computer operator (unsolicited key-in)

There are three possible types of foreground programs in RBM and they are classified according to the manner and location in which they are installed in foreground memory. Foreground memory is divided into two areas: resident and nonresident.

1. A resident foreground program is automatically loaded into its fixed area (absolute location) in resident foreground memory every time the system is booted in.

2. A semiresident foreground program is explicitly called from secondary storage (RAD or disk pack) into its portion of resident foreground memory for execution. The explicit call is made either by a resident foreground program or operator key-in. It is the responsibility of the caller to ensure that the required memory space is really available and not already occupied, since the program is unconditionally loaded.

3. A nonresident foreground program is also explicitly called by another foreground program or operator key-in from secondary memory but normally is loaded into the nonresident portion of foreground memory for execution. The space thus occupied is considered "active" and the program is fully protected by the Monitor from the background. If the nonresident space is already occupied when the call is made, the request is queued.[†]

Note that foreground programs, regardless of type or function, can only execute (e.g., perform read/write and compute operations in the foreground area. Compilations or assemblies can never take place in the foreground memory area. Therefore, before any foreground programs and tasks can be executed, they must first have been created by background jobs.

BACKGROUND JOBS

A job is the basic independent process performed in the background area of memory. Each job is independent of any other job, and consists of one or more directly or indirectly related job steps. A job step is the execution of a single language processor program, service processor program, or user program within a job. The program for each step is brought in for execution by a processor command that identifies the program. Jobs can call for Extended Symbol assemblies or FORTRAN compilations; that is, they can translate your source (symbolic) deck into a binary format called the relocatable object module, and then call in a service processor called the Overlay Loader to form the executable version of your program termed the absolute load module or program file.

Background jobs are frequently referred to as batch jobs, which means that RBM permits you to load a series of jobs for sequential processing (see "Job Stream Summary" below and Chapters 2 and 3 for definition) or build a job stack of several unrelated jobs. Naturally, all inputs that follow a given processor command within a batch must be written in the same language; that is, all must be written in FORTRAN or all in Extended Symbol, etc. Additionally, the parameters specified on the processor command will apply to every input module that follows it within the job step. The primary purpose in batching jobs is to reduce idle time between jobs and to increase throughput speed.

All background jobs, whether they are to become real-time programs or not, are loaded, assembled or compiled, and checked out in the same way. The only difference is that jobs intended for the foreground must be given permission to be loaded into protected foreground memory through operator FG key-ins and options on some control commands before the executable versions are loaded in a protected foreground area.

As implied above, some background jobs never become foreground programs. Once they have been processed into executable form, they are executed in the background memory area as background programs. Background programs are identified by name, (filename), the same as foreground programs. Background programs must execute in user mode and must perform all I/O through Monitor service routines, as opposed to foreground programs that execute in master mode and can either elect to use Monitor service routines or perform their own I/O.

In this explanation of job processing, we have implied that various services are performed by the system to aid your programming efforts, and we have alluded to "foreground" and "background" memory without defining them fully. Further discussion of foreground/background programs requires that we now define these as well as other terms.

THE MONITOR

The Monitor is the supervisor or executive part of the system that controls, coordinates, and provides services for both real-time and background programs. For background processing, the communications link between you and the Monitor is through a subprocessor called the Job Control Processor (JCP). The JCP reads, interprets, and

[†] In RBM, "queuing" refers to a list of entries maintained by the Monitor that identifies items waiting for service or attention. The following terms are sometimes interchangeably used with queuing: scheduling, sequencing, ordering, or dispatching.

RESPONSE TIME

Response time is the total time it takes for a task to begin executing meaningful instructions in response to some external signal. If a program must be brought into core, response time is the total time required to access the RAD, bring the root into core, initialize, and then respond to external interrupt(s). The amount of delay in response time that can be tolerated must be determined for each individual real-time program (and sometimes each task) within the system. However, many real-time applications at a given facility do not need response time as rapid as that available with resident programs, and of course, response time is not a factor in background programs.

The following section summarizes preceding material and presents additional information that is necessary before going on to the examples in Chapter 2 or 3.

JOB STREAM SUMMARY

Sigma RBM provides two levels of services for computer users: real-time (foreground) services and batch (background) services. The sequence of foreground programs is controlled by external interrupts, interval timers, a call by another foreground task, or the computer operator. Most operating system services are called in via system function calls within the program, with certain other services being implicitly provided.

Batch programs are processed serially in the order submitted. There are four primary subprocessors (often called service processors) to assist the Monitor in providing service to the background job stream:

Job Control Processor

Overlay Loader

RAD Editor

Utility Processor

To make use of these background service processors, each job submitted must conform to the general requirements of the operating system and the specific requirements of the service processor being utilized.

The portion of the operating system that responds to control commands preceded by an exclamation mark (!) and performs and coordinates many other system functions in the background is the Job Control Processor. The terms "JCP commands" and "Monitor commands" are frequently used interchangeably. The JCP accepts control command input from a specific input device designated as the Control Command Device (referred to in this manual as the "CC" device). This device is commonly a card reader, although other media may be used instead of cards (the use of a RAD or magnetic tape as the "CC" device generally is not recommended because they are less flexible).

In a typical batch operation, several jobs are combined into a single "batch stream" although each job retains its identify through its preceding !JOB (or !JOBC) control command. The entire job stream is terminated when the JCP interprets a !FIN control command that informs the Monitor that no more jobs are to be processed in the current batch stream.

In addition to the !FIN control command terminating the batch stream and the !JOB control command preceding each job, other Monitor control commands may be used within a job to request various services. For example, a !LIMIT control command may be used if the user wants to limit the processing time or other system resources expended. Should the job exceed the time limits defined on the !LIMIT command, it would be aborted.

There are many control commands that can be used in a job, with one of the most generally useful being the !ASSIGN command. An !ASSIGN command may be used to direct the flow of I/O to or from a specified peripheral device or RAD file. !ASSIGN commands enable the user to write a program containing symbolic reference to "logical" I/O devices rather than to "physical" devices by assigning a device-file name to an operational label (logical device name), RAD file, or physical device. This allows selection of a particular physical device to be deferred until the job deck is prepared, and also permits any logical device to be reassigned at certain points within job processing.

In addition to controlling the assignment of a logical device, an !ASSIGN command may be used to control a variety of I/O parameters for that device. For example, an !ASSIGN command may specify that a particular logical device is to read information from a specified RAD file. When the user's program is executed during a subsequent step within the job, the Monitor searches its Master File Directory for the specified file, and makes that file available whenever data is to be read via the assigned device. The !ASSIGN commands remain in effect until the next !JOB or !JOBC command is encountered.

A background job is a collection of one or more job steps. A job step is all the control commands required for the setup and execution of a single processor or user program within a job. These job steps can be one of the service processors, a standard language processor such as Extended Symbol or ANS FORTRAN IV, or a user-designed program.

Each processor, whether a service or language processor, is called for execution by means of a "processor" control command. A processor control command begins with an exclamation mark followed by a name identifying the requested processor; e.g., !FORTRAN or !XSYMBOL. Such commands may also contain parameters pertaining to the execution of the program; the exact form depending on which processor is being called. Data decks usually follow a processor command, although data may be input from a RAD file or other media.

Binary output from a language processor may be produced on punched cards, magnetic or paper tape, or a RAD file. Such output is always in Sigma Standard Object Language format and must be translated (link edited) into executable format by either the Absolute Loader or Overlay Loader before it can be executed by the computer. The Absolute Loader is called via an !ABS command, and is principally used to place the Overlay Loader on the RAD at System Load (SYSLOAD) time. While the Absolute Loader can also be used for placing user programs on the RAD, this is generally not recommended, and there are a number of restrictions in using it for this purpose (the Absolute Loader is described in the Sigma RBM Reference Manual under the !ABS command).

The Overlay Loader is called via an !OLOAD control command, and is a much more powerful and versatile processor for user program purposes. Throughout the rest of this manual, the term "Loader" always refers to the Overlay Loader unless otherwise stated.

The !OLOAD command may specify parameters related to the program elements to be loaded, the type of program, (foreground or background) being constructed, and other optional parameters. If no special options are needed, the command !OLOAD is sufficient. The name of the operational label used for the output generated by the Loader is always "OV", which is assigned by default to a special file located in the System Data area of the RAD, and is termed "RBMOV". The OV operational label is used to rewrite on this file. The output of the Loader is called a program file (load module), which is a RAD file containing a core image of the executable program.

The program in RBMOV (sometimes referred to as the "OV file") file is called for execution via an !XEQ command. Note that there is no protection for the program in the OV file. However, this file is not altered by the Monitor, and unless changed by the background job stream, may remain intact between jobs.

Object modules may be linked by the Loader to form an "overlay" program structure. The logical structure of an overlay program is defined for the Loader by means of !$ROOT and !$SEG Loader subcommands that must follow the !OLOAD command. Segments are identified by segment numbers and are defined by !$SEG commands for use by M:SEGLD service routine calls coded into the user's program. Segment 0 is always the root.

After a background program has been processed by the Loader, it may be brought into core for execution by means of an !XEQ or !name command. Foreground programs can be loaded by an !XEQ command or by a Monitor service routine call in another foreground task. Use of an !XEQ control command to load foreground programs must be preceded by an operator FG key-in.

File allocation, management, and manipulation is provided by calling in the RAD Editor service processor via a !RADEDIT control command. The !RADEDIT command itself requires no parameters; after the RAD Editor is called in, it reads subcommands that have a number sign (#) in column 2 and following an exclamation character in column 1. These subcommands identify the functions to be performed, such as !#COPY, !#TRUNCATE, !#DELETE, etc., and each contains user-defined parameters that specify the RAD areas and files to be processed.

Further file manipulation, such as file and record editing functions, copying files from one non-RAD device to another non-RAD device, etc., is provided by calling the Utility processor through a !UTILITY control command that also defines the type of Utility function to be used. Each Utility function has its own set of unique control subcommands that further define a given operation to be performed.

The operating system is capable of handling foreground tasks and background batch jobs concurrently because of the allocation of core into distinct foreground and background areas. All programs eventually intended for real-time applications are first assembled or compiled, processed by the Loader, and checked out through the background job stream. The programs are then loaded into their assigned files in the Foreground Programs area of the RAD. Thereafter, all new or modified foreground programs are first assembled or compiled in the background job stream in exactly the same fashion as any other batch job, and memory is allocated as required.

Now that basic orientation for the RBM system has been provided, some job examples for compiling or assembling and loading through the background can be considered. If you plan to write most of your program in FORTRAN you should go on to the next chapter. If you plan to write programs in assembly language, you should skip to Chapter 3.

2. HOW TO COMPILE AND LOAD FORTRAN JOBS

Xerox Basic FORTRAN, Xerox Basic FORTRAN IV, and Xerox ANS FORTRAN IV are the standard FORTRAN compilers currently available to Sigma 2/3 RBM users. In a typical foreground/background environment, FORTRAN users may range from engineers or other technical personnel who only occasionally write programs and have little interest in the internal functions of the RBM system, to real-time programmers whose knowledge of the software and hardware must be extensive. The discussion and examples that follow in this and other chapters are in increasing order of complexity that reflects this user range.

The examples in this chapter use the ANS FORTRAN IV specification options on the !FORTRAN processor card. Except for this major difference, all examples are equally valid for users of the other two compilers unless otherwise noted. ANS FORTRAN IV will process programs written for Basic FORTRAN and Basic FORTRAN IV.

COMPILATION WITH SOURCE LISTING

Figure 1 shows a FORTRAN job deck with the minimum control commands required to obtain a compilation of a source program and a source listing output to the line printer. The !JOB command informs the Monitor that a new job is being input. Optionally, the !JOB command could also contain an account number and user-defined name; e.g., !JOB 12345,FORTSAMP if job account was being used at the local facility.

Since binary output is not usually desired for an initial compilation and you will want to "desk check" the source listing before producing an object deck, the two !ASSIGN cards are used to suppress binary output to the BO (usually a card punch) and GO devices. If BO and GO were not assigned to 0, a request for binary output would be assumed by default.

The !FORTRAN card specifies a source listing (always defaulted) and the SQ option specifies that the compiler is to perform a sequence check of the source deck. Use of the SQ option is recommended for all initial compilations.

The !EOD card indicates that no further FORTRAN source statements are to be compiled and allows the compiler to exit to the Job Control Processor (JCP).

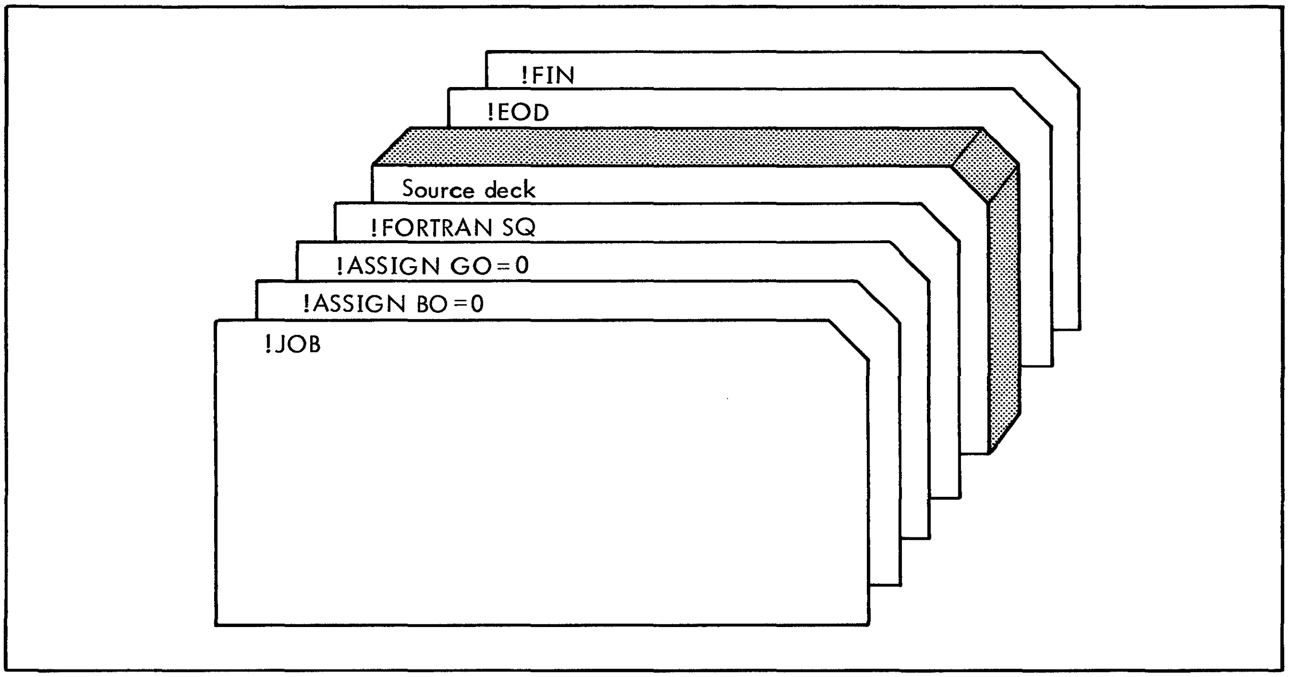


Figure 1. Compile with Source Listing and Suppressed Binary Output

The !FIN card informs the system that the job is completed and no other jobs are forthcoming. Should the job be one of a series in a "batch", the source decks would be followed by an !EOD card for each deck instead of !FIN until the final deck was input. This example is the simplest case of an ANS FORTRAN compilation under RBM.

COMPILE MAIN PROGRAM WITH SUBPROGRAMS

Compilation of a FORTRAN program with included subprograms poses no problems in terms of RBM interface. The single !ASSIGN card shown in Figure 2 suppresses binary output to the GO device but binary output to the BO devide will be produced in this case. The compiler will produce a source listing on the line printer. The system does not require any control commands between the subprogram modules.

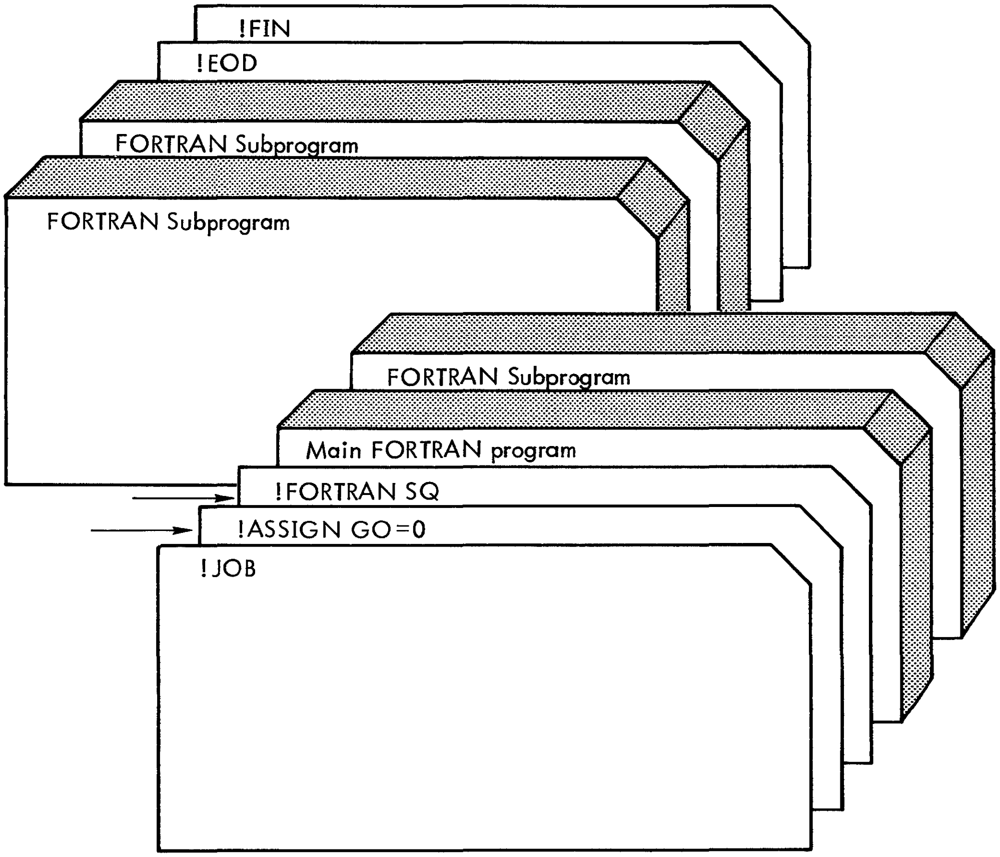


Figure 2. FORTRAN Compilation with Subprograms, Binary Output to BO Device, and Source and Object Listing to LO Device

EXECUTE OBJECT MODULE FROM CARD INPUT

After a source program has been successfully compiled into a binary object version that is free of obvious logical or coding errors, it is ready to be reloaded into the computer for execution. The example in Figure 3 shows the deck structure for loading a FORTRAN – produced binary object deck. Note that in this example we are assuming that the BI operational label is assigned to the card reader device at the local installation; if this was not the case, BI would have to be temporarily reassigned to this device via an !ASSIGN command.

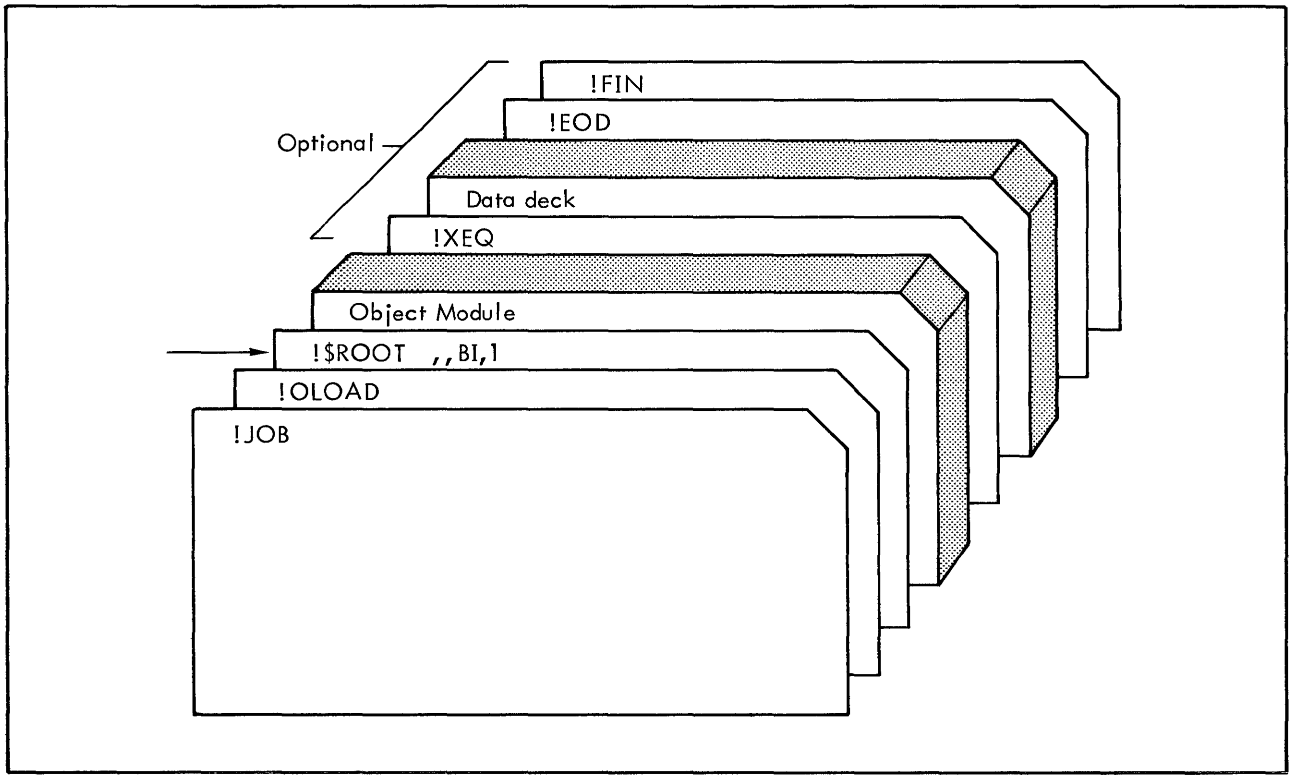


Figure 3. Object Program Input from Card Reader for Execution

The !OLOAD card calls in the Overlay Loader, and the BI and 1 parameters on the Loader !$ROOT card informs the Loader that it is to read one object module (binary deck) from the card reader, translate this into the load module (executable program or program file) and write this executable program into the RBMOV file. The RBMOV file is the default output file for the Loader and is located on the RAD. The default OV file will be reused the next time a program is loaded. The use of the !OLOAD card without parameters implies default of all options. Therefore, this will be a root only, background program with COMMON size taken from the object module (see the Overlay Loader chapter of the RBM Reference Manual for discussion of other options). The double comma on the !$ROOT card tells the Loader that the default cases are to be used for the "temp" and "exloc" options.

The !XEQ card causes the core image copy of the executable program to be loaded into core from RBMOV to process the data.

COMPILE AND EXECUTE FORTRAN PROGRAM

When your source program has been checked out to the point where any remaining coding errors are likely to be minor or the program is very simple, it is often useful to compile, load, and execute the program as one job. This procedure is commonly known as a "load-and-go" operation, and saves both computer time and unnecessary handling of the job.

In the load-and-go example illustrated in Figure 4, the !ATTEND card immediately following the !JOB card inhibits the Monitor Abort routine so that the system will go into a wait state instead of aborting the program in case any remaining program error is encountered. This will enable you to attempt corrective action at console while the program is still in memory, and is generally recommended for personally attended load-and-go jobs.

Since there are no commands preceding the !FORTRAN processor card, binary output will be written to both the BO and GO devices. The GO "device" is the RBMGO file on the RAD, and is the file from which the Overlay Loader will read its input. A source listing will be printed. The !OLOAD card calls the Overlay Loader, and the GO option on the !$ROOT card informs the Loader that it is to read one object module

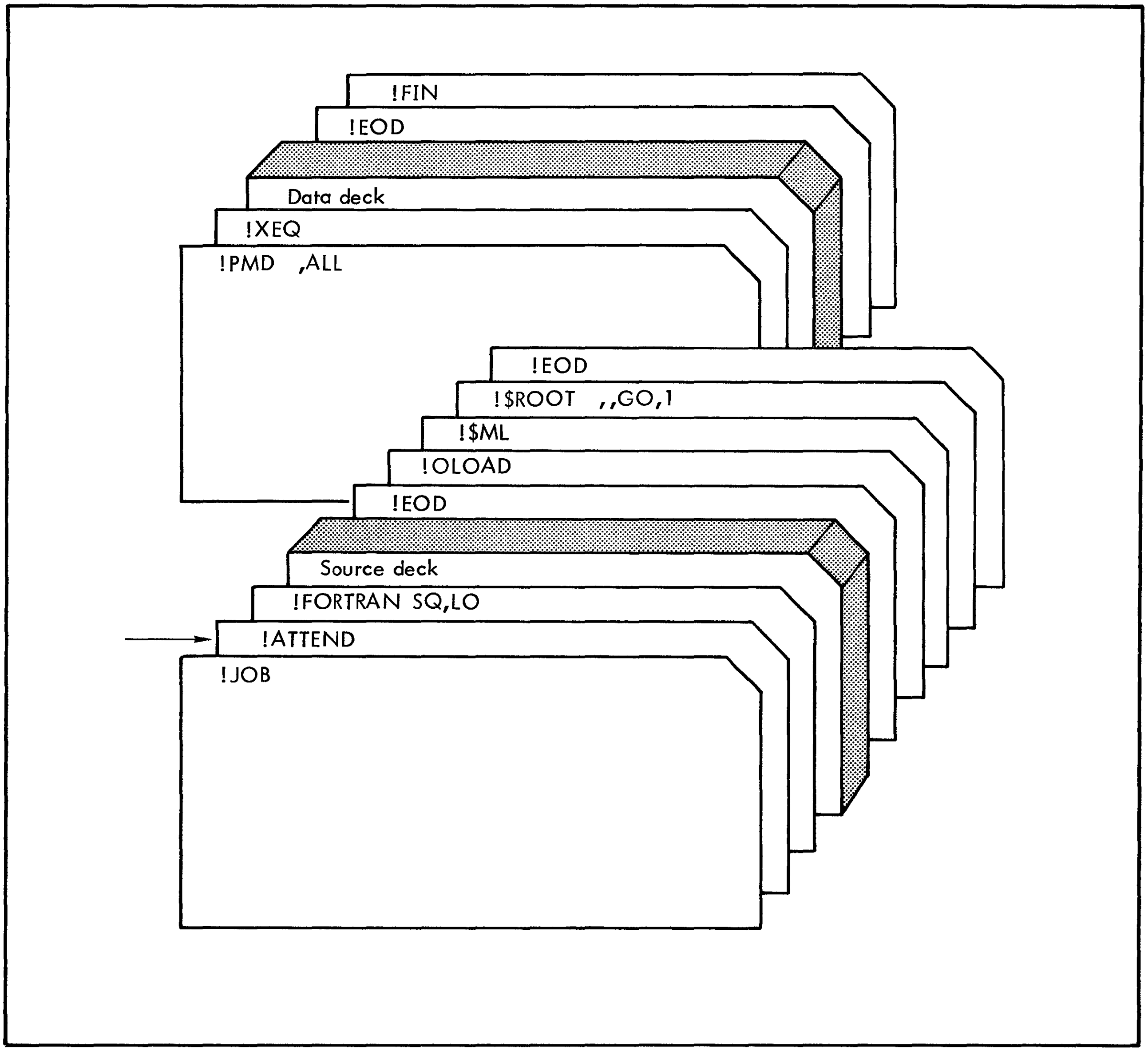


Figure 4. FORTRAN Load and Go Job

from the GO file, form the load module, and write it into the RBMOV file for subsequent loading into core for execution. The !$ROOT card also specifies that the "temp" and "exloc" default cases (double comma) are to be used.

The !$ML card informs the Loader that a Long map is to be output (an example and explanation of a load map is given in Chapter 6, "How to Build An Overlay Program").

The !PMD card provides for a core dump for further diagnosis if corrective action at the console is unsuccessful. Assuming !ATTEND was present, the !PMD card will cause a post-mortem dump to be output if you terminate a console recovery attempt with an X key-in (operator abort). Since ALL is specified, all of background memory and the CPU registers will be dumped in case of an abort for any reason. Generally, use of a !PMD command is advisable for load-and-go jobs regardless of the presence or absence of an !ATTEND card. The !PMD command is effective only for the job step following its appearance.

The !XEQ card calls the load module into core and gives control to the program for execution.

Figure 5 illustrates the job flow of a typical load-and-go job.

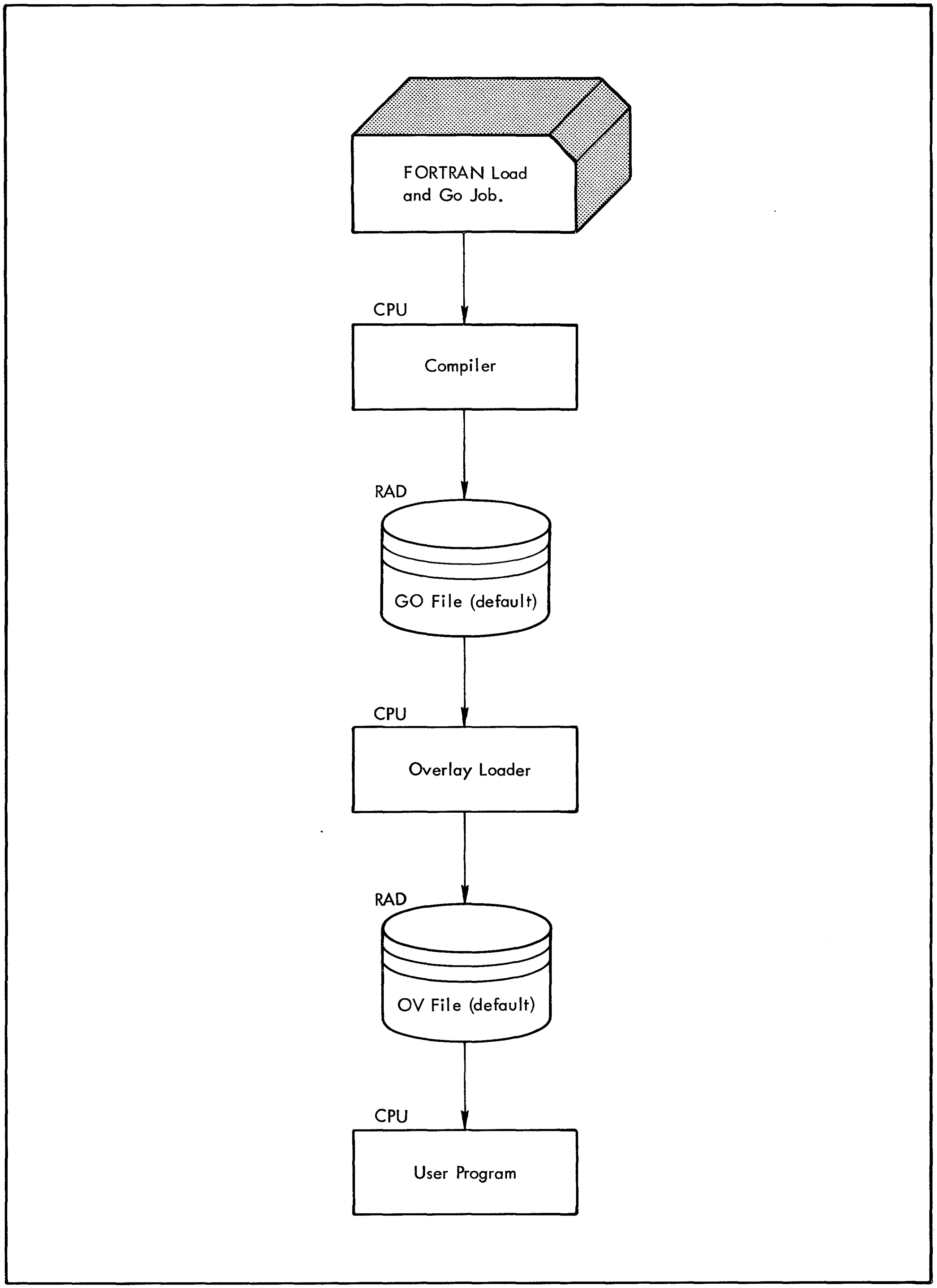


Figure 5. Load and Go Sequence

COMPILE, LOAD, AND GO FROM PERMANENT GO FILE

In the last example, we made the statement that the contents of the RBMGO file (the default GO file) is temporary and will be destroyed the next time a program is assembled or compiled. In some cases, it may be desirable to save the compiler output. In cases where you may be modifying or patching a program and do not wish to recompile every time, the compiler output file can be saved by defining your own GO file on the RAD (i.e., in the UD area).

Creating a user-defined file involves two control commands: the RAD Editor !#ADD card and the !ASSIGN card. The Editor !#ADD card is covered in more detail in the chapter "How To Create and Manipulate Files".

In the example in Figure 6, the !PAUSE KEYIN SY,S card is used to remove Monitor protection of previously defined RAD areas, and it functions in a similar manner to the FG key-in except that it permits access to protected RAD areas instead of foreground core memory.

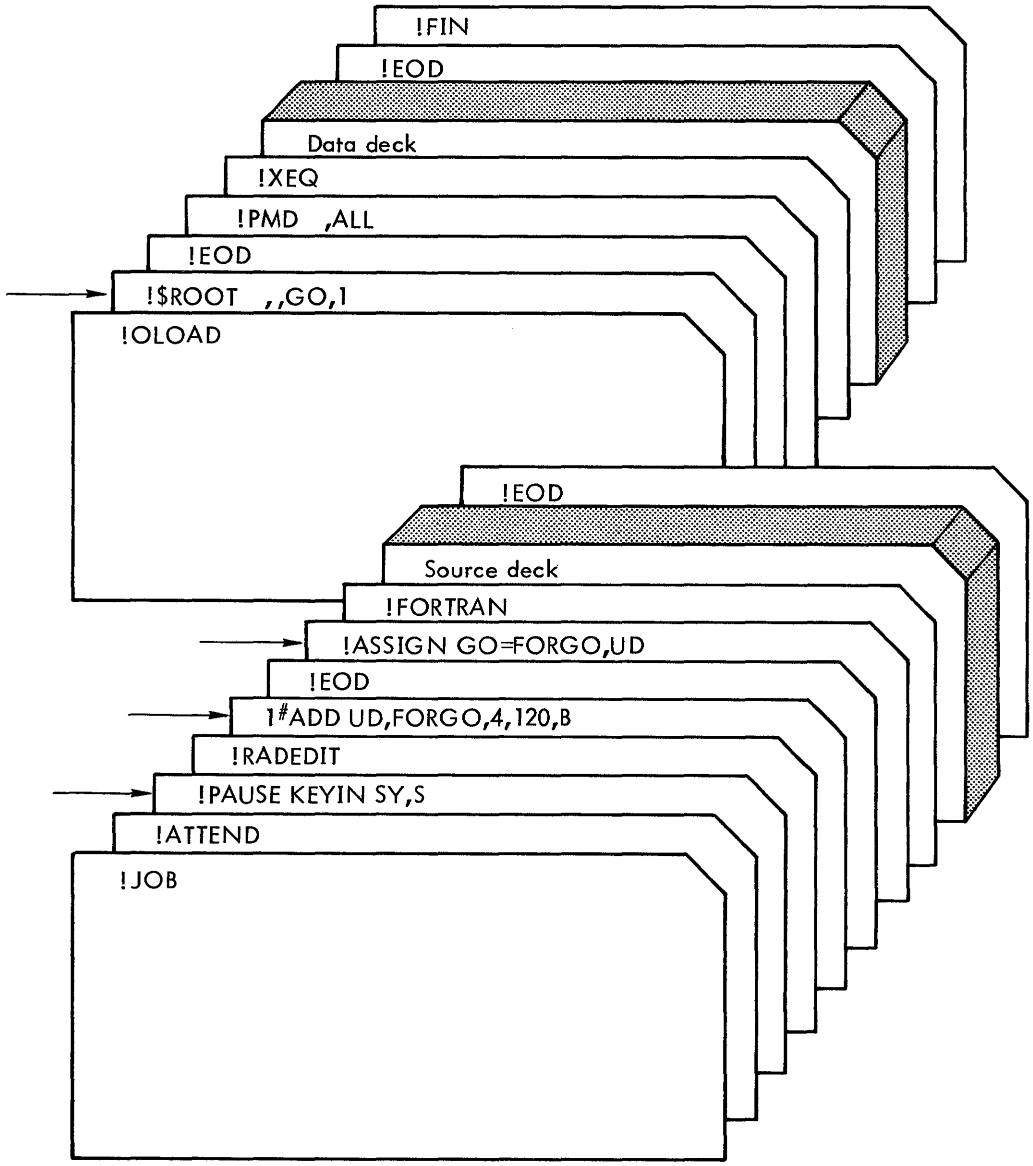


Figure 6. Compile, Load, and Go from User-Defined GO File

The !#ADD card following the !RADEDIT card informs the Editor that a new entry is to be added to the UD area, the name of the user-defined file is FORGO, and the file size is four records. The record must be 120 bytes to accommodate the Standard Object Language, and the file format should be blocked sequential access (B) for space economy.

The !ASSIGN command temporarily assigns the GO operational label to file FORGO (for this one job only) in the UD area of the RAD.

The !FORTRAN card specifies that a source listing is requested, a binary object deck is to be produced, and the Relocatable Object Module (ROM) is to be written to the GO operational label (which is reassigned to file FORGO). Identical ROMS are output on both BO and GO.

The Overlay Loader is called in (!OLOAD) and translates the ROM defined by the !$ROOT and (GO) into a load module and writes it in the RBMOV file (by default) for subsequent loading and execution. The double comma on the !$ROOT card specifies the default case for the "temp" and "exloc" parameters. Note that although the number of modules is specified (1), the "1" does not actually have to be specified in this case, since the !EOD on the GO file would terminate reading of the module.

BO instead of GO could have been assigned to file FORGO if a copy of the ROM was not desired from some selected device media. The choice is up to you, but there is a rule about assigning BO to a user file that should be remembered:

- The record size specified on the !#ADD command must be 120 bytes (60 words per record) and an EOF should be written by the user (!WEOF BO) to properly indicate end of data in the file. The compiler does not write an EOF to the BO operational label.

The Overlay Loader requires that all of its input object modules have 120-byte records and will abort the job if this is not so. Since the compiler does write an EOF to the GO operational label, no !WEOF command is necessary in the example; a file mark is written automatically at the end of the object module.

COMPILE, LOAD, AND GO FROM PERMANENT OV FILE

In the previous example, a program was compiled and the object module was written into a user-defined permanent GO file, but the Loader wrote the load module (executable program or program file) onto the RBMOV file for execution. Like RBMGO, the RBMOV file contents are considered temporary and may be altered from one job to another. Using the OV file is a useful procedure for programs not completely checked out or subject to frequent updating. However, once a program is completely debugged, you can define your own permanent OV file. The program can then be loaded into core for execution repeatedly, without the necessity of recompiling or recreation of the load module by the Overlay Loader.

The method for creating your own permanent OV file is quite similar to creating a permanent GO file, and again involves use of the !ASSIGN command and the RAD Editor !#ADD command.

In the example in Figure 7, the !RADEDIT card calls in the RAD Editor and the !#ADD card informs the Editor that a new entry is to be added to UP area. The name of the file is to be USEROV and there are four records within the file (filesize). The double comma specifies that the default record size is to be used and the format is to be random access (R). The file has write protection from everything except background programs (B).

The !FORTRAN card specifies a source and object listing, sequence check, a binary object deck, and a copy of the object module to be written into the RBMGO file.

The !ASSIGN command assigns the OV operational label to file USEROV (for this one job only) in the UP area of the RAD.

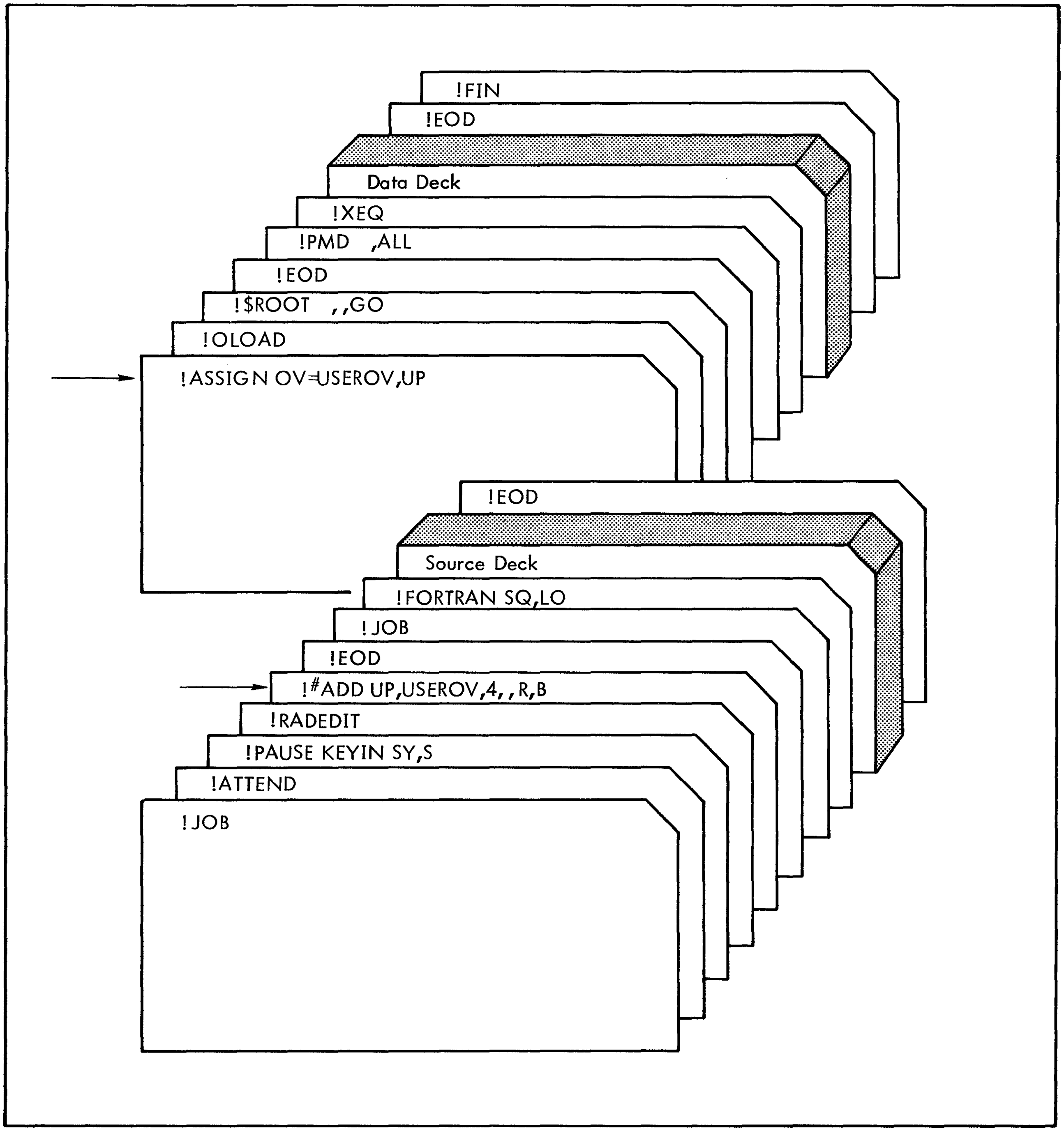


Figure 7. Compile, Load, and Go from Permanent OV File

The Overlay Loader translates the object module defined by the !$ROOT command into a load module and writes it in the USEROV file for subsequent execution. Future execution may be either by use of !ASSIGN OV=USEROV, UP and !XEQ cards, or the processor call !USEROV. You have created a permanent user program named USEROV.

Of course, there is nothing to prevent you from combining the creation of permanent GO and OV files into one job. This would merely involve adding the !ASSIGN and Editor !#ADD cards from the previous permanent GO file examples.

COMPILE AND EXECUTE IN FOREGROUND AREA

An example of loading a FORTRAN job from the background job stream that is to be executed in the nonresident foreground area of memory is illustrated in Figure 8. The deck structure for such jobs is identical to other load-and-go jobs except that the operator must key-in FG,S to access foreground memory, and the foreground option (F) must be specified on the !OLOAD card (the default option is B for background).

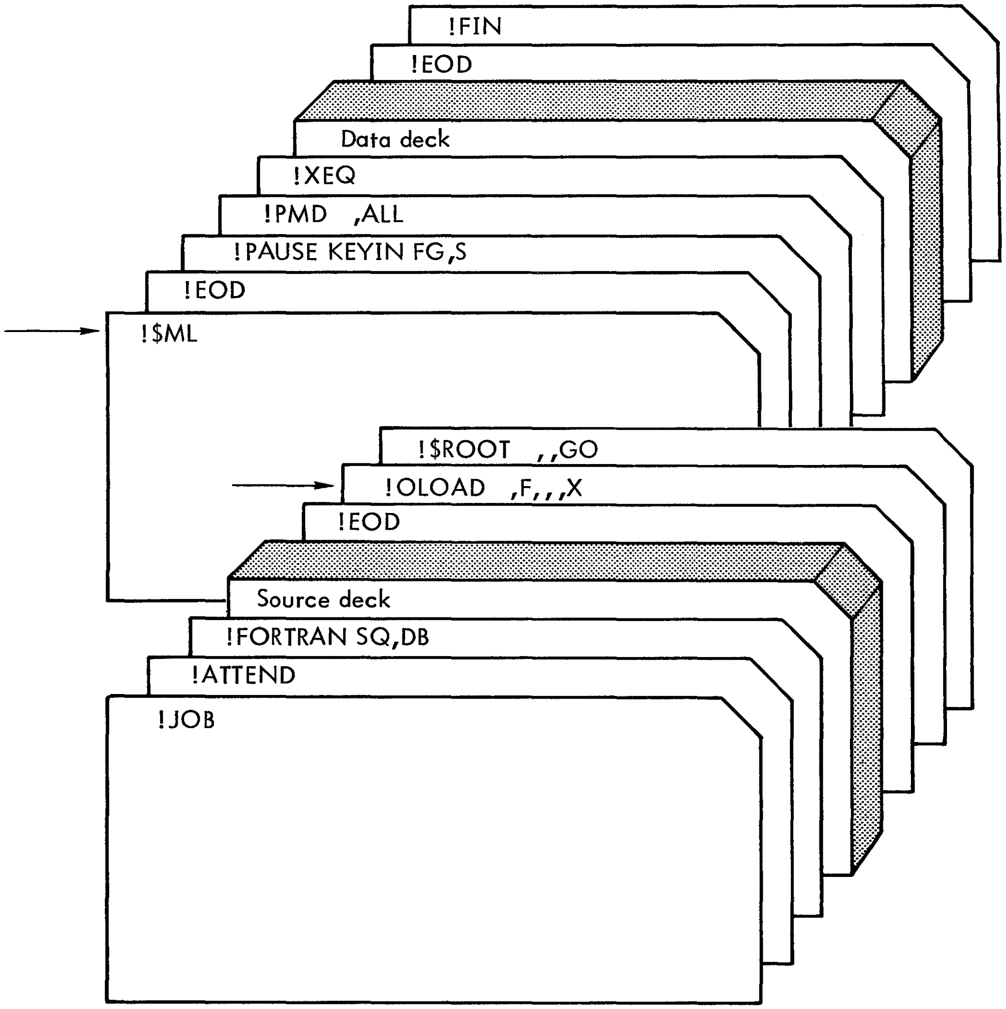


Figure 8. Compile, Load, and Execute in Foreground Area

All access to protected memory from the background job stream must be preceded by an FG key-in. Failure to do so is a foreground write protection violation and aborts the job unless an !ATTEND card is present. If !ATTEND is present and the !PAUSE KEY-IN FG,S card is accidently excluded, the Monitor will go into a wait state. The FG key-in must then be input and the command that caused the protection violation must be repeated.

The first comma (preceding "F") on the !OLOAD card informs the Loader by default that only a root segment is to be loaded; the "F" identifies the load module as a foreground task; the triple comma specifies that the step mode and Debug options are not being used. The X parameter requests the Loader to abort the job if a severity level greater than zero is encountered during the load process.

The double comma on the !$ROOT card informs the Loader that the default temp stack size (80 cells) is to be used and that the default beginning location for the load module is to be K:NFFWA (nonresident foreground first word address) in the nonresident area of foreground memory. The GO option specifies that the Loader is to read the single ROM (1) from the RBMGO file and write the load module into the RBMOV file by default.

The !$ML card causes the Loader to output a Long map when the load module is written into RBMOV.

The !PAUSE KEYIN FG,S card causes the system to go into a wait state so that the operator can perform the necessary FG,S combined key-in. This directs the Monitor to permit access to protected memory and continue processing.

The !XEQ command causes the load module on OV to be loaded into nonresident foreground memory for execution, beginning at location K:NFFWA.

HOW TO USE 026 SOURCE AND DATA DECKS

The RBM system expects all card images for source and data to be in the 029 character set. However, it is sometimes necessary or desirable to run FORTRAN jobs that were originally punched in the 026 character set format.

RBM has SYSGEN input parameter options called BR4 (026 card input), BP3 (026 card punch output), and B7 (7-track BCD magnetic tape) that will process I/O in BCD format instead of EBCDIC. If any or all of these required options are not already assigned to user-selected device file numbers at your local installation, it will be necessary to re-SYSGEN or perform a SYSGEN update before they can be used. See the subheading "Input Parameters" in the System Generation chapter of the RBM Reference Manual.

Figure 9 shows a complete FORTRAN job example with all input in 026 format, including main source program and subprograms on cards and the input data (to be read on FORTRAN device unit number F:112) on 7-track magnetic tape. The example assumes that a prior SYSGEN has assigned BR4 to device file number (DFN) 11 and that B7 has been assigned to DFN 12.

The presence of the !ATTEND and !PMD cards are suggested for diagnostic and corrective action when submitting a job that may be unfamiliar to the programmer submitting the job.

The first !ASSIGN card assigns the SI (source input) device to DFN 11 to read the source program and subprograms.

The !PAUSE command outputs the message on the card and then causes the Monitor to go into a wait state so that the operator can mount the 7-track tape containing the 026 input data to be processed.

The next !ASSIGN card assigns the FORTRAN device unit number (F:112) to DFN 12 to read in the data at execution time.

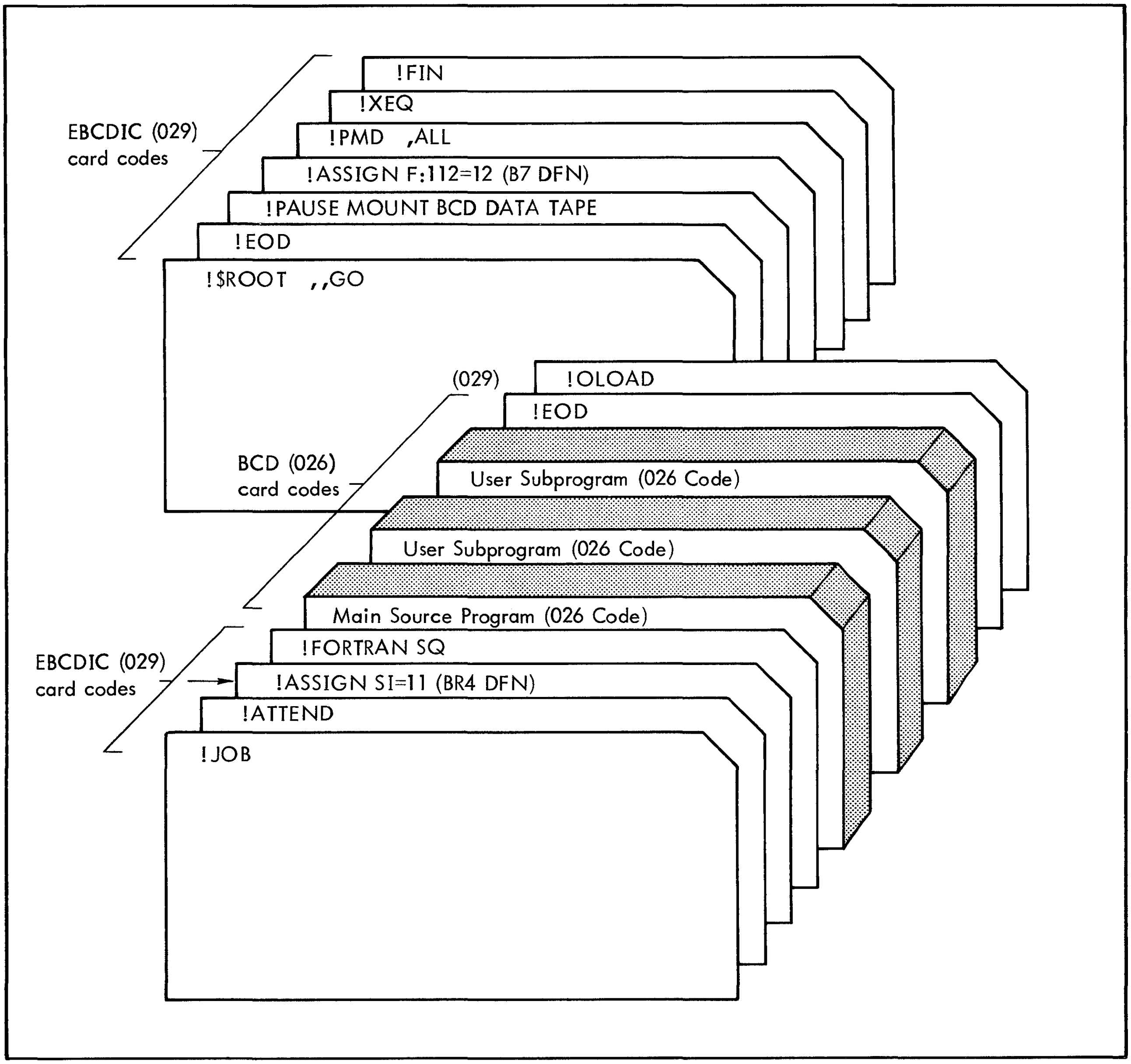


Figure 9. Job Setup with 026 Source and Data

3. HOW TO ASSEMBLE AND LOAD EXTENDED SYMBOL JOBS

Xerox Extended Symbol is the standard assembler available to RBM users. The information and examples in this chapter deal with the basic interface between RBM and your symbolic programs or updates; other and more complex material is given in the chapters concerning the Standard Procedure (S2) File, assembly language and FORTRAN routine interface, and real-time procedures.

EXTENDED SYMBOL ASSEMBLIES

Figure 10 shows an Extended Symbol job deck with the minimum control commands required to obtain an assembly listing output to the line printer. The !JOB command informs the system that a new job is being input. Optionally, the !JOB command may also contain a name and account number (e.g., !JOB SAMP1,12345).

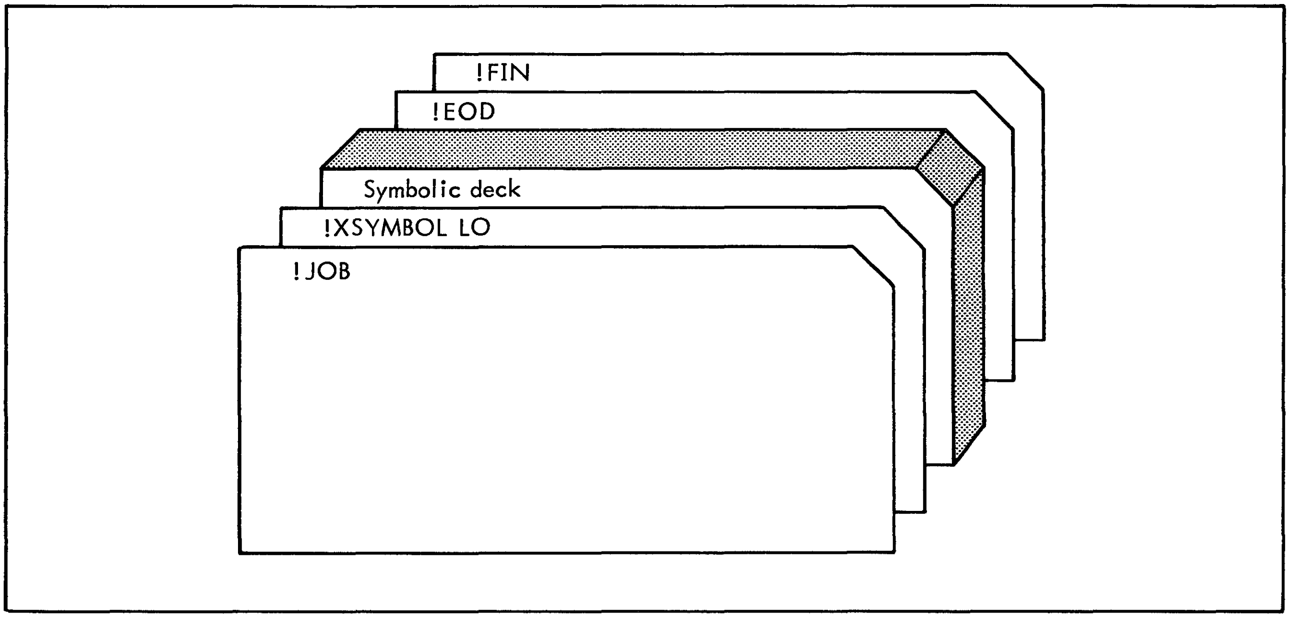


Figure 10. Assemble Single Program with Listed Output to LO Device

The !XSYMBOL card following the !JOB card informs the JCP that control is to be transferred to the assembler and LO specifies that an assembly listing is to be output (normally to line printer). The BO parameter is not specified because you will usually wish to "desk check" an initial assembly before producing an object deck.

Since in this program example we do not wish to use all the default options available (BO,GO,and LO), the desired option (LO) is specified. The only output will be the assembly listing.

The !EOD card must be present, to terminate execution of the assembler, before the !FIN card is encountered or the job will be aborted. The !FIN card informs the system that the job is completed and no other jobs are forthcoming.

ASSEMBLE, LOAD AND GO

When your source (symbolic) program has been "desk checked" to the point where any remaining errors are likely to be minor or nonexistent, it is often useful to assemble, load, and execute the resulting program file (load module) as one job. Such a job is commonly known as a "load-and-go" operation and saves both computer time and unnecessary handling of the job.

In the load-and-go example illustrated in Figure 11, the !XSYMBOL card specified by default that listing output is to be transmitted to the LO device, that binary output is to be produced (BO), and that the object program produced from the assembly is to be written on the RBMGO file, from which it is later read by the Overlay Loader as input.

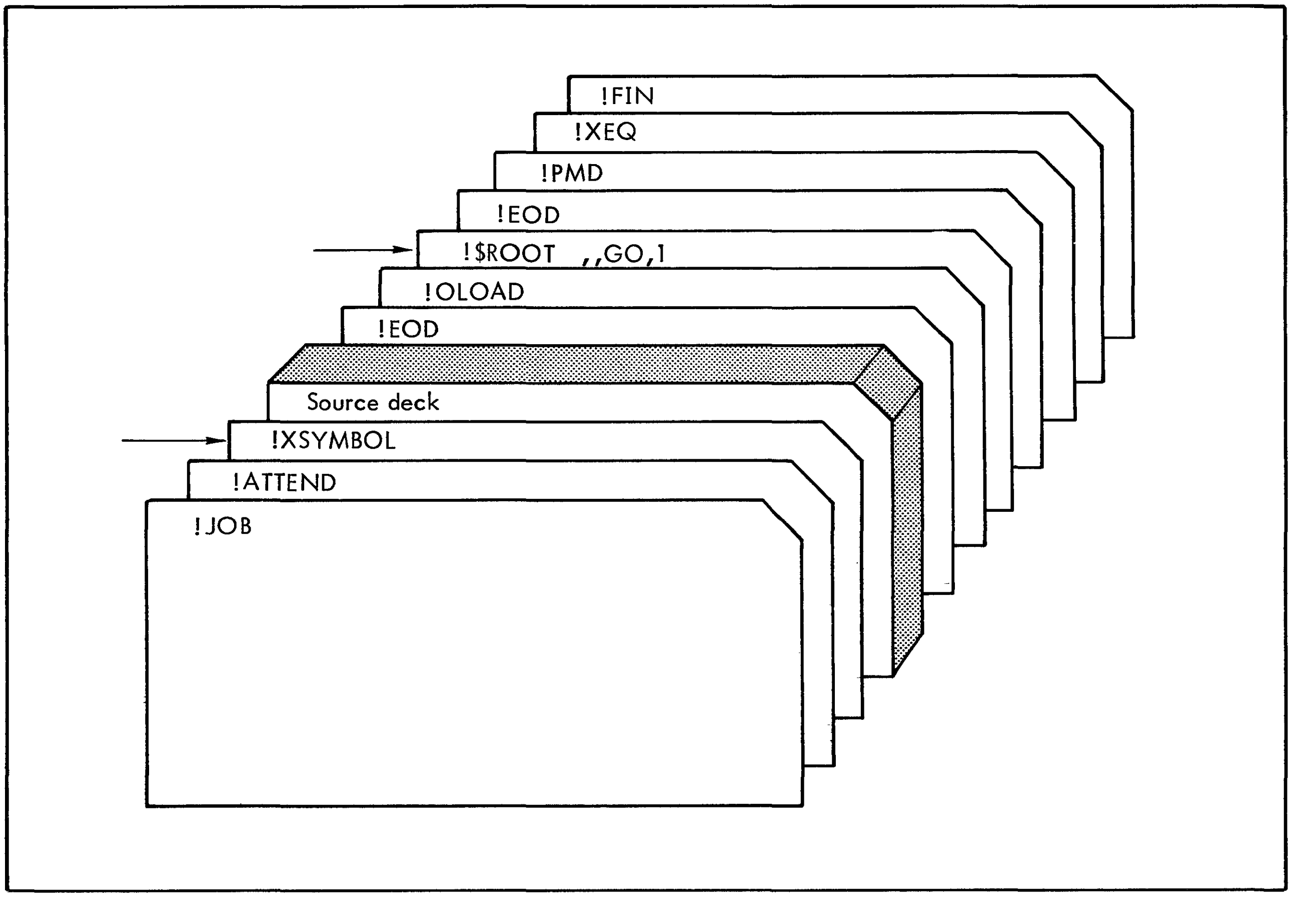


Figure 11. Assemble, Load, and GO

The !OLOAD card calls in the Overlay Loader, and GO option on the !$ROOT card directs the Loader to read the object module from the GO file, translate this into the load module (executable program or program file) and load it onto the RBMOV file. The OV file contents are also temporary and may be destroyed if another load module is loaded into the file.

The double comma on the !$ROOT card tells the Loader that the default cases are to be used for the "temp" and "exloc" options and the program will be executed in the background. The "1" following the GO parameter informs the Loader that only one object module is to be loaded.

The !PMD card applies only to the job step following it, and provides added information for future diagnosis if corrective action at the console proves unsuccessful. Assuming !ATTEND was present, the !PMD card will cause a post-mortem dump to be output if you terminate the console recovery attempt with an "X" key-in (operator abort). If the "U" option is specified, an unconditional dump will be output regardless of whether or not the program is aborted; if "U" is absent the dump will only be output if an abort (for any reason) takes place. If "ALL" is present, all background plus the CPU registers are dumped. Up to six pairs of "to" and "from" locations can be specified for selective dumping. If no options are specified on the !PMD card, only the CPU registers will be dumped (these registers are always dumped regardless of any specified limits). Generally, use of the !PMD command is advisable for load-and-go jobs regardless of the presence or absence of an !ATTEND card. The !ATTEND card inhibits the Monitor abort routine and will cause the Monitor to go into a wait state instead of aborting a job so that corrective action can be attempted at the console.

Since the !PMD card in the example does not contain any optional parameters, it will cause dumping of the CPU registers (only) if the program is aborted for any reason.

The !XEQ card causes the core image copy of the executable program located on OV to be loaded into background core and executed.

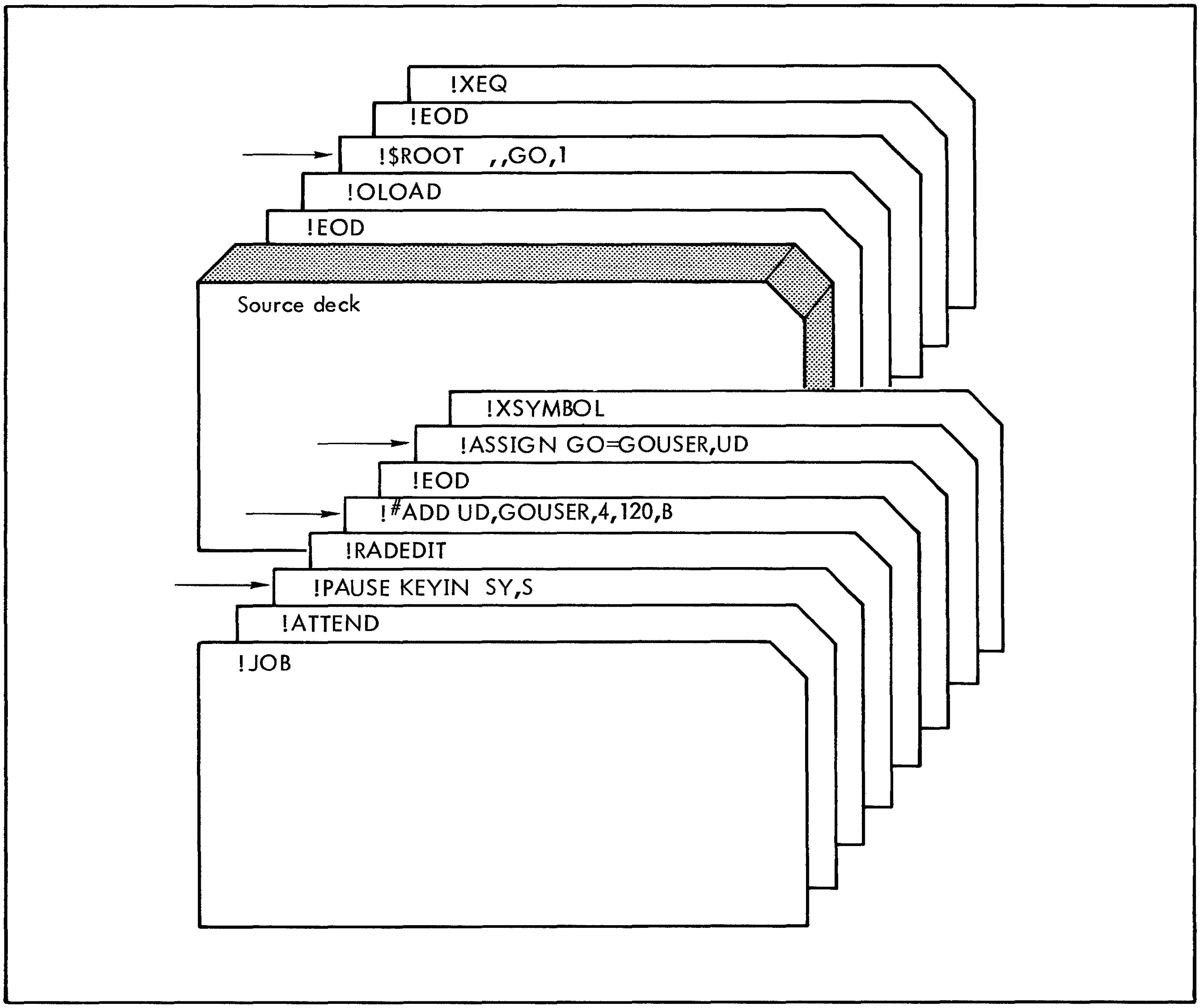
LOAD AND GO FROM PERMANENT GO FILE

It is sometimes desirable to save the assembler output. In cases where you may be modifying or patching a program every time it is loaded, the assembler output can be saved by defining your own file in the UD area (for instance).

Creating a user-defined GO file involves use of two control commands not previously discussed: the RAD Editor !#ADD command and the !ASSIGN command. The Editor !#ADD command is covered in the chapter "How To Create and Manipulate Files".

In the example in Figure 12 the PAUSE KEYIN SY,S card is used as a check to remove Monitor protection of previously defined RAD areas. SY is the RAD file analog of the FG key-in used for accessing foreground core memory.

The !#ADD card following the !RADEDIT card informs the Editor that a new entry is to be added to the UD area, the name of the user-defined file is to be GOUSER, and the filesize is four records. The record size must be 120 bytes to accommodate the Standard Object Language, and the file format should be blocked sequential access (B) for space economy.



```
!XEQ
!EOD
!$ROOT  ,,GO,1
!OLOAD
!EOD
!XSYMBOL
!ASSIGN GO=GOUSER,UD
!EOD
!#ADD UD,GOUSER,4,120,B
!RADEDIT
!PAUSE KEYIN  SY,S
!ATTEND
!JOB
```

Figure 12. Load and Go from User-Defined GO File

The !ASSIGN command temporarily assigns the GO operational label to file GOUSER (for this one job only) in the UD area of the RAD.

The !XSYMBOL card specifies by default that a listing is desired (LO), binary output is to be produced on some peripheral device (normally a card punch), and the Relocatable Object Module (ROM) is to be written to the GO operational label which is reassigned to file GOUSER for this job. Identical ROMs will be written on both BO and GO.

The Overlay Loader is called in (!OLOAD) and translates the ROM defined by the !$ROOT card (GO) into a load module and writes it in the RBMOV file by default for subsequent loading and execution. The double comma on the !$ROOT command specifies the default case for the "temp" and "exloc" parameters. Note that although the number of modules is specified (1), the "1" does not actually have to be specified in this case, since the !EOD on the GO file would terminate reading of the module.

BO instead of GO could have been assigned to file GOUSER if a copy of the ROM was not desired from some selected device media. The choice is up to you, but there is a rule about assigning BO to a user file that should be remembered:

- The record size specified on the !#ADD command must be 120 bytes (60 words per record) and an EOF should be written by the user (!WEOF BO) to properly indicate end of data in the file. The assembler does not write EOF to the BO operational label.

The Overlay Loader requires that all input object modules have 120-byte records and will abort the job if this is not so. Since the assembler does write an EOF to the GO operational label, no !WEOF command is necessary; a file mark is written automatically at the end of the object module.

MODIFY AN ASSEMBLY IN A PERMANENT GO FILE

Now that you have an assembled program located in a permanent file, it can be updated or modified without going through a reassembly. Using the GOUSER file from the previous example, the deck structure in Figure 13 wouldadd the patches and cause execution of the modified program.

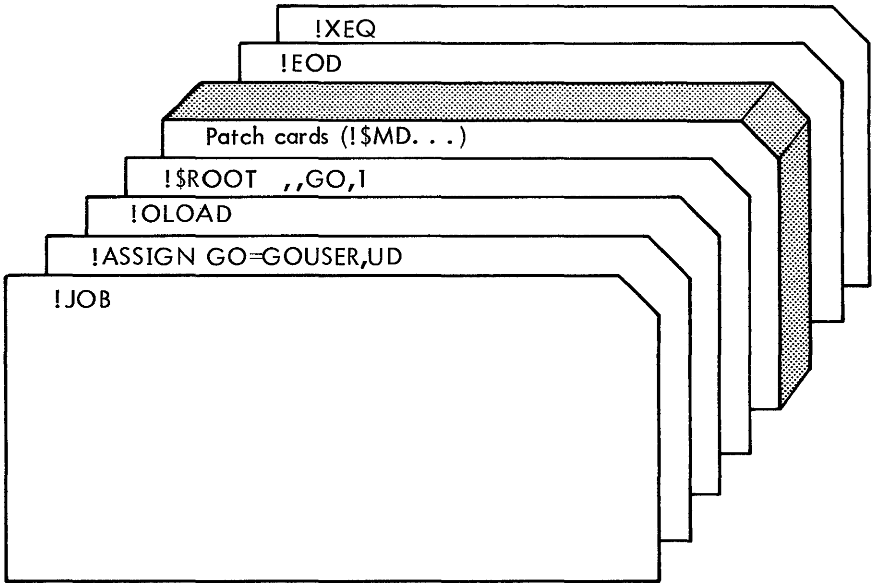


Figure 13. Assembly Update from Permanent GO File

ASSEMBLE, LOAD, AND GO FROM PERMANENT OV FILE

In the two previous examples a program was assembled and the object module was written into a user-defined permanent file, but the Loader wrote the load module (executable program or program file) onto the RBMOV file for execution. Like RBMGO, the RBMOV file's contents are frequently altered from one job to another. Using the RBMOV file is a useful procedure for programs not completely checked out or subject to frequent updating. However, once a program is completely debugged, you can define your own permanent file. The program can then be loaded into core for execution repeatedly, without the necessity of reassembly or recreation of the load module by the Overlay Loader.

The method for creating your own permanent file is quite similar to creating a permanent GO file, and again involves use of the RAD Editor !#ADD command and the !ASSIGN command.

In the example in Figure 14, the !RADEDIT card calls in the RAD Editor and the !#ADD card informs the Editor that a new entry is to be added to the UP area; the name of the file is to be USEROV, and there are four records within the file (filesize). The double comma specifies that the default record size is to be used and the format is to be random access (R). The file has write protection from everything except background programs (B).

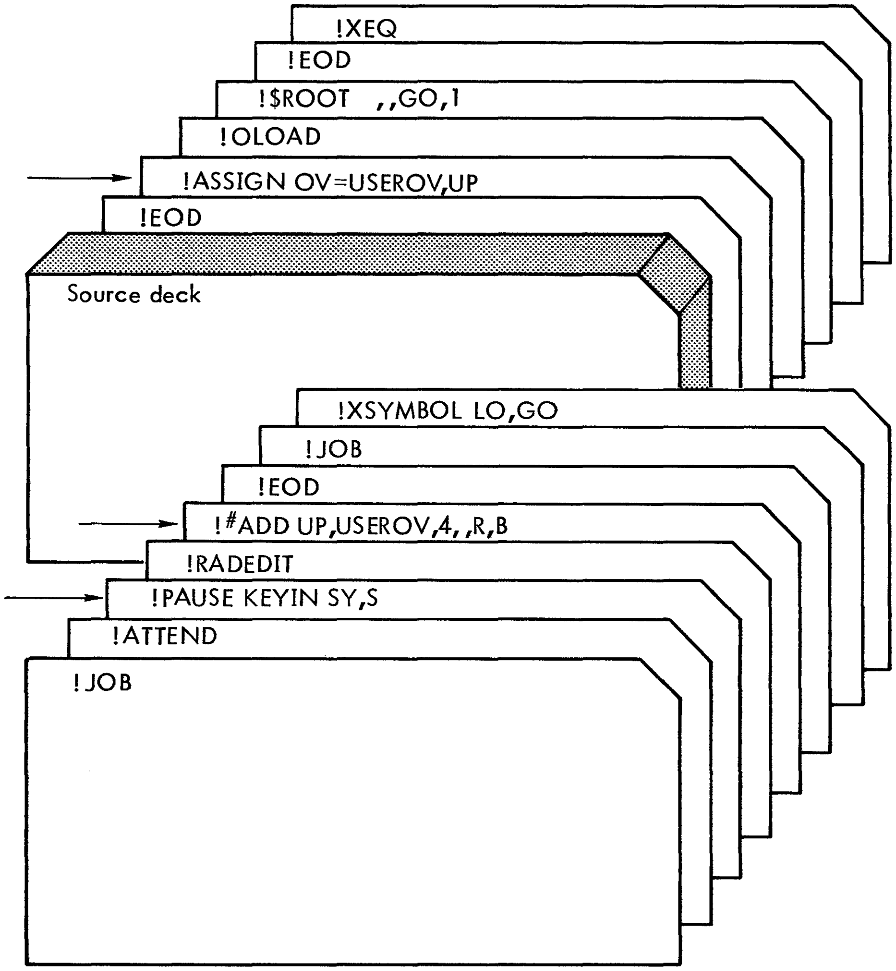


Figure 14. Assemble, Load, and Go from Permanent OV File

The !XSYMBOL card specifies that listed output is to be produced (LO) and the object program is to be written on the RBMGO file (GO).

The !ASSIGN command assigns the OV operational label to file USEROV (for this one job only) in the UP area of the RAD.

The Overlay Loader translates the object module, defined by the !$ROOT command, into a load module and writes it in the USEROV file for subsequent execution. Future execution may be either by use of !ASSIGN OV=USEROV, UP and !XEQ commands, or the processor !name call !USEROV. You have created a permanent user program named USEROV.

Of course, there is nothing to prevent you from combining the creation of permanent GO and OV files into one job. This would merely involve adding the !ASSIGN and Editor !#ADD cards from the previous permanent GO file examples.

ASSEMBLE IN BATCH MODE

A "batch" assembly is a series of successive assemblies performed with a single !XSYMBOL command. Batching offers processing and easier loading for the operator. There are three rules about batch assemblies that should be remembered:

- The assignments and options on the single !XSYMBOL card apply to all assemblies within the batch stream.
- Batch mode must be specified on the !XSYMBOL command via the BA option only if !EOD cards are used to separate the decks; otherwise, BA need not be specified.
- The last job in a batch must be terminated by double !EOD cards if the BA option is specified.

The example illustrated in Figure 15 shows three assemblies in a batch. Since !EOD cards are used as separators, BA must be specified on the !XSYMBOL card so that the assembler will reinitialize itself when it encounters the next source deck within the batch stream. Note that the parameters on !XSYMBOL could be in any order.

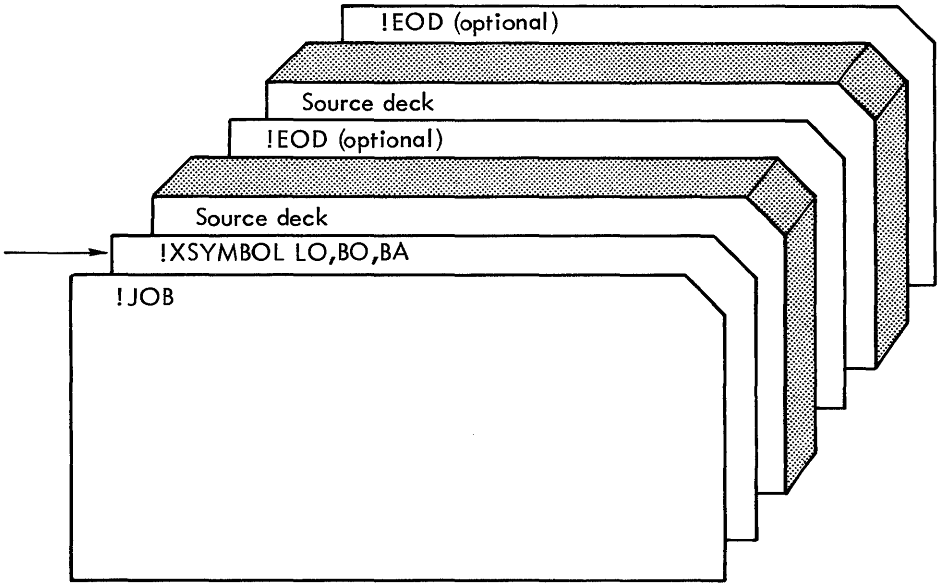


Figure 15. Assemble in Batch Mode

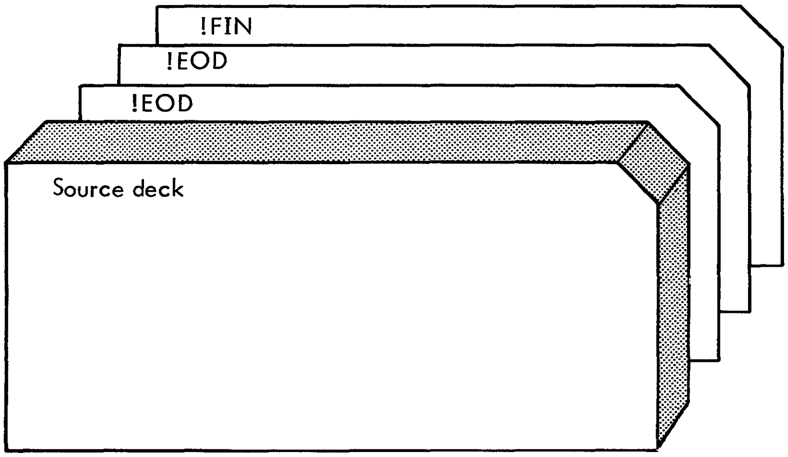


Figure 15. Assemble in Batch Mode (cont.)

ASSEMBLE AND EXECUTE IN FOREGROUND AREA

An example of loading an assembly from the background job stream that is to be executed in the nonresident foreground area of memory is illustrated in Figure 16. The deck structure for such jobs is identical to batch jobs except that the operator must key-in FG,S to access foreground memory and the foreground option (F) must be specified on the !OLOAD card (the default option is B for background).

All access to protected memory from the background job stream must be preceded by an FG key-in. Failure to do so is a foreground write protection violation and aborts the job unless an !ATTEND card is present. If an !ATTEND card is present and the !PAUSE KEY-IN FG,S card is accidently excluded, the Monitor will go into a wait state. The FG key-in must then be input and the command that caused the protection violation must be repeated.

The first comma (preceding "F"), on the !OLOAD card informs the Loader by default that only a root segment is to be loaded; the "F" identifies the load module as a foreground task; and the triple comma specifies that the step mode and Debug options are not being used. The X parameter requests the Loader to abort the job if a severity level greater than zero is encountered during the load process.

The double comma on the !$ROOT card informs the Loader that the default temp stack size (80 cells) is to be used and that the default beginning execution location for the load module is to be K:NFFWA (nonresident foreground first word address) in the nonresident area of foreground memory. The GO option specifies that the Loader is to read the single ROM (1) from the RBMGO file and write the load module into the RBMOV file by default.

The !$ML card causes the Loader to output a Long map when the load module is written into RBMOV.

The !PAUSE KEYIN FG,S card causes the system to go into a wait state and outputs the message on the card to the operator's console so that the operator can perform the necessary FG,S combined key-in. This directs the Monitor to permit access to protected memory and continue processing.

The !XEQ command causes the load module on OV to be loaded into nonresident foreground memory for execution, beginning at location K:NFFWA.

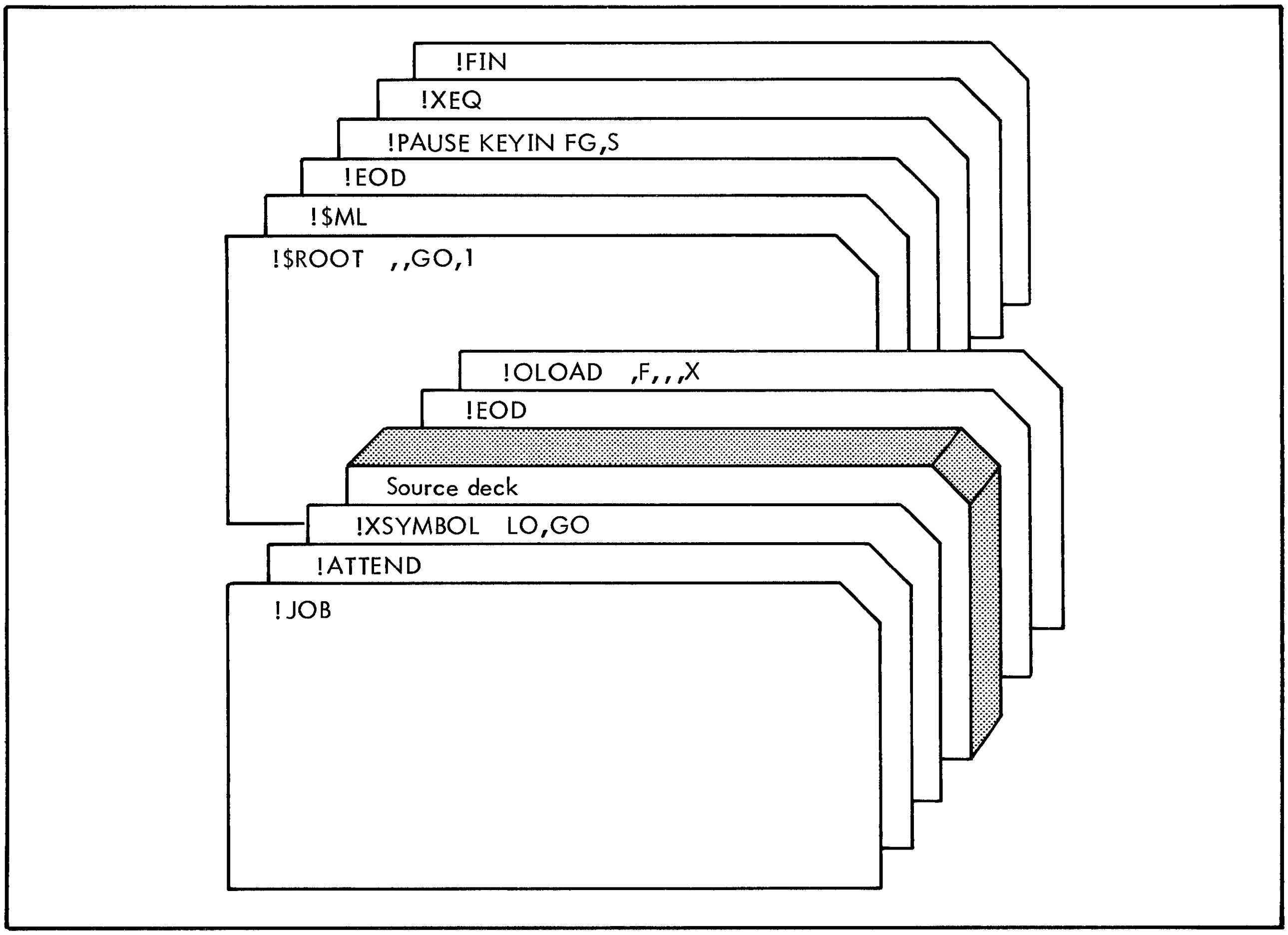


Figure 16. Load and Execute in Foreground Area

4. HOW TO CREATE AND MANIPULATE FILES

Whenever you want to allocate space or create or maintain permanent files on a RAD or disk pack, you must first call in the RAD Editor. The RAD Editor interprets and executes Editor control commands that define the operation, area mnemonic, and file name to be used.

Files to be saved achieve their permanent status by being created in the designated permanent area. Permanent areas are large blocks of RAD or disk pack space, each of which represents a grouping of files in terms of function. These permanent areas are initially set up at System Generation time and RBM will define the following areas by default if not defined by the user:

System Processor area (SP)	Contains language processors, nonresident portions of the Monitor, etc.
System Library area (SL)	Contains FORTRAN Library, etc.
System Data area (SD)	Generally contains the RBMGO and RBMOV files among other items.
Checkpoint area (CP)	Used for storing the background context when checkpointed by the foreground.
Background Temp area (BT)	Used as temporary scratch storage by background programs or processors.

During SYSGEN, the default cases for any of these areas may be overridden. The following areas are of direct concern to the RAD Editor:

System Processor area (SP)

System Data area (SD)

System Library area (SL)

Background Processor area (BP)

Foreground Processor area (FP)

User Processor area (UP)

User Data area (UD)

User Library area (UL)

Data area (Xn, where n is a hexadecimal digit)

aa (where aa represents a two-character mnemonic on the Dictionary)

You can put any type of file into any area desired. The area names are simply a convenience to expedite file housekeeping and management. The area names are formalized at SYSGEN and certain protection privileges are accorded to the areas, but what is put into these areas is up to you. Program files could be put into Data areas or vice versa.

The very permanence of Editor-created files suggests that you exercise economy by not using up any more permanent RAD or disk pack space than is strictly necessary, and instead, use temporary space in the BT area whenever possible.

The Background Temp area (BT) does not contain any permanent files and therefore is not the concern of the Editor since it cannot create files in this area. The Background Temp area contains the temporary scratch file (X1 through Xn) that are used by processors and the users.

- Management of permanent RAD areas and their files is handled by the RAD Editor via Editor control commands.

- Management of the Background Temp area (BT) scratch files is controlled through !DEFINE and !TEMP commands, or M:DEFINE Monitor service routine.

Since the Editor is itself a background processor, use of RAD Editor services is performed through the background job stream. This means that a foreground program never calls the RAD Editor to perform services. The allocation and subsequent manipulation of both foreground and background files are performed as background job steps.

Before discussing the use of the Editor and some of the control commands used to communicate with it, two terms must be clearly understood in the RBM context:

- A record is the amount of information processed by a single Write or Read instruction, and contains a user-specified decimal number of bytes. This number of bytes constitutes the RECORD parameter (record size).

- A file is an arbitrary, predetermined number of records that define the file's FILE parameter (file size). A file on the RAD must always have a name of three to eight EBCDIC characters by which it will be cataloged by the Editor in the proper permanent RAD area directory for all later calls to it. The file name is created by the user. Types of files include: program files that are interchangeably called executable programs or load modules; System and User Library files used by the Overlay Loader to satisfy external references in user's programs; and data files. Files are further categorized by FORMAT type: sequential, which may be U (unblocked), B (blocked), C (compressed); or random, which may be unblocked random (R) or (packed) random (P).

HOW TO CREATE A FILE

To create a file on the RAD for subsequent loading of either data or a background or foreground program, the Editor command

```
!#ADD   areaname,filename
```

is used. Since the Editor needs to know whether it has RAD space available whenever it encounters an !#ADD command defining an area and a file name, it also expects you to specify

file	(file size)
record	(record size)
format	(blocked, unblocked, compressed, random, blocked random)

on the same card, where as discussed previously, a file is a logically ordered group of records, and a record is the amount of information generally processed by one Read or Write request. FILE tells the Editor how many records to allocate a file. RECORD tells the Editor the maximum number of bytes per record. FORMAT tells the Editor the structure of the file. However, RECORD and FORMAT do not necessarily have to be specified if you are willing to let the Editor give you default options. For descriptions on default options, see RAD Editor chapter in the RBM Reference Manual. To create a data file, the sample command

```
!#ADD   D1,SAMP,20,,U
```

would define a file named SAMP to be allocated in the D1 area of the RAD. This file can contain up to 20 records. Since the "RECORD" was not specified, SAMP would have a default size of 360 bytes or 1024 bytes, depending on the RAD sector size where D1 is located.

Assume the default size is 360 bytes; therefore, the above !#ADD command would cause the Editor to reserve 7200 bytes of space in D1 under file name SAMP. So, the example has exactly the same effect as writing the command

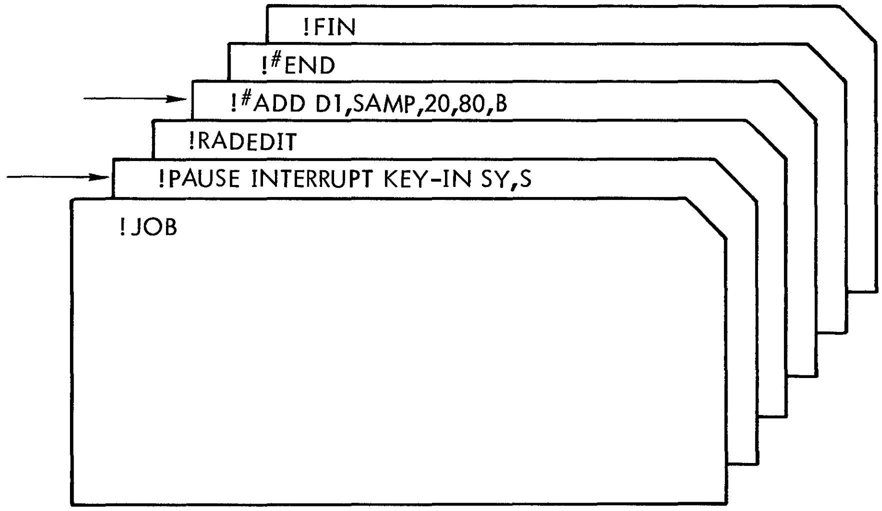
```
!#ADD   D1,SAMP,20,360,U
```

Using unblocked "FILE" format can sometimes waste space. For instance, if the purpose of the file SAMP is to hold the contents of 20 data cards (one EBCDIC card = 80 bytes of information), then 280 bytes in each one of the 20 records is wasted RAD space, and better efficiency is needed. Change the format of the example to blocked and the record size to 80 bytes:

```
!#ADD   D1,SAMP,20,80,B
```

For any blocked file, a 180- or 512-word blocking buffer is used to group as many records as will fit. If the blocking buffer is 180 words, in our new definition of SAMP above, four and one-half 80-byte records will fit in one block. Since it would take five blocks to contain 20 records, the amount of RAD space used would be five sectors.

This is very efficient use of RAD space, so use this last version of !#ADD to create a file as shown in Figure 17.



```
!FIN
!#END
!#ADD D1,SAMP,20,80,B
!RADEDIT
!PAUSE INTERRUPT KEY-IN SY,S
!JOB
```

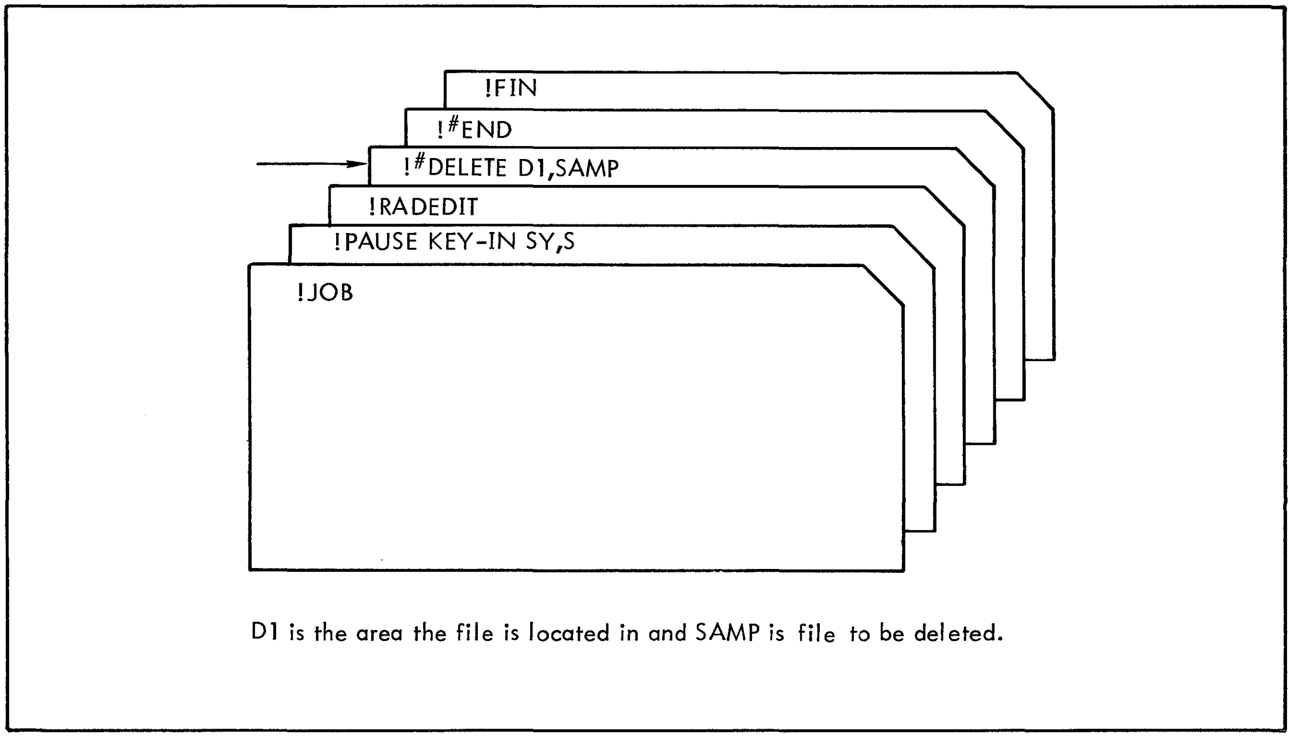
Note 1: Since the permanent file directories are software write-protected, an SY key-in must be initiated before updating or initializing a file directory if the area has an SY or FG protection code.

Note 2: For RADEDIT, !#END is equivalent to !EOD.

Figure 17. Create a RAD File

HOW TO DELETE A FILE

A file is deleted by the Editor !#DELETE command. Any files may be deleted from the permanent file directory. To delete the SAMP file used previously, the deck structure shown in Figure 18 would be used.



```
!FIN
!#END
!#DELETE D1,SAMP
!RADEDIT
!PAUSE KEY-IN SY,S
!JOB
```

D1 is the area the file is located in and SAMP is file to be deleted.

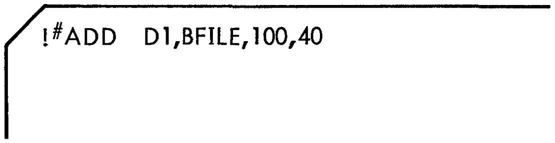
Figure 18. Deleting a File

If the file deleted is the last file within the area, the space is automatically recovered without squeezing (see "How to Squeeze a RAD AREA" later in this chapter).

HOW TO TRUNCATE A FILE

The Editor !#TRUNCATE command is used to delete unused but allotted space in a file by setting the EOT pointer equal to EOF.

Let's examine a hypothetical case. A blocked sequential file called BFILE of 100 records has been allotted. Thus,



```
!#ADD   D1,BFILE,100,40
```

Fifty records are copied into the file via the Utility COPY command (see Chapter 7 of this manual) and an EOF pointer is set at the end of the 50th record. Truncate the file as shown in Figure 19.

The resulting file will only contain 50 records (even though 100 records were allotted on the !#ADD card) because the !#TRUNCATE card cut down the size of the file to the actual number of sectors required to contain the 50 records by moving the EOT pointer equal to the EOF pointer.

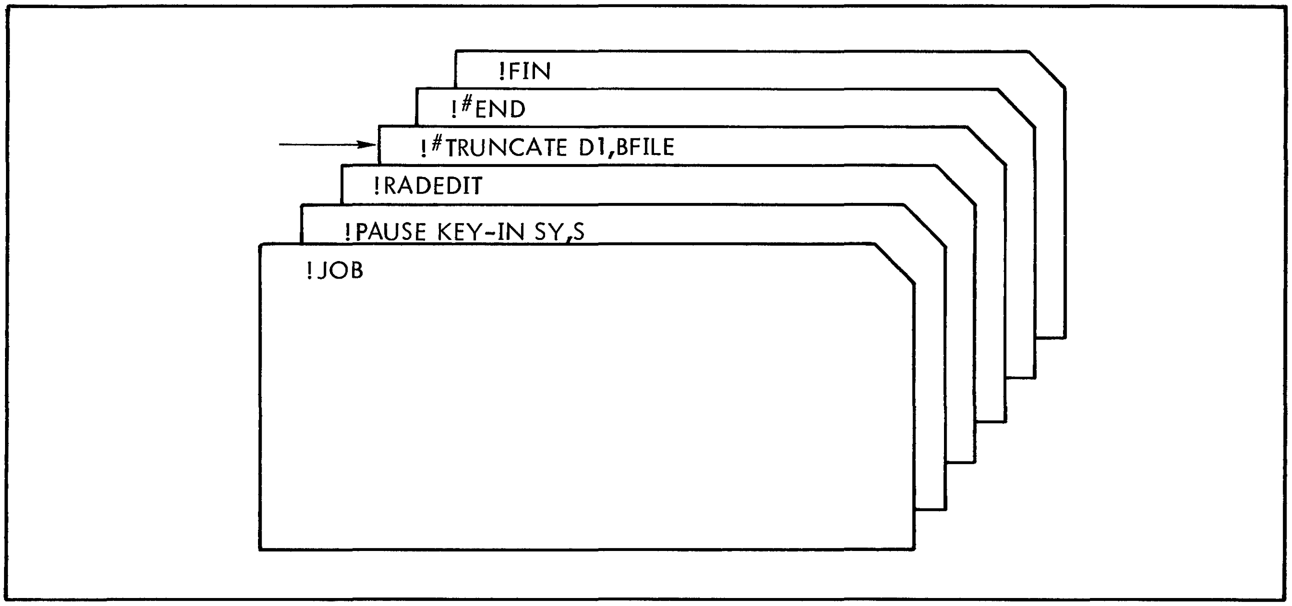


Figure 19. Truncating a File

However, the space deleted via the !#TRUNCATE card is still trapped; that is, it cannot be accessed either by other files or file BFILE. Since file size reduction has already been performed, the RAD area must be "squeezed".

HOW TO SQUEEZE A RAD AREA

To release unused space in a truncated file and to recover space occupied by deleted files so that the system can use it, the RAD Editor !#SQUEEZE command is used. This moves all files forward and leaves empty space at the end of the area. It is inserted after the !#TRUNCATE card as shown in Figure 20, but note that it is not always necessary to truncate a file before squeezing an area.

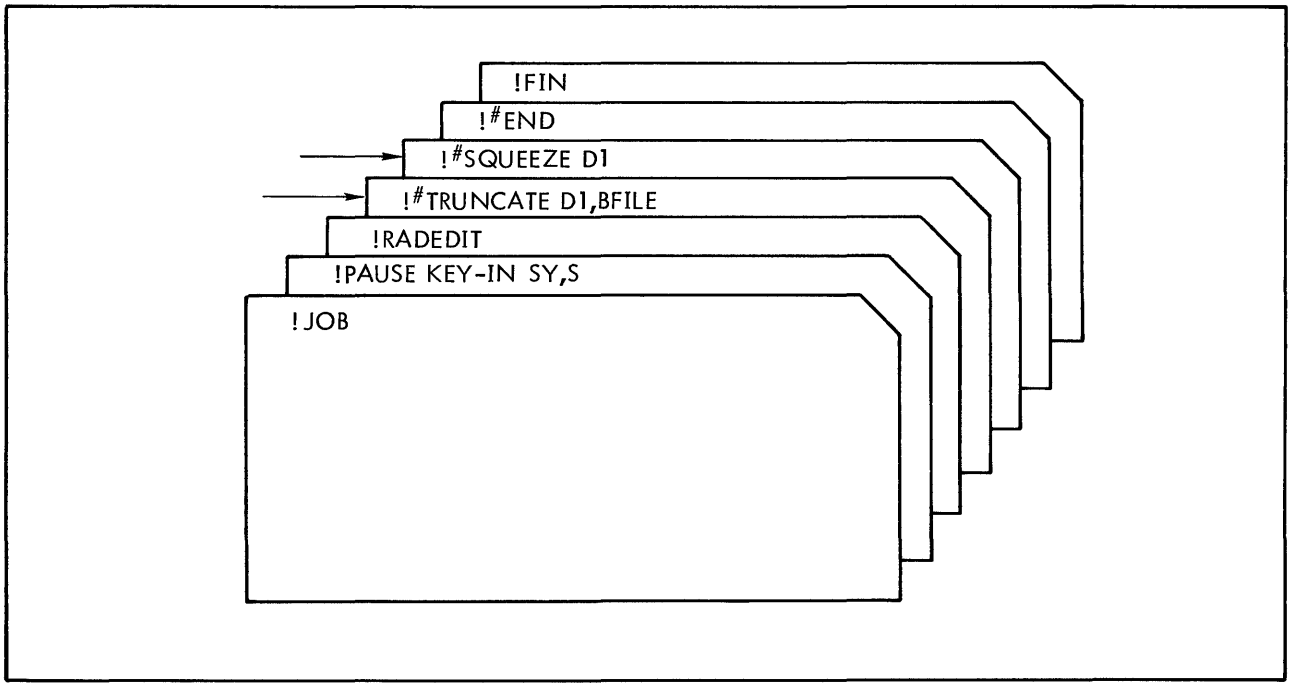


Figure 20. Squeezing a RAD Area

After the processes illustrated in Figures 19 and 20 have taken place, you may wish to know how much space your file actually takes in the designated RAD area. To find out, the Editor !#MAP command is used as shown in Figure 21.

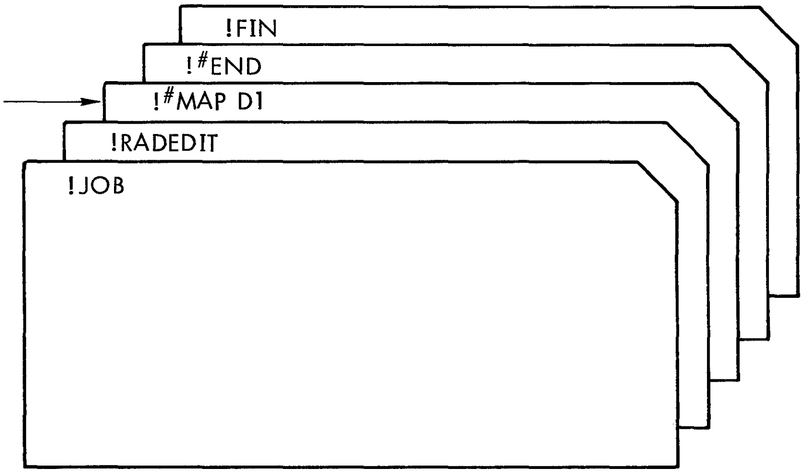
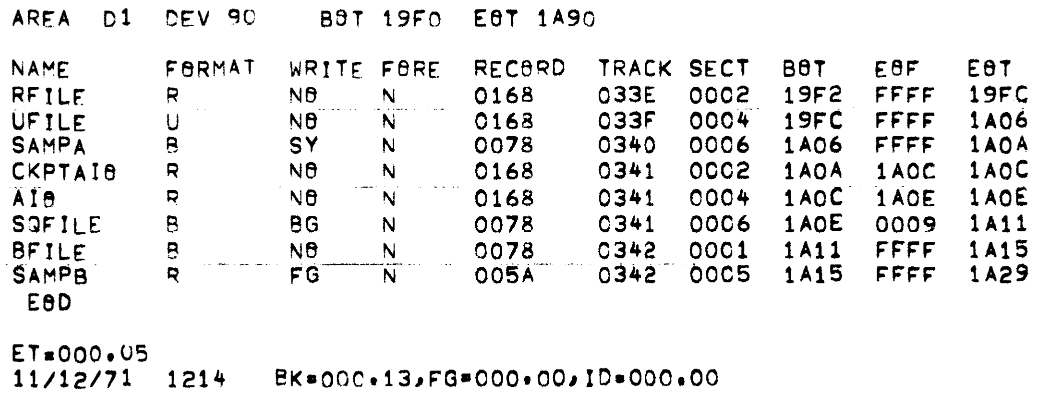


Figure 21. Output a RAD Map

This map provides a list of all the files in D1 area with their corresponding records and file sizes as shown in Figure 22.



```
AREA  D1  DEV 90    BØT 19F0  EØT 1A90

NAME      FØRMAT   WRITE FØRE  RECØRD  TRACK SECT  BØT   EØF   EØT
RFILE     R        NØ    N     0168    033E  0002  19F2  FFFF  19FC
UFILE     U        NØ    N     0168    033F  0004  19FC  FFFF  1A06
SAMPA     B        SY    N     0078    0340  0006  1A06  FFFF  1A0A
CKPTAIØ   R        NØ    N     0168    0341  0002  1A0A  1A0C  1A0C
AIØ       R        NØ    N     0168    0341  0004  1A0C  1A0E  1A0E
SQFILE    B        BG    N     0078    0341  0006  1A0E  0009  1A11
BFILE     B        NØ    N     0078    0342  0001  1A11  FFFF  1A15
SAMPB     R        FG    N     005A    0342  0005  1A15  FFFF  1A29
 EØD

ET=000.05
11/12/71  1214   BK=000.13,FG=000.00,ID=000.00
```

Figure 22. RAD Area Map Example

RAD area maps are a necessity, of course, when multiple users are creating files in a given area. Otherwise, the individual users would not know whether space is available in the area, whether a file with the same desired name already exists, etc.

HOW TO ACCESS A FILE

Now that we have created a file and initialized it with data, the problem of how to access the file remains. Files are read or written in background user programs in the same way that line printers, card readers, or other devices are accessed. The linkage between RAD files and your program is provided via !ASSIGN or commands inserted into your program deck. (See the !ASSIGN discussion in Chapter 2 of the RBM Reference Manual.) Foreground users access data files through Monitor service routines (Read/Write) coded into their programs.

HOW TO CREATE A PROGRAM FILE

Before discussing th technique for creating a program file that can be called into core for execution, a quick review of the two methods of accessing a file for either a Read or Write operation is necessary.

SEQUENTIAL ACCESS

When you use sequential access, you access a RAD file on a record-by-record basis (see definition of record given previously in this chapter) in exactly the same way that you access a data file on magnetic tape. This method can be used for blocked, unblocked, or compressed files.

RANDOM ACCESS

To perform random access you must supply the relative record number of the start of the Read/Write request and the number of bytes to be transferred, where "relative record number" is the number of a granule relative to the start of the file for an unblocked file, or the relative logical record number relative to the start of the file for a blocked file. (A granule is defined by the user to be one or more sectors.) The default (and typical) size is one sector for unblocked files. Addressing files by granules allows direct access to be independent of the RAD sector size.

CREATION PROCEDURES

All files are created in the same manner regardless of the functions for which they are to be used. This reduces general rules for program files to the following:

- To save a load module (executable program) in a user-defined file, the file must be created with an Editor !#ADD command before a load module is stored into it.
- The defined file must be a random access file.

When you call in the Overlay Loader via the !OLOAD command to create a load module the Loader will print out the size of the load module on the load map, assuming you used one of the Loader map options. This size is given in sectors and since a load module is a random access file, this is the value to use as the FILE parameter entry on the Editor !#ADD card.

If you do not know the size of the load module until after it has been created, how do you know how to !#ADD a file of precisely the right size? There are two solutions:

1. Create the module on the OV file, which is the default output file for load modules. Look up the granule size on the resulting load map and use this number as the FILE parameter on an Editor !#ADD card. Use an Editor !#FCOPY card to copy the OV contents to the newly created file.

   Example:

   Assume the load module created with the Overlay Loader used 20 granules in the RBMOV file. Allot a file called FTEST of 20 granules in area D1 and copy OV out to the new file. The deck structure given in Figure 23 would copy OV to the new file.

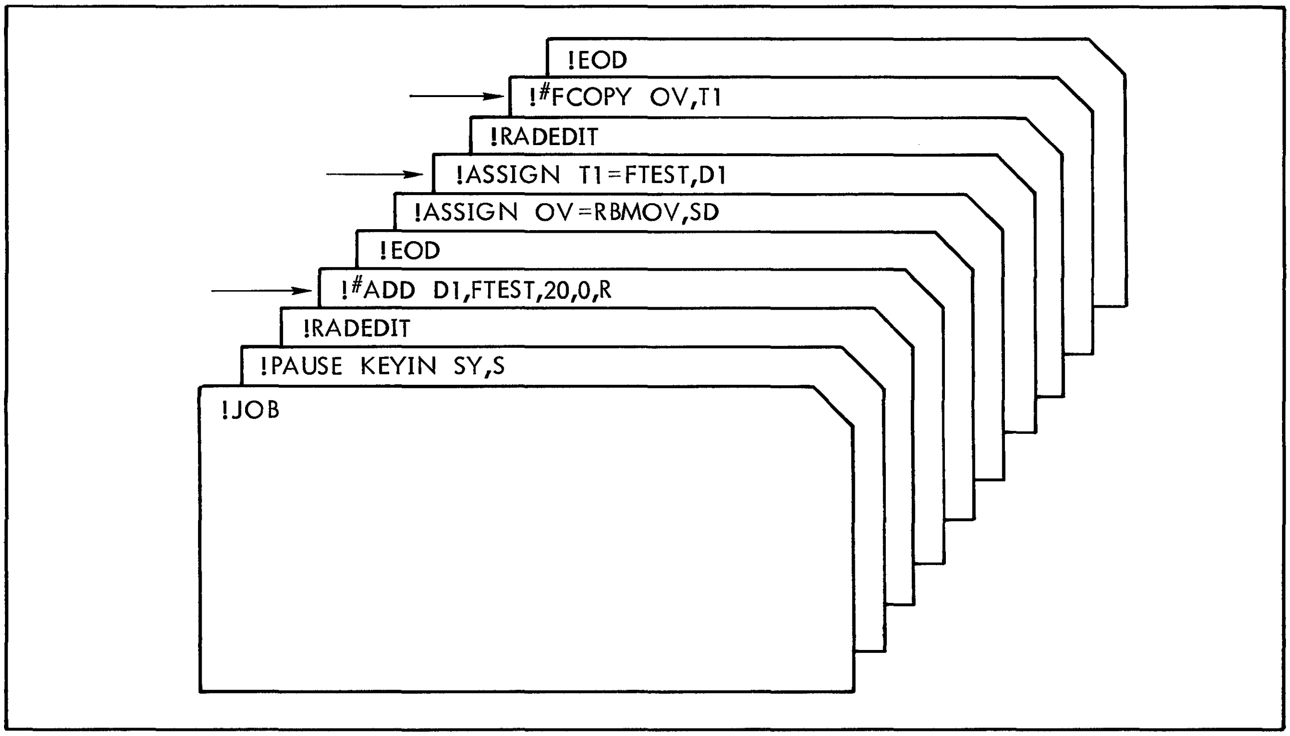


Figure 23. File Creation with Specific Granule Allotment

2. Use an !#ADD command to create a file that you know is sufficiently large for the load module. Put the load module into this file via an !ASSIGN command prior to an !OLOAD command in the command stack, and then follow with Editor !#TRUNCATE and !#SQUEEZE commands that have a corresponding file specification.

   Example:

   Assume a new file called TEST1 in area D1 is to contain a load module of unknown length. The deck structure given in Figure 24 would allocate all available space to the file, load the module into the file, and then recover the unused space.

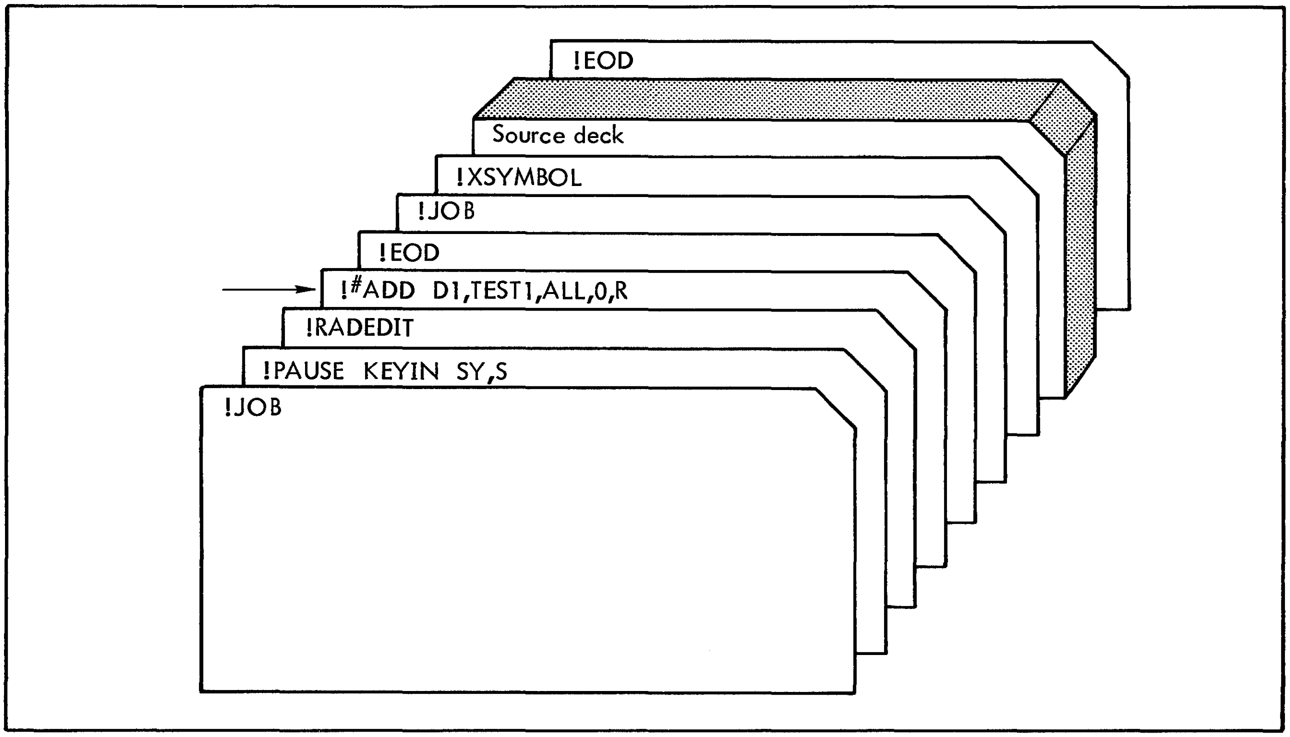
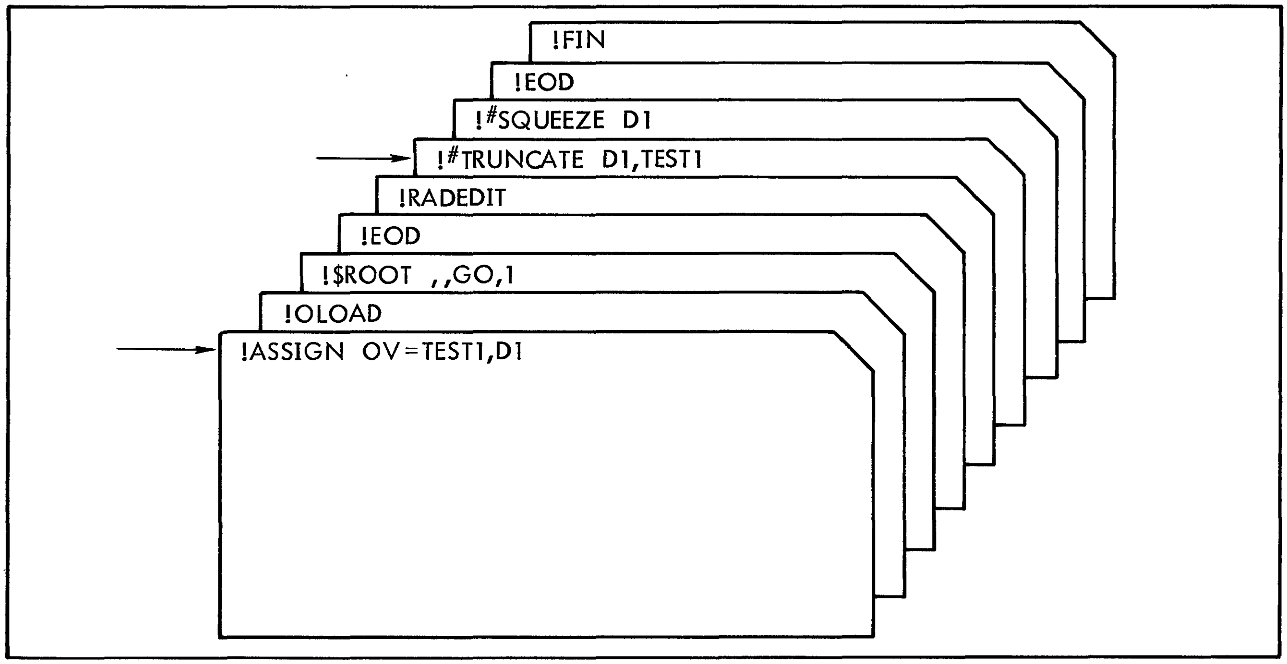


Figure 24. File Creation with Granule Over-Allotment



```
!FIN
!EOD
!#SQUEEZE D1
!#TRUNCATE D1,TEST1
!RADEDIT
!EOD
!$ROOT ,,GO,1
!OLOAD
!ASSIGN OV=TEST1,D1
```

Figure 24. File Creation with Granule Over-Allotment (cont.)

HOW TO CREATE A NEW LIBRARY

A library consists of six files: MODIR, EBCDIC, EDFRF, BDFRF, MDFRF, MODULE. These files are allocated in either the System Library (SL) or User Library (UL) areas, as appropriate (these are the only two areas that can have libraries), and the files must have the exact names given above and be in random format.

For instructions on how to compute the sizes of each file for a particular library, see the RAD Editor chapter in the RBM Reference Manual.

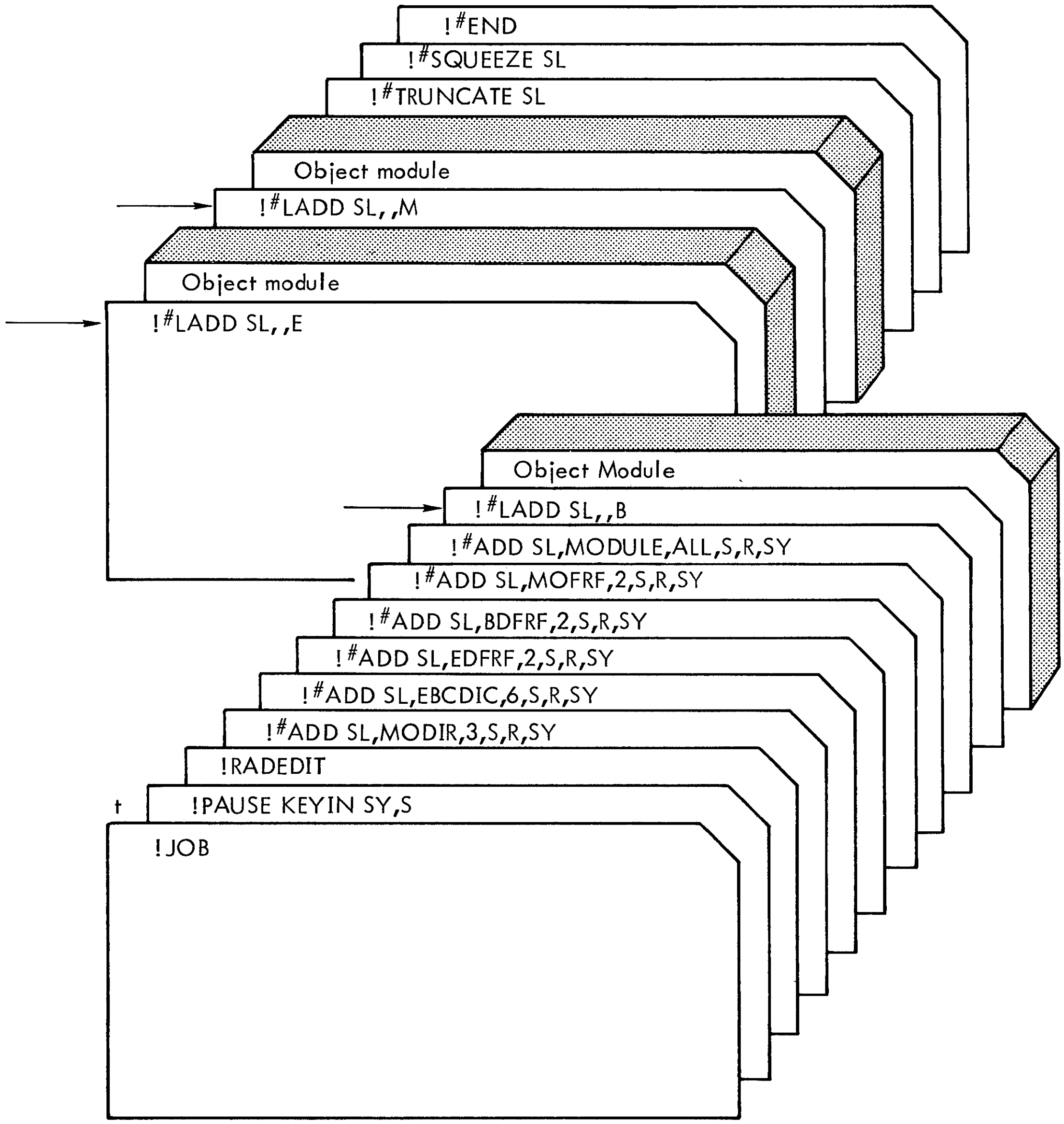
All library files are random files. In the !#ADD command example

```
!#ADD SL,MODIR,6,S,R,SY
```

the "6" parameter following the file name MODIR means a size of six records in the SL area. Since "RECORD" is S, you have specified that you wanted RECORD=SECTOR size. The FORMAT "R" specifies an unblocked random access file. The Write parameter "SY" means write permitted when the SY key-in is in effect. When computing the sizes of the files from the formulas in the RAD Editor chapter in the RBM Reference Manual, remember that the results will be in granules or the number of records needed. If you do not have enough information to compute the size of each file, allocate them a greater number of sectors than are required.

After all six files have been created with #ADD commands, the #LADD command enters the library routines into the defined four files, depending on the library code parameter on the !#LADD command: Basic (B), Main (M), or Extended (E) as defined in Figure 25. The same basic method is used to set up the User Library.

Note: If you ever plan to add new programs to a library or replace existing library routines with larger routines, omit the !#TRUNCATE command.



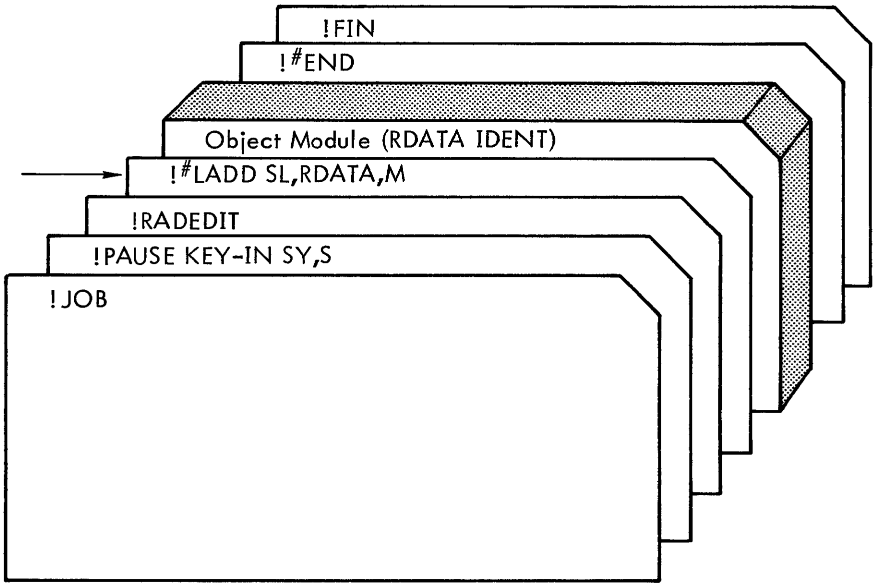
```
!#END
!#SQUEEZE SL
!#TRUNCATE SL
Object module
!#LADD SL,,M
Object module
!#LADD SL,,E
Object Module
!#LADD SL,,B
!#ADD SL,MODULE,ALL,S,R,SY
!#ADD SL,MOFRF,2,S,R,SY
!#ADD SL,BDFRF,2,S,R,SY
!#ADD SL,EDFRF,2,S,R,SY
!#ADD SL,EBCDIC,6,S,R,SY
!#ADD SL,MODIR,3,S,R,SY
!RADEDIT
† !PAUSE KEYIN SY,S
!JOB
```

[†] Note that SY,S Key-in is required to write into the SL area. This would also be required for creating a library in the UL area.

Figure 25. Input Library Files

HOW TO ADD A LIBRARY ROUTINE

Since the library already exists, the method for adding a new routine is quite simple, as illustrated in Figure 26. This example is essentially the same deck structure used to create the library, except that you ADD onto the end of the existing library files. The example assumes that BI has been assigned to the card reader.



```
!FIN
!#END
Object Module (RDATA IDENT)
!#LADD SL,RDATA,M
!RADEDIT
!PAUSE KEY-IN SY,S
!JOB
```

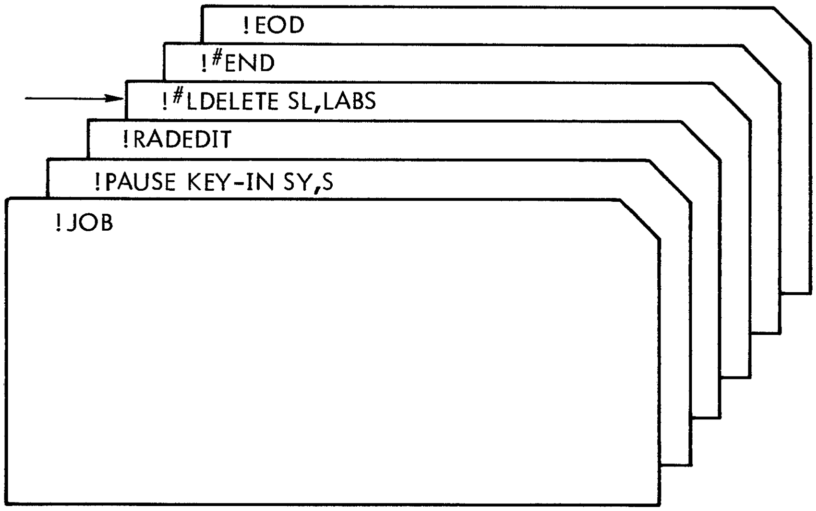
Note: The #LADD command adds an object module to the designated library. The "identification" parameter identifies the object module being loaded.

Figure 26. Add a Library Routine

HOW TO DELETE A LIBRARY ROUTINE

To delete a routine from a library it is first necessary to determine the name associated with each routine in that library via a RAD Editor !#LMAP command. Besides listing a name for each routine on the RAD map, it also lists all other entry points or data words in each routine.

The !#LDELETE command deletes an object module specified in the identification parameter from the designated library as shown in Figure 27.

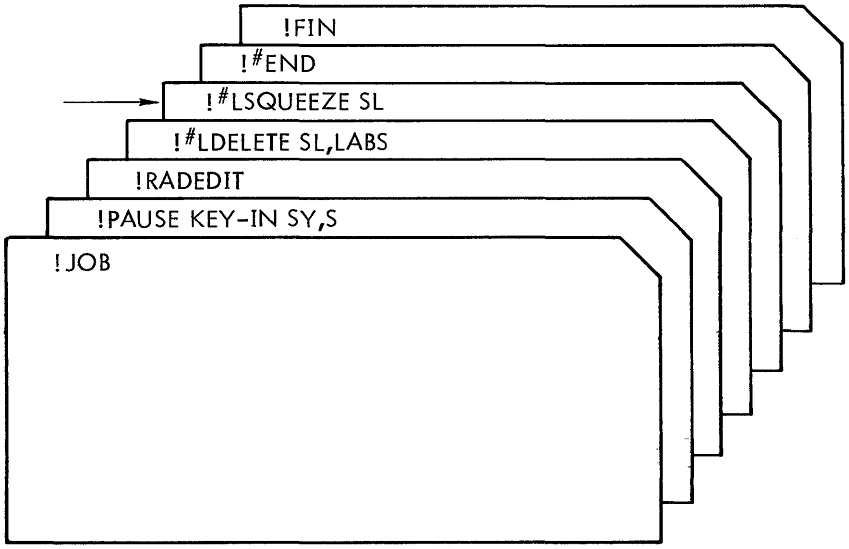


```
!EOD
!#END
!#LDELETE SL,LABS
!RADEDIT
!PAUSE KEY-IN SY,S
!JOB
```

Figure 27. Delete a Library Routine

HOW TO RECOVER UNUSED LIBRARY SPACE

It is sometimes desirable to recover previously used file space and to make it available for storing other library routines after a routine has been deleted. The !#LSQUEEZE command is used to release the space. The !#LSQUEEZE command shown in Figure 28 would restore the space formerly occupied by the LABS routine in the System Library files. This command does not change the file sizes allocated but compacts the data in the files.



```
!JOB
!PAUSE KEY-IN SY,S
!RADEDIT
!#LDELETE SL,LABS
!#LSQUEEZE SL
!#END
!FIN
```

Figure 28. Library Space Recovery

HOW TO REPLACE A LIBRARY MODULE

During the evolution of a routine in a User Library, updates are very common in the development of the final operational version. The !#LREPLACE command is used to replace an existing intermediate object module with a newer object module bearing the same identification. This command will not recover the space occupied by the replaced routine.

Example:

Assume a BASIC library routine called "BLIB" that is located in the User Library. To replace this routine with an updated version, the deck structure shown in Figure 29 would be used. (The example assumes that BI has been assigned to the card reader.)

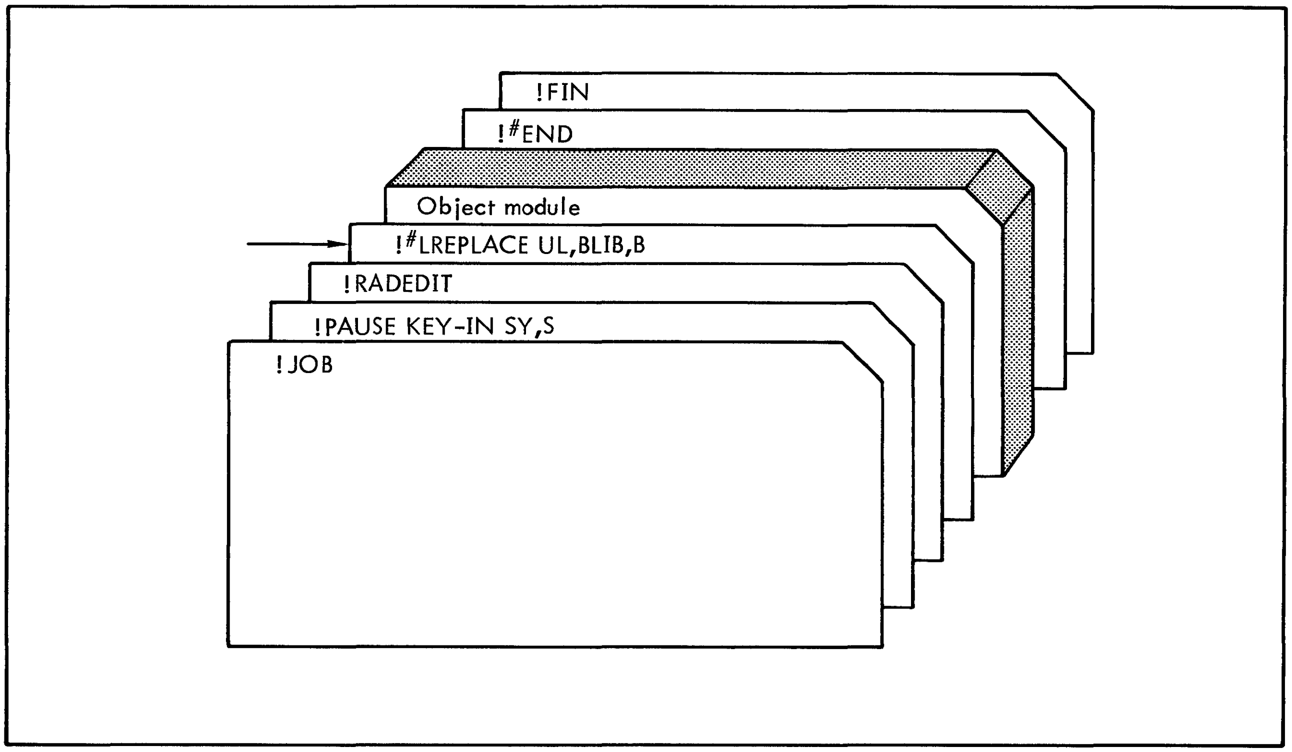


Figure 29. Replace Library Object Module

Note that the space used by the original BLIB would not be recovered, and that the new BLIB would reside at the end of the current library.

5. HOW TO BUILD AN OVERLAY PROGRAM

Use of the Overlay Loader with nonoverlay programs has been covered in the examples given in the FORTRAN and Extended Symbol chapters, and it is now necessary to discuss some of the ways the Loader can be used to create segmented (overlay) programs.

The only purpose in overlaying a program is to minimize core size requirements. Since there is a slight degradation in response time for each level of overlay within a foreground program, it is obvious that such programs should not have any more overlay levels than are absolutely necessary.

Before discussing overlay techniques, the term ROM must be fully defined:

- A ROM is a Relocatable Object Module, and is the only type of object module the Overlay Loader will accept to form the load module. "Relocatable" means that the execution location in core for the module is determined by the Overlay Loader at load time.

The other type of object module is absolute; that is, a fixed execution location is determined by the user at assembly time through use of the Extended Symbol (or SYMBOL) ASECT and ORG directives. Absolute object modules are loaded by another processor called the Absolute Loader, and are always executed in the same predetermined location unless reassembled.

Every reference to "object module" in this manual always means ROM, and this term is used by the Overlay Loader when it outputs a load map or diagnostic message.

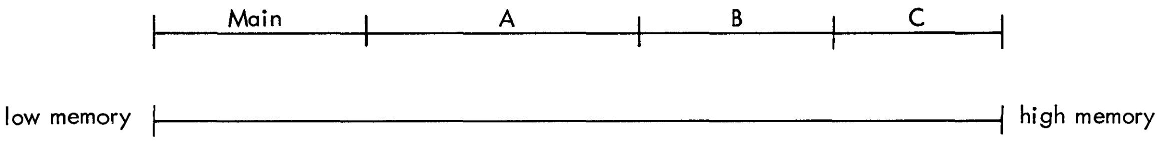
The material and examples in this chapter do not encompass every option available to the Loader user; rather, the chapter presents what constitutes an overlay´job and the interface between basic structures and several Loader control commands. A study of the examples will make the significance of other Loader commands and options (i.e., the !$TCB command described in later chapters) more apparent.

The presence or absence of a single option on the !OLOAD command (F), causes the Loader to define the resulting program as either background or foreground (B is the default case). All of the examples below are relevant to both foreground and background programs.

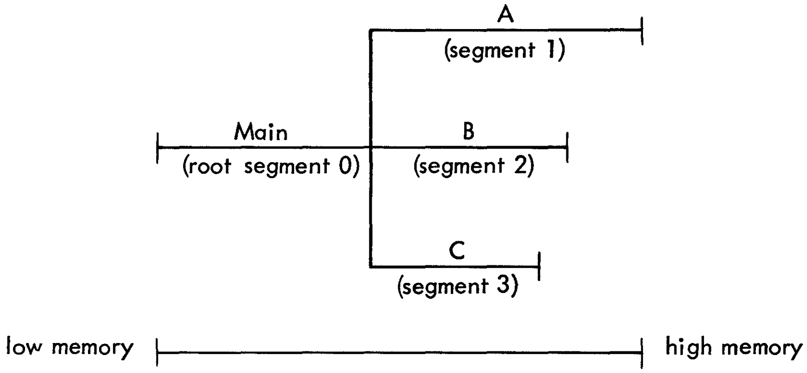
Assume the following program:

1. A Main program that calls in subroutines A, B, and C.
2. Subroutine A does not reference subroutines B or C.
3. Subroutine B does not reference subroutines A or C.
3. Subroutine C does not reference subroutines A or B.

This program could be loaded into memory in nonoverlay form to appear as

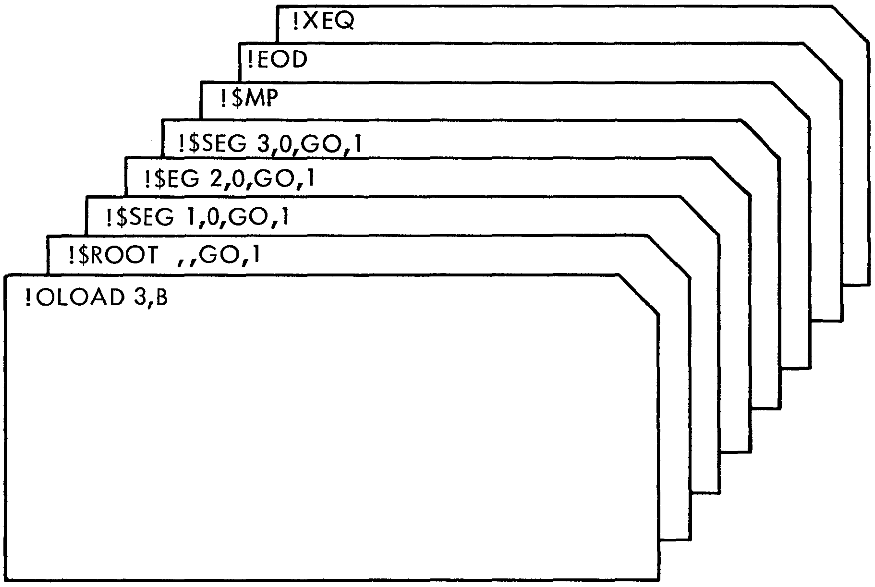


However, if the program was segmented, it would take up less memory and would appear as



where subroutine A is not residing in memory (saved on the RAD) when subroutine B is in memory, and vice versa. Note that the root segment is always resident and is designated as segment 0. The Main program (root segment) has the responsibility of calling the appropriate segment into memory (see "Communication Between Segments" later in this chapter).

Assuming the object modules are residing on the GO file in the order Main, A, B, and C, the structure could be created by the following set of commands which would create the overlay structure pictured above:



```
!XEQ
!EOD
!$MP
!$SEG 3,0,GO,1
!$EG 2,0,GO,1
!$SEG 1,0,GO,1
!$ROOT ,,GO,1
!OLOAD 3,B
```

The structure is defined by the !$ROOT and !$SEG cards. The first !$SEG card defines segment number 1 to be connected to segment number 0 (root). The second and third !$SEG cards correspondingly define segments number 2 and 3, also connected to the root.

For another example, assume the following program:

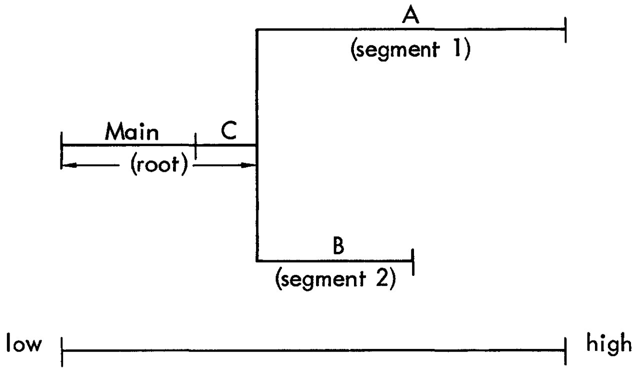
1. A Main program that calls subroutines A and B.

2. Subroutine A calls subroutine C.

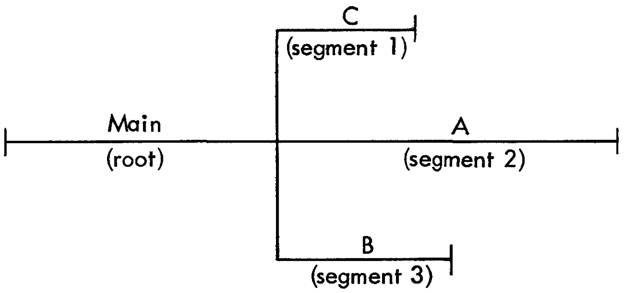
3. Subroutine B calls subroutine C.

4. Subroutine A does not reference subroutine B.

This program could be segmented in the following manner:

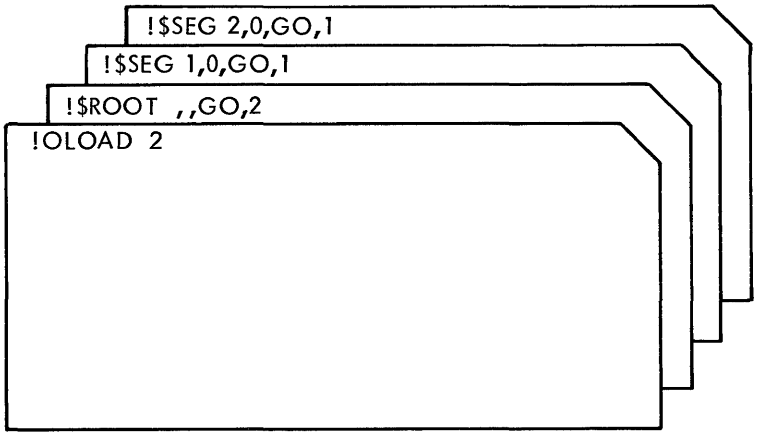


allowing C to be available to both subroutines A and B without an extra copy of C being necessary. However, assuming the order on the GO file is Main, C, A, and B, then using the previous control command string would give you the unworkable structure



which does not allow the required calls. Since this structure does not allow A or B to call C and then to continue upon C's return (C will wipe out A or B in memory), it is obvious that the simple structure we used above cannot be used to solve this problem.

The following control cards could be used instead:



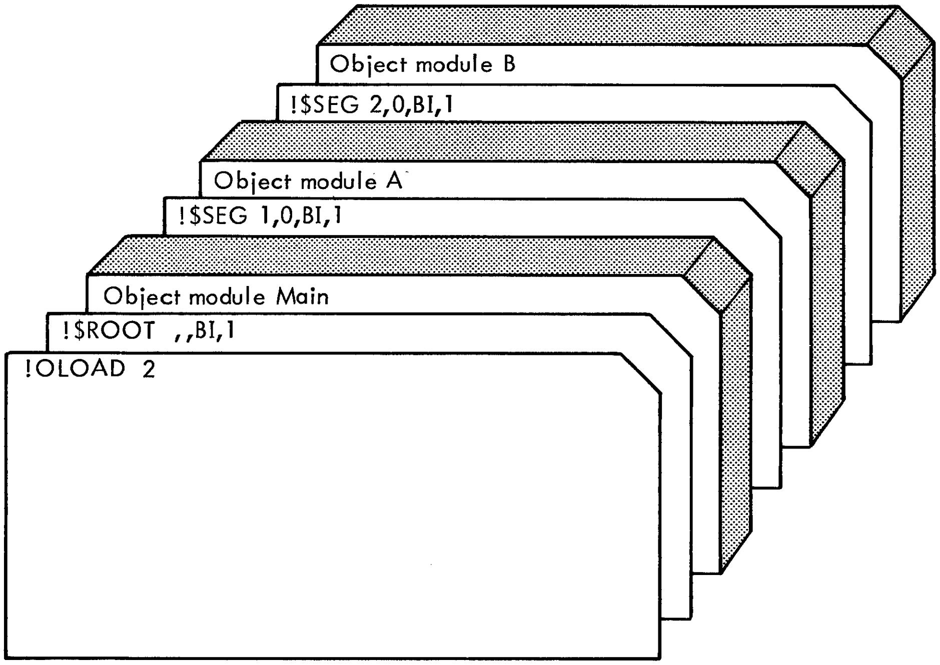
```
!$SEG 2,0,GO,1
!$SEG 1,0,GO,1
!$ROOT ,,GO,2
!OLOAD 2
```

These commands put the first two routines on the OV file in the root, and one routine each in segments 1 and 2.

Full understanding of the use of the !$ROOT and !$SEG commands is imperative for segmenting programs. Assume the following program:

1. The Main program calls in subroutines A and B.
2. Subroutine A does not reference routine B.
3. Subroutine B does not reference routine A.
4. Main, A, and B are to be input via the card reader (the binary input (BI) device).

The following command structure is needed (assuming the commands are also read in from the card reader):

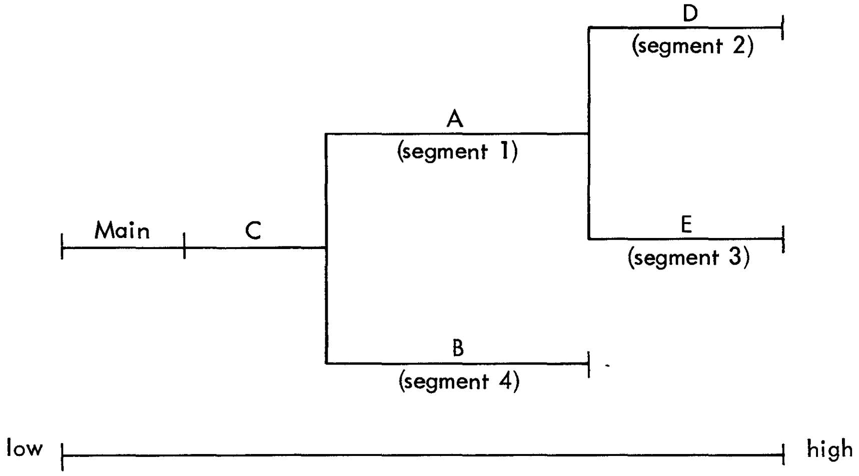


Whenever more than one module is needed, the number is required, or a blank may be used. If no number is specified the Loader expects to encounter an !EOD command following the object modules before any other Loader subcommand is encountered. Let's consider a slightly more complex example.

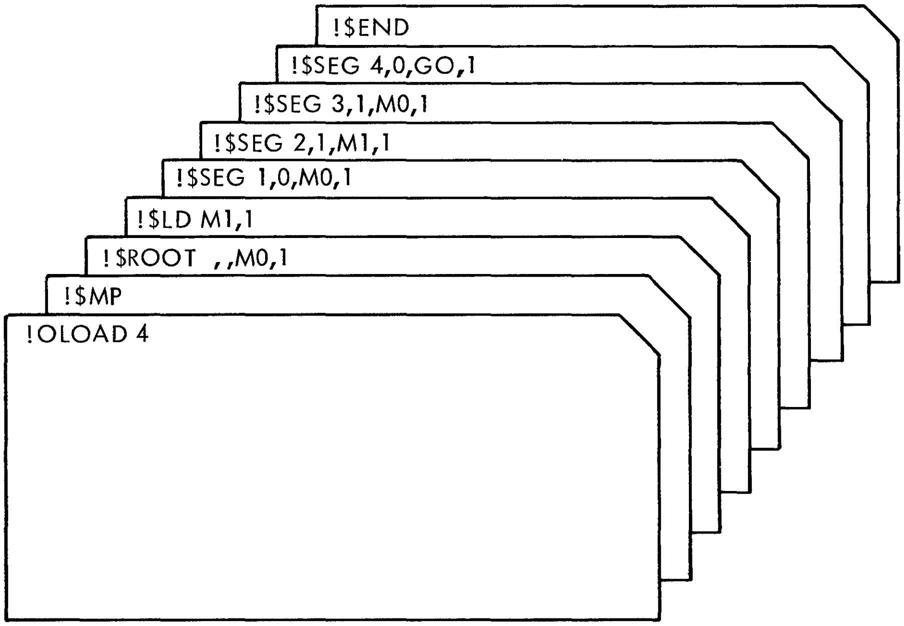
Assume the following program:

1. A Main program that calls subroutines A and B.
2. Subroutine A calls subroutine C, D, and E.
3. Subroutine B calls subroutine C.
4. Subroutine D calls subroutine C.
5. Subroutine E does not reference any routine.
6. Main, A, and E are on magnetic tape; oplabel M0.
7. C and D are on magnetic tape; oplabel M1.
8. B is on the GO file.

Thus, the overlay structure is

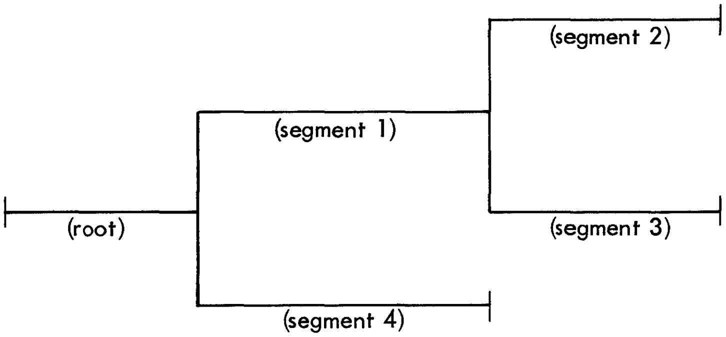


and the control command sequence could be



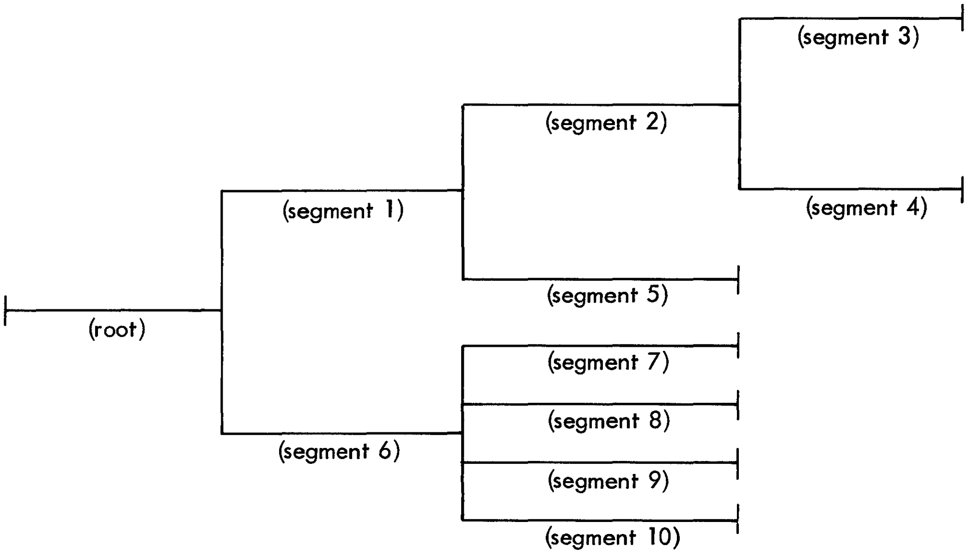
```
!$END
!$SEG 4,0,GO,1
!$SEG 3,1,M0,1
!$SEG 2,1,M1,1
!$SEG 1,0,M0,1
!$LD M1,1
!$ROOT  ,,M0,1
!$MP
!OLOAD 4
```

Whenever the overlay structure is such that some segments link to segments other than the root, there is an order to the !$SEG commands that must be followed. This can be illustrated by the following example:

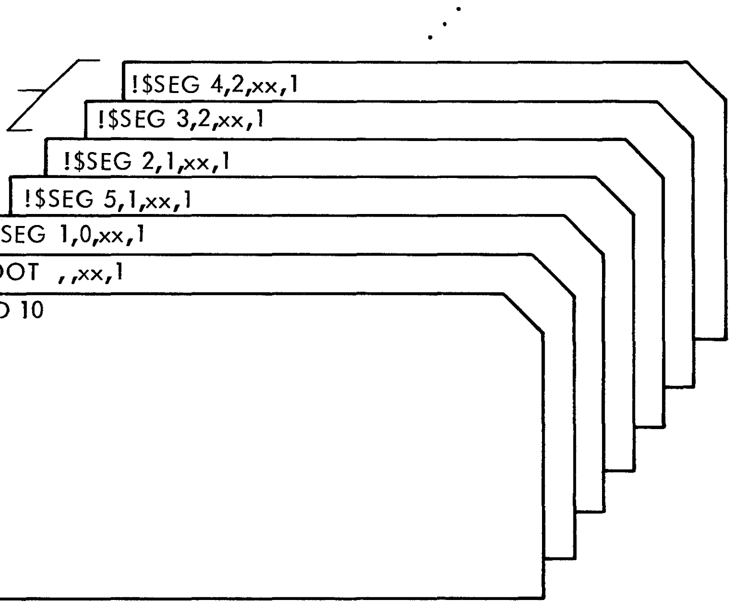


As soon as segment 1 is defined, all segments linking to it (segments 2 and 3) must be defined via !$SEG commands before !$SEG 4,0, can be encountered.

Let's take a look at another example:



In this case, after a !$SEG 1 command is input, either segment 2 or 5 may be defined. However, whenever segment 2 is defined, then all segments linked to it must be listed next (i.e., segments 3 and 4). There are several deck structures that could be used to construct the overlay above. One of these deck structures is shown below:

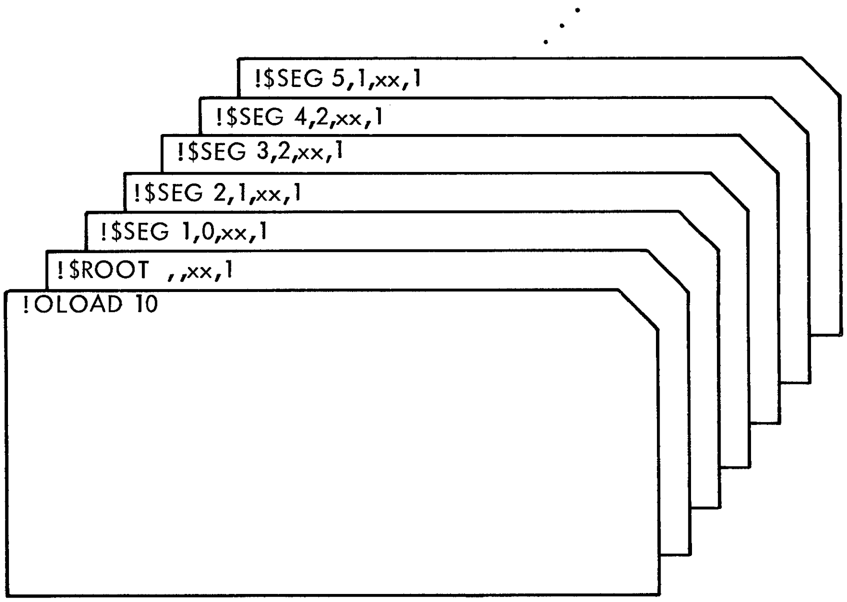


```
!$SEG 4,2,xx,1
!$SEG 3,2,xx,1
!$SEG 2,1,xx,1
!$SEG 5,1,xx,1
!$SEG 1,0,xx,1
!$ROOT ,,xx,1
!OLOAD 10
```

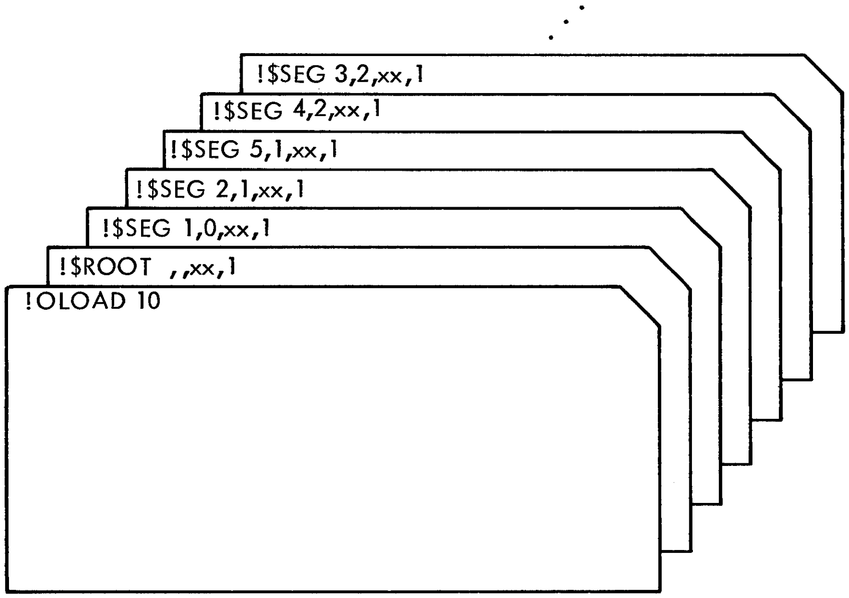
These two segment definitions could be interchanged since no other segment links to them.

where xx denotes the oplabel where the module resides

Another deck structure is as follows:



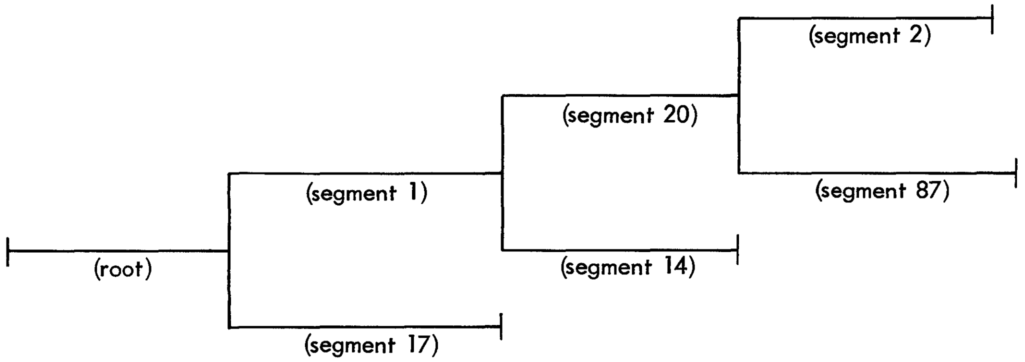
However, the following deck structure cannot be used since all segments linked to 2 must follow the !$SEG 2,1,xx,1 command:



There is no general rule regarding the actual numbers selected to designate segments. Segment numbers may range from 1 to 255 and may appear in any order. The segment numbers are needed for communication between the user program and the Segment Loader during execution.

COMMUNICATION BETWEEN SEGMENTS

Since the primary objective of the RBM operating system is to provide fast response and processing for real-time users, there is no automatic loading of overlay segments as the result of a reference to a routine within the segment. If the root only occasionally uses segment 2 in the following structure, then it can call it in, use it, and continue on without having waited for segments 1 and 20 to be brought in.



This permits the real-time user to manage the overlay process in the most efficient manner for a given program.

The exact procedure for calling in segments is different for the FORTRAN user and the assembly language user.

FORTRAN SEGMENT CALLS

A FORTRAN user calls in a segment with the statement

CALL SEGLD (I)

or

CALL SEGLD (I, J)

where

I is the segment number.

J is the file from which the segment is to be loaded.

In the case where J is not designated, the PI (processor input) file is assumed.

However, a call to SEGLDX causes an overlay segment to be loaded and control transferred to the transfer address of the segment. A call to SEGLDX has the form

CALL SEGLDX (I)

or

CALL SEGLDX (I, J)

where

I is the segment number.

J is the file from which the segment is to be loaded.

In the case where J is not designated, the PI (processor input) file is assumed.

ASSEMBLY SEGMENT CALLS

Assembly language users are provided with system calls for segment loading and have a choice as to how to load segments. They may

1. Load the segment and return.

2. Load the segment and transfer to the "starting address" of the segment upon completion of the load. The "starting address" is defined by a label on the END card of the assembly, or the FWA (first word address) of the segment by default.

3. In foreground, request the segment to be loaded and do not wait for the load to take place. Instead, specify a "loading-complete" receiver, which would transfer to a specified address when the I/O interrupt becomes active (any processing performed by the "end-action" routine should be kept as short as possible to prevent degradation of response time for lower-priority interrupts).

SEGMENT COMMUNICATION USING COMMON AREAS

Blank COMMON

Both FORTRAN and Extended Symbol users can define blank COMMON either in their source code or through the cmn parameter on the !OLOAD card.

In allocating COMMON for background programs, the Loader compares the cmn parameter with the first nonzero COMMON size allocation value encountered in loading and employs the larger of these two values. The COMMON base is set by subtracting the COMMON size from the upper limit of core memory.

For foreground programs having blank COMMON, cmn denotes the base (i.e., first word address) of COMMON. In this case, the effective upper limit of the program is cmn plus the largest COMMON size allocation value encountered in loading. For foreground programs in which COMMON is allocated but in which cmn has not been specified, the COMMON base is set by subtracting the first nonzero COMMON size allocation value encountered from the upper limit of nonresident foreground memory. Foreground programs that have no COMMON may use the cmn parameter to specify an upper limit for the program, if the address specified by cmn is higher than the root FWA. If the program exceeds the limit, the Loader aborts. The default value of the upper limit for foreground programs without COMMON is the upper limit of the nonresident foreground area.

Foreground loads may specify the cmn parameter at a lower address than the root FWA; in which case, the end of nonresident foreground is the program upper limit. A check is made at the end of the load to determine whether the COMMON allotment overlaps the root. If it does, the warning message "OLERR CO" is printed out but no error severity level is set.

See also the subsection "Blank COMMON Storage" in Chapter 8 of this manual.

Labeled COMMON

Labeled COMMON is defined in the source code and labeled COMMON areas precede the module in which they are first defined.

- A fresh copy of the labeled COMMON is brought into memory as each segment with a labeled COMMON is loaded. This means that any data the programmer wishes to save between segments that occupy the same memory area should be either in blank COMMON, or in labeled COMMON in a segment that does not get overlaid while the data is still pertinent.

- The FORTRAN IV compiler is capable of using labeled COMMON directly, but Extended Symbol must first REF the labeled COMMON block, which will give the start address of the block; access isachieved through the source code.

HOW TO READ A LOAD MAP

The primary intention in outputting a load map is to simplify the programmer's job when a set of programs fails to satisfy all external references. Additionally, various size and location information is output to simplify the task of file and core allocation. The Load Map shown in Figure 30 is the most extensive of the map options and is produced by the control command !$ML.

For the !$MP command, the same information is output except for the Public Library and Monitor service DEFs (the list of names between the lines marked OVERLAY TASK and ROOT), and Library symbols (those denoted by an L immediately to the right of the symbol name; X:ERROR, for instance).

The !$MS command outputs only the header lines; that is, the lines marked OVERLAY TASK, ROOT, SEGMENT, ERRSEV, and END MAP.

All size and location data is in hexadecimal. The terms "core location" and "address" in this discussion refer to core locations at execution time.

The OVERLAY TASK line gives information about the entire overlay cluster. From left to right, the various items have the following meaning:

BA: means a background program; foreground programs cause "FO" to be printed. A map will show either BA or FO.

ORG: means first (lowest) core location of the program's temporary stack (see RBM Reference Manual).

HLOC: means highest core location of the overlay cluster.

CBAS: means first core location of the blank COMMON area.

CSIZ: means size of the blank COMMON area in words.

UMEM: means unused memory, the difference between HLOC and CBAS (the amount of memory available to the Monitor for blocking buffers).

SECT: means the number of sectors required by the overlay cluster in a processor file.

Following the first line is a list of all the DEFs in the Public Library. These are flagged by the P to the right of the name; the M indicates that this routine was placed in the Public Library in the Main mode. The other possible modes are Extended and Basic and are flagged by an E and a B, respectively.

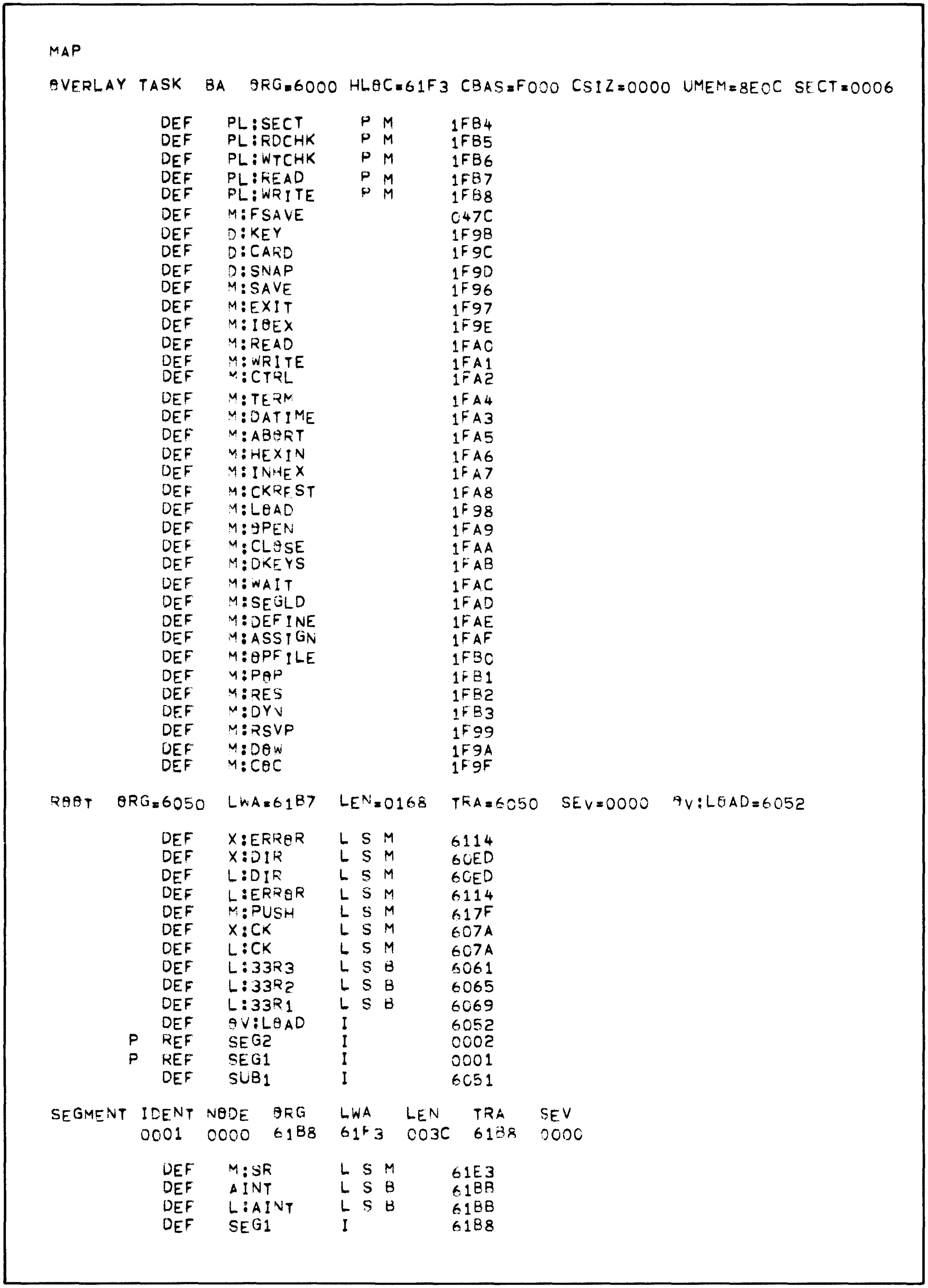
The remainder of the DEFs in this list are the various Monitor service routines. The numbers to the right of the Public Library DEFs and the Monitor service DEFs are their respective core locations in the Monitor's "Transfer Vector Table" (see Chapter 4 of the RBM Reference Manual), which is a table maintained by the Monitor for its own use in locating these routines.

This list will not change (for any load with an !SML command) unless a user changes the Public Library, in which case only the Public Library DEFs will change.

The ROOT line contains the following information:

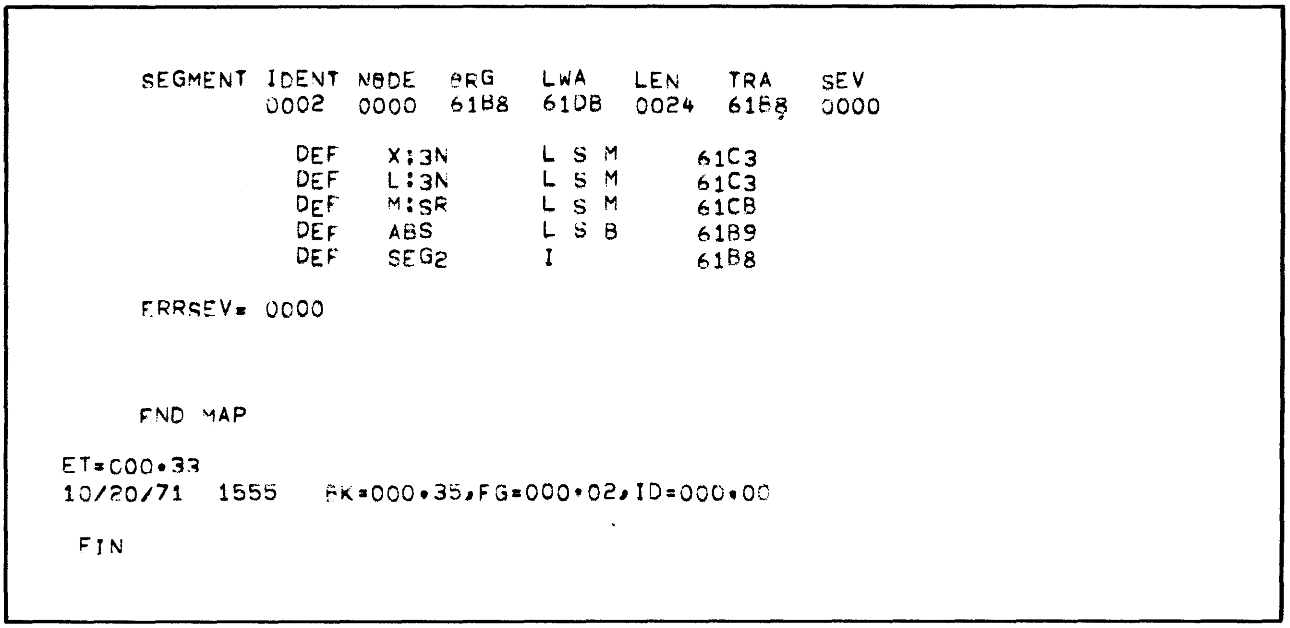
ORG: means the first core location of the root's program section for a background program, or the location of the TCB for a foreground program.

LWA: means the last core location of the root (including whatever library routines are in the root).



```
MAP

ØVERLAY TASK  BA  ØRG=6000 HLØC=61F3 CBAS=F000 CSIZ=0000 UMEM=8E0C SECT=0006

          DEF    PL:SECT      P M      1FB4
          DEF    PL:RDCHK     P M      1FB5
          DEF    PL:WTCHK     P M      1FB6
          DEF    PL:READ      P M      1FB7
          DEF    PL:WRITE     P M      1FB8
          DEF    M:FSAVE               047C
          DEF    D:KEY                 1F9B
          DEF    D:CARD                1F9C
          DEF    D:SNAP                1F9D
          DEF    M:SAVE                1F96
          DEF    M:EXIT                1F97
          DEF    M:IØEX                1F9E
          DEF    M:READ                1FA0
          DEF    M:WRITE               1FA1
          DEF    M:CTRL                1FA2
          DEF    M:TERM                1FA4
          DEF    M:DATIME              1FA3
          DEF    M:ABØRT               1FA5
          DEF    M:HEXIN               1FA6
          DEF    M:INHEX               1FA7
          DEF    M:CKREST              1FA8
          DEF    M:LØAD                1F98
          DEF    M:ØPEN                1FA9
          DEF    M:CLØSE               1FAA
          DEF    M:DKEYS               1FAB
          DEF    M:WAIT                1FAC
          DEF    M:SEGLD               1FAD
          DEF    M:DEFINE              1FAE
          DEF    M:ASSIGN              1FAF
          DEF    M:ØPFILE              1FB0
          DEF    M:PØP                 1FB1
          DEF    M:RES                 1FB2
          DEF    M:DYN                 1FB3
          DEF    M:RSVP                1F99
          DEF    M:DØW                 1F9A
          DEF    M:CØC                 1F9F

RØØT   ØRG=6050   LWA=61B7   LEN=0168   TRA=6050   SEV=0000   ØV:LØAD=6052

          DEF    X:ERRØR      L S M    6114
          DEF    X:DIR        L S M    60ED
          DEF    L:DIR        L S M    60ED
          DEF    L:ERRØR      L S M    6114
          DEF    M:PUSH       L S M    617F
          DEF    X:CK         L S M    607A
          DEF    L:CK         L S M    607A
          DEF    L:33R3       L S B    6061
          DEF    L:33R2       L S B    6065
          DEF    L:33R1       L S B    6069
          DEF    ØV:LØAD      I        6052
        P REF    SEG2         I        0002
        P REF    SEG1         I        0001
          DEF    SUB1         I        6051

SEGMENT IDENT NØDE  ØRG   LWA   LEN   TRA   SEV
        0001  0000  61B8  61F3  003C  61B8  0000

          DEF    M:SR         L S M    61E3
          DEF    AINT         L S B    61B8
          DEF    L:AINT       L S B    61BB
          DEF    SEG1         I        61B8
```

Figure 30. Loap Map Example



```
      SEGMENT IDENT NØDE  ØRG   LWA   LEN   TRA   SEV
              0002  0000  61B8  61DB  0024  61B8  0000

                DEF   X:3N      L S M      61C3
                DEF   L:3N      L S M      61C3
                DEF   M:SR      L S M      61CB
                DEF   ABS       L S B      61B9
                DEF   SEG2      I          61B8

      ERRSEV= 0000



      END MAP

  ET=C00.33
  10/20/71   1555    BK=000.35,FG=000.02,ID=000.00

   FIN
```

Figure 30. Load Map Example (cont.)

LEN: means the overall root size in words.

TRA: means the root's entry point, which is determined by the argument of the END statement in a program written in assembly language.

SEV: means the error severity level; the highest error severity encountered in loading the root (either assembled or Loader-generated).

OV:LOAD: means the OV Load Table location, which is a 5-word-per-entry table generated by the Loader at the end of the root's program section. It contains information to allow the Monitor to locate and load into core the segments of the program at run-time. If there is only a root, no OV Load Table is generated, and instead of a core location, the word NONE is printed.

Following the ROOT line is a list of DEFs and REFs encountered while loading the root (including library routines). Each symbol (name) is error-flagged or identified by one or two characters as follows:

For DEFs: D means duplicate.

U means a DEF statement was declared (in the code) but no value was given (there was no label in the routine matching the argument of the DEF statement).

LC means a Labeled COMMON block was defined.

For REFs: U means unsatisfied; the Loader could find no matching DEF while loading this path.

P means a primary Reference (REF).

S means a secondary Reference (SREF).

To the right or each name are from one to three characters denoting the input source for that name. An "L" signifies a library file, "S" signifies the System Library (a "U" would signify the Users Library and a "P" would signify Public Library), and a "B", "E", or "M" signifies Basic, Extended, or Main mode respectively. An "I" to the right of a DEF means that it was encountered during the program load of this segment; for a REF, the "I" means that a matching DEF was found in a higher level segment.

The numbers in the rightmost column denote the first core location of the routine containing that DEF or the segment number containing a DEF which matches that REF; i.e., primary REF SEG1 is "satisfied" in segment number one.

Note that there are several pairs of Library routines whose core addresses are the same, e.g., X:DIR and L:DIR. This signifies that these are different DEFs (entry points) in a single module.

When the Loader satisfies an external REF in the same segment, the symbol table entry for that REF is changed to a DEF. Thus, although the program whose map is illustrated contained a REF SUB1 in the root, it is not printed since a DEF SUB1 was encountered during root loading.

The information in a SEGMENT line differs from that in a ROOT line in three items, as follows:

1. There is no OV:LOAD entry since only the root has an OV:LOAD table.
2. The number under the IDENT is the first parameter on the !$SEG card for that segment. It is the Segment Identifier.
3. The number under the "NODE" is the second parameter on the !$SEG card. It is the segment identifier of that segment to which this segment is connected.

The listed information following the SEGMENT line is exactly similar to that following a ROOT line.

The ERRSEV line shows the highest error severity level encountered during the entire program load and the END MAP line is self-explanatory.

LOADER PROCESS SUMMARY

During the loading process, the Loader must exist in background core together with the absolute load module version of the segment being loaded, plus various tables that are required for linkage. One such table is the Segment Table, which requires 10 (N+2) cells, where N is the number of segments specified on the !OLOAD card. The Segment Table is located at the highest available core locations.

Below the Segment Table are the various Symbol Tables, all of which have the same entry format. Each entry refers to a specific definition or reference. The entries are of variable length, from five to eight words, depending on the number of characters in the DEF or REF symbol.

The Permanent Symbol Table is a set of DEFs of the Monitor service routines; typically 230 words in length where there is no Public Library. If a Public Library exists, a Permanent Symbol Table entry is generated for each DEF in the library, again from five to eight words in length.

Below the Permanent Symbol Table, the Loader builds the Root Symbol Table, working from high core toward low core. The Loader is also building the absolute core image of the program, working from low core toward high core. (If the program should meet the Symbol Table, table overflow occurs and the load is aborted.) Whenever a REF is seen in the root, the currently existing Symbol Tables are searched for a satisfying DEF. If none is found, the REF is added to the root symbol table.

When the program portion of the root is completely loaded, it is written to the OV file, and the specified or default libraries are searched for any unsatisfied REFs in the Symbol Table. When all the library DEFs that match any unsatisfied REFs have been loaded, the library portion of the root is written to the OV file. Segment loading then proceeds similarly to the root loading, except that the program and library portions of each segment are loaded together, and then written to the OV file. The Segment Symbol Tables are built below the Root Symbol Table (again working from high toward low core) until a new path (or portion of a path) is begun, at which time that part of the

Segment Symbol Table that refers to the just-completed path is written to a temporary file (oplabel XI). Assuming no overflow or other problems are encountered, this process continues until the program is completely loaded. Then PASS2 of the load process commences, where all forward REFs are linked and any requested maps are output.

The important factors in segmenting a program are these:

- Program segment sizes (including library code)
- Number of segments
- Number of DEFs and REFs in each ROM
- Response time requirements

The diagram in Figure 31 illustrates the layout of core during the load process.

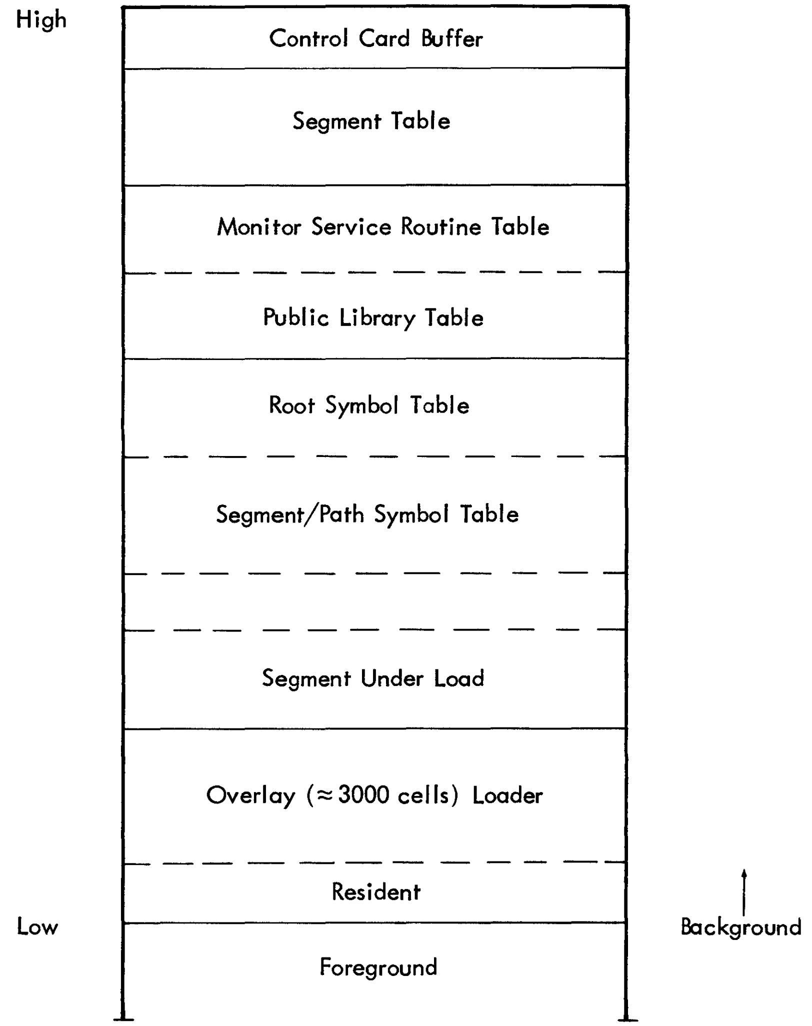


Figure 31. Core Layout During Loading

6. HOW TO USE MONITOR SERVICE ROUTINES

Most previous discussions and examples in this manual concerning communication with the Monitor have dealt with control commands and key-ins used in the background job stream prior to execution of an operational program. Monitor service routines are the means used by an executing program or task to request I/O, privileged operations, or other services from the Monitor. Service routines are the only means by which the background can perform I/O or privileged instructions.

You request a Monitor service routine by establishing a pointer to an argument list, a return link such as a register copy and increment (RCPYI), and a branch instruction coded into your source program prior to assembly, or compilation. The techniques for branching to the service routines are primarily the concern of assembly language users, since FORTRAN users normally call these services indirectly through FORTRAN Library routines. Note however, that Xerox ANS FORTRAN has an in-line assembly language capability via an "S" in column 1, and this capability permits branches to Monitor service routines without going through the FORTRAN Library.

There are three easy ways to access the various Monitor service routines. The first two are basically the same; that is, by branching indirectly through a fixed RBM Zero Table location. These locations are obtained by referencing the Transfer Vector Table for Monitor Services in Chapter 4 of the RBM Reference Manual. This table will supply the address associated with the desired routine, and Note 1 in the same table gives the steps to follow when coding.

In small developmental routines, it is often convenient to use the actual Zero Table location, as in this call to M:WRITE:

```
            LDX         =ARGLIST (pointer to argument list)

            RCPYI       P,L (return link)

            B           *X'CA' (indirect branch through vector location)
            .
            .
ARGLIST     DATA        X'3005', 'LO', BUFF, 18 (parameter list)

BUFF        TEXT        'ƀWRITE TO PRINTER'
```

However, in larger and more complex programs, it is often convenient to create a set of EQU directives that equate mnemonically satisfying labels to the appropriate core locations, as follows:

```
V:READ      EQU         X'C9'     }
                                  }   Definition of Zero Table locations
V:WRITE     EQU         X'CA'     }
            .
            .



            .
            .
            LDX         =ARGLIST  }

            RCPYI       P,L       }   Program section: call to M:READ

            B           *V:READ   }
            .
            .
```

As shown above, the branch indirect is accomplished exactly as in the first example; the difference being only one of convenience.

The third method is to reference the appropriate service routine at the beginning of each program element. This is easier from a programming standpoint, since you need not know the fixed core locations. The Loader will create the required linkage at load time. However, the cost of such simplicity is increased core requirements. During loading, each REF creates an additional Symbol Table entry. (See the Overlay Loader, Chapter 7 in the RBM Reference Manual and Chapter 5 in this manual.) Programs using this technique may require reloading when a new version of the Monitor is installed. An example of this type of access would be

```
            (Routine start)
            .
            .
            .
            REF         M:READ, M:WRITE, M:LOAD,. . .
            .
            .
            .
            RCPYI       P,X

            B           LABEL

            DATA        X'3801', 'X1', BUFF, 360, 2

LABEL       RCPYI       P,L

            B           M:WRITE
            .
            .
            .
```

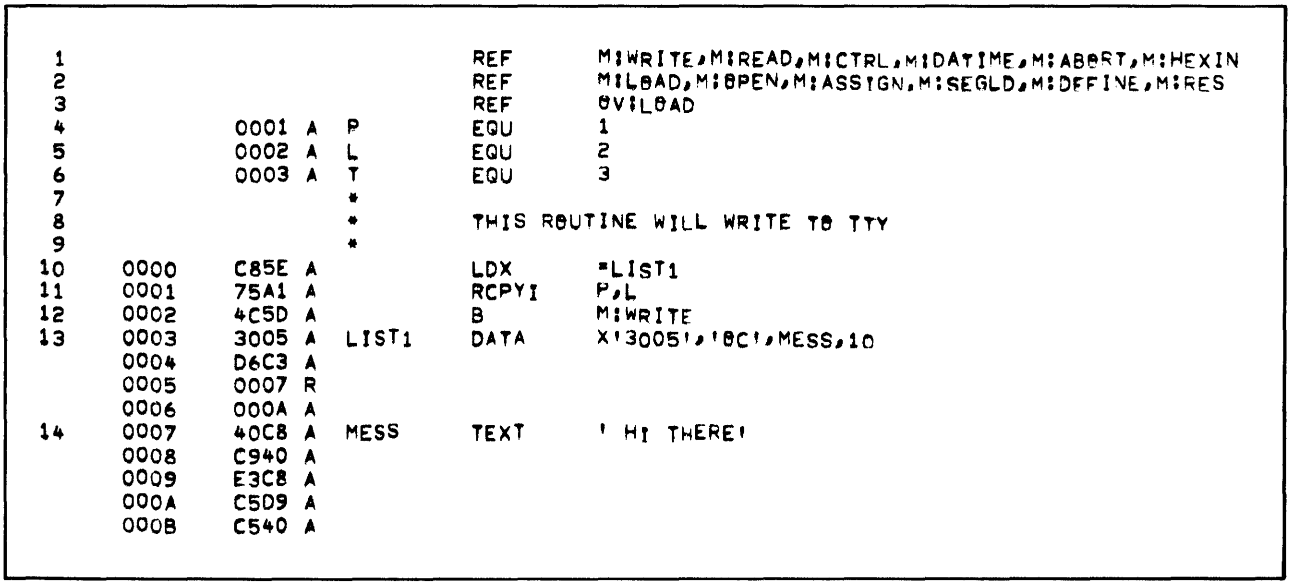
The above example also illustrates a slightly different method of entering the address of the M:WRITE argument list into the index (X) register.

All of the examples described assume the existance of a set of register equates in the program of the form:

```
P           EQU         1
L           EQU         2
            etc.
```

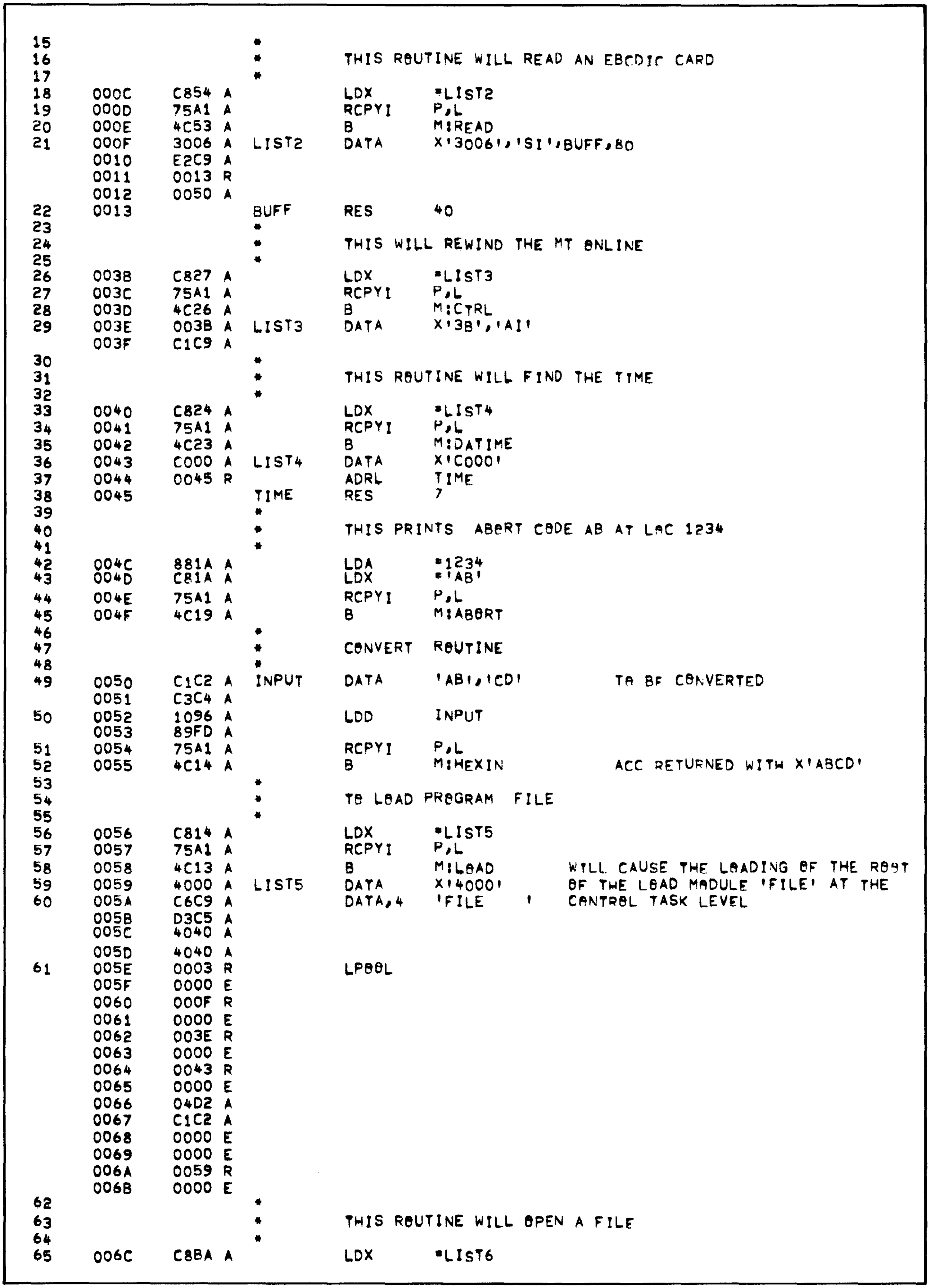
See Chapter 4 of the RBM Reference Manual for detailed descriptions and argument list requirements for each Monitor service routine.

Figure 32 shows assembled examples of some of the most frequently used service routines (see also Chapter 9 in this manual, "How To Use Standard Procedure Files"). Note that none of the examples are set up to be executed, but only illustrate how Monitor service routines are coded within a program.



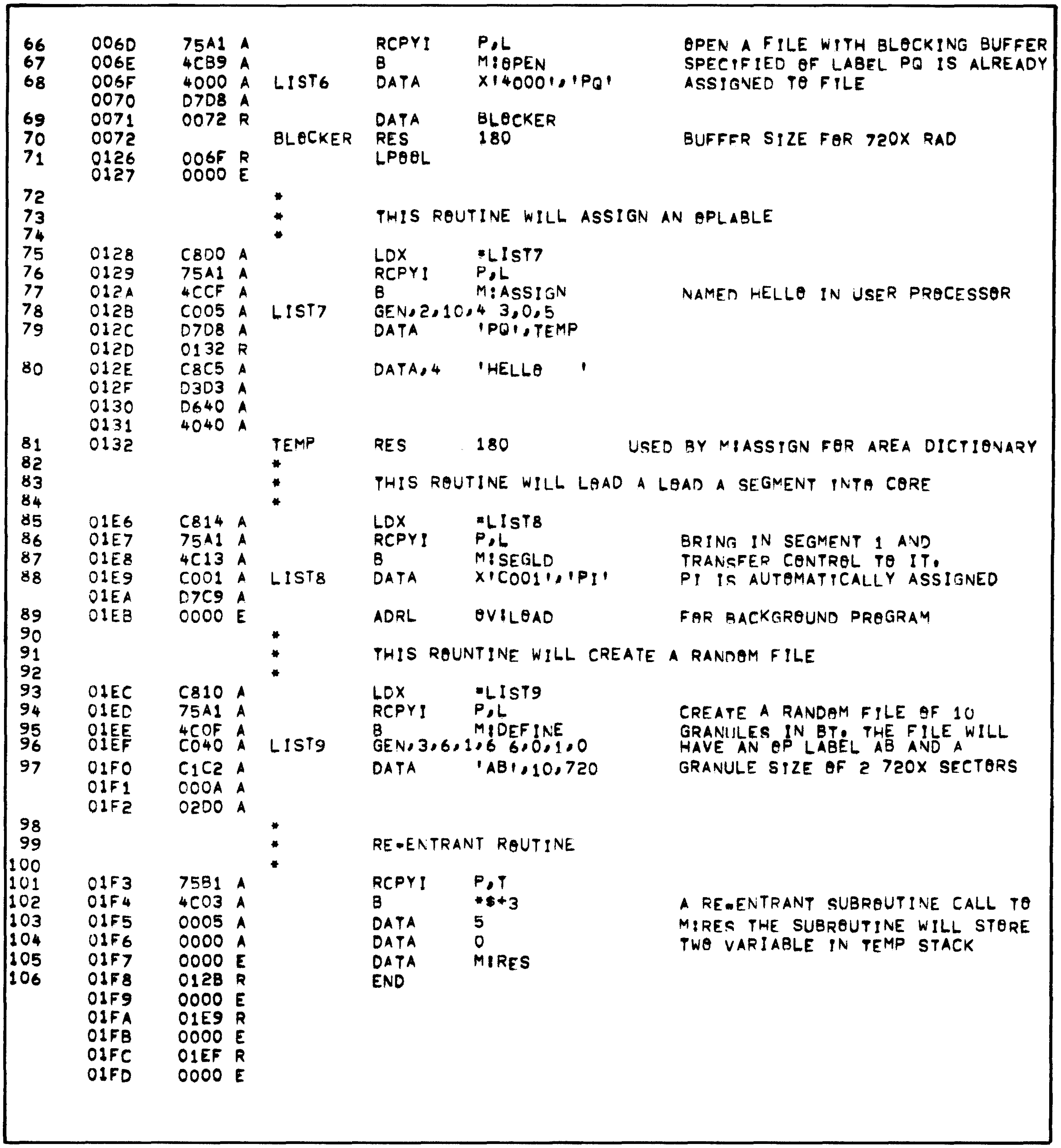
```
  1                              REF      M:WRITE,M:READ,M:CTRL,M:DATIME,M:ABØRT,M:HEXIN
  2                              REF      M:LØAD,M:ØPEN,M:ASSIGN,M:SEGLD,M:DEFINE,M:RES
  3                              REF      ØV:LØAD
  4             0001 A  P        EQU      1
  5             0002 A  L        EQU      2
  6             0003 A  T        EQU      3
  7                     *
  8                     *        THIS RØUTINE WILL WRITE TØ TTY
  9                     *
 10     0000    C85E A           LDX      =LIST1
 11     0001    75A1 A           RCPYI    P,L
 12     0002    4C5D A           B        M:WRITE
 13     0003    3005 A  LIST1    DATA     X'3005','ØC',MESS,10
        0004    D6C3 A
        0005    0007 R
        0006    000A A
 14     0007    40C8 A  MESS     TEXT     ' HI THERE'
        0008    C940 A
        0009    E3C8 A
        000A    C5D9 A
        000B    C540 A
```

Figure 32. Monitor Service Routine Examples



```
                     *
                     *        THIS ROUTINE WILL READ AN EBCDIC CARD
                     *
000C   C854 A                 LDX      =LIST2
000D   75A1 A                 RCPYI    P,L
000E   4C53 A                 B        M:READ
000F   3006 A   LIST2         DATA     X'3006','SI',BUFF,80
0010   E2C9 A
0011   0013 R
0012   0050 A
0013            BUFF          RES      40
                     *
                     *        THIS WILL REWIND THE MT ONLINE
                     *
003B   C827 A                 LDX      =LIST3
003C   75A1 A                 RCPYI    P,L
003D   4C26 A                 B        M:CTRL
003E   003B A   LIST3         DATA     X'3B','AI'
003F   C1C9 A
                     *
                     *        THIS ROUTINE WILL FIND THE TIME
                     *
0040   C824 A                 LDX      =LIST4
0041   75A1 A                 RCPYI    P,L
0042   4C23 A                 B        M:DATIME
0043   C000 A   LIST4         DATA     X'C000'
0044   0045 R                 ADRL     TIME
0045            TIME          RES      7
                     *
                     *        THIS PRINTS  ABORT CODE AB AT LOC 1234
                     *
004C   881A A                 LDA      =1234
004D   C81A A                 LDX      ='AB'
004E   75A1 A                 RCPYI    P,L
004F   4C19 A                 B        M:ABORT
                     *
                     *        CONVERT  ROUTINE
                     *
0050   C1C2 A   INPUT         DATA     'AB','CD'          TO BE CONVERTED
0051   C3C4 A
0052   1096 A                 LDD      INPUT
0053   89FD A
0054   75A1 A                 RCPYI    P,L
0055   4C14 A                 B        M:HEXIN            ACC RETURNED WITH X'ABCD'
                     *
                     *        TO LOAD PROGRAM  FILE
                     *
0056   C814 A                 LDX      =LIST5
0057   75A1 A                 RCPYI    P,L
0058   4C13 A                 B        M:LOAD       WILL CAUSE THE LOADING OF THE ROOT
0059   4000 A   LIST5         DATA     X'4000'      OF THE LOAD MODULE 'FILE' AT THE
005A   C6C9 A                 DATA,4   'FILE    '   CONTROL TASK LEVEL
005B   D3C5 A
005C   4040 A
005D   4040 A
005E   0003 R                 LPOOL
005F   0000 E
0060   000F R
0061   0000 E
0062   003E R
0063   0000 E
0064   0043 R
0065   0000 E
0066   04D2 A
0067   C1C2 A
0068   0000 E
0069   0000 E
006A   0059 R
006B   0000 E
                     *
                     *        THIS ROUTINE WILL OPEN A FILE
                     *
006C   C8BA A                 LDX      =LIST6
```

Figure 32. Monitor Service Routine Examples (cont.)



```
 66     006D     75A1 A             RCPYI      P,L                 ØPEN A FILE WITH BLØCKING BUFFER
 67     006E     4CB9 A             B          M:ØPEN              SPECIFIED ØF LABEL PQ IS ALREADY
 68     006F     4000 A  LIST6      DATA       X'4000','PQ'        ASSIGNED TØ FILE
        0070     D7D8 A
 69     0071     0072 R             DATA       BLØCKER
 70     0072             BLØCKER    RES        180                 BUFFER SIZE FØR 720X RAD
 71     0126     006F R             LPØØL
        0127     0000 E
 72                      *
 73                      *          THIS RØUTINE WILL ASSIGN AN ØPLABLE
 74                      *
 75     0128     C800 A             LDX        *LIST7
 76     0129     75A1 A             RCPYI      P,L
 77     012A     4CCF A             B          M:ASSIGN            NAMED HELLØ IN USER PRØCESSØR
 78     012B     C005 A  LIST7      GEN,2,10,4 3,0,5
 79     012C     D7D8 A             DATA       'PQ',TEMP
        012D     0132 R
 80     012E     C8C5 A             DATA,4     'HELLØ   '
        012F     D3D3 A
        0130     D640 A
        0131     4040 A
 81     0132             TEMP       RES        180           USED BY M:ASSIGN FØR AREA DICTIØNARY
 82                      *
 83                      *          THIS RØUTINE WILL LØAD A LØAD A SEGMENT INTØ CØRE
 84                      *
 85     01E6     C814 A             LDX        *LIST8
 86     01E7     75A1 A             RCPYI      P,L                BRING IN SEGMENT 1 AND
 87     01E8     4C13 A             B          M:SEGLD            TRANSFER CØNTRØL TØ IT.
 88     01E9     C001 A  LIST8      DATA       X'C001','PI'       PI IS AUTØMATICALLY ASSIGNED
        01EA     D7C9 A
 89     01EB     0000 E             ADRL       ØV:LØAD            FØR BACKGRØUND PRØGRAM
 90                      *
 91                      *          THIS RØUNTINE WILL CREATE A RANDØM FILE
 92                      *
 93     01EC     C810 A             LDX        *LIST9
 94     01ED     75A1 A             RCPYI      P,L                CREATE A RANDØM FILE ØF 10
 95     01EE     4C0F A             B          M:DEFINE           GRANULES IN BT. THE FILE WILL
 96     01EF     C040 A  LIST9      GEN,3,6,1,6 6,0,1,0           HAVE AN ØP LABEL AB AND A
 97     01F0     C1C2 A             DATA       'AB',10,720        GRANULE SIZE ØF 2 720X SECTØRS
        01F1     000A A
        01F2     0200 A
 98                      *
 99                      *          RE-ENTRANT RØUTINE
100                      *
101     01F3     75B1 A             RCPYI      P,T
102     01F4     4C03 A             B          *$+3               A RE-ENTRANT SUBRØUTINE CALL TØ
103     01F5     0005 A             DATA       5                  M:RES THE SUBRØUTINE WILL STØRE
104     01F6     0000 A             DATA       0                  TWØ VARIABLE IN TEMP STACK
105     01F7     0000 E             DATA       M:RES
106     01F8     012B R             END
        01F9     0000 E
        01FA     01E9 R
        01FB     0000 E
        01FC     01EF R
        01FD     0000 E
```

Figure 32. Monitor Service Routine Examples (cont.)

7. HOW TO USE UTILITY

The Utility Processor is a highly flexible collection of media operation routines composed of six primary functions: Copy, Dump, three Editors and a Control Function. Each function operates independently of the others but in conjunction with a root called the Utility Executive. The Executive scans the !UTILITY control command for parameters and if none are found, it is assumed that only the control functions are desired by the user.

The Control Function Processor consists of file and record positioning commands, pause and message commands, an assignment command, a write file mark command, and a prestore command.

When a specific routine is requested, the Executive checks to see that it is available, reads it into core, and initializes the required tables and flags. The Executive is used to process all utility control commands, which are always read using the SI operational label.

The "prestore" function requires an additional explanation. Prestore is a condition wherein the control commands, or sometimes data, is "prestored" in a temporary file (X5) for later processing. The prestore function may be invoked by control command (!*PRESTORE) or, under certain conditions, by the requested routine. Termination of the prestore function occurs when a !EOD, !*END, or file mark is encountered. The conditions under which prestore is invoked will be discussed more fully in the description of the Utility routines.

All of the Utility routines share the use of certain operational labels: SI, UI, UO, LO, DO, OC, and BI. Several functions will optionally accept user-supplied operational labels, and these cases will be described in the individual descriptions.

Before going on to discussions and examples of the various functions, three points should be mentioned about the material covered in this chapter:

- When using any of the functions for the first time, you should supplement the material in this chapter with study of the appropriate subchapter in the Utility section of the RBM Reference Manual (Chapter 9).

- All Utility examples in this chapter show !ASSIGN cards with device mnemonic operational labels instead of device file numbers in the device assignments. This is a SYSGEN-implemented capability that is recommended where a number of intermittent users are on the system, and particularly if a large number of Utility jobs are being run. See Chapter 19, "How To Assign and Use Device Operational Labels" in this manual for more details.

- In these examples, !EOD cards may be replaced by !*END cards.

HOW TO COPY AND VERIFY

The Copy routine is called by a !UTILITY COPY command to copy information from one device to another device and, optionally, to verify (compare) the copies. Note that RAD or disk pack files are also considered "devices".

Copy requires a !*OPLBS command to define the output device(s). This command must precede any file positioning or processing commands, but must follow a !*PRESTORE command (if present), and may follow !*ASSIGN commands. Data to be copied is always input from the UI device. The UI assignment must be specified on an !ASSIGN or !*ASSIGN command, if different from the SYSGEN assignment. The copies are output on the device(s) specified on the !*OPLBS command. A command string may contain more than one !*OPLBS command. In the verify function, Copy uses oplabel X4 as one input, and the oplabel specified on the last prior !*OPLBS command as the other input; or, data from X4 is compared to an in-core buffer if the CORE parameter was used on the !UTILITY COPY command. If UI and SI are assigned to the same device (or device file number, if the device is a RAD), SI is prestored. The number of records and files copied or verified is output to the DO device upon completion.

HOW TO COPY CARD READER INPUT TO LINE PRINTER

The example illustrated in Figure 33 will read one file from a card reader and list the contents of the file on the line printer.

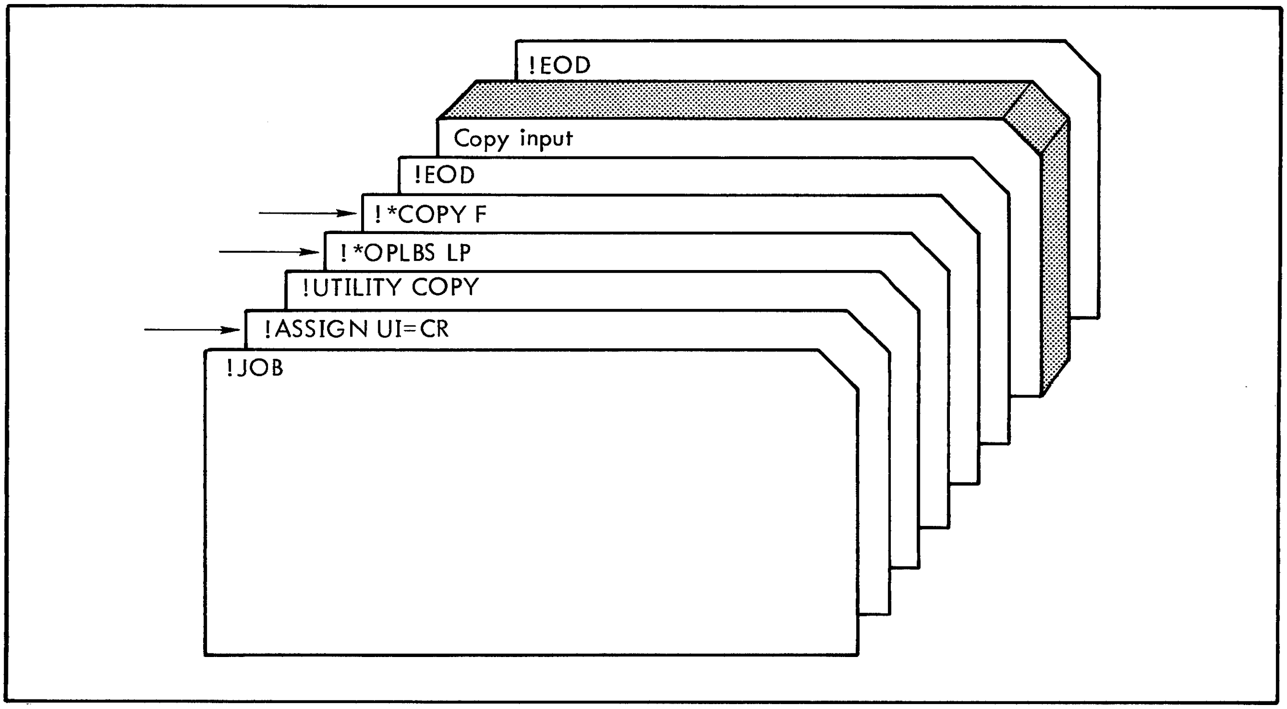


Figure 33. Copy Card Input to Line Printer

The !ASSIGN card assigns UI to the permanent operational label CR. Note that CR is a SYSGEN-defined label assigned to the file number (DFN) associated with the card reader. If device mnemonic operational labels were not implemented at the local facility, UI would have to be assigned to the DFN (e.g., 2) or another less-easily remembered operational label assignment to the card reader DFN.

The !UTILITY COPY card calls in the Utility Executive and the Executive calls in the Copy routine. Since the CORE option is not specified, a copy of the input will not be stored in core memory.

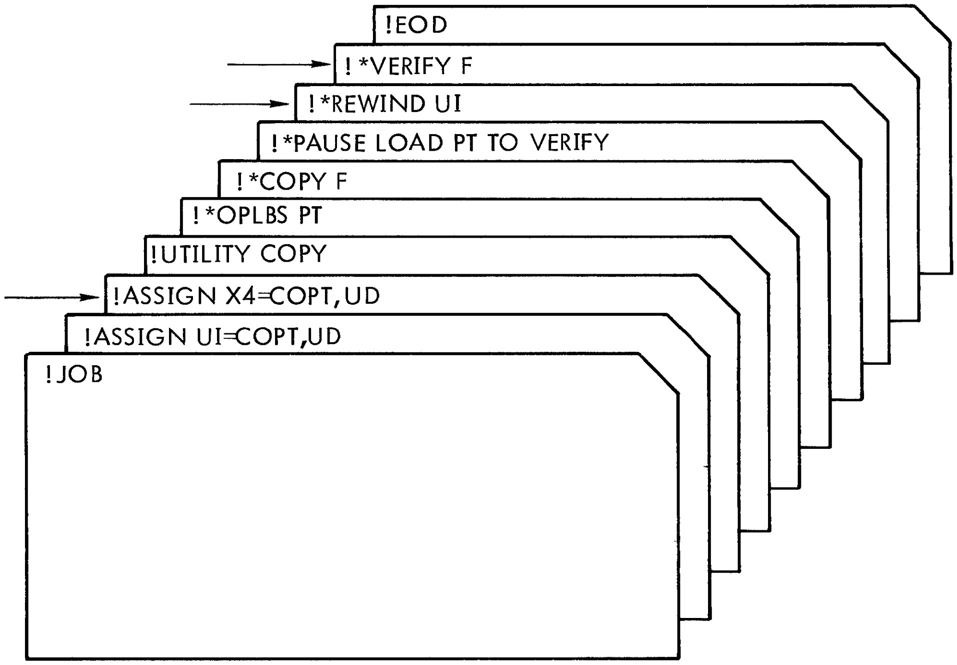
The !*OPLBS card specifies that the output device for the copy is to be the line printer (LP). At this point, the Copy routine determines if prestore should be invoked. Since the control commands (SI) and the Utility inputs (UI) oplabels are assigned to the same device (card reader), prestore will be used. The control commands, through the first !EOD card, will be written on temporary RAD file X5. The control commands are now read and executed as though oplabel SI had been assigned to X5.

The !*COPY card causes the records from the card reader to be written to the line printer until one complete file is copied (specified by the F parameter). If "R" was specified instead of "F", one record would be copied. The second !EOD card indicates end of file for the card reader inputs. The !EOD following the !*COPY card not only terminates prestore, but will cause Utility to exit when read from X5.

If we had wanted to copy from paper tape (for instance) to the line printer, the !ASSIGN card would be changed to read UI=PT, and the input would be followed by an !EOD (NL). Since SI and UI would be assigned to different devices, no prestore would be used.

HOW TO COPY AND VERIFY FROM A RAD FILE TO PAPER TAPE

In the example in Figure 34, the input to be copied on paper tape is located in a user-defined permanent RAD file called COPT in the UD (User Data) area.



```
!EOD
! *VERIFY F
! *REWIND UI
! *PAUSE LOAD PT TO VERIFY
! *COPY F
! *OPLBS PT
!UTILITY COPY
!ASSIGN X4=COPT,UD
!ASSIGN UI=COPT,UD
!JOB
```

Figure 34. Copy and Verify File from RAD Area to Paper Tape

The first !ASSIGN command defines the input "device" from which the data is to be copied as file COPT in the UD area of the RAD. The UI device must always be defined when using the COPY routine. This may be done either at SYSGEN or at run-time.

The second !ASSIGN card defines X4 to also be file COPT in the UD area. This card also causes adjustment of the byte count (from the COPT input) to 80 or 120 bytes before being copied to paper tape. The byte count depends on the contents of the first byte in the data to be copied. Note that non-standard binary can be copied to or from paper tape by specifying the byte count in the BIN mode on either the !*COPY or the !*VERIFY commands. See the COPY run-time discussion in Utility Chapter of the RBM Reference Manual.

The !*OPLBS card specifies that the output device for the copy is to be paper tape (SYSGEN-defined device mnemonic operational label PT).

The !*COPY card specifies that a file (F) is to be copied rather than a record (R), and by default, specifies that a single file is to be read.

The !*PAUSE card causes the system to go into a wait state to give the operator time to rewind and load the paper tape output for verification.

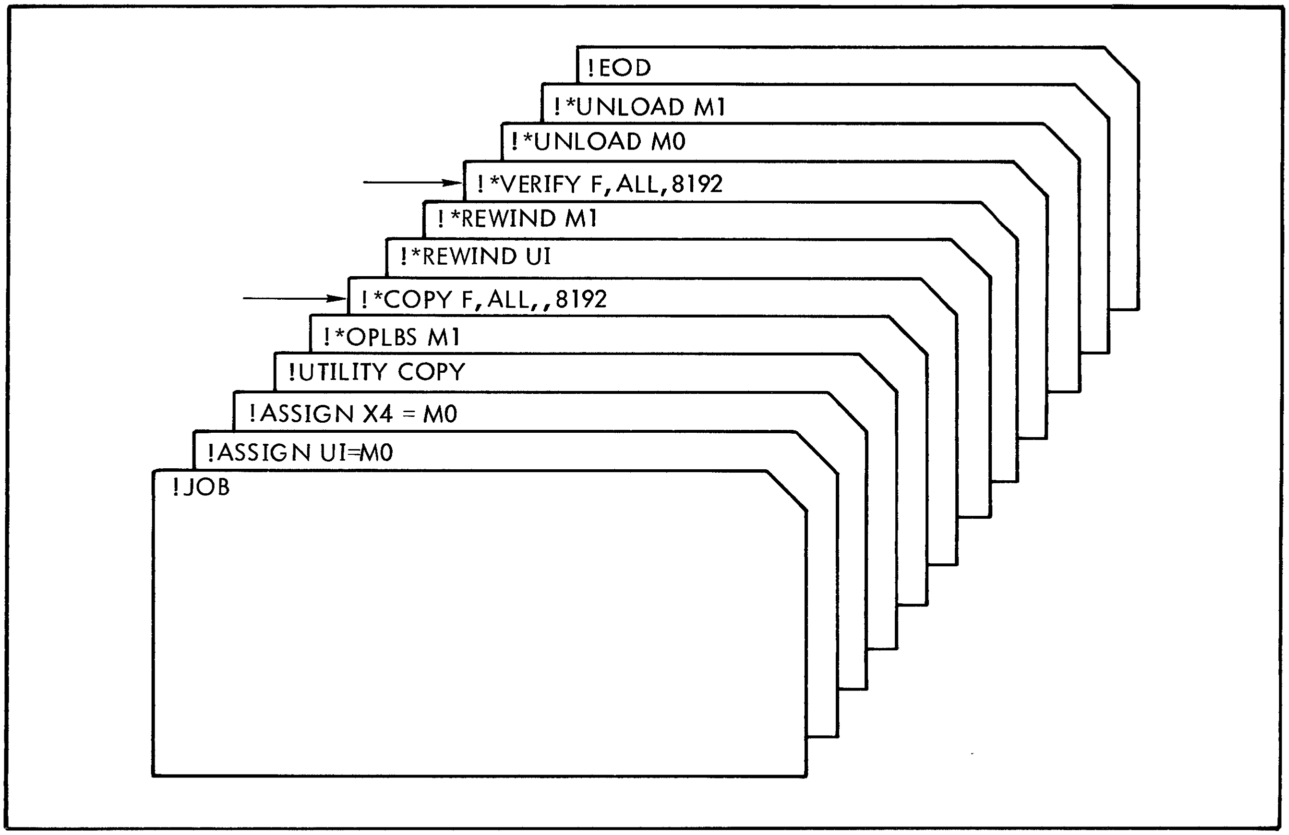
The !*REWIND UI card causes the RAD to "rewind" to the beginning of the file prior to the verify process. Since UI and X4 are assigned to the same device, the card could alternatively specify !*REWIND X4.

The !*VERIFY card causes the two files to be verified (compared) to the end of one file (F).

The !EOD card causes control to be returned to the Monitor.

HOW TO COPY MAGNETIC TAPE TO MAGNETIC TAPE

The example in Figure 35 will copy and verify all files from one magnetic tape to another until a double end-of-file is encountered.



```
!EOD
!*UNLOAD M1
!*UNLOAD M0
!*VERIFY F,ALL,8192
!*REWIND M1
!*REWIND UI
!*COPY F,ALL,,8192
!*OPLBS M1
!UTILITY COPY
!ASSIGN X4 = M0
!ASSIGN UI=M0
!JOB
```

Figure 35. Copy and Verify Magnetic Tape to Magnetic Tape

The first two !ASSIGN cards assign oplbs UI and X4 to the same magnetic tape input device (M0).

The !*OPLBS card defines M1 as the device that receives the copy output.

The !*COPY card specifies that all files (ALL) are to be copied until a double EOF is encountered, and the double comma specifies that the FORM parameter is not being used since no output is to go to the line printer or keyboard/ printer. Since we will, in this example, assume that the maximum record size is not known, the maximum permissable record size of 8192 bytes is specified. If any record read exceeded this maximum, a "CALLING SEQUENCE ERROR" message would be output and Utility would abort with a UT abort code.

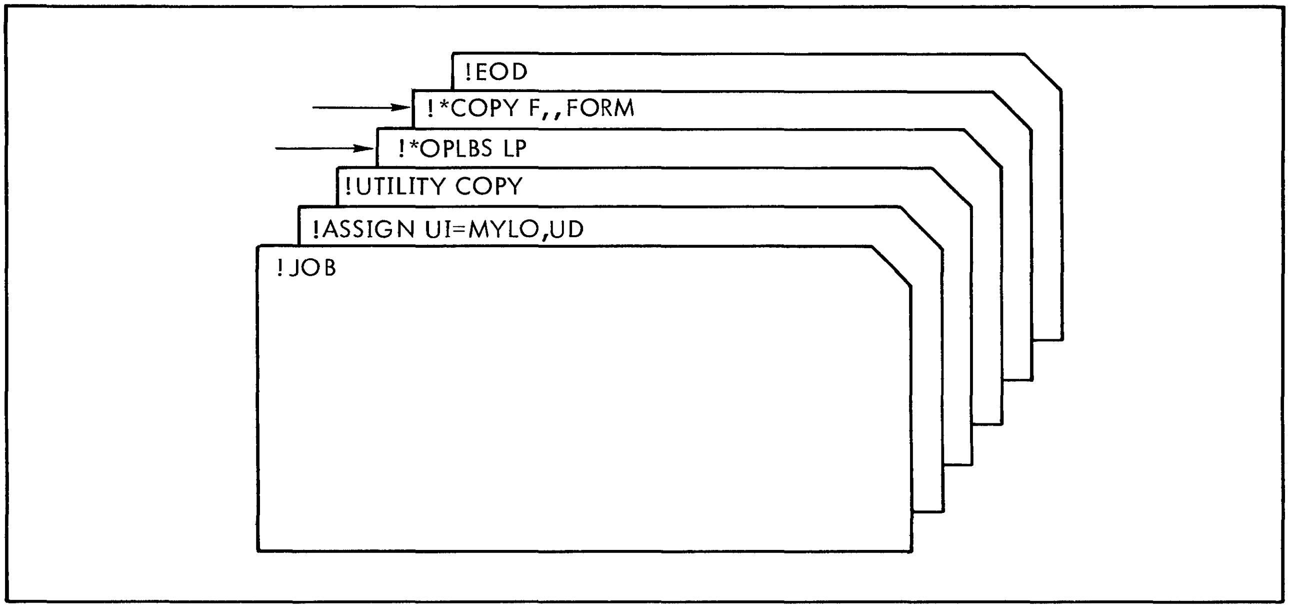
The !*REWIND UI (or !*REWIND X4) and !*REWIND UO will rewind the M0 and M1 magnetic tapes in preparation for verification.

The !*VERIFY card causes the M0 and M1 magnetic tapes to be read and compared.

The two !*UNLOAD cards cause the tapes to be rewound and placed in manual mode.

HOW TO COPY A FILE TO LINE PRINTER

An example of how to copy the listing output from either an assembly or compilation to the line-printer is given in Figure 36. The input (UI) could be read in from a magnetic tape or compressed RAD file.



```
!EOD
!*COPY F,,FORM
!*OPLBS LP
!UTILITY COPY
!ASSIGN UI=MYLO,UD
!JOB
```

Figure 36. Copy LO File to Line Printer

In this example, the !ASSIGN card assigns UI to a compressed RAD file called MYLO located in the User Data area.

The !*OPLBS card defines the line printer (LP) as the device to receive the copy, and the !*COPY card specifies by default (double comma) that one file is to be copied. The FORM parameter specifies that the first byte of each record is used for line printer or keyboard/printer format control and is not to be printed.

HOW TO PRESTORE CONTROL COMMANDS

The prestore mode separates control command functions from data when both are read in from the same input device, and then delays execution of the control commands until an !EOD is encountered. It does this by prestoring all Utility commands on a temporary RAD file (X5) up to an !EOD, and they are then executed in sequence to process the input data.

When the Copy routine is being used, the Utility Executive will automatically prestore commands when SI and one or more other oplbs are assigned to the same input device[†] but you can "force" prestore through the use of a Utility !*PRESTORE card.

In the example shown in Figure 37, a card deck is verified against a paper tape, with the data deck and control commands being read in from the same device. We will use the UO operational label as an input source to compare against X4.

The first two !ASSIGN cards assign SI (for PRESTORE input) and UO (for VERIFY output) to the card reader (CR device operational label), and the third !ASSIGN card assigns the X4 device to paper tape.

The !*PRESTORE command (which must always follow the !UTILITY card) causes all Utility control commands to be loaded into the X5 RAD file and delays their execution until the !EOD card following the !VERIFY card is read.

[†] Prestore takes place under different conditions in the Object Module Editor. See the OMEDIT subsection in Chapter 9 of the RBM Reference Manual.

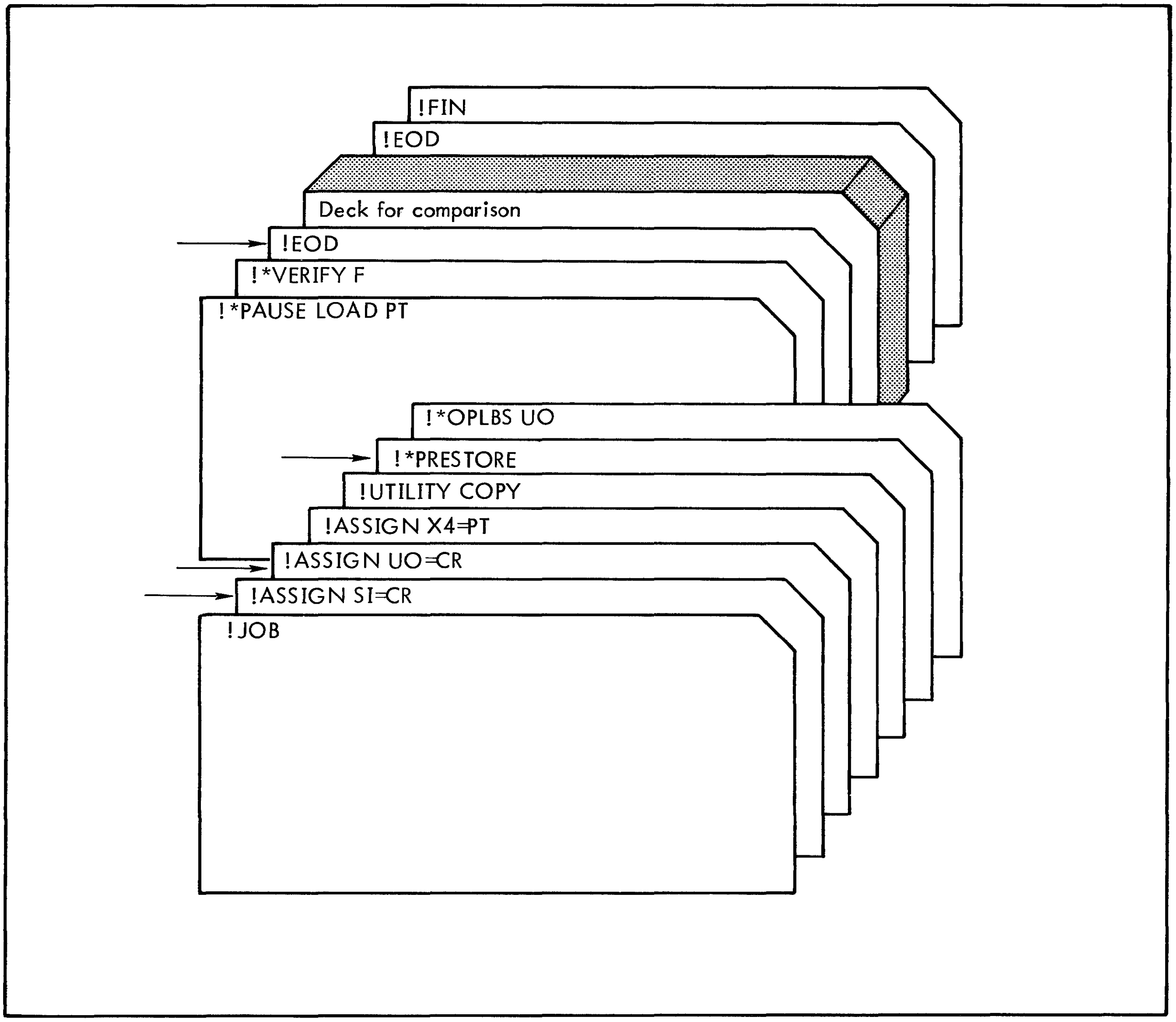


Figure 37. Verify Card Deck and Paper Tape with Forced Prestore

The !*OPLBS card defines UO as the output (in this case input) device, which is the card reader since UO was previously assigned to CR.

The !*VERIFY card causes card deck to be read from UO and compared with the data written on paper tape (X4).

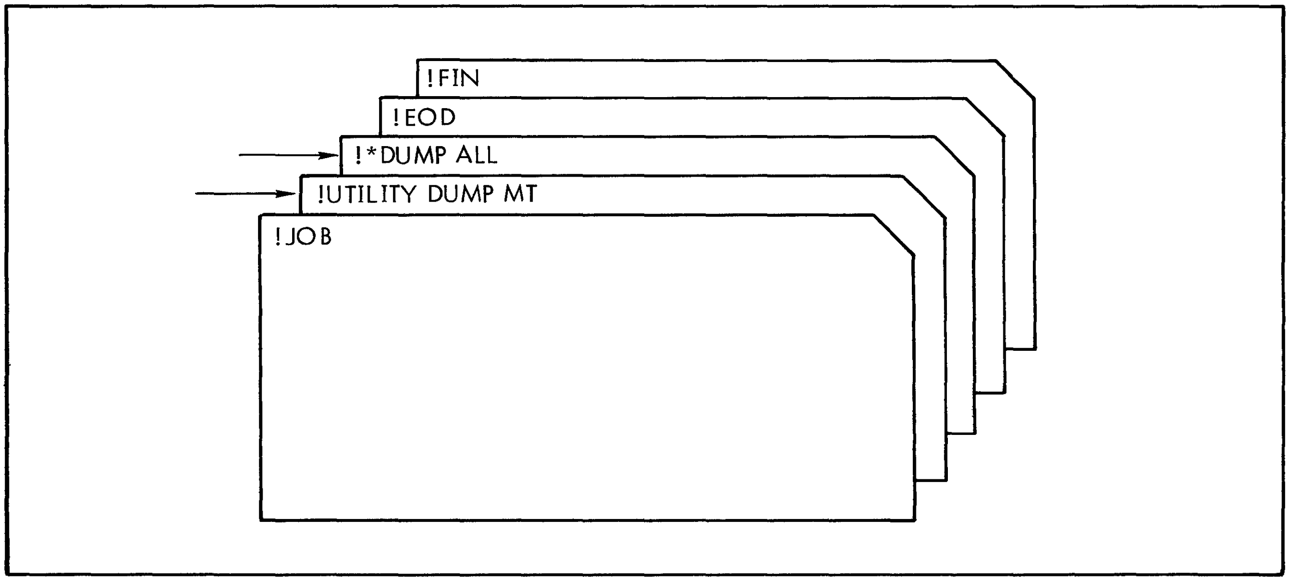
HOW TO USE UTILITY DUMP

The Dump routine will dump (print) records or files from one media onto any device you specify. You can also specify whether the dump is to be in EBCDIC or hexadecimal format. Dump is called by a !UTILITY DUMPcommand. If an operational label is specified with this command, input is from the device assigned to that label; otherwise, input is from the UI device. Output is always to the LO device. The number of records may be specified by the first parameter on a !*DUMP command. If this parameter is blank, records will be dumped to a file mark. If the parameter is ALL, records will be dumped to a double EOF. Maximum record size may also be specified. Input data may be binary or EBCDIC. If the HEX parameter is specified on the !*DUMP command, the output is in the hexadecimal equivalent of the input (assumed binary). If HEX is omitted, records with a binary indication in the first byte (see the DUMP description in Chapter 9, RBM Reference Manual) are output in hexadecimal; all others are assumed EBCDIC and are output as such.

If SI and the input device are assigned to the same device or RAD DFN, SI is prestored.

HOW TO DUMP A MAGNETIC TAPE

The deck example given in Figure 38 will dump a magnetic tape assigned to operational label MT onto whatever device is assigned to LO (normally a line printer).



```
!FIN
!EOD
!*DUMP ALL
!UTILITY DUMP MT
!JOB
```

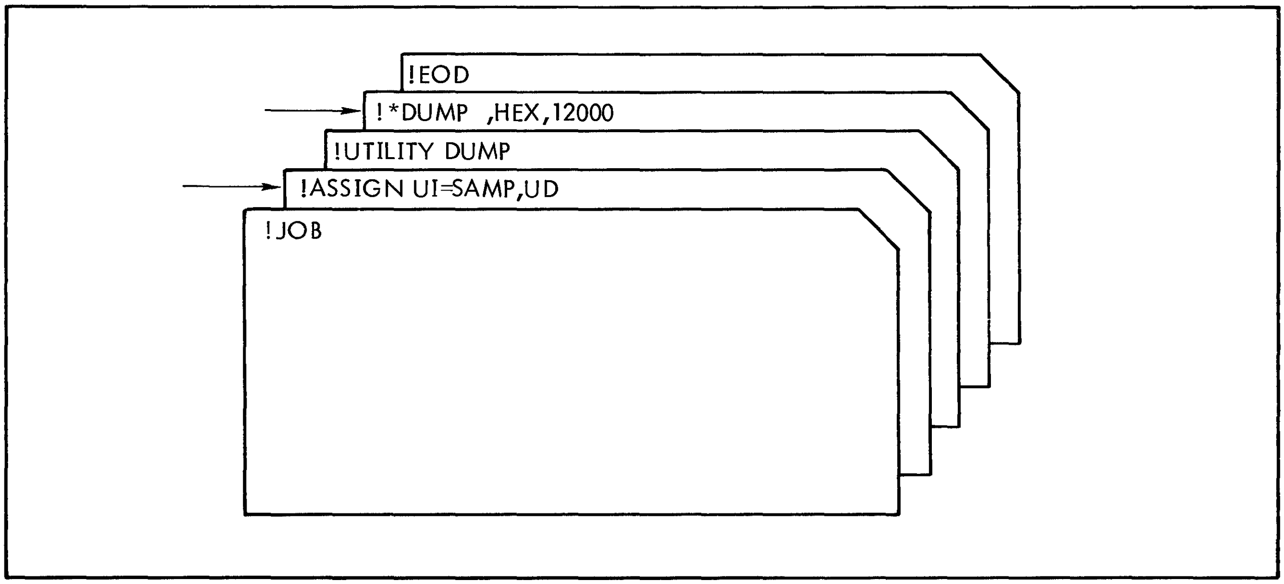
Figure 38. Dump a Magnetic Tape

The !UTILITY DUMP card calls in the Dump routine and specifies that input is to be read from MT. If an oplb was not specified, input would be read by default from the device to which UI is assigned.

The !*DUMP card specifies that all records on the tape are to be dumped until a double EOF is encountered (ALL). Since neither the "mode" or "size" are specified, the default options of EBCDIC format for the output and the standard record size (120 bytes) will be used.

HOW TO DUMP A RAD FILE

The example given in Figure 39 will dump a user-defined sequential access RAD file called SAMP from the User Data area of the RAD.



```
!EOD
!*DUMP ,HEX,12000
!UTILITY DUMP
!ASSIGN UI=SAMP,UD
!JOB
```

Figure 39. Dump Sequential Access RAD File

The !ASSIGN card assigns UI to file SAMP, in the User Data (UD) area of the RAD.

The !UTILITY dump card calls in the Dump routine and specifies the input device as UI by default, which for the length of this one job, is temporarily assigned to the SAMP file in the UD area.

The !*DUMP card causes the SAMP file to be dumped out to the LO operational label. The HEX parameter specifies the dump is to be in hexadecimal format and maximum size of the record to be dumped is 12000 bytes (which would probably be determined by looking at a RAD map). Note that since this is a sequential access file, the maximum record size specified must be an even number.

HOW TO USE THE OBJECT MODULE EDITOR

Like the other two Utility Editors, the Object Module Editor generates or maintains (updates) magentic tape, paper tape, RAD, or disk pack files. As its name implies, OMEDIT works with binary object modules that are output from assemblies or compilations.

OMEDIT is called by a !UTILITY OMEDIT command and itself has no specification parameters. OMEDIT operates in two modes: list and modify, and either a !*LIST or !*MODIFY card must follow the !UTILITY OMEDIT card.

In the list mode, object modules are input from the UI device, checksum and sequencing are checked, and the "ident" (the result of an IDNT directive in Extended Symbol or a subroutine name in FORTRAN) is printed on the LO device. Checksum and sequence errors are flagged on LO, and listing continues.

In the modify mode, two alternatives are available: GEN and INSERT. If the GEN parameter is used, it must be followed by a !*INSERT command. OMEDIT then copies binary records from BI to UO, performing checksum and sequence checks.

If the INSERT parameter is used, it may be followed by !*INSERT commands or by !*DELETE commands. Binary records are copied from UI to UO. A !*DELETE card causes the named modules to be omitted from the UO records. The !*INSERT command causes modules from the BI device to be written, in sequence, to the UO device. The first ident on the !*INSERT command specifies the BI module to be inserted; the second ident, if present, specifies the UI module it is to follow. If the second ident is absent, the BI module will be the next module written to UO.

Prestoring of control commands or binary data will occur under the following conditions:

Oplabels Assigned to the Same Device	Prestored
SI, BI	SI
SI, UI	SI
BI, UI	BI
SI, BI, UI	SI, BI

OMEDIT will not terminate and exit until two successive !EODs are encountered from UI or BI.

HOW TO LIST OBJECT MODULES FROM GO FILE

The example in Figure 40 will read the object module(s) located on the default GO file (RBMGO) from the System Data area of the RAD, and list the contents until a double !EOD is encountered.

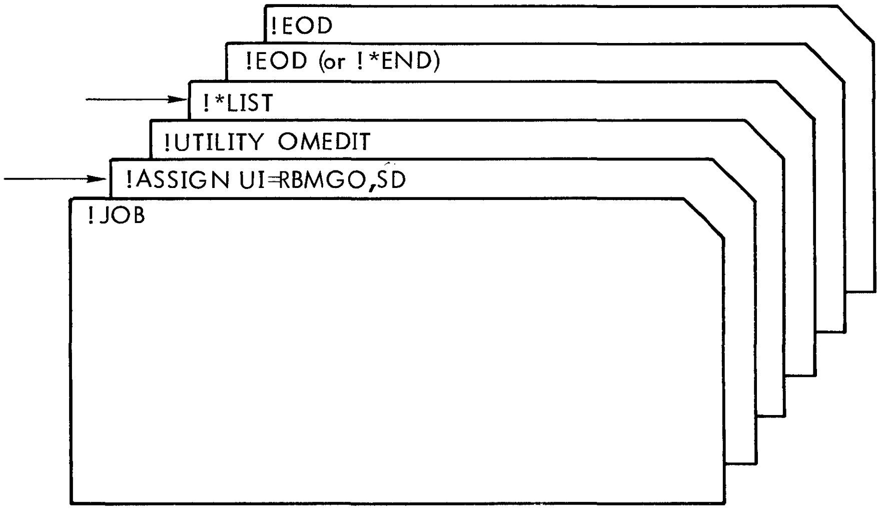


Figure 40. List Object Module from RAD File

The !ASSIGN card assigns UI to the RBMGO file, and the !*LIST card will cause the contents to be listed on the LO device (normally the line printer).

HOW TO UPDATE OBJECT MODULES FROM CARDS

The example in Figure 41 will read in a set of update modules (BI) that modify the original binary object modules (UI). The updates, original modules, and control commands are all read in from the same device. The updated version of the program is to be written on magnetic tape. Since BI (updates) and UI (old modules) are assigned to the same device (SI), the complete BI file will be automatically prestored on a temporary RAD file before the update takes place. All inserts and !*INSERT commands must be in the proper sequence.

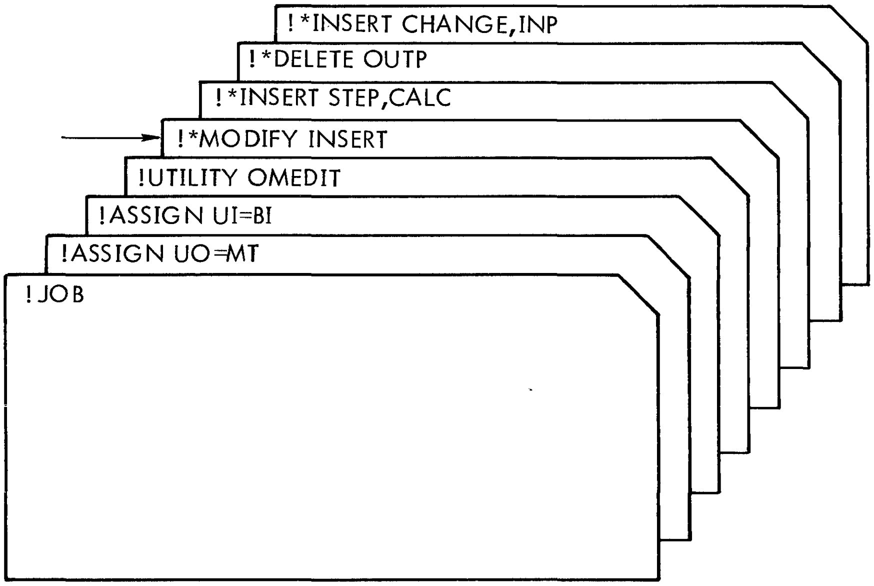


Figure 41. Update Object Modules from Card Reader to Magnetic Tape

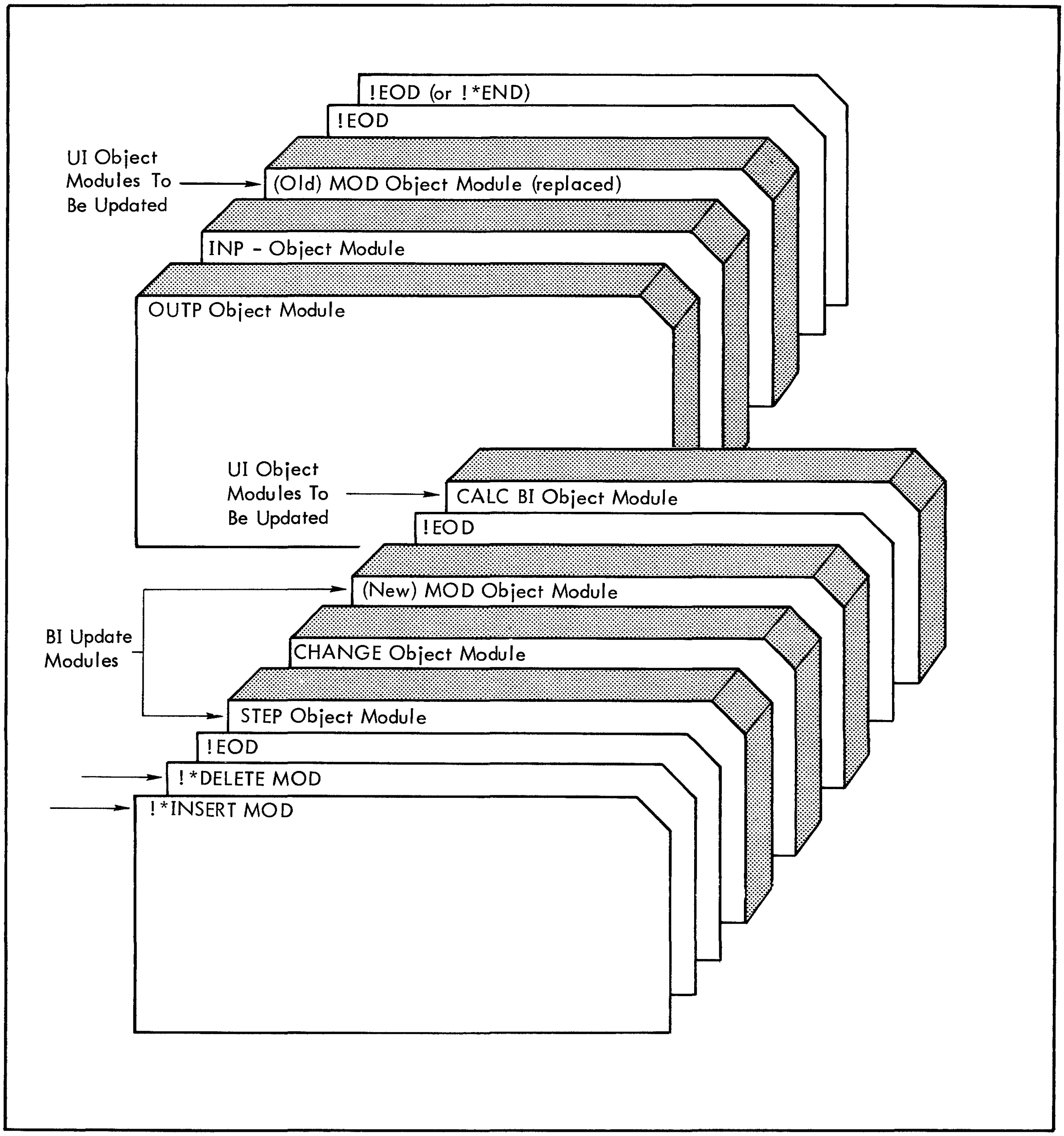


Figure 41. Update Object Modules from Card Reader to Magnetic Tape (cont.)

The deck structure in Figure 41 will perform the following functions:

1. Insert module STEP after module CALC.
2. Delete module OUTP.
3. Insert module CHANGE after module INP.
4. Insert "new" module MOD after module CHANGE.
5. Delete "old" module MOD.
6. Write updated version on magnetic tape assigned to UO.

The card images of the resulting updated version of the program written on magnetic tape would appear as shown in Figure 42, as compared to the UI version shown in Figure 41.

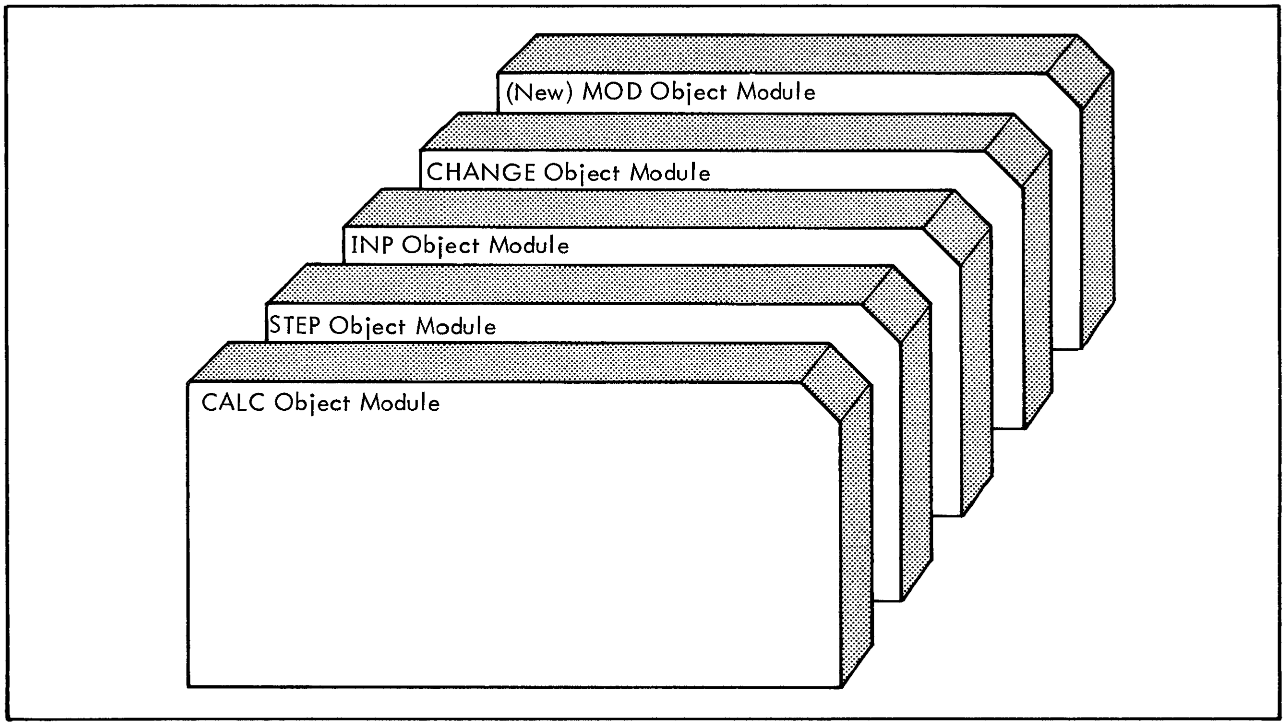


Figure 42. OMEDIT Update Example

HOW TO USE THE RECORD EDITOR

The Record Editor edits FORTRAN or assembly language source input by record number in the following manner:

- Generates source data files.
- Lists source data files with associated line numbers.
- Modifies source data files.

A !*LIST command places RECEDIT in the list mode. Source files are read from UI and listed on LO, with associated line numbers starting with "1". An !EOD or file mark will cause line numbering to restart with 1. An optional "number" parameter on the !*LIST command indicates the number of files to be read; if "number" is omitted, one file will be read. If double !EODs are encountered, the list mode is terminated.

The !*MODIFY command initiates the modify mode (and will terminate the list mode). The GEN parameter on this command causes source images to be copied from SI to UO. If the LIST parameter is also present, UO and the associated line numbers will be listed on the LO device.

If the GEN parameter is absent, updating is implied. If LIST is present, the LO listing will contain the UI line numbers. In the update mode, UI is read, modified by control commands and source images from SI, and written to UO. Lines specified by number on a !*DELETE command are omitted from UO.

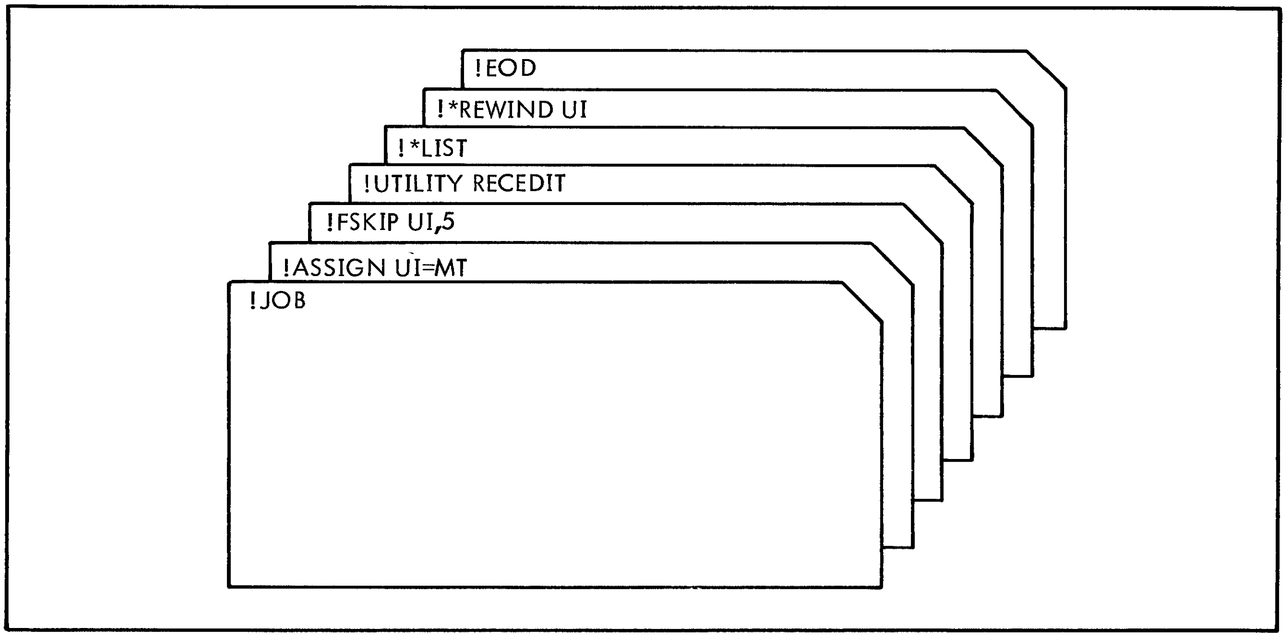
Source images following (on SI) an !*INSERT command are written on UO following the UI line number that is specified on the !*INSERT command. A source image following a !*CHANGE command replaces the UI source image whose number is specified on the !*CHANGE command. If more than one source image follows the !CHANGE card, those following the "changed" one are inserted before the next UI image is copied. Two numbers on the !*CHANGE command causes deletion of all UI images inclusively between the numbers. Source images on SI following either !*INSERT or !*CHANGE commands are inserted on UO until the next control command is encountered. When a !EOD is encountered from SI, the remainder of the UI file is copied to UO.

If SI and UI are assigned to the same device, SI is prestored.

It is worth mentioning that assembly language users have some record editing functions available to them through the assembler. See the specification options on the !XSYMBOL control command in the Extended Symbol/LN, OPS Reference Manual, 90 10 52, for record updating capability.

HOW TO LIST A SPECIFIED FILE FROM MAGNETIC TAPE

The example given in Figure 43 would list the sixth file from a magnetic tape.



```
!EOD
!*REWIND UI
!*LIST
!UTILITY RECEDIT
!FSKIP UI,5
!ASSIGN UI=MT
!JOB
```

Figure 43. List Specific File for Magnetic Tape

The !ASSIGN card assigns UI to the MT operational label for a magnetic tape, and the !FSKIP card causes UI to be skipped past the first five EOF marks.

The !*LIST card then causes all records in file six or the UI tape to be listed until the next EOF is encountered and UI is then rewound.

HOW TO MODIFY A SOURCE MODULE TO A RAD FILE

The example in Figure 44 will read source record updates, the original source deck, and all control cards from the card reader. The updated source module will then be written in a compressed, blocked RAD file called MYSORS (user-defined) in the User Data area. Prestore will be imposed since SI=UI.

The !*MODIFY card specifies that both the records deleted and the records inserted will be listed on LO, including UI line numbers deleted and the line number preceding the one inserted. Since updating is to be performed, the GEN parameter must not be present on the card.

The !*DELETE card specifies that lines 31 and 32 are to be deleted from the source records, and the !*INSERT card specifies that source records are to be inserted after line 49. In our example, four new records are inserted.

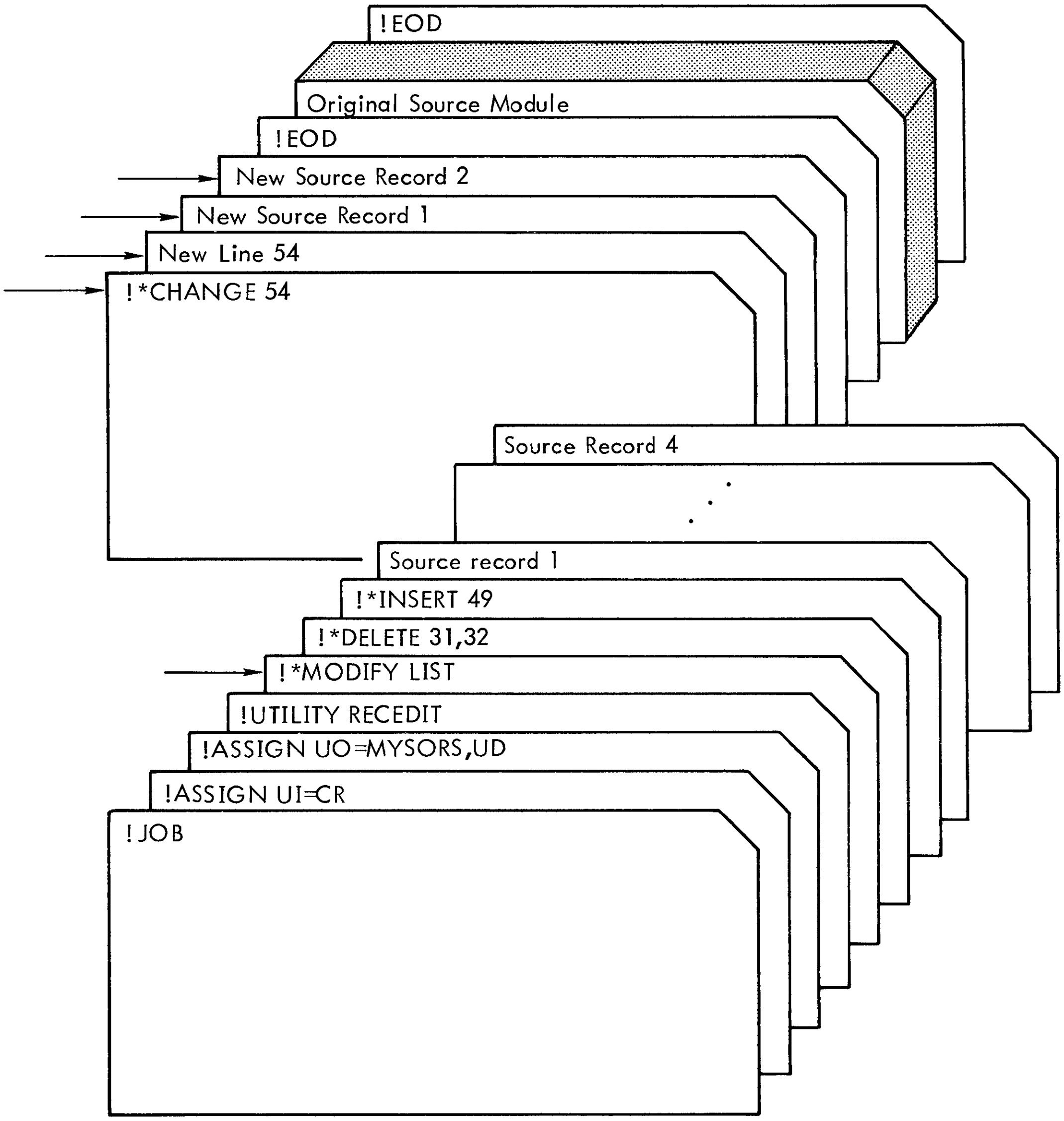


Figure 44. Update Source Records in Source Module

The !*CHANGE card specifies that line 54 is to be replaced and that new source records may be inserted after the new line 54. In this case, two new records are inserted.

HOW TO USE THE SEQUENCE EDITOR

SEQEDIT performs the same functions and operates similarly to RECEDIT; the principal difference being that SEQEDIT operates on sequence numbers in the sequence field of the source image (columns 73–80 of a source card), thus providing more flexibility than RECEDIT. Again, source is read from UI, modified by commands and source data from SI, and the update is written to UO. SEQEDIT is not recommended for paper tape use.

SEQEDIT is called by a !UTILITY SEQEDIT command. Three optional parameters are allowed on the call: GEN, IGN, and ALL (in that order). The GEN parameter specifies that the source is copied from SI to UO.

The absence of GEN implies UPDATE mode. The IGN parameter indicates that sequence errors are ignored on SI if in GEN mode, or UI if in UPDATE mode. The presence of the ALL parameter causes SEQEDIT to continue until double !EODs are encountered.

A special command !*IDENT is available to break the sequence field into an "ident" and a numerical section. This facilitates updating of multiple files or multi-program files.

Insertions and replacements are accomplished by the source images (on SI) themselves, rather than by specific commands. If an image on SI has the same sequence field as an image on UI, the SI image is written to UO instead of the image from UI. If an SI image has a sequence number between two UI images, the SI image will be inserted, on UO, between those two UI images. If SI contains a block of images with blank sequence fields that follows an image with a sequence number, UO will contain the numbered image (be it insertion or replacement), followed by the blank-sequence images.

Deletion of images from UI is accomplished by a !*DELETE or a !*SUPRESS command that contains the sequence number to be deleted. If two numbers are present, UI images will be deleted inclusively between the numbers. The difference in the two commands is that !*DELETE causes the deleted images to be listed on LO while !*SUPRESS does not.

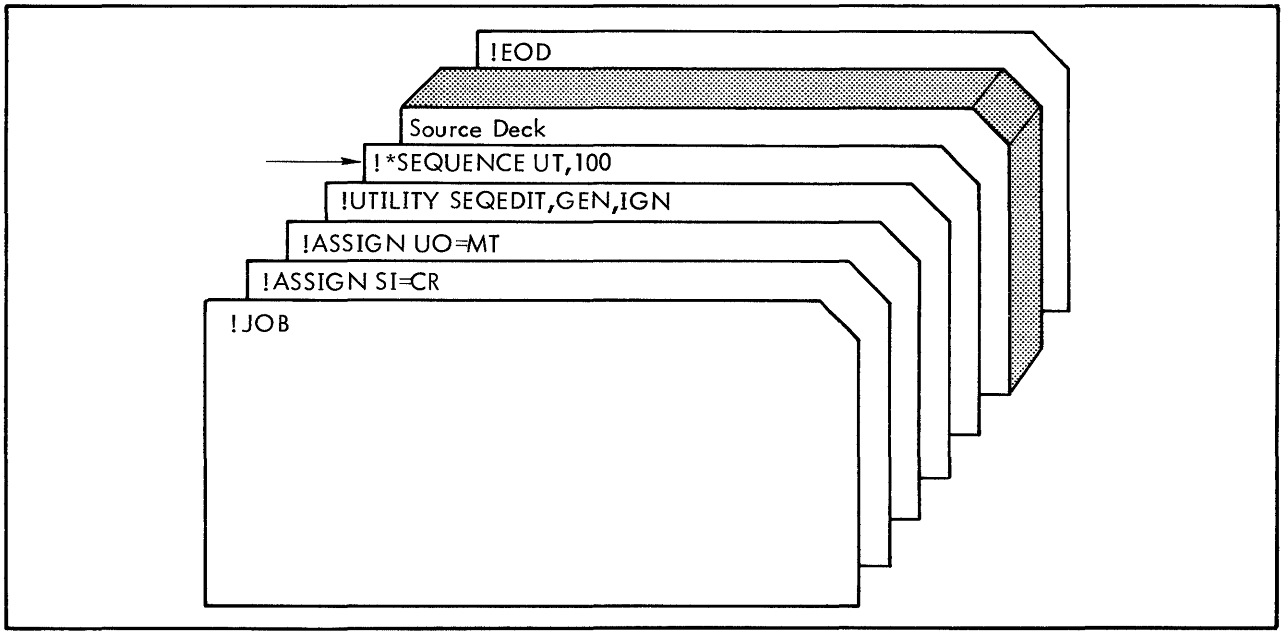
The !*SEQUENCE command may be used to sequence a file being generated, or to resequence files being updated. If multiple files are being updated, a new !*SEQUENCE command must be used for each file.

If UI and SI are assigned to the same device, SI is prestored.

HOW TO GENERATE AND SEQUENCE A FILE ON MAGNETIC TAPE

The job example in Figure 45 will generate and sequence a new file on magnetic tape.

The !UTILITY SEQEDIT card specifies that a single file is to be generated on UO, and the presence of the GEN parameter also informs the Sequence Editor that no updates are to take place. (Updating and generation cannot take place within the same call to the Sequence Editor.) IGN indicates that SI sequence errors are to be ignored.



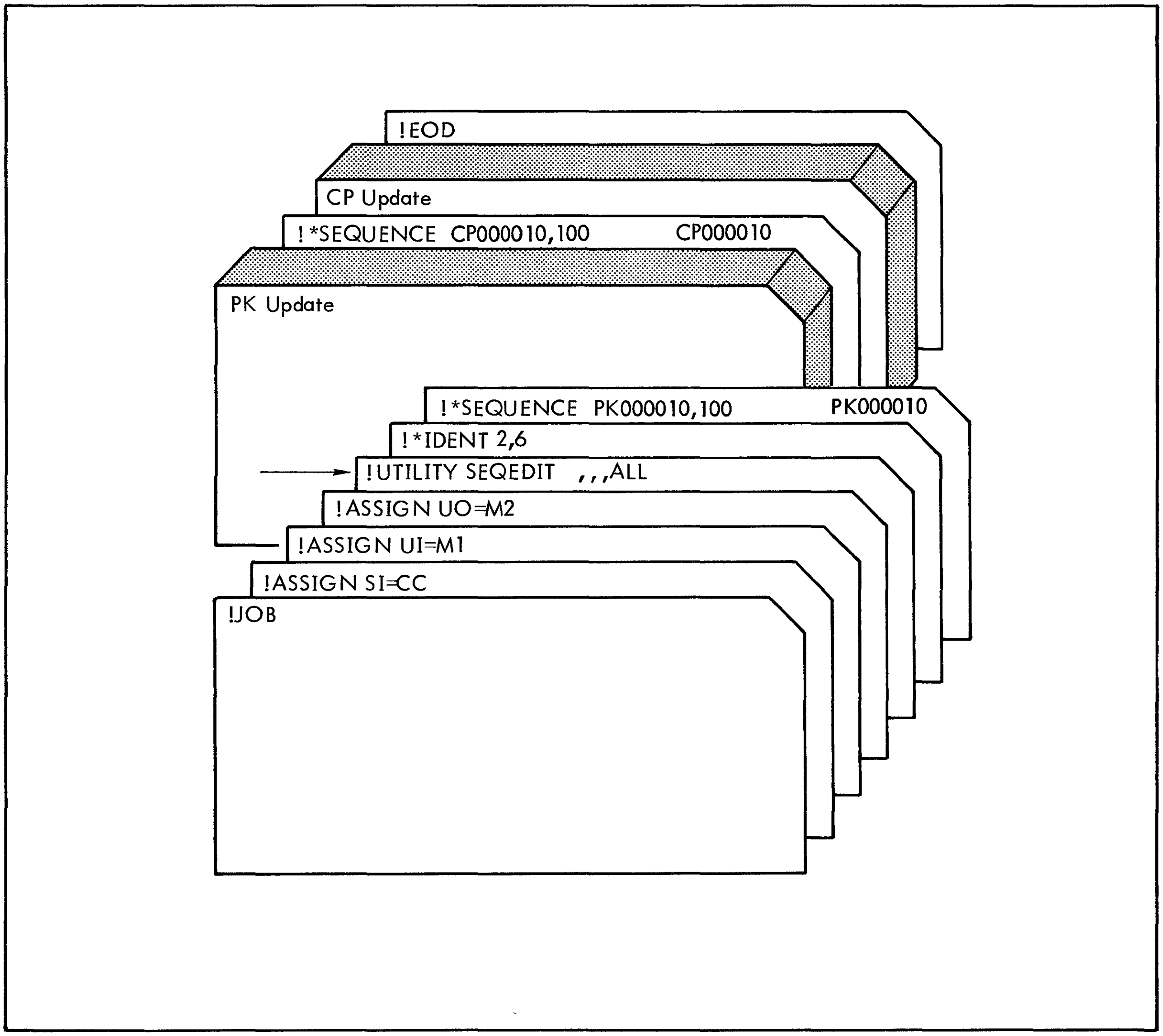
```
!EOD
Source Deck
!*SEQUENCE UT,100
!UTILITY SEQEDIT,GEN,IGN
!ASSIGN UO=MT
!ASSIGN SI=CR
!JOB
```

Figure 45. Generate and Sequence a File on Magnetic Tape

The !*SEQUENCE card specifies that UT is the ident field and that the sequence numbers are to be in increments of 100. This will permit a large number of later insertions without interfering with the original sequence numbers. Thus, the first sequence number would appear as UT000100, the second number as UT000200 and so on. The sequencing will take place as the file is being generated.

HOW TO UPDATE AND RESEQUENCE TWO FILES ON MAGNETIC TAPE

The example in Figure 46 will update and resequence two separate files on a magnetic tape and write the updated versions on a new magnetic tape.



```
!EOD
CP Update
! *SEQUENCE  CP000010,100          CP000010
PK Update
! *SEQUENCE  PK000010,100          PK000010
! *IDENT 2,6
!UTILITY SEQEDIT  ,,,ALL
!ASSIGN UO=M2
!ASSIGN UI=M1
!ASSIGN SI=CC
!JOB
```

Figure 46. Update and Resequence Two Magnetic Tape Files

The first !ASSIGN card assigns SI to CC, which defines the device from which the updates and control commands will be read. The next two !ASSIGN cards assign M1 as the device from which to read the two files to be updated, and M2 as the output device to write the updated and resequenced version.

The triple comma on the !UTILITY SEQEDIT card specifies that the GEN, IGN options are not being exercised, and the ALL option specifies that updating is to continue until two EOFs are encountered on the UI device (M1).

The !*IDENT card, used only when updating files, specifies that the number of characters in the ident portion of the sequence field is two characters, and the 6 specifies the number of characters in the sequence number subset of the sequence field. The card holds true for both updates.

The first !*SEQUENCE specifies that resequencing will begin when a card containing PK000010 in columns 73-80 is found, and that resequencing will be in increments of 100, beginning with PK000010 (if the increment number was missing, the increment would be 10 by default). The "PK000010" parameter (in columns 73-80) specifies the sequence number at which the resequencing is to commence to incorporate the update.

The PK update will be added to the original file beginning at sequence number PK000010, and all line numbers from that point will be in increments of 100 (e.g., PK000110, PK000210, etc).

The second !*SEQUENCE card causes identical functions to be performed with the CP update.

8. HOW TO INTERFACE ANS FORTRAN IV AND EXTENDED SYMBOL SUBROUTINES

This chapter defines the conventions and techniques required for interfacing Extended Symbol and ANS FORTRAN IV programs and subprograms with one another and with FORTRAN Library routines. The material attempts only to clarify those points directly related to interface problems, and more detailed coverage of the items discussed will be found in the appropriate language reference manuals listed in "Related Publications". It is assumed that you are already conversant with the use of the language processors, Loaders, and characteristics of the library routines that are available on your system.

GENERAL CONCEPTS AND CONVENTIONS

EXTERNAL REFERENCES

Separately assembled or compiled program sections or library routines may refer to symbolic locations within other sections via external references. In Extended Symbol, this is accomplished with the DEF and REF directives. A DEF "name" declares that the value of "name" is accessible to the outside world. A REF "name" implies that the value of "name" is to be obtained from another program section and is to replace all references to "name" within this section. In particular, if "name" is a DEF within a selected library, the REF will cause that library routine to be loaded as part of the program. Since, during loading, the value of the "name" is its location, this means that the location of the name will replace the references to "name".

In FORTRAN, a DEF is implied in a SUBROUTINE or FUNCTION subprogram, where "name" is the name of the subroutine or function. A REF is created either by the explicit "CALL name (x)", or by the use of a function name within an expression (e.g., A = name (x)). Additionally, if "name" appears in an EXTERNAL statement, all references to "name" will result in the creation of a REF.

TEMPORARY STACK

When operating in a priority-interrupt environment, it is essential that a routine that might be used concurrently by different tasks (i.e., interrupt levels) is coded reentrantly. To achieve reentrancy, the called routine must not store call-dependent data values within itself at fixed locations. The method adopted by Sigma 2/3 standard software is for the program calling a reentrant routine to provide a certain amount of scratch storage for any storage the reentrant routine may require. This is called a temp stack, and it is expected that the calling program will make available the start address of its own temp stack before calling the reentrant routine. To provide for this structure, ANS FORTRAN IV is careful to ensure that the temp stack does not attempt to contain any preset data. (See also Chapter 14 of this manual for a detailed explanation of temp stack and assembly language reentrancy.)

Any variables (scalar or array) that appear in a DATA statement in an ANS FORTRAN IV program or subprogram will be allocated within the body of the program along with the code. All other variables are allocated in the temp stack. This technique has no impact on nonreal-time programs. However, it does allow the real-time programmer to have "named constants" within his program. An example of an occasion where the named constant concept is valuable is where the routine is to be parameterized. If this "constant" variable is used in lieu of an actual constant, it is possible to easily alter the control.

Real-time programmers may elect to use variables that have been preset as other than "named constants". Any such usage must be done with extreme caution.

FLOATING ACCUMULATOR

The Floating Accumulator is simply a name for the first six cells in a calling program's temp stack. Since various routines employ temp storage for different purposes, this convention should not be thought of as an actual area used only for floating-point calculations. Within a sequence of floating operations, however, it is treated much like the A-register in Sigma 2/3 class 1 instructions.

COMPLEX ACCUMULATOR

In order to handle the computation of complex arithmetic functions, ANS FORTRAN IV has introduced the Complex Accumulator. The Complex Accumulator is the name for the n + 4 to n + 15 cells of the portion of the temp stack that is reserved by a FORTRAN Main program. The location of this area is held in the n + 3 cell of the Main program, and is passed to each FORTRAN program that is called.

BLANK COMMON STORAGE

The Overlay Loader provides for both program-relocatable and blank COMMON-relocatable load items. Note that the Loader will not actually place data into blank COMMON storage, but will resolve address references relative to a specified COMMON base. It is often desired to share COMMON storage between FORTRAN and Extended Symbol. Its use should be made clear by the example below, which defines identical COMMON areas:

FORTRAN

```
             COMMON DAT1(10,20),TEMP(20),ICNT,ITEMP(5),MOD
```

Extended Symbol

```
  (Assume above compilation uses the standard XDS allocation)
  INTSIZE     EQU     1                       INTEGER WORD SIZE
  REALSIZE    EQU     2                       REAL WORD SIZE
  *
  DAT1        COMMON REALSIZE*(10*20)         400 LOCATIONS
  TEMP        COMMON REALSIZE*(20)             40
  ICNT        COMMON INTSIZE                    1
  ITEMP       COMMON INTSIZE*(5)                5
  MOD         COMMON INTSIZE                    1
```

Blank COMMON is discussed in detail in the chapter "How to Build An Overlay Program" earlier in this manual.

NAMED COMMON STORAGE

ANS FORTRAN IV has introduced an additional form of named COMMON that is available for use by both the FORTRAN and Extended Symbol programmers. An Extended Symbol program may reference named COMMON that is defined in a FORTRAN program. In general, the Extended Symbol access to a named COMMON is roughly equivalent to the technique that would be used to access locations within an Extended Symbol routine that has only its first location DEF'd. The example below shows the general form of the technique to be used:

FORTRAN

```
             COMMON/ALPHA/DAT9,TEMPA(20),TEMPB(300),TEMPC(2)
```

Extended Symbol

```
  REALSIZE    EQU     2                       REAL WORD SIZE
  DAT9        EQU     0
  TEMPA       EQU     DAT9+REALSIZE
```

```
TEMPB      EQU      TEMPA+(20*REALSIZE)
TEMPC      EQU      TEMPB+(300*REALSIZE)
  .
  .
LDX        =ALPHA
LDA        DAT9,1
  .
  .
LDX        =ALPHA
LDA        TEMPA,1                          TEMPA(1)
  .
  .
LDX        =ALPHA
LDA        TEMPA+(9*REALSIZE),1             TEMPA(10)
  .
  .
LDX        =ALPHA
LDA        =TEMPC
RADD       A,X
LDA        0,1                              TEMPC(1)
```

CODING EXTENDED SYMBOL ROUTINES FOR CALLS FROM FORTRAN

Both SUBROUTINE and FUNCTION subprograms may be coded in assembly language for subsequent call by a FORTRAN program. The name(s) identifying the entry point(s) to the routine is (are) declared in a DEF directive.

STANDARD CALLING SEQUENCE

FORTRAN generates the following code when a SUBROUTINE call or FUNCTION reference occurs:

```
REF        name
RCPYI      P,L
B          name
DATA       X'n'     argument keyword
ADRL       arg1     one entry for each
  .          .
  .          .      actual argument in the call
ADRL       argk
```

The "argument keyword" specifies the addressing mode of each argument address. It consists of 1-8 two-bit codes such that bits 0-1 refer to arg_1, 2-3 to arg_2, etc. More than eight arguments in a call will have an argument keyword preceding each group of eight argument addresses. The two-bit codes have the following meaning:

00 - no more arguments (refers to argument k+1, which is nonexistant).

01 - absolute address (the ADRL contains the actual address of the argument).

10 - indirect relative address (the ADRL value, added to the present contents of the B-register, gives the address of the argument).

11 - relative address (the ADRL value, added to the present contents of the B-register, gives the actual address of the argument).

These codes have been covered in detail here because they will be used extensively in following subsections of this chapter.

ARGUMENT TRANSFER ROUTINES

There are several argument transfer routines available in the FORTRAN Library to relieve the Extended Symbol programmer of the necessity for deciphering these calling sequences. The most general routine, M:PUSH, will be covered here. The other routines are, in general, subsets of M:PUSH and may be used as the application warrants. Essentially, M:PUSH does the calculations necessary to produce an absolute address for each of the arguments in the call, and moves these new addresses into a specified temp stack for easy accessibility in the called routines.

The calling sequence for M:PUSH is

```
RCPYI       P,T
B           *$+3
DATA        n           (number of words to reserve)
ADR L       TEMP        (address of temp stack)
ADRL        M:PUSH
```

where n is equal to three plus the number of argument addresses (r) that will be converted and passed into the temp stack from the calling parameters, plus the number of temporary cells needed by the routine. M:PUSH exits with the A-register unchanged, the entry contents of the B-register in location TEMP+1, and the return address (calculated from the number of arguments and the entry contents of the L-register) in location TEMP+2. In locations TEMP+3 to TEMP+3+n-1 will be the absolute addresses of the calling arguments. The B-register will point to location TEMP. The E-register contains the number of arguments processed by M:PUSH upon return.

It should be noted that the alternate entry point M:PUSHC must be used if the Extended Symbol routine is to call (directly or indirectly) another FORTRAN routine. The M:PUSHC entry operates the same as the M:PUSH entry with the exception that n must be equal to four plus the number of argument addresses. TEMP+1 and TEMP+2 have the same contents as for M:PUSH. TEMP+3 contains the address of the complex accumulator (obtained from the calling routine). Locations TEMP+4 to TEMP+4+n-1 will contain the absolute addresses of the calling arguments.

SUBPROGRAM EXIT

The call to M:PUSH or M:PUSHC is normally placed as early in the subprogram as possible. Care must be taken to preserve the entry value of the L-register upon calling M:PUSH. Exit from a subprogram that used M:PUSH or M:PUSHC is effected simply by

```
B           M:POP       RETURN TO CALLING PROGRAM
```

TEMP STACK DEFINITION

The recommended method for terminating the source code of a subprogram is

```
LPOOL                   DROP ANY LITERALS
RES         n           SET UP TEMP STACK
END                     NO TRANSFER ADDRESS
```

Note that "TEMP" and "n" are the same as in the call to M:PUSH. The LPOOL is included as a safety measure because no data can follow the temp stack if the program is to be converted to use dynamic storage. If this routine is placed in the Public Library by the Overlay Loader, the call to M:PUSH will automatically be changed to indicate dynamic temp allocation, and the trailing temp stack will be removed from the subprogram.

ARGUMENT CHECKING

M:PUSH ensures that the calling argument list is at least no larger than the n-3 locations allocated for it by the call to M:PUSH. More arguments cause a PU abort code. However, the subprogram may well be interested in whether there are less than the specified number of arguments in the call. Note that the L-register, upon entry to the subprogram, contains the address of the first argument keyword. Thus, to check for a minimum of three arguments, we might code

```
RCPY      L,X
LDA       0,1         GET KEYWORD
AND       =X'0C00'    CHECK THIRD KEY
BAZ       ERROR       ERROR IF ABSENT
```

A FUNCTION subprogram makes use of the same techniques, the only difference being that a meaningful result must be left in the A-register and/or the Floating Accumulator upon exit. Conventions associated with such math routines are covered in the next subsection.

CALLING MATH LIBRARY ROUTINES FROM EXTENDED SYMBOL

Often a FUNCTION or SUBROUTINE subprogram coded in Extended Symbol will need to make use of the floating-point and I/O routines in the Math Library. These routines may be grouped in three main classes with regard to the calling and usage conventions: Math routines, Arithmetic routines, and I/O routines.

The address modes that are used in the reference of parameters may be classified as follows:

TYPE 1 01 Key: direct (or absolute) address.

Type 1 data refers to constants that are incorporated directly in a program. They should never be altered by the program, and are referenced as Type 1 only by the program in which they are assembled. COMMON variables may not be assembled into the program but are addressed as Type 1 data.

TYPE 2 10 Key: indirect, base-relative address.

Type 2 address references are those made to subprogram dummy variables. In this case, M:PUSH has moved the addresses of the calling program's data into the subprogram's temp stack.

TYPE 3 11 Key: base-relative address.

Type 3 data refers to non-blank-COMMON variables used within a given program. Variables should never be given initial values at assembly time, but instead should be initialized at execution time by moving Type 1 data into the subprogram's temp stack. (Blank COMMON variables must be initialized at execution time.)

The above conventions ensure the reentrancy of any subprogram that may be used in a real-time environment. Note that the DATA statement in FORTRAN sets up Type 1 data, but allows this data to be modified at execution time. A FORTRAN subprogram should thus avoid ever modifying an item declared in a DATA statement if it is to be used in a reentrant mode. FORTRAN reentrancy is further discussed in Chapter 17.

MATH ROUTINES

The calling sequence for a Math routine is the same as that used for standard function references. All registers, with the exception of the B-register, should be assumed to be volatile. The result will be returned in the Floating Accumulator.

ARITHMETIC ROUTINES

Although Math routines may be called with several arguments and thus use the keyword for address resolution, an Arithmetic routine expects at most one argument, and the address resolution is specified by selecting the appropriate entry point to the routine. The second argument of an Arithmetic routine is expected to be in the Floating Accumulator.

The call to an Arithmetic routine is of the form

```
    RCPYI        P,L
    B            p:IDn
    DATA         argument

    RETURN
```

where

p   is L for standard precision.

     X   for extended precision.

:ID   is the base name of the routine.

n   is 1, 2, 3, or 4, and corresponds to the address type of the argument.

Each Arithmetic routine except p:33CPn (real compare) increments the L-register one location past the return address before returning. This allows consecutive calls on Arithmetic routines with only an initial RCPYI P,L.

9. HOW TO USE STANDARD PROCEDURE (S2) FILES

A Standard Procedure (S2) file is an easy-to-use mechanism for allowing common symbols and often-used procedures to be stored in a special format so that they can be used automatically during an assembly, without being duplicated in each source program that uses them.

A basic set of procedures that define the Sigma 2/3 machine instructions are supplied with the Extended Symbol assembler. This set of procedures should be considered a minimal base upon which to build other installation-specific, or user and program-specific S2 files. There is no limit to the number of S2 files you may have on a system. However, RBMS2 is the only default S2 file and therefore does not need an !ASSIGN card. (!ASSIGN cards are required for all other S2 files.)

WHAT MAY BE STORED IN AN S2 FILE

There are two logical portions of an S2 file: the assembler's global symbol table, and the skeleton ("sample") procedure definitions. These form the initial symbol and sample table areas within the assembler. Additional symbols and procedure definitions are added as they are defined in the source program.

Any procedure definition may be stored in an S2 file. It may contain LOCAL directives, calls on other procedures defined in the same S2 file, or even calls on procedures that will be defined later in the source program that uses this S2 file. In general, anything that may legally occur within a CNAME, PROC,...,PEND group will be correctly stored on an S2 file.

Some care must be taken in storing Main program global symbols (and their values) on an S2 file, but the resulting convenience is often as great as that of common procedure definitions. Likely candidates for standard S2-defined symbols are register designators and Zero Table constant identifiers. Such constants should be defined as absolute values via the EQU directive. In following the two rules itemized below, such symbols should not be placed within procedures in order to save core storage. A global definition such as, A EQU 7, will require six cells of storage if its definition is invoked by a procedure name, but will require only two cells if left at the Main program level when creating the S2 file.

- Symbol values (at Main program level) stored on an S2 file should not be redefined by the source, nor should any attempt be made to store Main program-level LOCAL symbols on an S2 file.

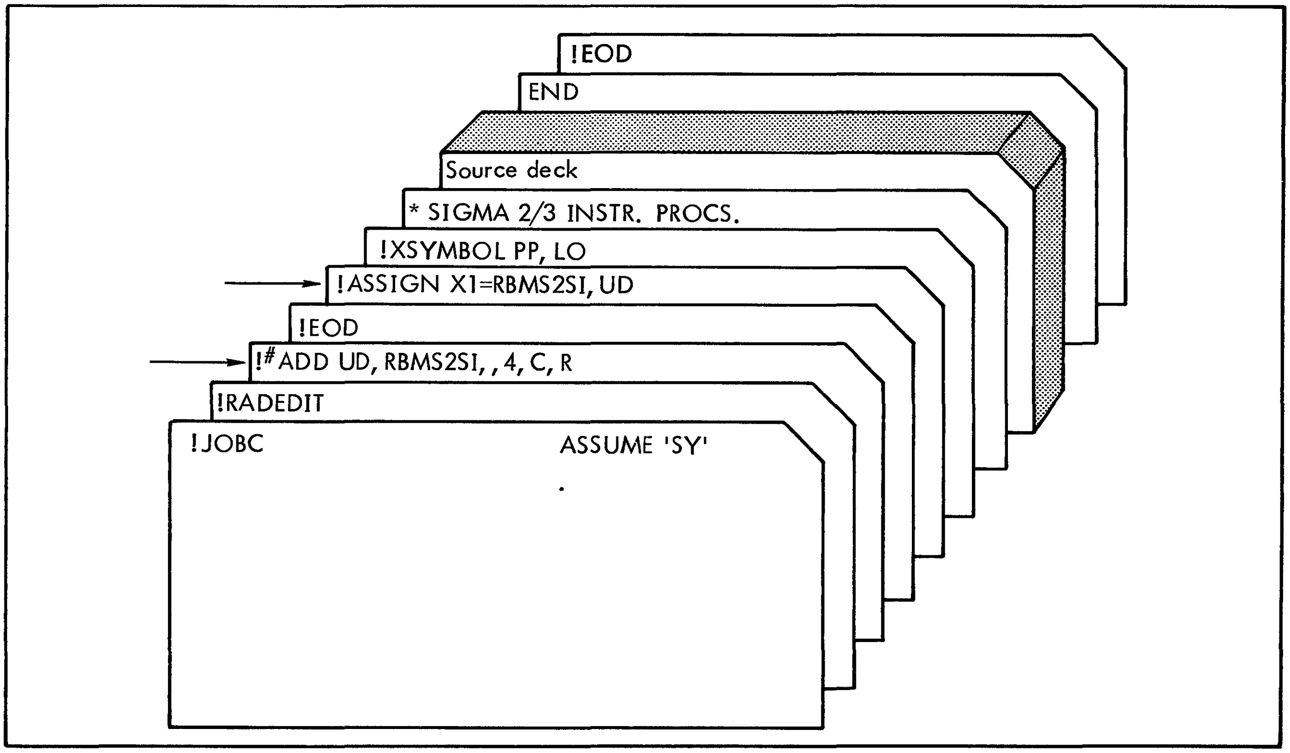
- No symbols stored on an S2 file should be used as arguments in a DEF, REF, or SREF directive (unless within a procedure definition).

The action of the assembler is unpredictable if the above rules are violated.

In summary, any legal procedure definition may be stored on an S2 file as may any unredefinable, nonexternal, global symbol value. A code-generating program should not be used to create an S2 file if the code generation occurs during creation of the file.

HOW TO CREATE AN S2 FILE

It is not possible to read in an existing S2 file, add to it, and create a new S2 file. For this reason it may be desirable to at least keep a source copy of the Sigma 2/3 instruction procedure set on bulk storage (RAD or disk pack). If a listing of this file is obtained at SYSLOAD time, it may be posted (or distributed) so as to serve as a base for special S2 files. The example in Figure 47 illustrates one method of preserving the source during SYSLOAD.

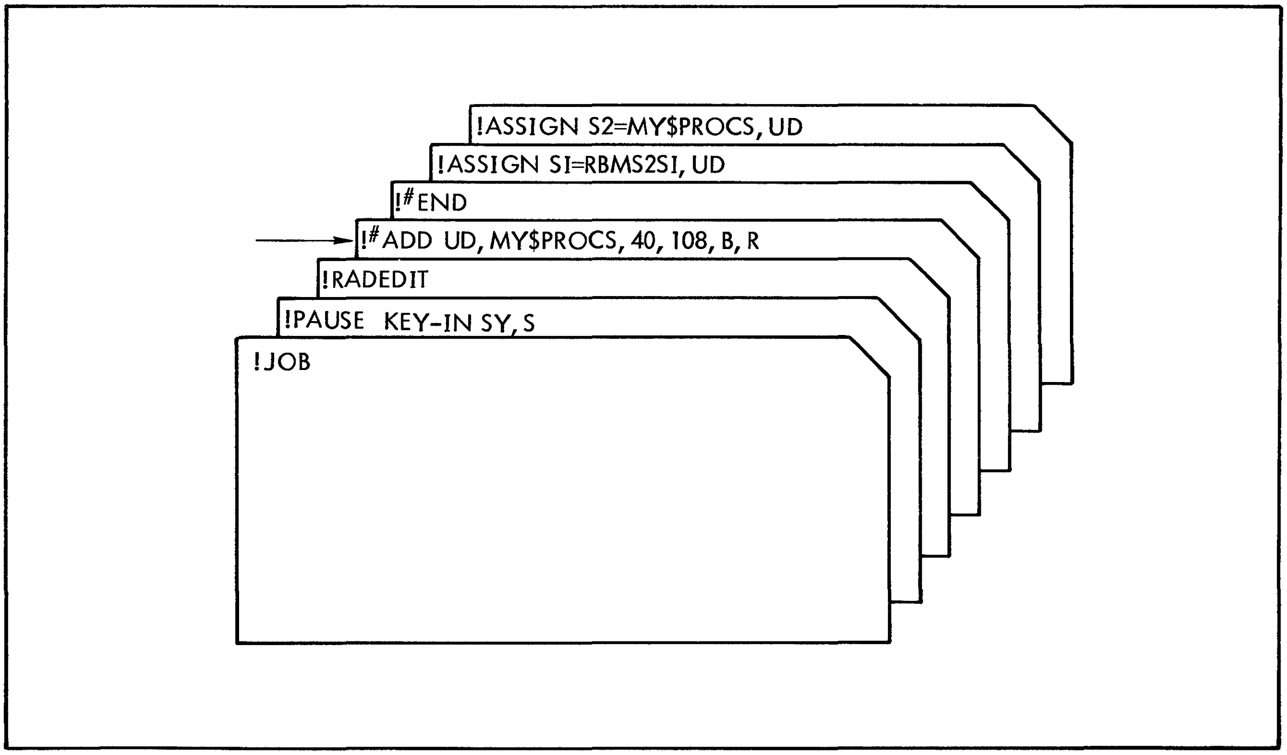


```
!EOD
END
Source deck
* SIGMA 2/3 INSTR. PROCS.
!XSYMBOL PP, LO
!ASSIGN X1=RBMS2SI, UD
!EOD
!#ADD UD, RBMS2SI, , 4, C, R
!RADEDIT
!JOBC                                ASSUME 'SY'
```

Figure 47. Save Instruction Procedures During SYSLOAD

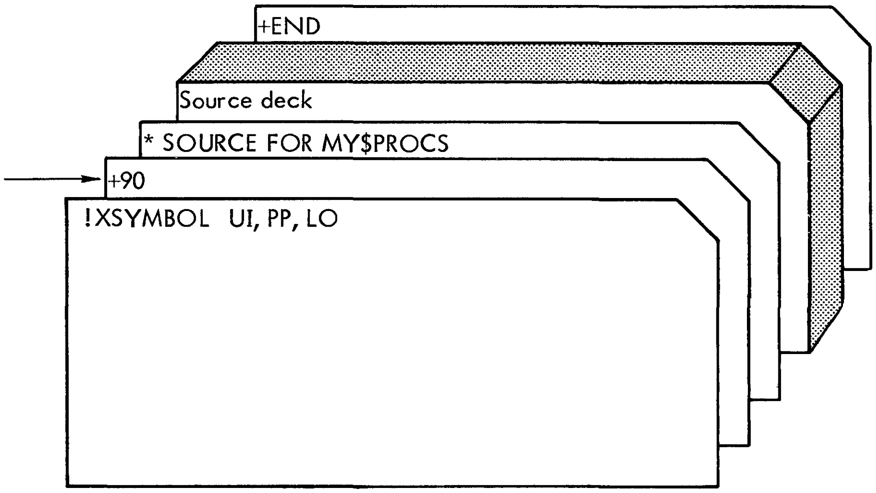
This example creates the standard (default) RBMS2 file and saves the source in a file called RBMS2SI in the User Data area. (X1 is a copy of the source and is normally used only for the assembly listing).

Assuming that the above procedure was used, the example in Figure 48 shows how a new S2 file is created using RBMS2 as a base. (Also assume that the listing from the example in Figure 47 showed the last line before the END line to be line number 90.)



```
!ASSIGN S2=MY$PROCS, UD
!ASSIGN SI=RBMS2SI, UD
!#END
!#ADD UD, MY$PROCS, 40, 108, B, R
!RADEDIT
!PAUSE  KEY-IN SY, S
!JOB
```

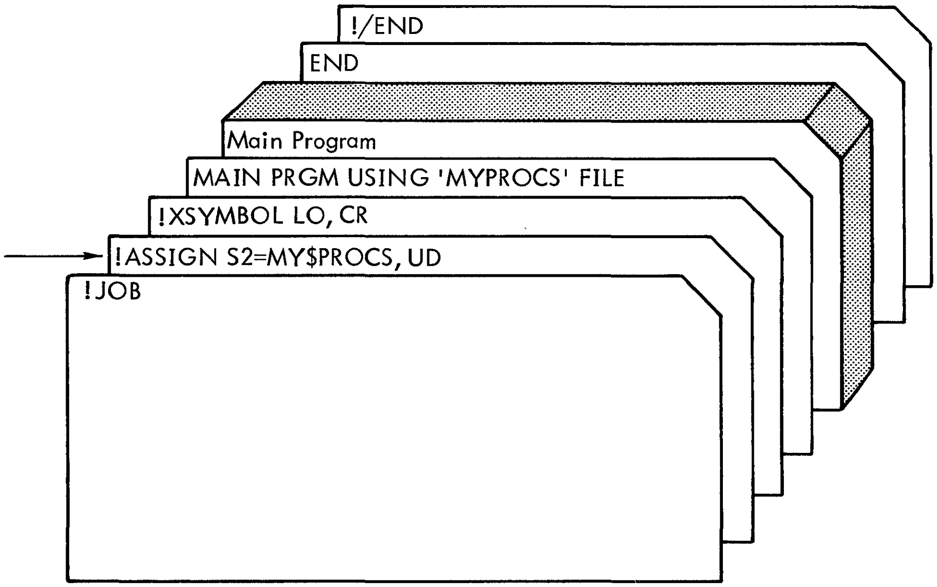
Figure 48. Create An S2 File



```
+END
Source deck
* SOURCE FOR MY$PROCS
+90
!XSYMBOL  UI, PP, LO
```

Figure 48. Create An S2 File (cont.)

The example in Figure 48 causes the source from RBMS2SI to be read, the source for the new procedures to be inserted immediately before the original END line, and the resulting set of procedures written to the MY$PROCS file in the User Data area. Note that MY$PROCS has been created using a blocked file with a record size of 108 bytes, which is the required format for S2 files. Subsequent assemblies using this new file might be done as shown in Figure 49.



```
!/END
END
Main Program
MAIN PRGM USING 'MYPROCS' FILE
!XSYMBOL LO, CR
!ASSIGN S2=MY$PROCS, UD
!JOB
```

Figure 49. Assemblies Using S2 File

The !ASSIGN card in Figure 49 prevents the automatic assignment of S2 to the default RBMS2 file.

The following update packet illustrates legal usage in creating an expanded S2 file:

```
+90
*             : OPERATIONAL REGISTER EQUATES
Z             EQU        0            ZERO
P             EQU        1            PROGRAM
              :
E             EQU        6            EXTENSION
A             EQU        7            ACCUMULATOR
*             : MONITOR CONSTANTS
K:X8000       EQU        X'09'        X'8000'
K:X4000       EQU        X'0A'        X'4000'
              :
K:M15         EQU        X'33'        X'FFF1'
K:M16         EQU        X'34'        X'FFF0'
*             : PROCEDURES
BAL           CNAME                   BRANCH AND LINK (TO SUBROUTINE)
              PROC
              RCPYI      P, CFR(2)
              B          AFR(1), AF(2), AF(3)
              PEND
              :
XR            CNAME                   EXCHANGE REGISTER
              PROC
              REOR       AFR(1), AFR(2)
              REOR       AF(2), AF(1)
              REOR       AF(1), AF(2)
              PEND
+END
```

The above packet defines Z - A as standard symbols denoting operational registers, various symbols for standard zero-table constants, and a set of user-defined procedures such as BAL, A SUBRNAME, which loads a return address into the A-register and branches to "SUBNAME"; or XR T, A, which exchanges the contents of the T and A-registers without altering the condition codes.

10. HOW TO REDUCE ASSEMBLY LANGUAGE HARDWARE REQUIREMENTS

The standard RBM processors make certain assumptions about resource allocation, and they operate most efficiently when these assumptions are met. A well-planned installation will provide these resources in the majority of instances, but exceptional problems can sometimes be accommodated within the existing resources via special techniques.

The common resource constraints are either fast core storage, or bulk (RAD or disk pack) storage. In the case of Extended Symbol, these two constraints are related, since this assembler uses several blocked files on bulk storage that require blocking buffers in fast core for each blocked file. Assuming that Extended Symbol is loaded with the standard blocking buffer parameters, an !XSYMBOL call will cause m*n words of fast core to be reserved for I/O buffers, where

m = blocking buffer size = 180 for a 720X RAD-only system.

= 512 for a disk-pack-only, or a mixed RAD/disk pack system.

n = the number of the following operational labels that are currently assigned to blocked files:

SI, UI, SO, LO, GO, BO, X1, X3, S2, DO

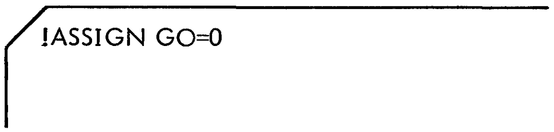
Unless reASSIGNed, the GO, X1, X3, and S2 files are assigned to blocked files by default. In addition, the X2 file requires a buffer that must be the size of a sector of the device to which the X2 operational label is assigned. On a disk-pack-only system, this would immediately remove over 2500 words from Extended Symbol's working storage (enough for assembly of approximately 2000 additional lines of program). Unless noted otherwise, the techniques given below apply to minimizing core storage requirements. Some consideration is also given to situations where such file elimination might be inadvisable.

COPING WITH EXISTING RESOURCES

Without modification of a program, the only improvement in resource demands during an assembly will be achieved by cutting down on the number of RAD/disk pack files and their associated blocking buffers.

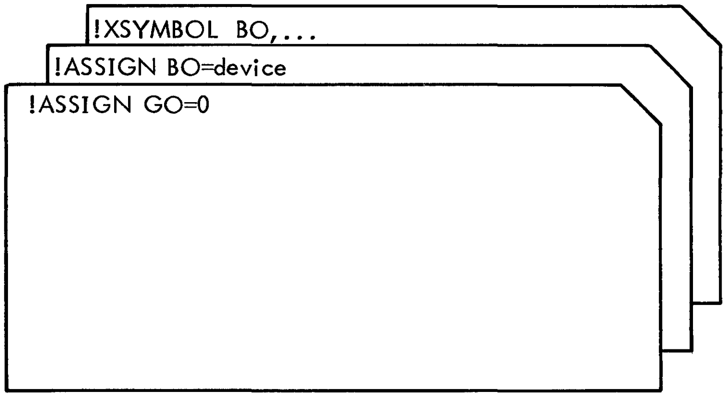
GO FILE ELIMINATION

The GO file has a default blocking buffer, and is probably the safest to eliminate. If you have a suitable binary file output device such as magnetic tape, high-speed paper tape, or card punch, it may be feasible to bypass binary output to RAD or disk pack. The command



```
!ASSIGN GO=0
```

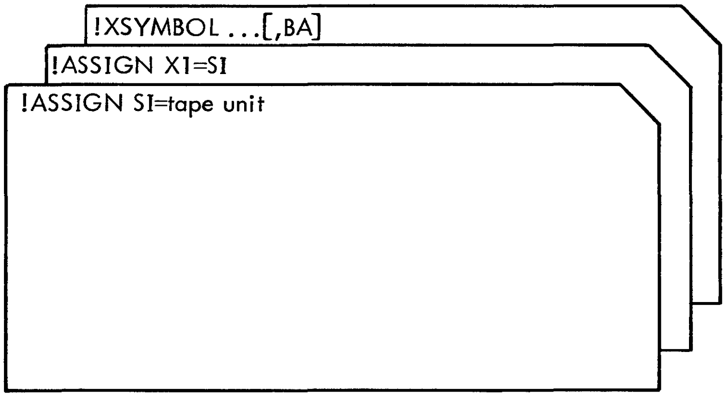
saves a blocking buffer whether binary output is requested or not. If binary output is required, the following cards are recommended for saving RAD and buffer space:



```
!XSYMBOL BO,...
!ASSIGN BO=device
!ASSIGN GO=0
```

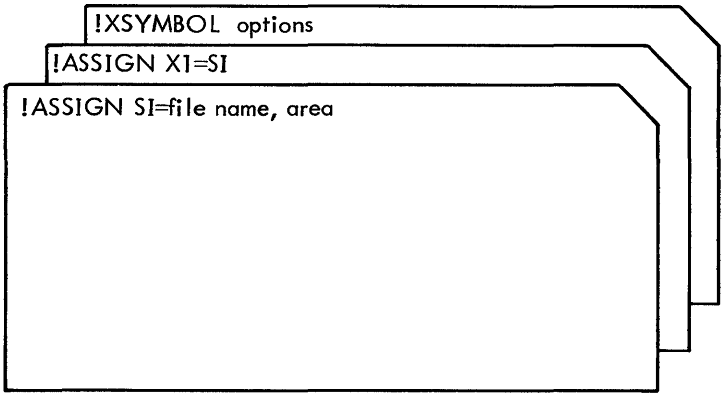
X1 FILE ELIMINATION

The X1 file has a default blocking buffer and is used by Extended Symbol to maintain a copy of the source (with possible updates) for listing purposes. If you are not using the update feature of Extended Symbol, it may be possible to eliminate the X1 file. If you have magnetic tape and the one or more source programs are separated by file marks on the tape, the following commands will eliminate X1 and its buffer:



```
!XSYMBOL...[,BA]
!ASSIGN X1=SI
!ASSIGN SI=tape unit
```

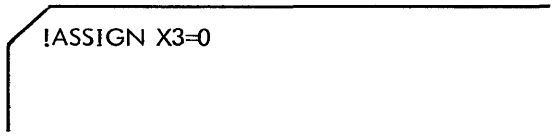
In the case where your source program is already on a RAD/disk pack file, it is still possible to assemble this one program and eliminate X1. This may be done as follows:



```
!XSYMBOL options
!ASSIGN X1=SI
!ASSIGN SI=file name, area
```

X3 FILE ELIMINATION

The X3 file has a default blocking buffer, but this file is only used if the source program uses LOCAL directives at the main-program (i.e., not within a PROCedure) level. If it is known that a program does not use main-level LOCAL symbols, the !XSYMBOL command should be preceded by



```
!ASSIGN X3=0
```

It is not necessarily a good practice to avoid main-level LOCALs however, since their use can generally reduce resource requirements more than their avoidance (see, "Coding for Existing Resources" in this chapter).

S2 FILE ELIMINATION

The final file with a default blocking buffer is S2. The S2 file should be left assigned to RAD/disk pack if at all possible. While the procedure for running S2 from paper-tape, cards, etc. is easy to perform, the method is highly dependent upon the particular hardware configuration.

REDUNDANT FILE ASSIGNMENTS

In general, unnecessary or redundant assignment of assembler files to RAD or disk pack should be avoided. This precept includes assigning unused default files to device zero. A redundant assignment would include something like assigning BO=GO, or DO=LO, where GO and LO were assigned to RAD/disk pack files.

CODING FOR EXISTING RESOURCES

There is an upper limit to the size of a program that may be assembled in a given system configuration. In Extended Symbol, the limit generally depends upon core size balanced against the number of unique symbols in a program.

The most obvious technique for reducing core requirements during an Extended Symbol assembly is to reduce the number of unique symbols in the program. This can be accomplished by using several LOCAL regions in the program and using the same LOCAL symbols for each region, as shown in the following example:

```
            .
            .
            .
            LOCAL      $10, $20, $30
AROUTINE    RES        0
            .
            .
            .
$10         RES        0
            .
            .
            .
$20         RES        0
            .
            .
            .
$30         RES        0
            .
            .
            .
            LOCAL      $10, $20
BROUTINE    RES        0
            .
            .
            .
$10         RES        0
            .
            .
            .
$20         RES        0
            .
            .
            .
            END
```

The above technique is practical in most programs since programs are often divided into many separate routines; the routine name being global to the whole program, but many symbols within the routine are referenced only by that routine.

Note that main-level LOCAL symbols do require RAD or disk space during an assembly. The requirements are three words per symbol per LOCAL region. The above example would thus require 15 words of bulk storage. Since a blocking buffer is required for the LOCAL (X3) file, this technique must be used consistently over moderate-to-large programs before the savings in core storage becomes effective.

11. HOW TO USE HARDWARE INTERRUPTS

This chapter forms a natural division in the organization of the User's Guide. All previous topics have dealt either with purely background applications or with those services and procedures used in common by both foreground and background. The descriptions and suggestions for using the hardware interrupt system given below mark the entry into the real-time world of RBM, and it is recommended that some attention be given to this material before going on to succeeding chapters. By studying the capabilities and implications of the hardware interrupt system, you will be better able to call on these resources in a way best suited to your own foreground programming requirements.

In particular, all foreground users should at least be cognizant of the internal interrupt structure and purposes of the RBM Tasks that comprise the Monitor, since RBM is itself a real-time program that must respond to time-critical events such as I/O interrupts and operator interrupts. The interrupt levels of the RBM Tasks and their interrelationship with user tasks and programs are described briefly at the end of this chapter as an example of one way to use hardware interrupts. (A description of the Tasks' functions is given in the Real-Time Programming chapter of the RBM Reference Manual.)

Note that some foreground programs can utilize interrupt levels without being real-time programs, and one suggestion for such use is given later in this chapter.

PURPOSE OF HARDWARE INTERRUPTS

The Sigma hardware architecture includes a powerful hardware priority interrupt system. This consists of a multilevel interrupt structure composed of both external and internal levels arranged in an expandable, flexible, and partially pre-selected order. RBM has been specifically designed to use this hardware priority interrupt structure to the fullest extent possible, and this particular structure's purpose is to efficiently allocate the CPU. The implications of this structure are described in detail to suggest ways that you can use Sigma interrupts. For if you do not take advantage of these features when designing a real-time system, the full power of the hardware approach is lost. To get the maximum utilization out of a Sigma 2/3, your real-time system design should be based on a clear understanding of the power and flexibility of the hardware interrupt system.

The detailed implications of Sigma interrupts are as follows:

- Fast real-time response: On the occurrence of some predetermined external or internal event, the Sigma 2/3 can stop its current operation and switch to an entirely different operation within a few instructions. This permits real-time programs to operate in a time-critical manner.

- Priority response: Since each hardware level has an implied priority (by its position in the interrupt chain) and since a level only interrupts the CPU if it is the highest active level, the CPU is guaranteed to always be working on the most important (highest priority) operation that needs attention in the system.

- Low overhead: No time is spent by the CPU in posting to software queues, periodically scanning these queues, and checking to see if something new has arrived on the ready queue that is higher priority than the current operation; instead this is all done automatically by the hardware priority interrupt system. In fact, this system can be viewed as a separate "processor" that executes in parallel with the CPU, maintaining a "queue" in the interrupt hardware and operating in a microprogrammed fashion to do the scheduling for the Sigma 2/3 CPU. Thus, the Sigma 2/3 CPU can be involved with solving user problems instead of trying to decide what to do next.

- Noninterference: When events of a lower priority than the currently executing program become ready, this fact is noted by the hardware interrupt system but the CPU is not diverted away from its currently more important operation (even for a microsecond) to record the fact and make a decision relative to the priority of these two tasks; this decision is accomplished automatically in the interrupt hardware.

- Asynchronous operation: The hardware interrupt system is truly asynchronous; that is, it executes a task at a specific interrupt level only when that level goes active as the result of some specific event. If the times of such an event are variable and random, this asynchronous but immediate response is highly important. (Asynchronous operation is considered in detail later in this chapter.)

- Anonymous operation: The design of tasks need not be concerned with what other tasks may be operating when an interrupt occurs, or what context the interrupted task was using upon entry. Each task saves (in its own area) the registers and other temporary working storage that it will modify, does its own function, and then restores these registers independently of what the other task was. Thus, this is a strictly LIFO (last-in, first-out) method of scheduling that minimizes debugging and design problems. Another way of defining anonymous operation is that an interrupted task is never aware that it was interrupted, except where timing considerations (such as real-time clock pulses) are a factor.

- Flexibility: From one to 100 interrupt levels can be used and they can be arranged in various priority levels; some above and some below the I/O level and other RBM levels. By selecting the number of interrupts, the groups of interrupts, the priority of these interrupts, and the source of activation pulse for each level, each installation can fit its own unique demands.

SUMMARY OF HARDWARE INTERRUPT FEATURES

A complete description of the features of the hardware interrupts can be found in the Sigma 2 or 3 Computer Reference Manuals. However, the key points as related to your program design can be summarized in this chapter.

The interrupt "environment" (CONNECT, ARM, ENABLE, INHIBIT) is controlled either directly through special RD or WD instructions coded into your program, or indirectly through RBM Monitor service routines accessed by calls, key-ins, or control commands. The hardware interrupts for Sigma 2/3 computers possesses the following characteristics:

- No level may advance to an active state while a higher level is active.
- Under program control, individual levels (or "groups") may be set to ignore incoming signals (DISARMED) or to postpone reaction to these signals until some later time (DISABLED or group INHIBITED).
- The initial condition of all interrupt levels (except the override group, when the options exist) is DISARMED and DISABLED.
- Interrupt levels may be TRIGGERED either by program control or by external signals.
- All levels (except the override group) may be inhibited by a single instruction; the inhibit may also be removed by a single instruction.
- The internal and external interrupts can be inhibited either separately or at the same time.
- The previous state of interrupt inhibits can be saved, and new inhibit conditions set or reset in a single instruction.
- The previous interrupt inhibit state is saved (in the old PSD) on a task entry sequence, and the first instruction of the task is always executed before another interrupt can take place; thus, this first instruction can inhibit all further interrupts if desired.
- No level may advance from the waiting state to the active state unless it is ENABLED and not inhibited.
- The hardware priority sequence may be arranged in virtually any priority order, either above or below the I/O group. The override group is always high.
- An interrupt level should not be DISARMED while it is active because the results are unpredictable. Since the hardware priority search during the EXIT sequence is based on certain mutually exclusive states of the interrupt flip-flops, DISARMING causes a level to be ignored even though it is active.
- Only one-level of signal is remembered by interrupts. That is, if a level is already waiting to go active, another TRIGGER will have no additional effect on it; and if a level is already active, another TRIGGER is ignored.

Some of the implications of these characteristics are as follows:

- Real-time programs can be debugged by using software triggering for certain levels before the real-time hardware is connected.

- Real-time programs can work in groups by using internal WD (Write Direct) instructions to TRIGGER various levels; thus, a high priority level may be connected to external signals and collect data at the high priority, and then TRIGGER a lower priority to process the data at its leisure.

- Under program control you can select which interrupt levels to initially ARM and ENABLE, and can reject or postpone future signals based on program logic or program computations.

- Real-time programs that have to inhibit interrupts anywhere (except as the first instruction on task entry) should do so by a save-and-inhibit sequence (a special RD instruction). Later, they should never "remove" interrupt inhibits but should always "restore" them to their previous state; either by executing an EXIT sequence (and using the old PSD), or by a "restore" sequence that uses the information from the save-and-inhibit sequence. This is because some tasks may inhibit externals and not internals, or vice-versa.

- If an active task wishes to have all future signals to its own level ignored, it should DISABLE this level and EXIT rather than DISARM it and then EXIT. Some future, lower-priority task can DISARM the disabled level if this is really necessary. One should not DISARM an active level. One should also not ARM an active level since this interferes with the current state.

INTERRUPT TASK SCHEDULING

The possibilities for real-time task scheduling based on these hardware interrupts are very broad, subject only to the LIFO requirement. The several suggestions given below are meant to be general guidelines only.

A particular interrupt level can be used to uniquely identify an event that requires processing. At the same time, it establishes the priority of this processing relative to other levels. Or, an interrupt level (such as the I/O level) may only identify a class of possible events, and further information may be required to identify the specific event.

When a series of tasks, each of which is connected to separate interrupt levels, all use the same database (tables or files), the other tasks in the group can be DISABLED while any one of them is modifying critical portions of the database, and it still permits higher priority interrupts in another group to become active if necessary. A general inhibit instead of a DISABLE would not permit this. Also, any signals to the DISABLED tasks will be "remembered" for later processing, which would not occur if a DISARM were used.

A task connected to a level higher than the I/O level can be activated from some critical real-time event (such as an over-temperature condition in a process plant), and this task is guaranteed 100 μs response to any signals, assuming this is the highest real-time user level, since RBM never inhibits interrupts for longer than 100 μs.

A user task at a high priority level may sample some data input devices periodically and then TRIGGER other, lower-priority levels associated with some particular condition that requires further processing, and such processing can be performed at a lower level that is commensurate with the importance of the particular condition.

RBM itself uses some interrupt levels for its own processing as identified in the RBM Reference Manual. But generally speaking, RBM does not interfere with the real-time interrupt levels and user programs are free to make their own scheduling rules.

It is by no means necessary to limit the use of interrupt levels to real-time operations. A tape-to-printer routine in the foreground can be connected to an interrupt level and can use the AIO Receiver and no-wait I/O operations to schedule and synchronize itself in order to buffer output to a printer, and so permit other (lower priority) tasks (including the background) to execute while I/O is in progress for this task.

SOFTWARE SCHEDULING OF SUBTASKS

We stated earlier by implication that attempts to schedule CPU allocation through user software, rather than taking full advantage of the hardware features, would result in degradation of the system. This is true at the primary task level but software scheduling within a task can be useful. While the primary (task) scheduling in RBM is strictly off the hardware priority interrupt system, it is possible for a task at a particular hardware interrupt level to organize itself into a series of subtasks.

Suppose that in your installation a very large number of distinct real-time events are possible. And suppose that many of them are processed in little groups, each related in some way to an external event. It is not necessary to have a separate task with a separate interrupt level for each of them. There is a concept of secondary scheduling that can be controlled by user software, at your discretion, through any of a number of schemes.

When the primary task is activated by its hardware interrupt, it might identify a subtask (or subfunction or subevent) to be performed by means of status information read in from the external equipment. Communications equipment and analog or digital converters very often operate this way. Or, a fixed number of subtasks under the primary task might be processed sequentially if their execution sequence is always fixed and always known. Or again, the priority of each subtask might correspond to a bit in a software status word and the primary task might search the status word from left to right, looking for the highest priority subtask to process. These bits might be set by the primary task or by other subtasks, based on conditions during the processing of these subtasks.

Many other methods are also possible. A primary task could be thought of as special foreground executive, with the job of scheduling the activity of a set of related subtasks. As a special RBM service, there are 32 dedicated locations in low core (mail boxes) available to all real-time programs to aid in this intratask communication.

The rules for determining which events should be processed as primary tasks and which as subtasks are very simple:

1. At least a portion of all primary tasks must be resident to answer a hardware priority interrupt. Subtasks can be resident or can be nonresident overlay segments; if overlaid, the overlays are controlled from the primary task by calls to RBM service routines.

2. All subtasks must operate to completion (in regard to other subtasks at the same level) or until they explicitly release control back to their primary task executive before another subtask at this same level can be scheduled. Thus, the types of events that can operate as subtasks are restricted in regard to other subtasks at this level, since this is basically synchronous operation. But they are not restricted relative to other primary tasks. Primary tasks at separate interrupt levels can interrupt each other immediately when an event occurs that needs attention. Thus, primary tasks are basically asynchronous and are much more responsive than subtasks.

3. Control of subtasks is centralized at their primary task and is exercised through software. Primary tasks are controlled with decentralized hardware scheduling.

RBM ORGANIZATION

To better illustrate the idea of programs, tasks, and subtasks, the detailed structure of RBM should be examined, since RBM is itself a real-time program with several tasks and subtasks. RBM uses up to nine of the fixed hardware priority levels on a Sigma 2/3 and one assignable external interrupt level that is controlled by software triggering from other tasks or from Monitor service routines. This priority interrupt structure is illustrated in Figure 50. The RBM Control Task level is designed primarily for operator control and for control of the background, and must not interfere with the foreground. Thus, it must always be assigned to an interrupt level below all the foreground priority levels. The resident part of this level causes the various RBM subtasks to be loaded from the RAD as needed.

The other RBM tasks perform a minimum of analysis at their level, set status bits in the RBM Control Task control word, trigger the RBM Control Task level, and then exit. For example, when the operator activates the Control Panel interrupt, which is just below the I/O interrupt and above most of the real-time tasks, this RBM Control Panel Task sets a bit in the RBM Control Task status word to signify that the operator key-in subtask is needed, and then triggers the RBM Control Task. Thus, operator key-ins do not interfere with foreground operation, since these key-ins are designed to control batch background processing. Similarly, if the background tries to execute a privileged instruction or tries to branch to protected core, the Memory Protect task is activated; this task sets the Abort subtask flag, triggers the RBM Control Task, and then exits. From the address in the program status doubleword, the RBM Abort subtask can tell the exact location causing the protection violation. This information is printed in the abort message to aid in debugging background programs. The printing of messages and the deactivation of the background does not interfere with foreground operation. Thus, the entire set of RBM tasks works as an asynchronous whole to control the operation of the system.

Discussion on the RBM handling of input/output to achieve multi-task operation is in order at this point. Remember this fundamental rule:

- All input/output is initiated and checked for completion at the priority level of the requesting task. Further, all input/output uses interrupt control to coordinate I/O activity.

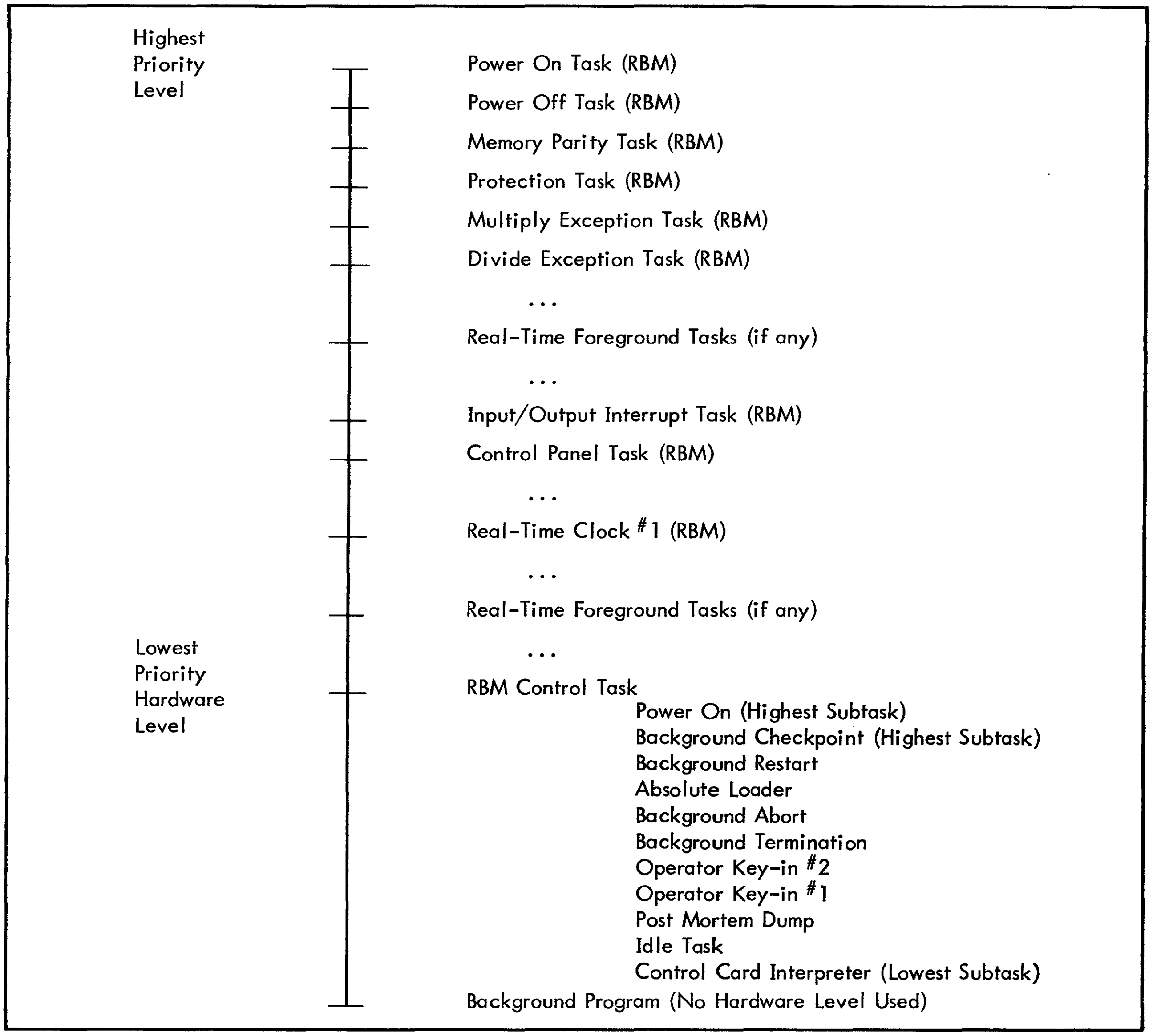

Highest Priority Level
Power On Task (RBM)
Power Off Task (RBM)
Memory Parity Task (RBM)
Protection Task (RBM)
Multiply Exception Task (RBM)
Divide Exception Task (RBM)
. . .
Real-Time Foreground Tasks (if any)
. . .
Input/Output Interrupt Task (RBM)
Control Panel Task (RBM)
. . .
Real-Time Clock #1 (RBM)
. . .
Real-Time Foreground Tasks (if any)
Lowest Priority Hardware Level
. . .
RBM Control Task
Power On (Highest Subtask)
Background Checkpoint (Highest Subtask)
Background Restart
Absolute Loader
Background Abort
Background Termination
Operator Key-in #2
Operator Key-in #1
Post Mortem Dump
Idle Task
Control Card Interpreter (Lowest Subtask)
Background Program (No Hardware Level Used)

Figure 50. RBM Hardware Priority Interrupt Levels

To prevent the problem of I/O hang-up on shared devices like the RAD, the I/O Interrupt Task in RBM saves end-action status information in a task context area called a Device File Table that is unique to each task. For example, if the background initiates an operation on the RAD and then is interrupted by the foreground before the operation is complete, the I/O Interrupt Task saves the device status at channel end in the specified background Device File Table and frees this device for further use. The foreground may then use this device. Later, when control returns to the background and when the data is needed, a check is made to determine if the operation was completed successfully. If any retries are necessary, they are performed here. Otherwise, the operation is complete. Standard error recovery is provided for all devices, but user programs can elect to treat errors in any manner they choose.

A very important service that is in keeping with the philosophy of asynchronous operation is the AIO Receiver. This permits a foreground task to initiate I/O with a no wait option. When the Monitor has then successfully initiated the I/O operation, it returns control to the foreground task; the foreground task can then set a flag for itself that I/O is pending and exit to a lower priority task (or to the background). Later, when this level becomes active, processing will continue for that task. This feature can be used in the foreground to permit more efficient use of the computer. Thus, users have a choice about releasing control during I/O operations. This is an efficient way to buffer or queue I/O operations for foreground tasks. (A foreground program could have one task to do nothing but queue and buffer for other tasks, for example.)

12. HOW TO CREATE A TASK CONTROL BLOCK

A Task Control Block (TCB) is a convenient means for storing and organizing the information required to allow various foreground tasks to operate and interrupt each other in an orderly manner. The Monitor assumes that a TCB is the first loadable item within a foreground program. The TCB is used by the Monitor service routines M:SAVE, M:EXIT, M:LOAD, M:OPEN, M:CLOSE, and also when a C: (connect) control command or C key-in is read. (See the next chapter, "How To Connect Tasks To Interrupts" for more details about TCB and C: interface.)

You have two alternatives in the creation of a Task Control Block for foreground use: code your own TCB (if programming in Extended Symbol), or allow the Overlay Loader to create it. If you wish to code your own TCBs, refer to Chapter 6 of the RBM/RT, BP Reference Manual, 90 10 37 for detailed information on TCB composition. The information given below deals entirely with Loader-built TCBs.

If the Loader builds the TCB, it does so completely; that is, no initialization of the TCB by the user is allowed.

A foreground root may contain one or more tasks, with each task connected to its own interrupt. This is accomplished through multiple Overlay Loader !$TCB commands within the root loading sequence. The first !$TCB command must precede the !$ROOT command and is called the "initial" TCB. The only difference between it and subsequent !$TCB commands is that a "temp" parameter on the initial !$TCB command is ignored; instead, the value of the "temp" parameter from the !$ROOT command is used.

The other two parameters on the initial !$TCB command, w_1 and w_2, are placed by the Loader in the next two locations of the TCB. For simplicity, these words are usually written as hexadecimal numbers (e.g., preceded by a +), although if desired, they could be written in decimal. Groups of bits within these two words are used as indicators and interrupt locations. See the "Task Control Block" table in Chapter 6 of the RBM/RT, BP Reference Manual, 90 10 37 for more detailed TCB construction.

The first parameter, w_1, is constructed as follows:

Bits 0-3 contain the A register bit number for a Write Direct instruction. This is a number from 0 through F (hexadecimal) associated with each individual interrupt within its group. Refer to Table 1 in Chapter 2 of the Sigma 3 Computer Reference Manual, 90 15 92. The list of numbers below the heading "Write Direct Register Bit (3)" of this table is the A register bit number referred to above.

Bit 4 is a flag indicating whether the Monitor should set core locations 1 through 7 when M:SAVE or F:SAVE is called. If any Monitor service routines are called within the task, bit 4 should be zero indicating that the Monitor is to set these core locations.

Bit 5 is not used but should be zero. Bit 6 indicates whether the interrupt should be triggered when the task is loaded ("1" means trigger, and "0" means not to trigger).

Bits 7-15 contain the core location (in hexadecimal) of the particular interrupt to be associated with this task. Refer again to Table 1 in Chapter 2 of the Sigma 3 Computer Reference Manual 90 15 92. The leftmost column in this table gives the location for each interrupt.

Assuming that w_1 was written as +C30C, the indicators would mean that the interrupt wired to location X'10C' (268) is associated with this task, the Monitor is to set core locations 1-7 (the usual case), and interrupt X'10C' is to be triggered when its task is loaded into core. Remember that the Overlay Loader does not load programs into core; this is the function of the M:LOAD Monitor service routine.

The second parameter, w_2, is constructed as follows:

Bits 0, 1, 2, 4 and 8-11 are always zero, and bit 3 is always 1. Bits 5, 6 and 7 contain an "operation code" that is used with the Write Direct instruction. The "Interrupt System Control" section in Chapter 2 of the Sigma 3 Computer Reference Manual, 90 15 92, describes the action taken with each of these codes. Bits 12-15 of w_2 contain the Group Number of this interrupt. These numbers are listed in the rightmost column in Table 1, in Chapter 2 of the Sigma 3 Computer Reference Manual.

Assuming w_2 contained +1200 in conjunction with a w_1 containing C30C as described previously, this would cause interrupt X'10C' to be armed and enabled when the task is brought into core. This means that when the program task is loaded into core, interrupt X'10C' would be armed and enabled (the "2" in 2 S+1200) and then triggered (since bit 6 in 1 is a 1), causing the computer to set its P register to the address contained in location X'10C'. This address would have been stored in this location by M:LOAD, using information placed in the task's TCB by the Overlay Loader.

The flexibility of the "operation code" technique for interrupt control allows the user a wide variety of interrupt handling methods. For instance, if this example used a code of 3 (that is, if w_2 were +1300), the interrupt would have been armed and disabled on loading. The Monitor would have tried to trigger it, but no action would have occurred. However, since an interrupt that is armed and disabled "remembers" a trigger, a different task could enable this level, which in turn, would cause interrupt X'10C' to go "active" (transferring control to its task) as soon as it became the highest priority active interrupt.

You may define multiple tasks within a single root by having additional !$TCB commands subsequent to the initial !$TCB command. Each must be followed by one or more !$LD commands to load the ROMs for that task. A !$TCB command may also be followed by a !$BLOCK command to specify any oplbs that may require blocking buffers in that task.

The "temp" parameter on !$TCB commands subsequent to the initial one is used to reserve temporary space in the same way as on the !$ROOT card. If the "temp" is absent, a default value of 80 (X'50') is supplied by the Loader.

Since there is a heavy interface between the !$TCB commands and C: control commands or key-ins, it is recommended that you now turn to the next chapter, "How To Connect Tasks To Interrupts".

13. HOW TO CONNECT TASKS TO INTERRUPTS

The function of linking a foreground task to its interrupt, and optionally controlling the state of the interrupt is performed through the Monitor !C: control command or the C: operator key-in. The !C: control command and C: operator key-in function in precisely the same way and have identical parameters. For brevity, they will be termed "Connect" commands in the rest of this chapter. A frequent use of the Connect commands is to check out real-time systems that use externally triggered interrupts. Once the foreground is initialized, operation of the various tasks may be checked without the necessity of actually applying signals to the interrupt lines.

The first and mandatory parameter for a Connect command is the first core location of the task's TCB.[†] The second and optional parameter is an "operation" code. This is a number from 0-7, and if present, is used by the Monitor in place of the code contained in bits 5, 6, and 7 or word 2 of the TCB; that is, w_2 on the !$TCB command. However, the data in the TCB is not changed.

For instance, assuming a foreground task had been loaded with a !$TCB command where w_2 was +1100, it would be brought into core with it's interrupt disarmed. A C: key-in with a code of 2 could then be used to ARM and ENABLE the interrupt. A second C: key-in (for the same TCB) with a code of 7 would then TRIGGER the interrupt, and if no higher-priority interrupt were ACTIVE, the task would receive control.

Note that if you request the Overlay Loader to build a TCB, but have supplied a transfer address (that is, a label in the argument field of your END statement), M:LOAD will honor this as an initialization entry point and this address will also be used for interrupt entry. If this is not appropriate, you must alter it in your code. If the Loader builds the TCB but no transfer address is supplied, the interrupt entry will be the first word of the program. The Loader will output a "OLERR TA" message and set an error level of 1; however, this is only a warning message and does not affect program execution.

[†] The location of the task's TCB is the location immediately following the keyword "ORG" on a load map. See the load map example in Chapter 6, "How To Build an Overlay Program".

14. HOW TO ATTAIN REENTRANCY IN ASSEMBLY LANGUAGE SUBROUTINES

Reentrancy in a subroutine permits the subroutine to be interrupted during its execution for one task by a higher priority task, entered and executed by the higher priority task, and later reentered and continued for the original task with all previous environment saved. The advantage of reentrancy, of course, is the savings in memory space achieved by the sharing of procedural code.

Reentrancy is made possible through the use of two special hardware registers: the base register and the link register.

The base register (B) is used by reentrant routines to point to a temporary scratch area (called the "temp stack"), that is allocated by the Loader, and is unique to each task. The base register contains an absolute core address that is the start address of the temp stack. Note that the Sigma 2/3 instruction set permits use of both a base register and an index register (with or without indirect addressing) which is a powerful technique for manipulating data and address values.

The link register (L) is used in reentrancy to save the return address in all subroutine calls. Since no subroutine area can be modified, a method for subroutine calling that uses a branch-and-store instruction counter will not work, because it would store the return address in the subroutine area. However, with the link register as a separate register for the return address, linking is quite easy.

With the exception of the B register, all of the hardware registers can be used as a temporary scratch area in a manner similar to temp stack usage.

There are two inter-dependent software parts that are responsible for providing reentrancy: the task and the reentrant subroutine. If the proper conditions are not met in both items, no reentrancy is possible. That is, the task is not itself reentrant, but if it calls a reentrant subroutine and the subroutine requires more working storage than can be provided by the general registers, then the calling task must provide a temporary storage area for the reentrant subroutine. The reentrant subroutine will use this area as required.

When a task's interrupt occurs, the pointer to the temp stack of the interrupted task is switched by the interrupting task via the M:SAVE Monitor service routine. This pointer (K:DYN) is set in the task's TCB to identify the temporary work area for the reentrant subroutine. In order for the subroutine to get the B-register set to the unused part of the stack, it should call M:PUSH upon entry. To release this space before the subroutine exits, it should call M:POP. This temporary space is illustrated in Figure 51.

The address of the temp stack is in word 3 of the Task Control Block. This address can also be found in location 6 (K:BASE). The best method for using the stack for temporary storage of up to "n" words is to use the M:RES and M:POP Monitor service routines, where the calling sequence

```
RCPYI     P,T

B         *$+3

DATA      n  (number of cells)

DATA      0

ADRL      M:RES
```

would save the previous value of B in the temp stack and set B to the FWA of the temporary scratch area (within the temp stack) being allocated, and the sequence

```
LDA       =RETURN

STA       2,,1

B         M:POP
```

would set up the return to TEMP+2, after releasing the current temp storage stack and restoring the previous value of B.

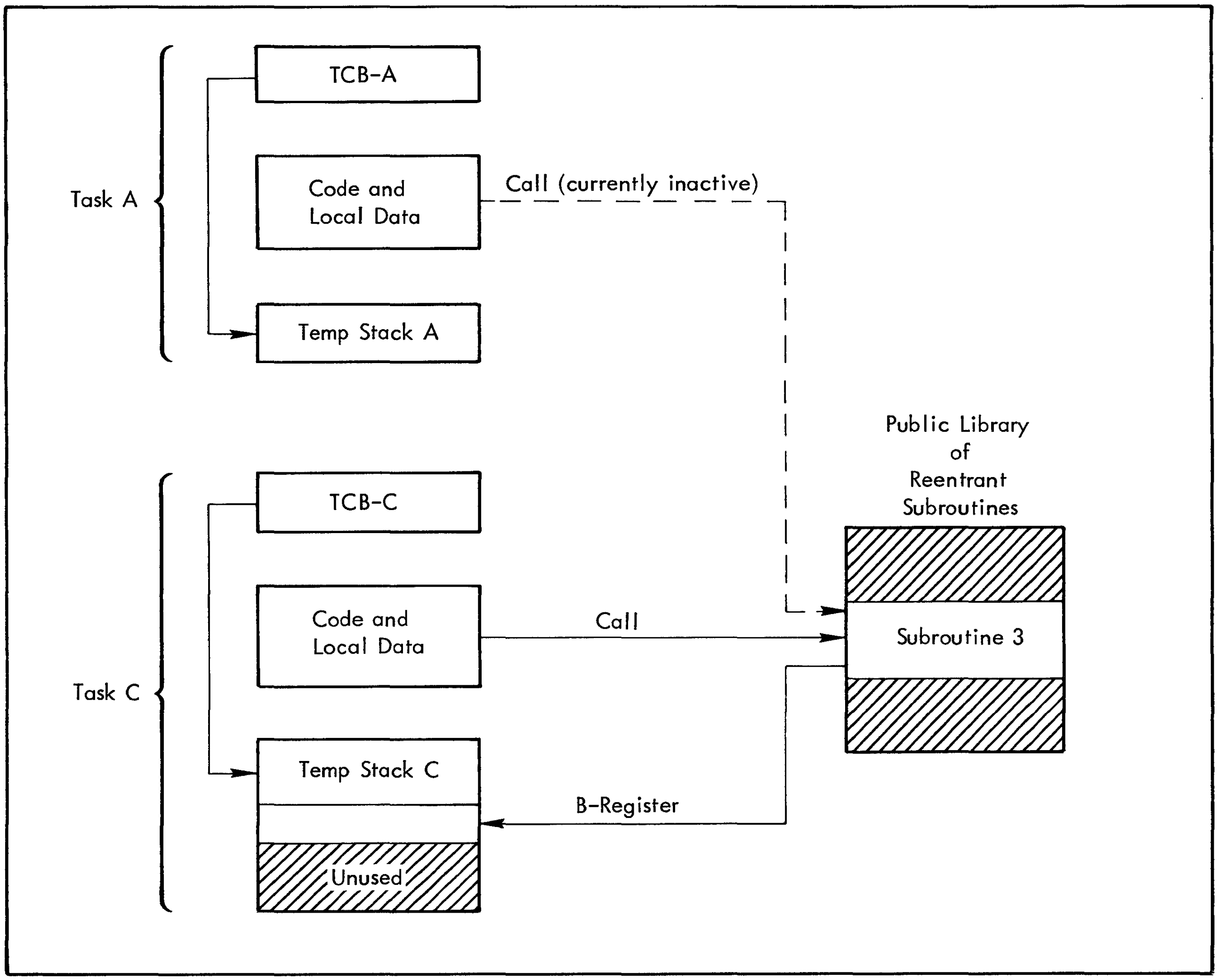


Figure 51. Reentrant Subroutine Calling Example

The size of the required temp stack is determined by the maximum nesting of subroutine calls. For example, assume the following events:

Task C calls Subroutine 2, which requires 15 words of temporary space.

Subroutine 2 calls Subroutine 3, which requires 8 words of temporary space.

Task C must therefore provide a temporary stack with a minimum of 23 words.

Let's further assume that Task C has a total of 50 words of temp stack. The temp stack would then appear as illustrated in Figure 52 when Subroutine 3 was executing.

During the execution of Subroutine 3, the base register does not point to the beginning of the temp stack, but instead, points to the beginning of space required for Subroutine 3.

If Task C had called Subroutine 3 directly instead of indirectly from Subroutine 2, the space required for Subroutine 3 would have been at the top of the stack.

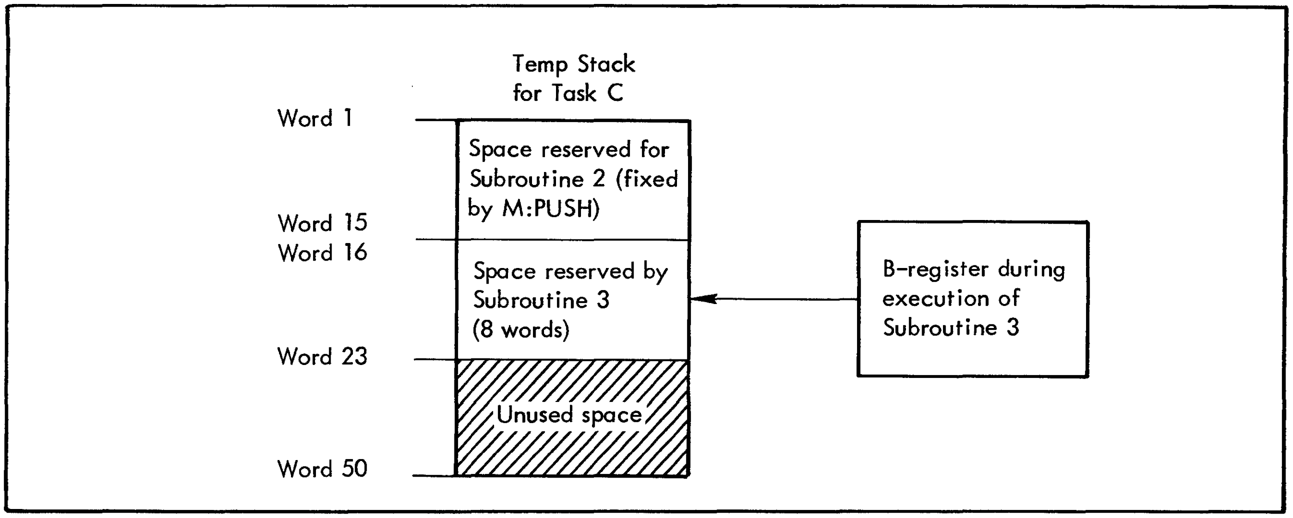

Temp Stack
for Task C
Word 1
Space reserved for Subroutine 2 (fixed by M:PUSH)
Word 15
Word 16
Space reserved by Subroutine 3 (8 words)
B-register during execution of Subroutine 3
Word 23
Unused space
Word 50

Figure 52. Temp Stack Usage Example

In summary, then:

- Tasks that call reentrant subroutines must reserve adequate temp stack space and get this space pointed to from the TCB, via a call to M:SAVE.
- Subroutines designed to be reentrant must call M:PUSH (or M:RES) to set the B-register and reserve space, must use base addressing to reference this space, and must call M:POP to release this space.

Procedures for assembly language calls to reentrant FORTRAN Library routines are discussed in detail in Chapter 8.

15. HOW TO WRITE AN ASSEMBLY LANGUAGE INTERRUPT HANDLER

The sample assembly language program example illustrated in Figure 53 and 54 will output the message

KEY-IN THE DATE AND TIME BEFORE PROCESSING ANY JOBS

on the OC device when any of the following conditions are encountered:

- Each time the system is booted in from a RAD or tape.
- Whenever a trigger is initiated by either a C: control command or C: operator key-in.
- Whenever a !name processor command is encountered, preceded by an FG operator key-in.

Figure 53 shows the source listing and required control commands and Figure 54 shows the assembled program. The message

OLERR TA

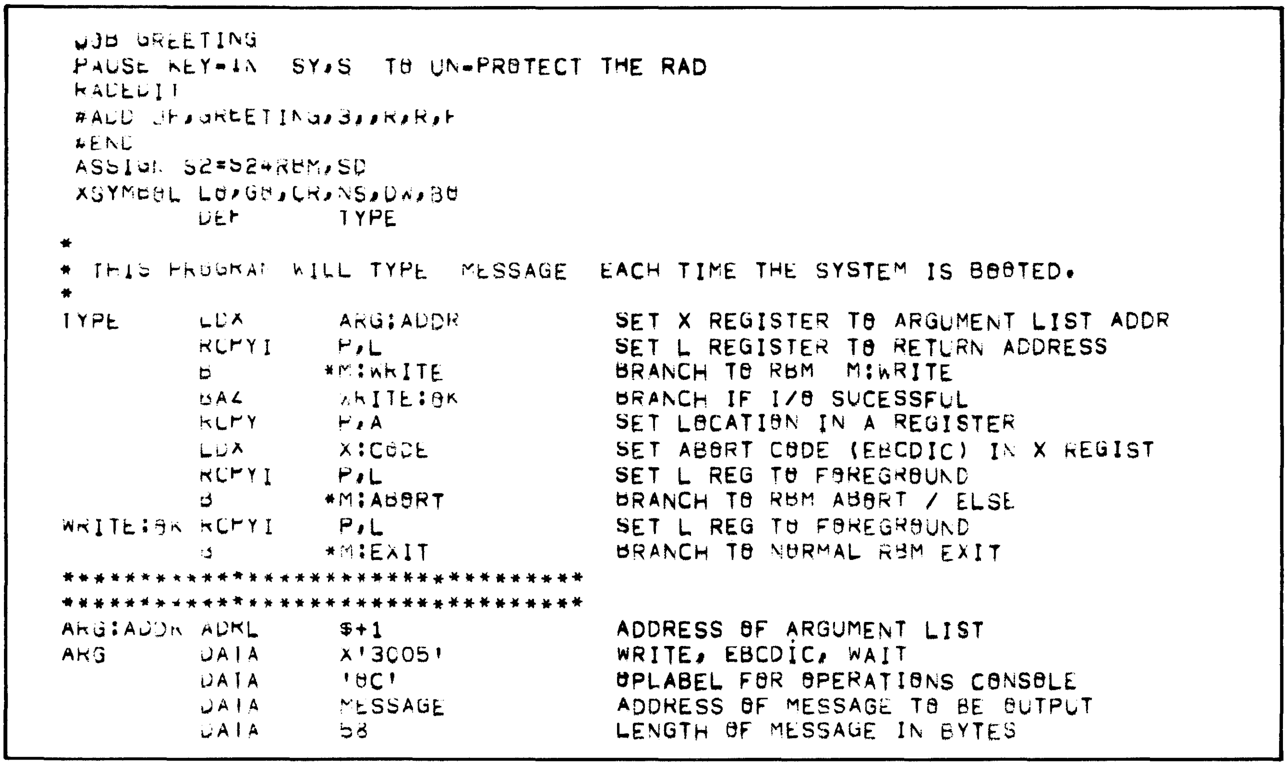
that appears following !OLOAD in Figure 53 is expected and is to be ignored.

At execution the program is loaded into memory and is armed/disarmed, enabled/disabled, and/or triggered in accordance with the specification in the Task Control Block (TCB). The TCB in this program arms and enables the interrupt and triggers when the program is loaded in.

When the message

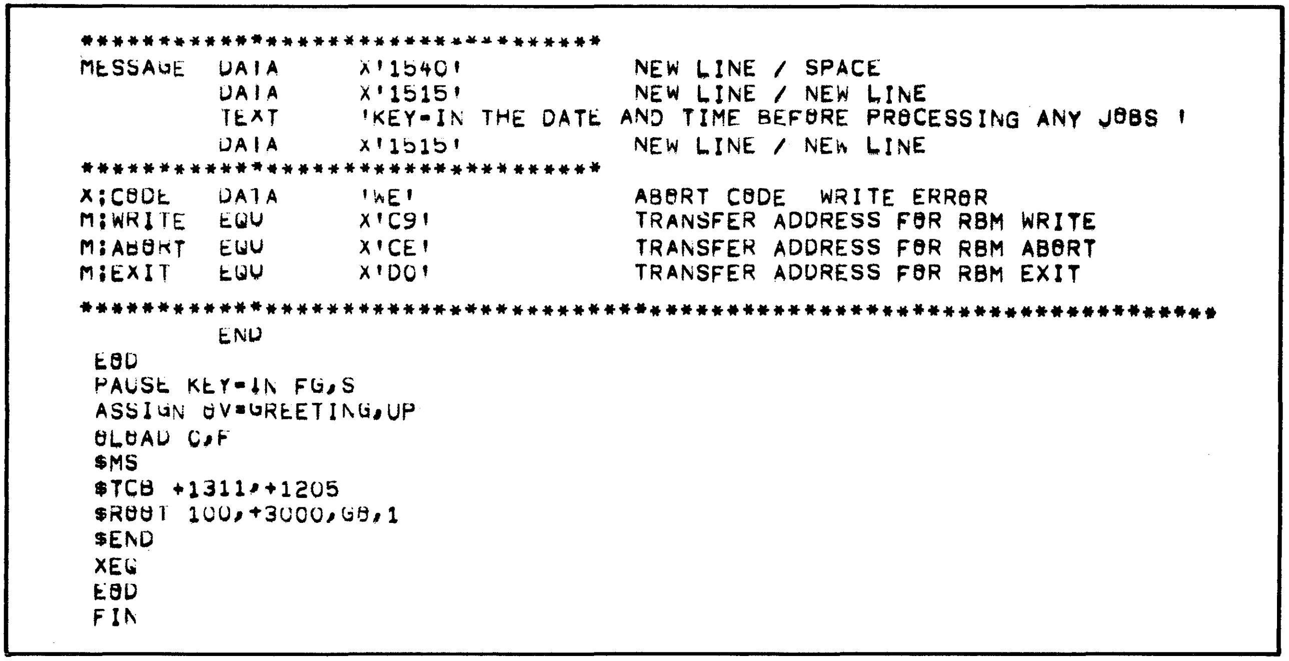
KEY-IN THE DATE AND TIME BEFORE PROCESSING ANY JOBS

appears on the OC device, the program has been loaded correctly.



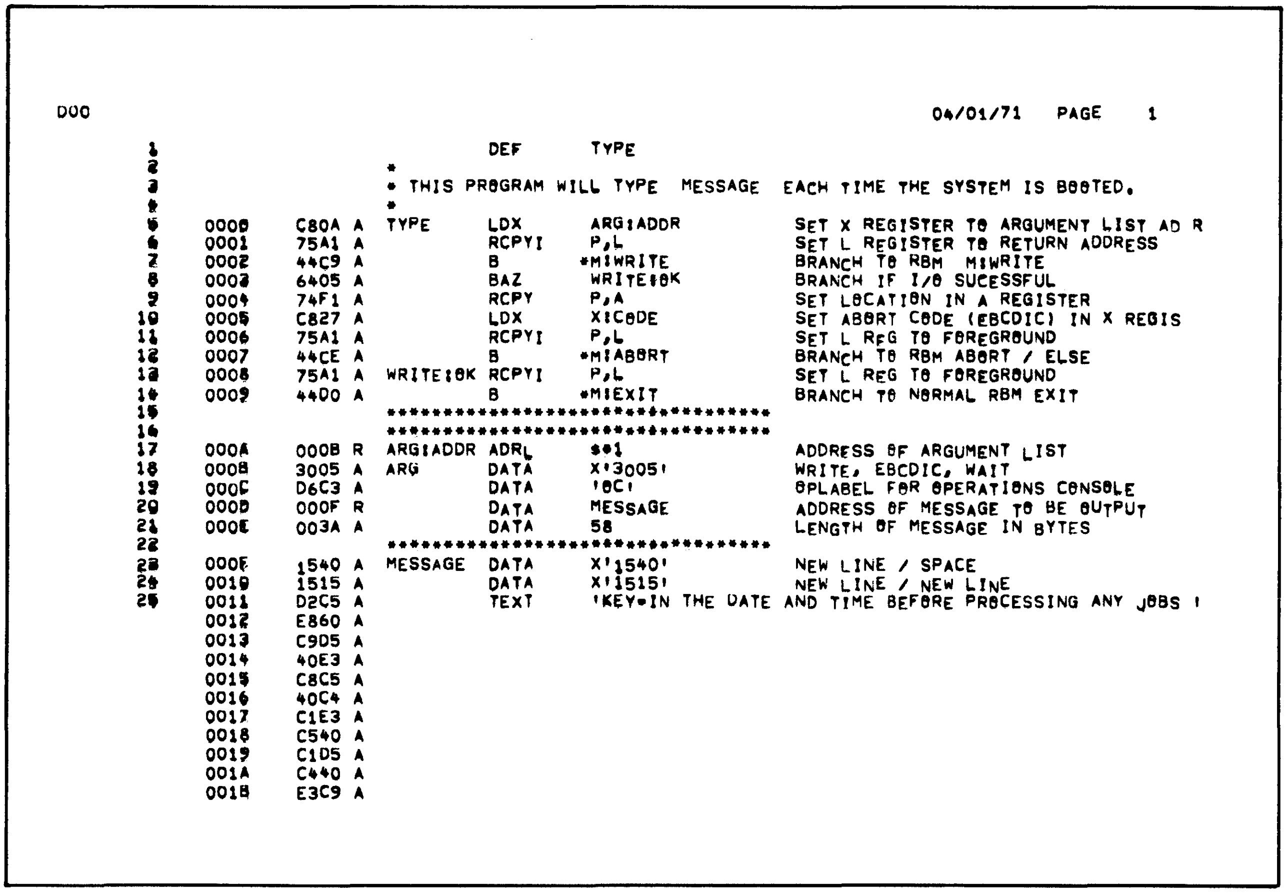
```
 !JOB GREETING
 PAUSE KEY-IN  SY,S  TO UN-PROTECT THE RAD
 RADEDIT
 #ADD JF,GREETING,3,,R,R,F
 #END
 ASSIGN S2=S2+RBM,SD
 XSYMBOL LO,GO,CR,NS,DW,BO
          DEF       TYPE
*
* THIS PROGRAM WILL TYPE  MESSAGE  EACH TIME THE SYSTEM IS BOOTED.
*
TYPE      LDX       ARG:ADDR              SET X REGISTER TO ARGUMENT LIST ADDR
          RCPYI     P,L                   SET L REGISTER TO RETURN ADDRESS
          B        *M:WRITE               BRANCH TO RBM  M:WRITE
          BAZ       WRITE:OK              BRANCH IF I/O SUCESSFUL
          RCPY      P,A                   SET LOCATION IN A REGISTER
          LDX       X:CODE                SET ABORT CODE (EBCDIC) IN X REGIST
          RCPYI     P,L                   SET L REG TO FOREGROUND
          B        *M:ABORT               BRANCH TO RBM ABORT / ELSE
WRITE:OK  RCPYI     P,L                   SET L REG TO FOREGROUND
          B        *M:EXIT                BRANCH TO NORMAL RBM EXIT
*********************************
*********************************
ARG:ADDR  ADRL      $+1                   ADDRESS OF ARGUMENT LIST
ARG       DATA      X'3005'               WRITE, EBCDIC, WAIT
          DATA      'OC'                  OPLABEL FOR OPERATIONS CONSOLE
          DATA      MESSAGE               ADDRESS OF MESSAGE TO BE OUTPUT
          DATA      58                    LENGTH OF MESSAGE IN BYTES
```

Figure 53. Interrupt Handler Source Listing



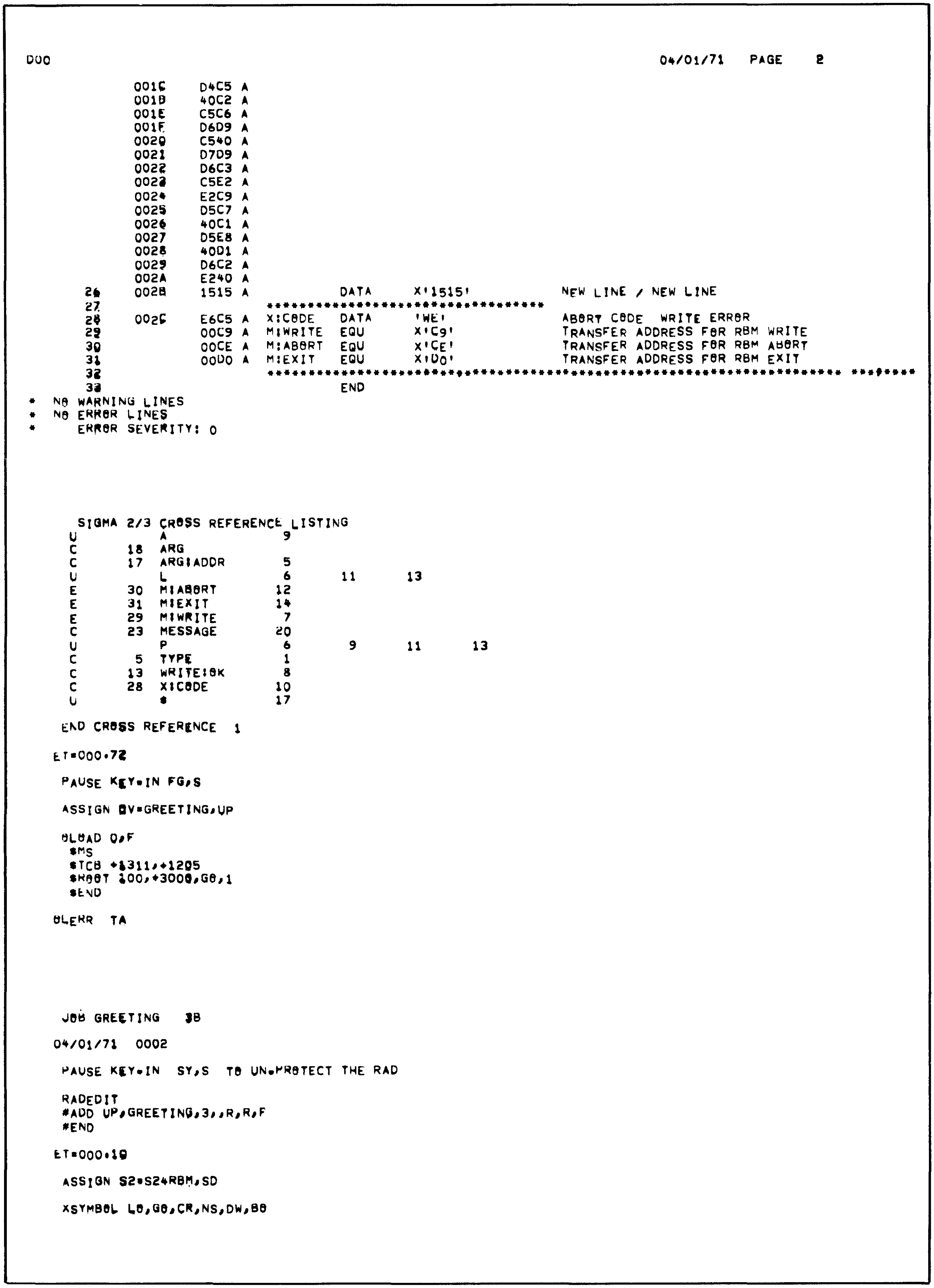
```
*********************************
MESSAGE  DATA      X'1540'             NEW LINE / SPACE
         DATA      X'1515'             NEW LINE / NEW LINE
         TEXT      'KEY-IN THE DATE AND TIME BEFORE PROCESSING ANY JOBS '
         DATA      X'1515'             NEW LINE / NEW LINE
*********************************
X;CODE   DATA      'WE'                ABORT CODE  WRITE ERROR
M;WRITE  EQU       X'C9'               TRANSFER ADDRESS FOR RBM WRITE
M;ABORT  EQU       X'CE'               TRANSFER ADDRESS FOR RBM ABORT
M;EXIT   EQU       X'DO'               TRANSFER ADDRESS FOR RBM EXIT
************************************************************************
         END
 EOD
 PAUSE KEY-IN FG,S
 ASSIGN OV=GREETING,UP
 OLOAD C,F
 $MS
 $TCB +1311,+1205
 $ROOT 100,+3000,GO,1
 $END
 XEQ
 EOD
 FIN
```

Figure 53. Interrupt Handler Source Listing (cont.)



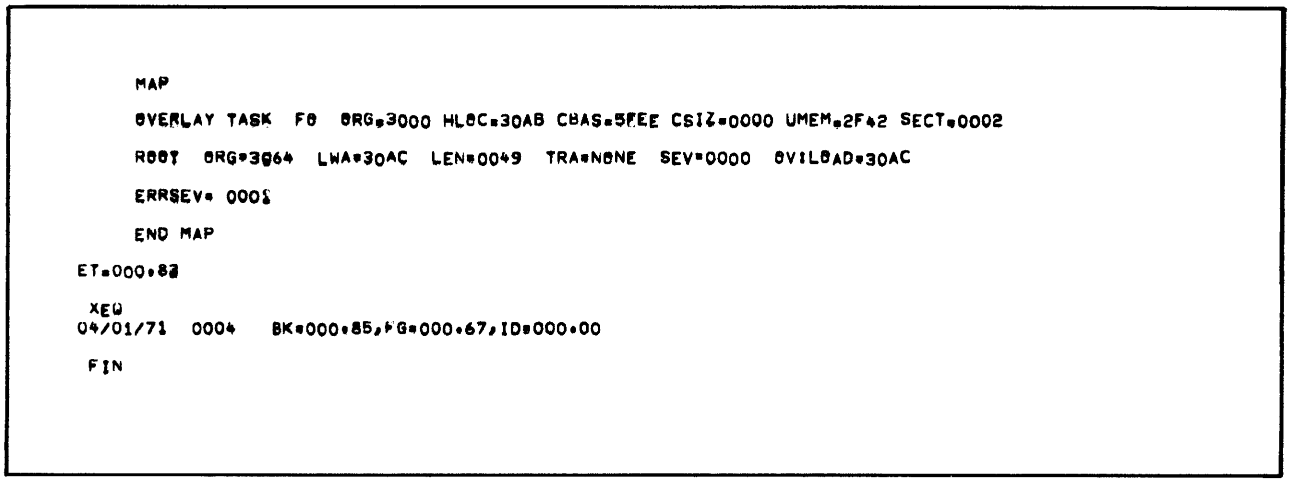
```
DOO                                                                       04/01/71    PAGE    1

                                  DEF       TYPE
                          *
                          * THIS PROGRAM WILL TYPE  MESSAGE  EACH TIME THE SYSTEM IS BOOTED.
                          *
     0000     C80A A  TYPE        LDX       ARG;ADDR            SET X REGISTER TO ARGUMENT LIST AD R
     0001     75A1 A              RCPYI     P,L                 SET L REGISTER TO RETURN ADDRESS
     0002     44C9 A              B         *M;WRITE            BRANCH TO RBM  M;WRITE
     0003     6405 A              BAZ       WRITE;OK            BRANCH IF I/O SUCESSFUL
     0004     74F1 A              RCPY      P,A                 SET LOCATION IN A REGISTER
     0005     C827 A              LDX       X;CODE              SET ABORT CODE (EBCDIC) IN X REGIS
     0006     75A1 A              RCPYI     P,L                 SET L REG TO FOREGROUND
     0007     44CE A              B         *M;ABORT            BRANCH TO RBM ABORT / ELSE
     0008     75A1 A  WRITE;OK    RCPYI     P,L                 SET L REG TO FOREGROUND
     0009     44D0 A              B         *M;EXIT             BRANCH TO NORMAL RBM EXIT
                      *********************************
                      *********************************
     000A     000B R  ARG;ADDR    ADRL      $+1                 ADDRESS OF ARGUMENT LIST
     000B     3005 A  ARG         DATA      X'3005'             WRITE, EBCDIC, WAIT
     000C     D6C3 A              DATA      'OC'                OPLABEL FOR OPERATIONS CONSOLE
     000D     000F R              DATA      MESSAGE             ADDRESS OF MESSAGE TO BE OUTPUT
     000E     003A A              DATA      58                  LENGTH OF MESSAGE IN BYTES
                      *********************************
     000F     1540 A  MESSAGE     DATA      X'1540'             NEW LINE / SPACE
     0010     1515 A              DATA      X'1515'             NEW LINE / NEW LINE
     0011     D2C5 A              TEXT      'KEY-IN THE DATE AND TIME BEFORE PROCESSING ANY JOBS '
     0012     E860 A
     0013     C9D5 A
     0014     40E3 A
     0015     C8C5 A
     0016     40C4 A
     0017     C1E3 A
     0018     C540 A
     0019     C1D5 A
     001A     C440 A
     001B     E3C9 A
```

Figure 54. Interrupt Handler Assembly Listing



```
DOO                                                                          04/01/71    PAGE    2
             001C    D4C5 A
             001D    40C2 A
             001E    C5C6 A
             001F    D6D9 A
             0020    C540 A
             0021    D7D9 A
             0022    D6C3 A
             0023    C5E2 A
             0024    E2C9 A
             0025    D5C7 A
             0026    40C1 A
             0027    D5E8 A
             0028    40D1 A
             0029    D6C2 A
             002A    E240 A
      26     002B    1515 A            DATA     X'1515'            NEW LINE / NEW LINE
      27                      *********************************
      28     002C    E6C5 A   X:CØDE   DATA     'WE'               ABØRT CØDE  WRITE ERRØR
      29             00C9 A   M:WRITE  EQU      X'C9'              TRANSFER ADDRESS FØR RBM WRITE
      30             00CE A   M:ABØRT  EQU      X'CE'              TRANSFER ADDRESS FØR RBM ABØRT
      31             00D0 A   M:EXIT   EQU      X'D0'              TRANSFER ADDRESS FØR RBM EXIT
      32                      ********************************************************************** *******
      33                               END
*   NØ WARNING LINES
*   NØ ERRØR LINES
*      ERRØR SEVERITY: 0



    SIGMA 2/3 CRØSS REFERENCE LISTING
   U           A             9
   C       18  ARG
   C       17  ARG:ADDR      5
   U           L             6       11       13
   E       30  M:ABØRT      12
   E       31  M:EXIT       14
   E       29  M:WRITE       7
   C       23  MESSAGE      20
   U           P             6        9       11       13
   C        5  TYPE          1
   C       13  WRITE:ØK      8
   C       28  X:CØDE       10
   U           $            17

  END CRØSS REFERENCE  1

 ET=000.72

  PAUSE KEY-IN FG,S

  ASSIGN ØV=GREETING,UP

  ØLØAD 0,F
   $MS
   $TCB +1311,+1205
   $RØØT 100,+3000,GØ,1
   $END

 ØLERR  TA




  JØB GREETING   3B

 04/01/71  0002

  PAUSE KEY-IN  SY,S  TØ UN-PRØTECT THE RAD

  RADEDIT
  #ADD UP,GREETING,3,,R,R,F
  #END

 ET=000.10

  ASSIGN S2=S24RBM,SD

  XSYMBØL LØ,GØ,CR,NS,DW,BØ
```

Figure 54. Interrupt Handler Assembly Listing (cont.)



```
     MAP

     ØVERLAY TASK  FØ  ØRG=3000 HLØC=30AB CBAS=5FEE CSIZ=0000 UMEM=2F42 SECT=0002

     RØØT  ØRG=3064  LWA=30AC  LEN=0049  TRA=NØNE  SEV=0000  ØV:LØAD=30AC

     ERRSEV= 0001

     END MAP

ET=000.82

 XEQ
04/01/71  0004   BK=000.85,FG=000.67,ID=000.00

 FIN
```

Figure 54. Interrupt Handler Assembly Listing (cont.)

16. HOW TO WRITE AND EXECUTE A REAL-TIME PROGRAM

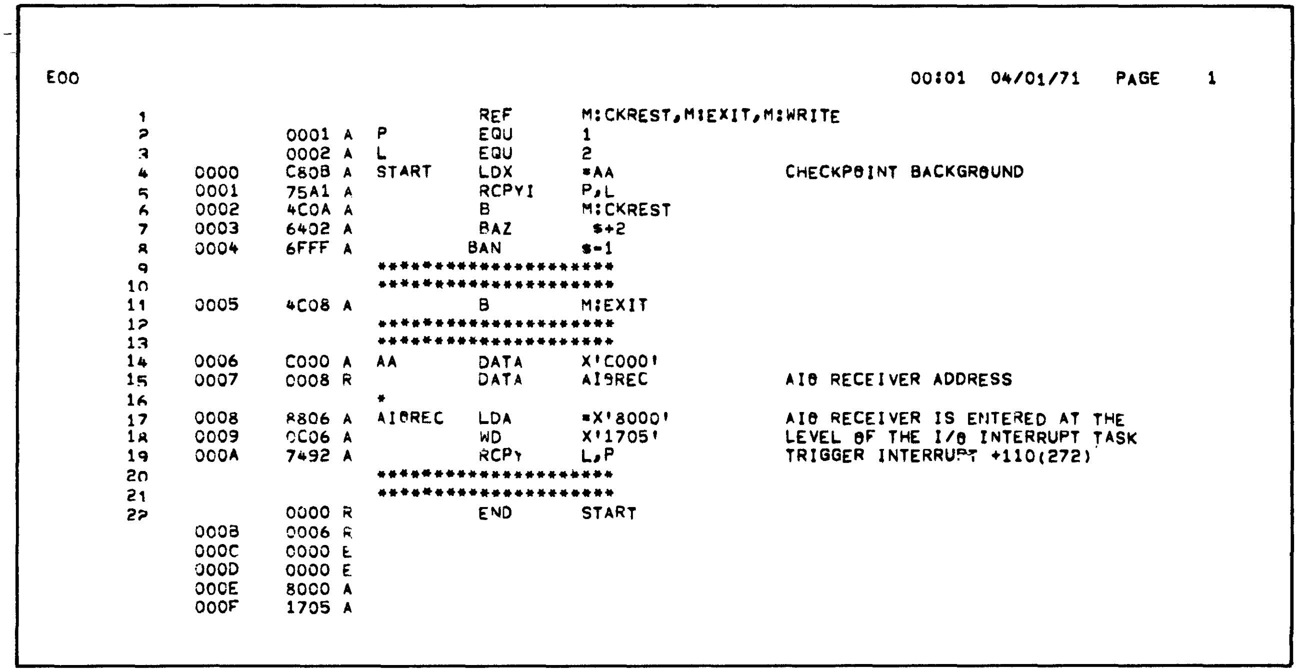
The source listings in Figures 55 and 56 illustrate the interface between two real-time task examples. The first task calls for a checkpoint of the background, specifies the Checkpoint Complete Receiver (via the M:CKREST Monitor service routine and is used similarly to the AIO Receiver), and then exits itself. The task is reentered at channel end. The Checkpoint Complete Receiver then triggers a second, higher priority task to restart the background.

The deck structure given in Figure 57 would load and cause execution of the two real-time tasks illustrated in Figures 55 and 56.

When both tasks have successfully executed, the message

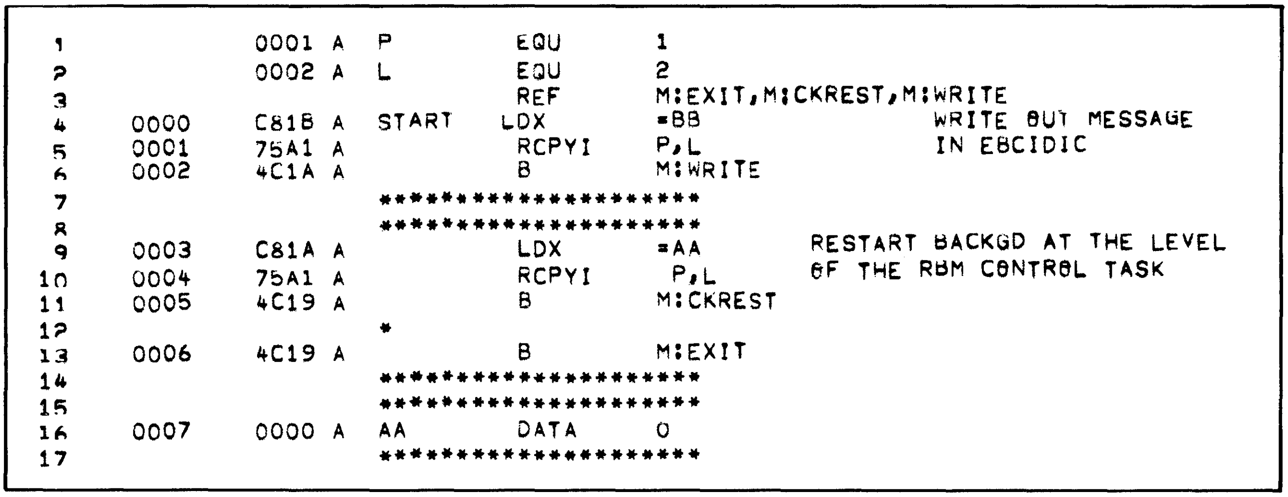
```
CKPOINT SPECIFIES AIO RECEIVER
!!BKG RESTART
```

will be output on the operator's console.



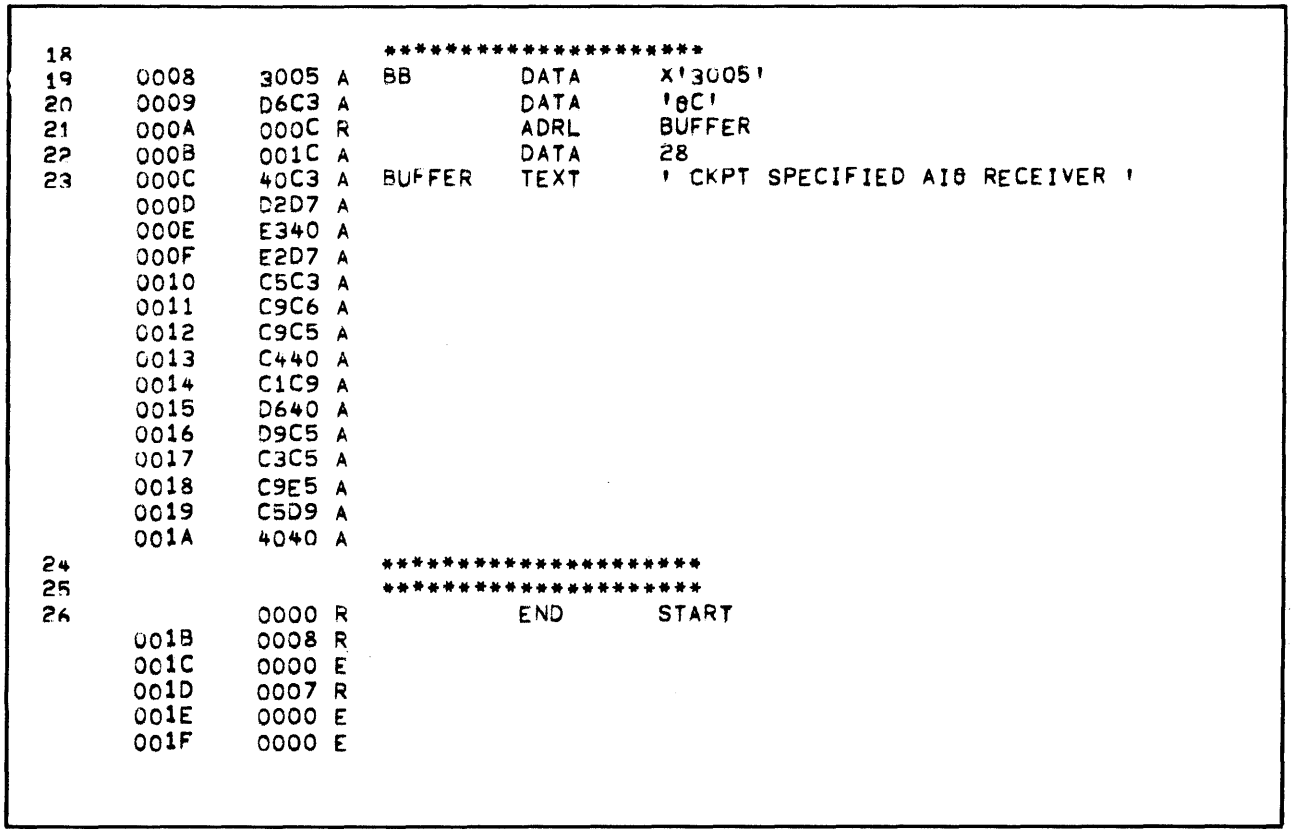
```
E00                                                                              00:01  04/01/71   PAGE    1

                        REF       M:CKREST,M:EXIT,M:WRITE
         0001 A  P      EQU       1
         0002 A  L      EQU       2
  0000   C808 A  START  LDX       =AA                  CHECKPØINT BACKGRØUND
  0001   75A1 A         RCPYI     P,L
  0002   4C0A A         B         M:CKREST
  0003   6402 A         BAZ        $+2
  0004   6FFF A        BAN        $-1
                 ********************
                 ********************
  0005   4C08 A         B         M:EXIT
                 ********************
                 ********************
  0006   C000 A  AA     DATA      X'C000'
  0007   0008 R         DATA      AI9REC               AIØ RECEIVER ADDRESS
                 *
  0008   8806 A  AIØREC LDA       =X'8000'             AIØ RECEIVER IS ENTERED AT THE
  0009   0C06 A         WD        X'1705'              LEVEL ØF THE I/Ø INTERRUPT TASK
  000A   7492 A         RCPY      L,P                  TRIGGER INTERRUPT +110(272)
                 ********************
                 ********************
         0000 R         END       START
  000B   0006 R
  000C   0000 E
  000D   0000 E
  000E   8000 A
  000F   1705 A
```

Figure 55. Real-Time Task Example, Checkpoint Call and Exit (Task 1)



```
         0001 A  P       EQU      1
         0002 A  L       EQU      2
                         REF      M:EXIT,M:CKREST,M:WRITE
  0000   C81B A  START  LDX       =BB                 WRITE ØUT MESSAGE
  0001   75A1 A          RCPYI    P,L                 IN EBCIDIC
  0002   4C1A A          B        M:WRITE
                 *********************
                 *********************
  0003   C81A A          LDX      =AA          RESTART BACKGD AT THE LEVEL
  0004   75A1 A          RCPYI     P,L         ØF THE RBM CØNTRØL TASK
  0005   4C19 A          B        M:CKREST
                 *
  0006   4C19 A          B        M:EXIT
                 *********************
                 *********************
  0007   0000 A  AA      DATA     0
                 *********************
```

Figure 56. Real-Time Task Example, Restart Background (Task 2)



```
                             *********************
     0008     3005 A   BB         DATA      X'3005'
     0009     D6C3 A              DATA      'OC'
     000A     000C R              ADRL      BUFFER
     000B     001C A              DATA      28
     000C     40C3 A   BUFFER     TEXT      ' CKPT SPECIFIED AIO RECEIVER '
     000D     D2D7 A
     000E     E340 A
     000F     E2D7 A
     0010     C5C3 A
     0011     C9C6 A
     0012     C9C5 A
     0013     C440 A
     0014     C1C9 A
     0015     D640 A
     0016     D9C5 A
     0017     C3C5 A
     0018     C9E5 A
     0019     C5D9 A
     001A     4040 A
                             *********************
                             *********************
              0000 R              END       START
     001B     0008 R
     001C     0000 E
     001D     0007 R
     001E     0000 E
     001F     0000 E
```

Figure 56. Real-Time Task Example, Restart Background (Task 2) (cont.)

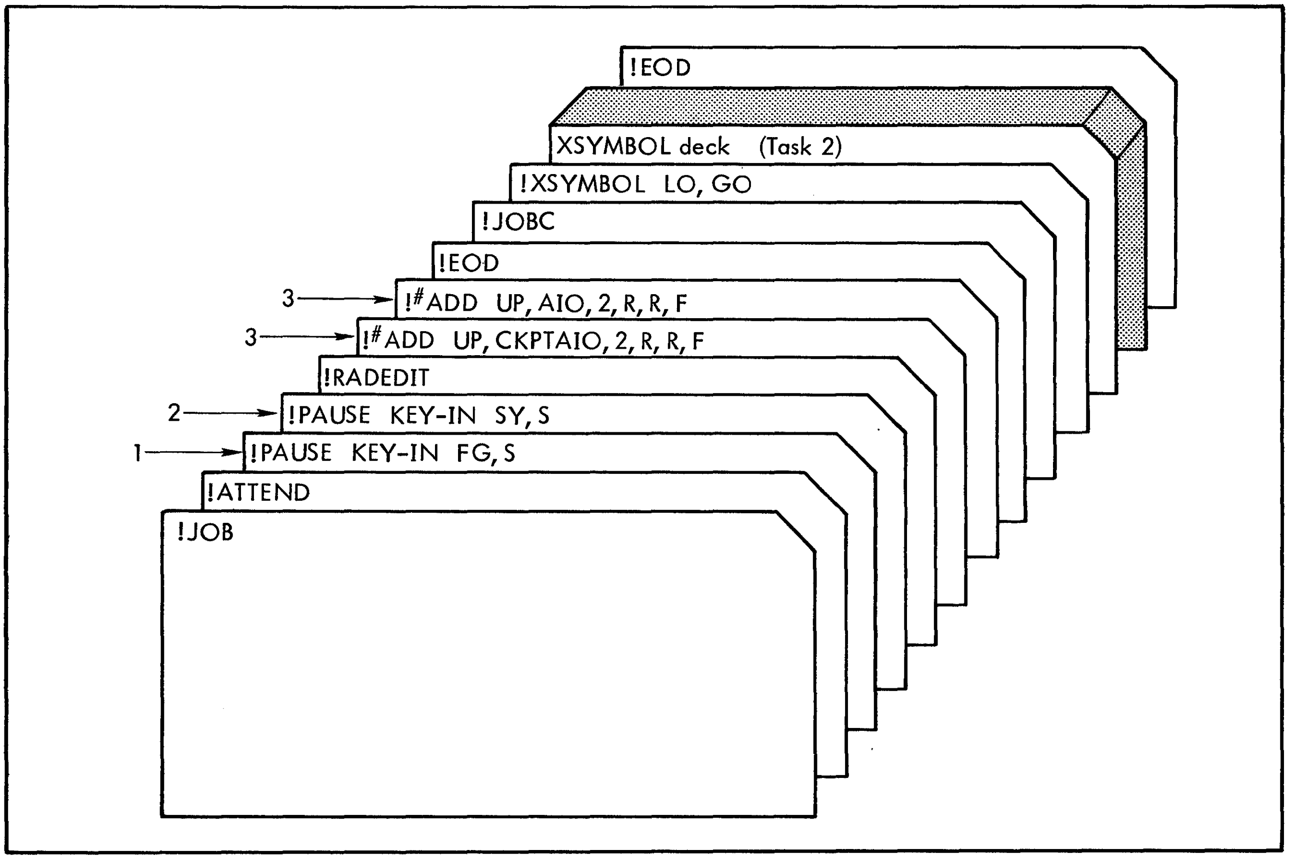
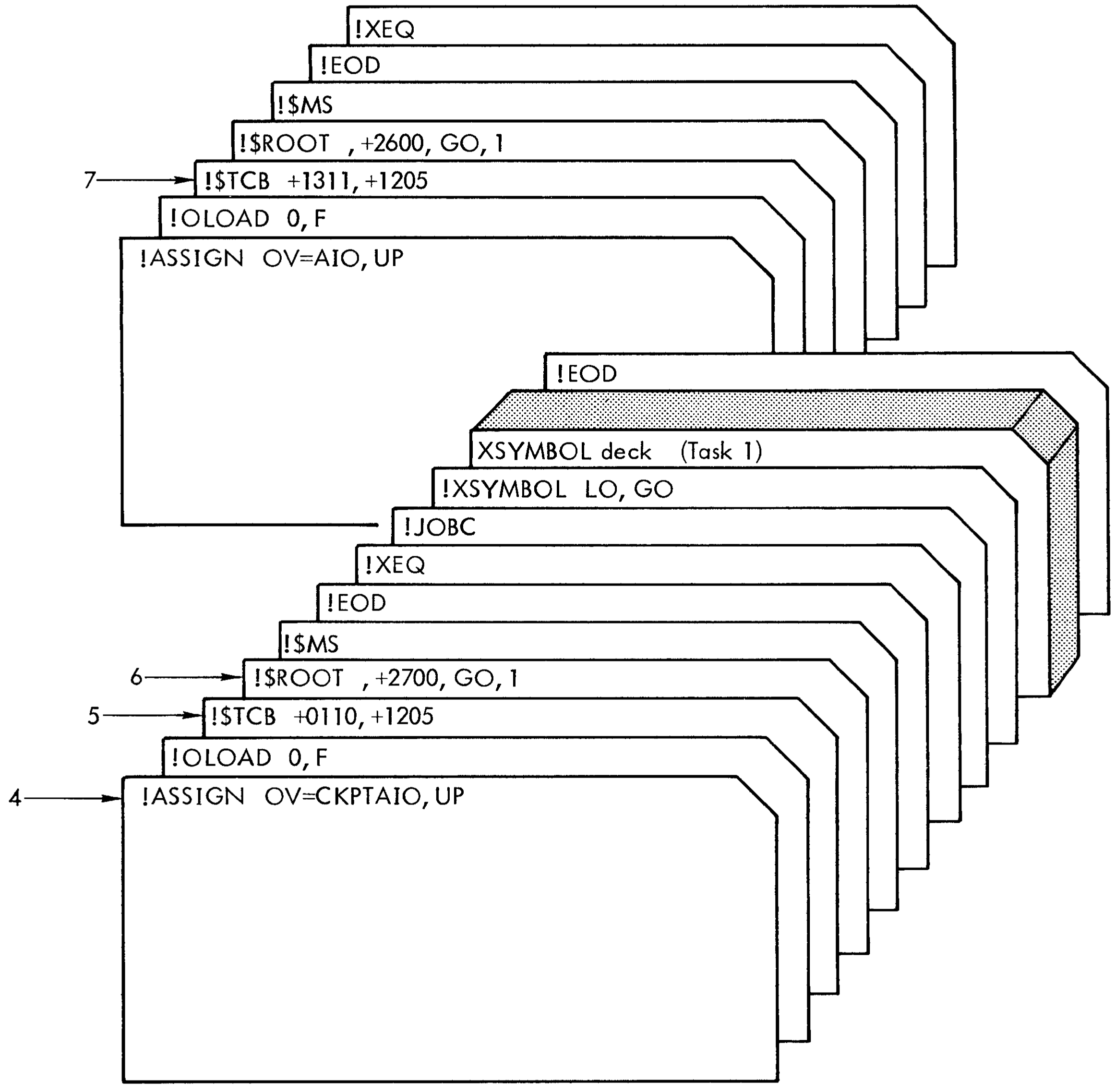


Figure 57. Deck Example For Loading and Executing Real-Time Tasks



where the flagged control commands have the relevance and meaning given below:

1. Permits loading into the foreground.

2. Permits modifications to the RAD area.

3. Creates two files: AIO and CKPTAIO. The files are to be two records (sectors) long, random access, resident foreground, and have RBM write protection.

4. Core image output by the Overlay Loader goes directly on file CKPTAIO.

5. Task is connected to interrupt +0110 (or 272_{10} in decimal) in external group 5. The interrupt is armed and enabled but is not to be triggered when loaded into memory for execution.

6. Starting address is +2700 (temp stack FWA).

7. Task is connected to interrupt +0111 (273_{10} in decimal) in external group 5. The interrupt is armed and enabled, and is to be triggered when loaded into memory for execution.

Figure 57. Deck Example for Loading and Executing Real-Time Tasks (cont.)

When designing and coding your own real-time programs, there is a cardinal rule to be remembered. It was touched upon in previous chapters but is so important and fundemental to RBM design that it deserves added emphasis:

- Each and every foreground task must be connected to a hardware priority interrupt and therefore will execute if and only if its interrupt level is ACTIVE. In particular, a foreground task must not continue execution if the interrupt level is removed from ACTIVE status for any reason.

The single exception to this rule is during the initialization phase of a foreground program, which is run at the RBM Control Task level. (Exit from initialization must return to the RBM Control Task.) It is assumed that initialization activity is of short duration.

The priority level of user foreground tasks must be above the priority level of the RBM Control Task and below the I/O priority level if any I/O is to be performed by the user task.

17. HOW TO CREATE A FORTRAN REAL-TIME SYSTEM

Using ANS FORTRAN IV, you have the ability to construct a Sigma 2/3 real-time system that may be entirely written in FORTRAN if desired.

REENTRANCY

To effectively use a FORTRAN program in a real-time environment, it is necessary to structure the subprograms so that they are reentrant. The Main program and TASK Main programs are, as noted elsewhere, not reentrant; however, any subroutine that may be reached from the Main program and a TASK Main program or two TASK Main programs must be reentrant if the system is to function properly. As indicated in Chapters 13 and 15, Sigma 2/3 programs achieve reentrancy through separation of program and data and the use of a dynamic temp stack allocated by the Overly Loader.

The standard object code output by the ANS FORTRAN compiler is designed so that it may be transformed into reentrant subprograms. Such a transformation is achieved through the following requirements:

- The subprogram must use M:RES, M:MPUSH, M:PUSHC, or M:PUSHK for its storage allocation.
- The temp stack must be allocated at the very end of the subprogram.
- The temp stack must not contain any preset data.
- The program area must not be modified during execution.

When made reentrant, a subprogram is set so that it uses the dynamic temp (see the "Public Library FORTRAN Routines" subsection later in this chapter).

TASKS

The key to the generation of a real-time FORTRAN system is the TASK Main program, which is a Main program having a TASK statement as its first statement. The TASK statement provides a means of naming (other than with F:MAIN) a Main program so that it may be used by the CONNECT subroutine. Thus, TASK Main programs are the interrupt entry points used in constructing a real-time FORTRAN system having more than one entry point. Note that tasks themselves are not reentrant; however, they provide temporary space to any reentrant subprograms and to Monitor service routines.

BASIC STRUCTURE

An example of a possible real-time FORTRAN system is shown in the schematic given in Figure 58. In the example, the Main program provides the initialization for the rest of the real-time system. Initial entry would be to the Main program, which would then connect the tasks ALPHA and BETA.

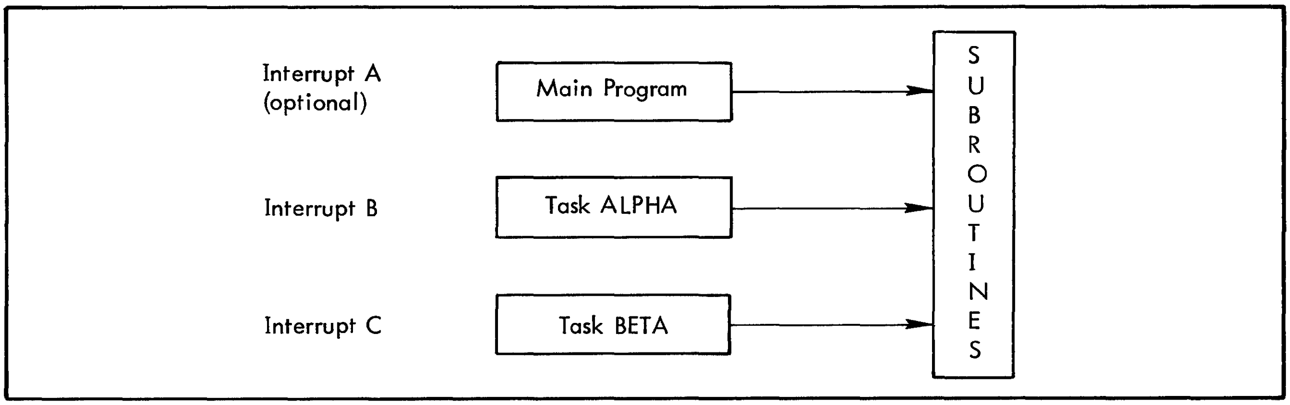


Figure 58. Sample Real-Time FORTRAN System Schematic

If the main program is to be connected to an interrupt, it will be necessary to have the Main program alter a preset variable so as to flag the fact that the later entries are not to do initialization.

Figure 59 shows the deck setup that might be used to construct the schematic sample in Figure 58. The first !$TCB card instructs the Overlay Loader to initiate the root segment through the w_1, w_2 specification.

The !$ROOT card tells the Loader the size of the Main program's temp stack (t_1) and also where to find the Main program and the subroutines BI.

The second !$TCB card sets the Loader to expect a TASK Main program (t_2) which will be task ALPHA, and also causes the Loader to allocate temp stack space for the task. The task module with its !$BLD command (ALPHA deck) _must_ immediately follow the !$TCB card; otherwise the CONNECT subroutine will malfunction. The use of !$TCB commands is further discussed in Chapter 12 of this manual.

INITIALIZATION

In initialization, you have the option of allowing the Overlay Loader to do all the work (thus avoiding the problem of determining whether the Main program entry is really the initial entry), or you can use the CONNECT subroutine in an initialization routine.

If the Overlay Loader is to do the initialization, you must then specify a w_1, w_2 on the !$TCB cards (see the !$TCB command in the Overlay Loader chapter in the RBM Reference Manual, and "How To Create Task Control Blocks" in this manual).

If the Main program is to perform an initialization function, then only the Main program need have a w_1, w_2 field entry.

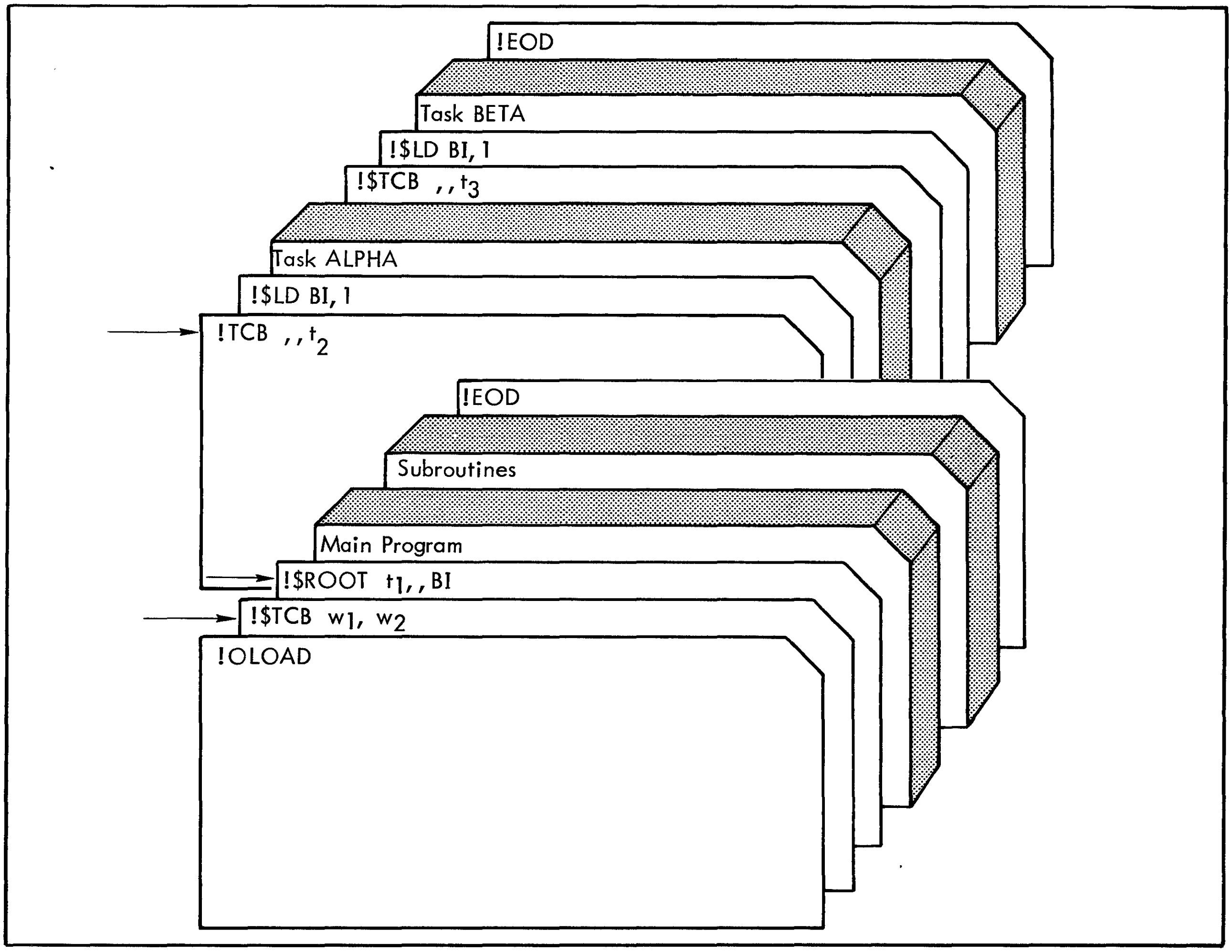


Figure 59. Overlay Loader Controls For Sample Real-Time FORTRAN System

You could use a mixture of these two approaches if desired. A Main program and one or more tasks might be initiated by the Loader. Then, at the occurrence of some specified set of conditions, other tasks could be connected to their respective interrupts.

SUBROUTINE SHARING

Caution must be exercised if subroutines could possibly be shared by two or more interrupts after activation. Where this condition exists, you must be able to ensure that a subroutine is either in the Public Library or that it uses only dynamic storage (directly or indirectly).

PUBLIC LIBRARY FORTRAN ROUTINES

Since all routines in the FORTRAN Library fulfill the requirements for conversion to a reentrant subprogram, it is possible to convert a FORTRAN subroutine into a Public Library routine. As previously stated, a routine to be converted must use M:RES, M:PUSH, M:PUSHC, or M:PUSHK for storage allocation, and the static temp stack must be empty and allocated at the end of the program.

With these conditions fulfilled, the Overlay Loader can be used to convert a FORTRAN routine to a Public Library version. In general, the conversion involves altering the storage allocation calling sequence so that it requests dynamic temp and by stripping the static temp stack from the end of the program. The chapter "How To Write Reentrant Subroutines In Assembly Language" in this Manual and the Overlay Loader chapter in the RBM Reference Manual give more specific details.

18. HOW TO DEBUG ASSEMBLY LANGUAGE PROGRAMS

The most useful features of the RBM Debug package are conditional dumps and the capability to insert code. Both of these features require a region of memory that we will call the "insertion block".

HOW TO DEFINE AN INSERTION BLOCK

The insertion block is defined with the Debug command:

```
D start, end
```

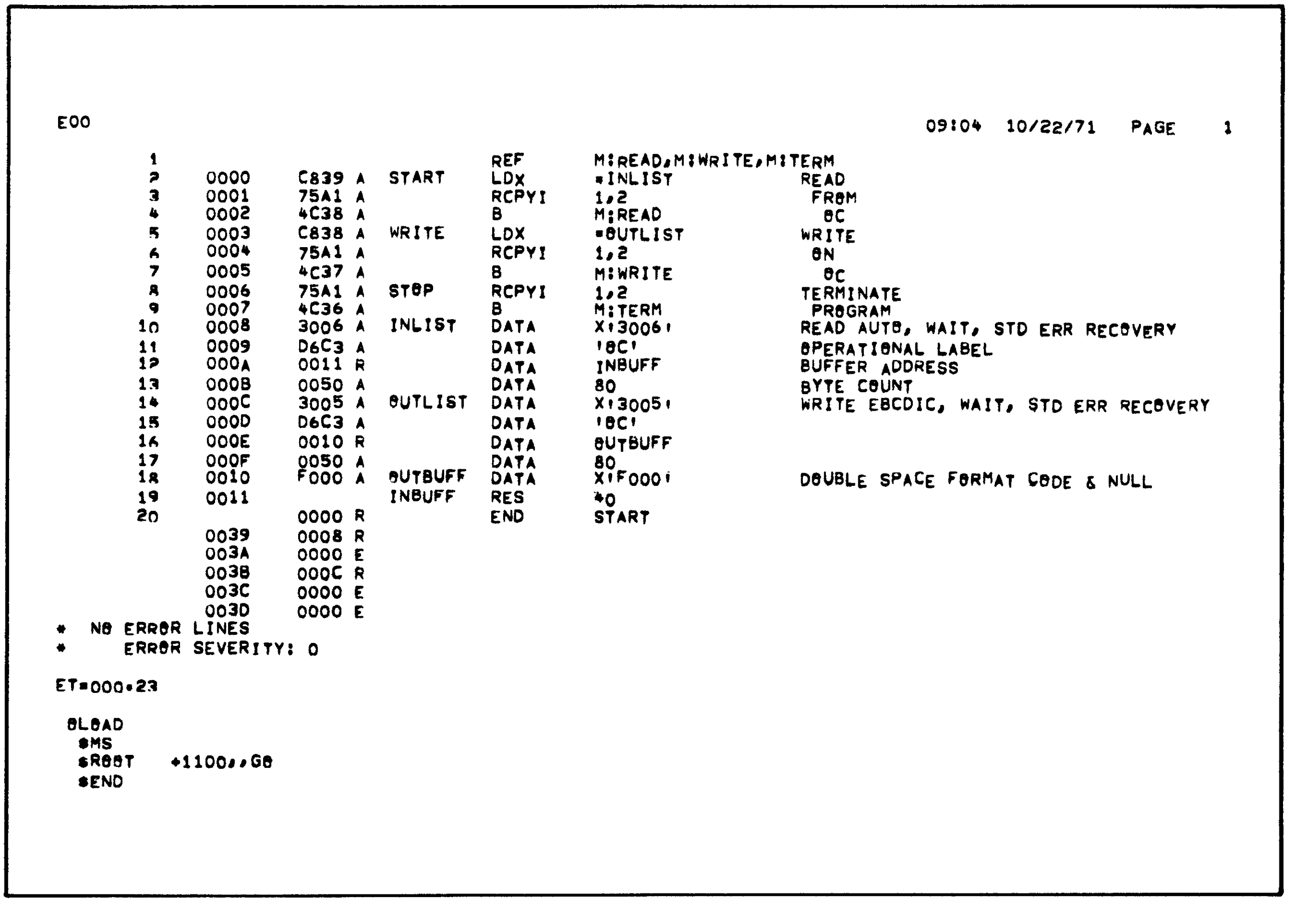
and must be given before any code insertions or snapshots may be specified. The most convenient way to define the insertion block limits is to initiate program execution with an !XED control command that will cause the message

```
!!DKEYIN
```

to be output on the OC device after the program is loaded into core. At this point, you can type in the insertion block definition, type in the conditional snapshots and/or code insertions, and then begin execution.

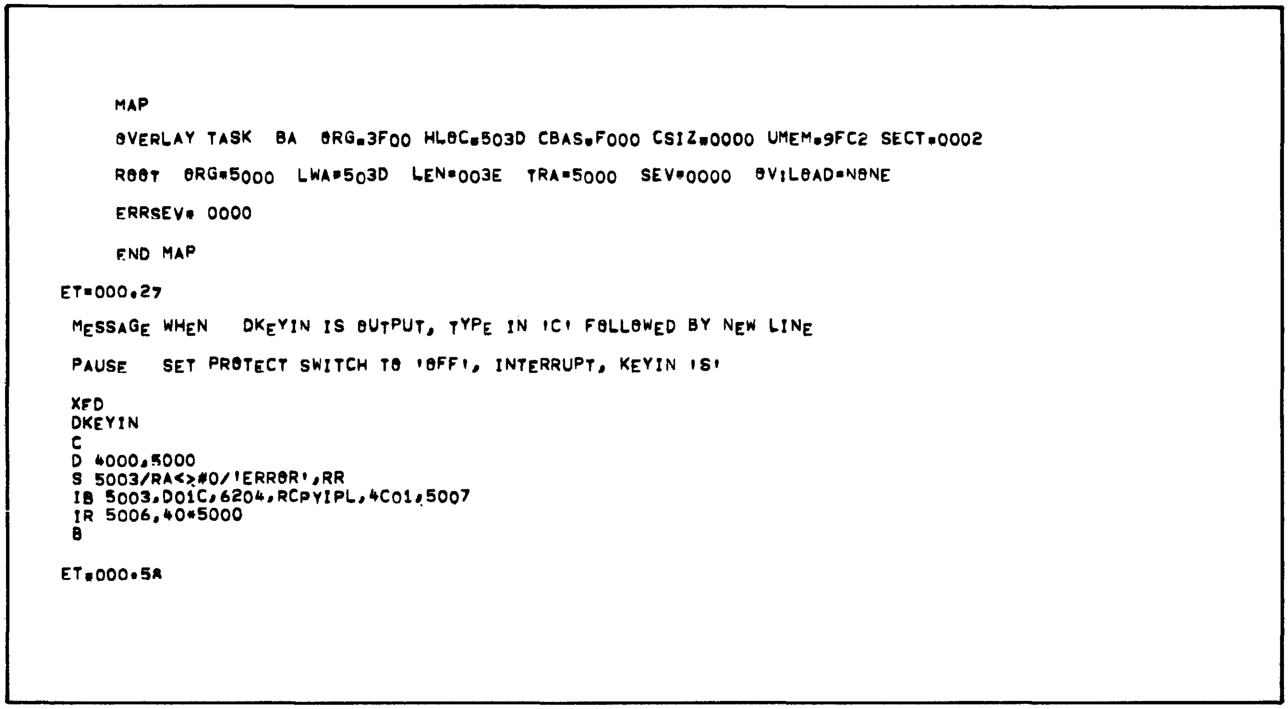
HOW TO INSERT SNAPSHOTS AND CODE

The listing in Figure 60 is an example of a background program using a conditional snapshot and two code insertions.



```
EOO                                                                       09:04  10/22/71   PAGE    1
                                   REF     M:READ,M:WRITE,M:TERM
           0000    C839 A  START   LDX     =INLIST           READ
           0001    75A1 A          RCPYI   1,2                FROM
           0002    4C38 A          B       M:READ              OC
           0003    C838 A  WRITE   LDX     =OUTLIST          WRITE
           0004    75A1 A          RCPYI   1,2                ON
           0005    4C37 A          B       M:WRITE             OC
           0006    75A1 A  STOP    RCPYI   1,2               TERMINATE
           0007    4C36 A          B       M:TERM             PROGRAM
           0008    3006 A  INLIST  DATA    X'3006'           READ AUTO, WAIT, STD ERR RECOVERY
           0009    D6C3 A          DATA    'OC'              OPERATIONAL LABEL
           000A    0011 R          DATA    INBUFF            BUFFER ADDRESS
           000B    0050 A          DATA    80                BYTE COUNT
           000C    3005 A  OUTLIST DATA    X'3005'           WRITE EBCDIC, WAIT, STD ERR RECOVERY
           000D    D6C3 A          DATA    'OC'
           000E    0010 R          DATA    OUTBUFF
           000F    0050 A          DATA    80
           0010    F000 A  OUTBUFF DATA    X'F000'           DOUBLE SPACE FORMAT CODE & NULL
           0011            INBUFF  RES     40
                   0000 R          END     START
           0039    0008 R
           003A    0000 E
           003B    000C R
           003C    0000 E
           003D    0000 E
*  NO ERROR LINES
*     ERROR SEVERITY: 0

ET=000.23

 OLOAD
  $MS
  $ROOT   +1100,,GO
  $END
```

Figure 60. Example, Background Conditional Snapshot With Two Code Insertions



```
     MAP

     ØVERLAY TASK  BA  ØRG=3F00 HLØC=503D CBAS=F000 CSIZ=0000 UMEM=9FC2 SECT=0002

     RØØT  ØRG=5000  LWA=503D  LEN=003E  TRA=5000  SEV=0000  ØV;LØAD=NØNE

     ERRSEV= 0000

     END MAP

ET=000.27

 MESSAGE WHEN   DKEYIN IS ØUTPUT, TYPE IN 'C' FØLLØWED BY NEW LINE

 PAUSE   SET PRØTECT SWITCH TØ 'ØFF', INTERRUPT, KEYIN 'S'

 XFD
 DKEYIN
 C
 D 4000,5000
 S 5003/RA<>#0/'ERRØR',RR
 IB 5003,D01C,6204,RCPYIPL,4C01,5007
 IR 5006,40*5000
 B

ET=000.58
```

Figure 60. Example, Background Conditional Snapshot With Two Code Insertions (cont.)

The example in Figure 60 reads one 80-character record from the OC device, outputs the same record on the OC device, and then terminates.

The !$ROOT card causes the program to be loaded for execution at X'5000'. Since the beginning of background is at X'3F00' in the system used for this example, the region from X'3F00' to X'3FFF' is used as dynamic temp space for the program.

When the !!DKEYIN message is output, the operator types in "C", which causes Debug to read further commands from the Debug DI device.

The S (snapshot) command tests for register A being equal to the value 0, following the call to M:READ. If A does not equal zero, the message

ERROR

is output on the Debug DO device, followed by a hexadecimal dump of the register contents.

The IB (Insert Before) command inserts a test for an EOF condition (A=3) before the snapshot. The symbolic equivalent of the inserted code is

```
CP          =3

BNC         $+3

B   *$+1    *$+1

DATA        STOP
```

The IR (Insert Replace) command inserts an unconditional branch back to START, following the call to M:WRITE. The symbolic equivalent of the inserted code is

```
B           START
```

HOW TO DEBUG A FOREGROUND PROGRAM

The use of Debug with a foreground program involves the use of a high-level interrupt for use by RBM while the foreground program is active. The background program example given previously in Figure 60 can be made to operate in the foreground as shown in Figure 61.

The !ROOT card shown in Figure 61 causes the program to be loaded so that the label START is at location X'3400'. The start of the program is computed as

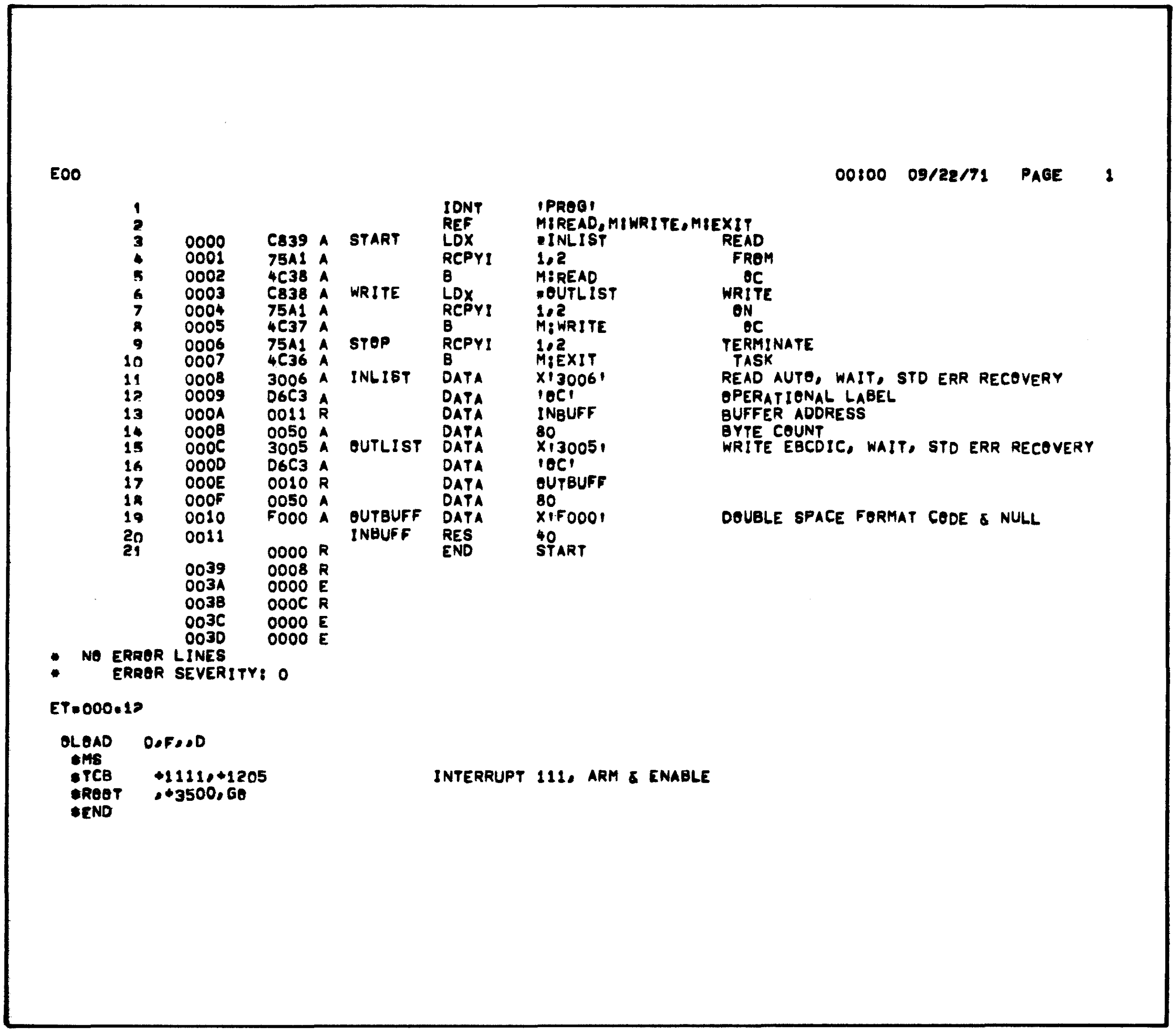
TCB address = exloc (X'3300') + temp (X'E5')

START = TCB address (X'33E5') + TCB size (X'1B')

Since the default temp size is X'50', the !$ROOT command could also be

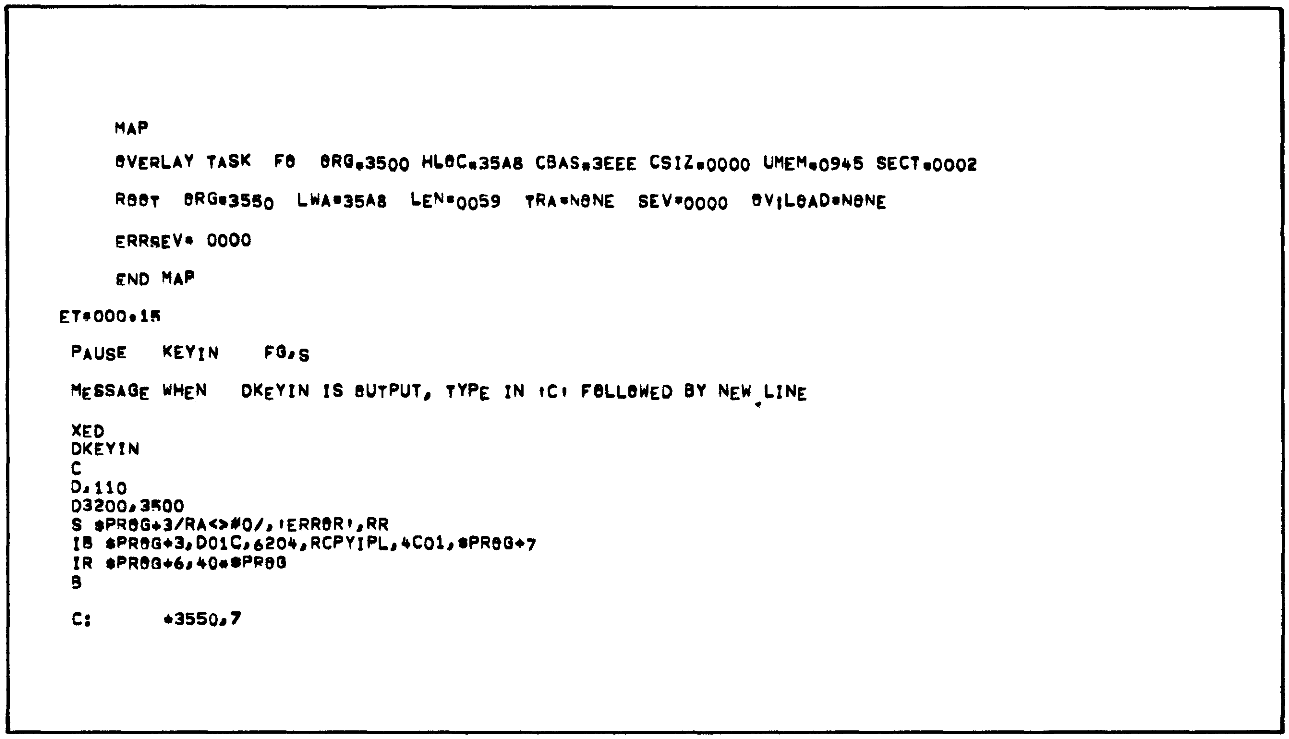
```
!$ROOT ,+3395, GO
```

to cause the label START to be on a convenient boundary. The TCB would still be at X'33E5'.



```
E00                                                                    00:00  09/22/71   PAGE    1

      1                                IDNT     'PRØG'
      2                                REF      M:READ,M:WRITE,M:EXIT
      3   0000     C839 A  START       LDX      *INLIST              READ
      4   0001     75A1 A              RCPYI    1,2                   FRØM
      5   0002     4C38 A              B        M:READ                 ØC
      6   0003     C838 A  WRITE       LDX      *ØUTLIST             WRITE
      7   0004     75A1 A              RCPYI    1,2                   ØN
      8   0005     4C37 A              B        M:WRITE                ØC
      9   0006     75A1 A  STØP        RCPYI    1,2                  TERMINATE
     10   0007     4C36 A              B        M:EXIT                TASK
     11   0008     3006 A  INLIST      DATA     X'3006'              READ AUTØ, WAIT, STD ERR RECØVERY
     12   0009     D6C3 A              DATA     'ØC'                 ØPERATIØNAL LABEL
     13   000A     0011 R              DATA     INBUFF               BUFFER ADDRESS
     14   000B     0050 A              DATA     80                   BYTE CØUNT
     15   000C     3005 A  ØUTLIST     DATA     X'3005'              WRITE EBCDIC, WAIT, STD ERR RECØVERY
     16   000D     D6C3 A              DATA     'ØC'
     17   000E     0010 R              DATA     ØUTBUFF
     18   000F     0050 A              DATA     80
     19   0010     F000 A  ØUTBUFF     DATA     X'F000'              DØUBLE SPACE FØRMAT CØDE & NULL
     20   0011             INBUFF      RES      40
     21            0000 R              END      START
          0039     0008 R
          003A     0000 E
          003B     000C R
          003C     0000 E
          003D     0000 E
*  NØ ERRØR LINES
*     ERRØR SEVERITY: 0

ET=000.12

 ØLØAD    0,F,,D
  •MS
  •TCB     +1111,+1205                    INTERRUPT 111, ARM & ENABLE
  •RØØT    ,+3500,GØ
  •END
```

Figure 61. Foreground Conditional Snapshot With Two Code Insertions



```
     MAP
     ØVERLAY TASK  F0  ØRG=3500 HLØC=35A8 CBAS=3EEE CSIZ=0000 UMEM=0945 SECT=0002
     RØØT  ØRG=3550  LWA=35A8  LEN=0059  TRA=NØNE  SEV=0000  ØV;LØAD=NØNE
     ERRSEV= 0000
     END MAP
ET=000.15
 PAUSE   KEYIN    FG,S
 MESSAGE WHEN   DKEYIN IS ØUTPUT, TYPE IN 'C' FØLLØWED BY NEW LINE
 XED
 DKEYIN
 C
 D,110
 D3200,3500
 S $PRØG+3/RA<>#0/,'ERRØR',RR
 IB $PRØG+3,D01C,6204,RCPYIPL,4C01,$PRØG+7
 IR $PRØG+6,40=$PRØG
 B
 C:       +3550,7
```

Figure 61. Foreground Conditional Snapshot With Two Code Insertions (cont.)

HOW TO USE $NAME AND @NAME

The Debug package provides two methods of referring to program locations by name rather than by hexadecimal value. Both methods involve the use of an arbitrary symbol of up to eight alphanumeric characters preceded by a $ or by an @ sign. Examples:

$PROGRAM

$SEG1

@START

@S

REQUIREMENTS FOR $NAME

1. The source program must include an IDNT statement. Example:

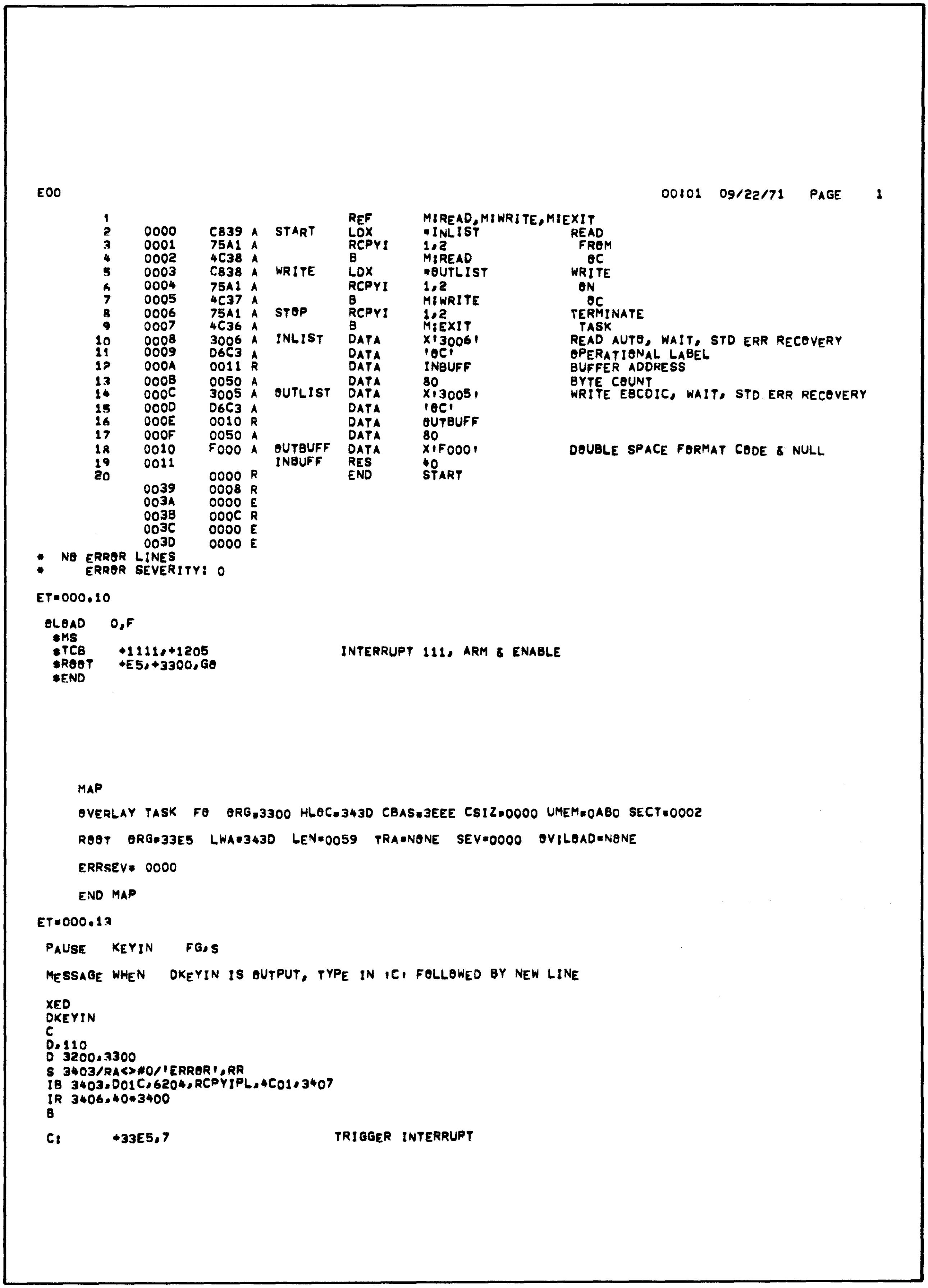
   ```
   IDNT 'PROGRAM'
   ```

2. The !OLOAD command used to load the program must contain the character D as the fourth parameter. Example:

   ```
   !OLOAD 0,F,,D
   ```

3. The Debug insertion block must be large enough to contain a blocking buffer for reading from the operational label ID. The blocking buffer is allocated as the last K:BLOCK words of the insertion blocks, where K:BLOCK words of the insertion blocks have the value 180 or 512. This value is contained in location X'EE' in all RBM systems from Version D00 upward. If snapshots or insertions are to be made, the insertion block must be large enough to contain the blocking buffer and the additional space required for the insertions or snapshots.

The example in Figure 62 shows how the $NAME feature can be used with a foreground program.



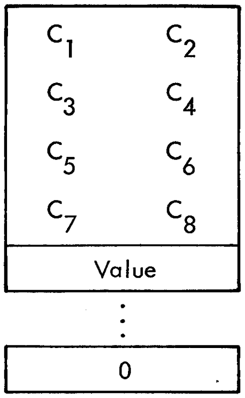
```
E00                                                                     00:01  09/22/71   PAGE    1
        1                               REF      M:READ,M:WRITE,M:EXIT
        2     0000    C839 A  START     LDX      *INLIST              READ
        3     0001    75A1 A            RCPYI    1,2                   FROM
        4     0002    4C38 A            B        M:READ                 OC
        5     0003    C838 A  WRITE     LDX      *OUTLIST             WRITE
        6     0004    75A1 A            RCPYI    1,2                   ON
        7     0005    4C37 A            B        M:WRITE                OC
        8     0006    75A1 A  STOP      RCPYI    1,2                  TERMINATE
        9     0007    4C36 A            B        M:EXIT                TASK
       10     0008    3006 A  INLIST    DATA     X'3006'              READ AUTO, WAIT, STD ERR RECOVERY
       11     0009    D6C3 A            DATA     'OC'                 OPERATIONAL LABEL
       12     000A    0011 R            DATA     INBUFF               BUFFER ADDRESS
       13     000B    0050 A            DATA     80                   BYTE COUNT
       14     000C    3005 A  OUTLIST   DATA     X'3005'              WRITE EBCDIC, WAIT, STD ERR RECOVERY
       15     000D    D6C3 A            DATA     'OC'
       16     000E    0010 R            DATA     OUTBUFF
       17     000F    0050 A            DATA     80
       18     0010    F000 A  OUTBUFF   DATA     X'F000'              DOUBLE SPACE FORMAT CODE & NULL
       19     0011            INBUFF    RES      40
       20             0000 R            END      START
              0039    0008 R
              003A    0000 E
              003B    000C R
              003C    0000 E
              003D    0000 E
*  NO ERROR LINES
*     ERROR SEVERITY: 0

ET=000.10

 OLOAD   O,F
  *MS
  *TCB     +1111,+1205                   INTERRUPT 111, ARM & ENABLE
  *ROOT    +E5,+3300,GO
  *END



     MAP

     OVERLAY TASK  FO  ORG=3300 HLOC=343D CBAS=3EEE CSIZ=0000 UMEM=0ABO SECT=0002

     ROOT  ORG=33E5  LWA=343D  LEN=0059  TRA=NONE  SEV=0000  OV:LOAD=NONE

     ERRSEV= 0000

     END MAP

ET=000.13

 PAUSE   KEYIN    FG,S

 MESSAGE WHEN   DKEYIN IS OUTPUT, TYPE IN 'C' FOLLOWED BY NEW LINE

 XED
 DKEYIN
 C
 D,110
 D 3200,3300
 S 3403/RA<>#0/'ERROR',RR
 IB 3403,D01C,6204,RCPYIPL,4C01,3407
 IR 3406,40=3400
 B

 C:      +33E5,7                        TRIGGER INTERRUPT
```

Figure 62. Foreground Debug Example Using $NAME

REQUIREMENTS FOR @NAME

When it is desirable to be able to refer to locations within a program, the use of a Debug symbol table is required. The symbol table may be assembled into the program or it may be constructed in an unused area of core with Debug Modify commands. Use of the @NAME facility allows you to write snapshots for subroutine entry or exit points, to simulate input values, etc., for programs being assembled and executed within the same job.

The structure of the Debug symbol table is

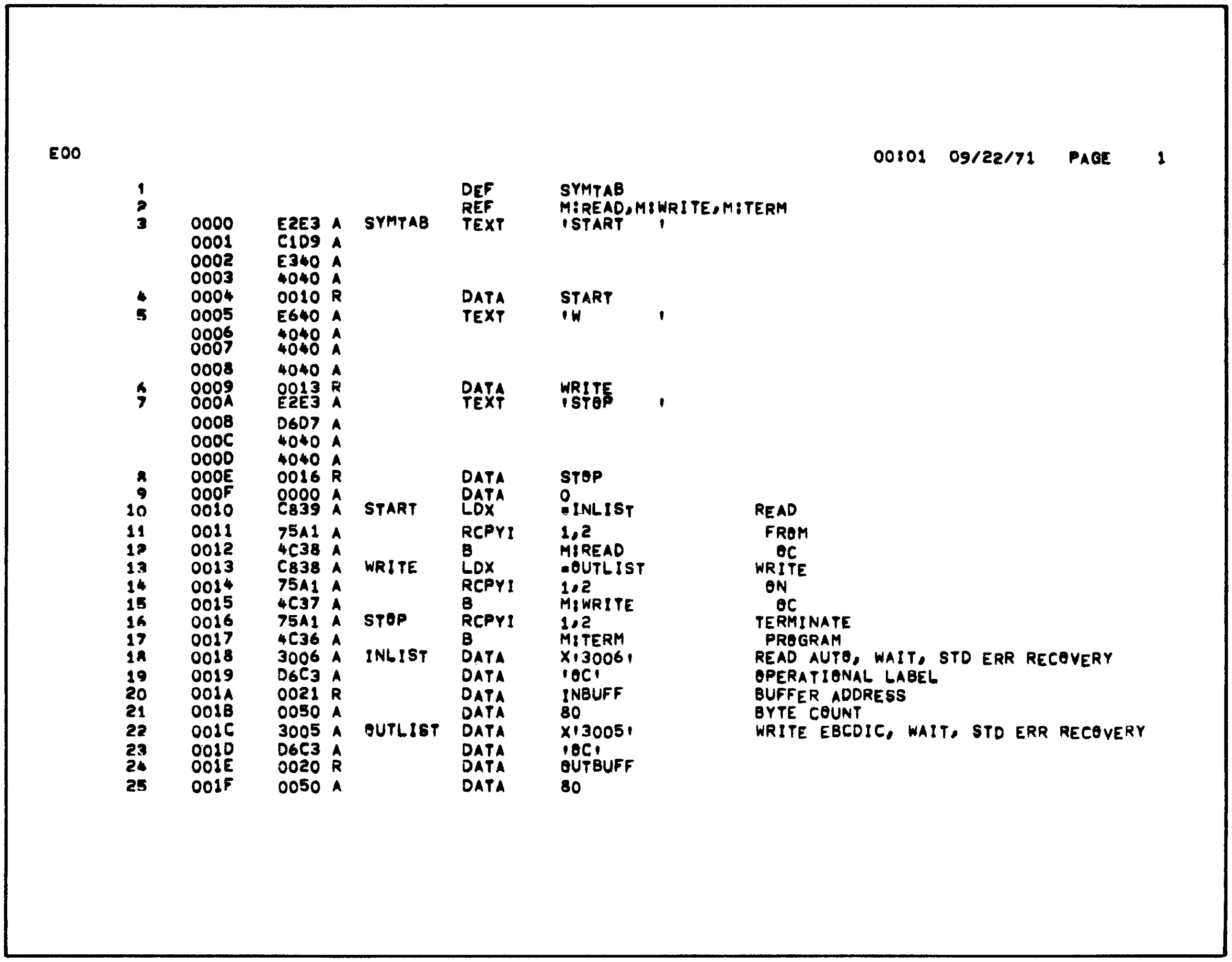


Symbol name, left-justified and padded with blanks to a total of 8 characters.

The location value to be used.

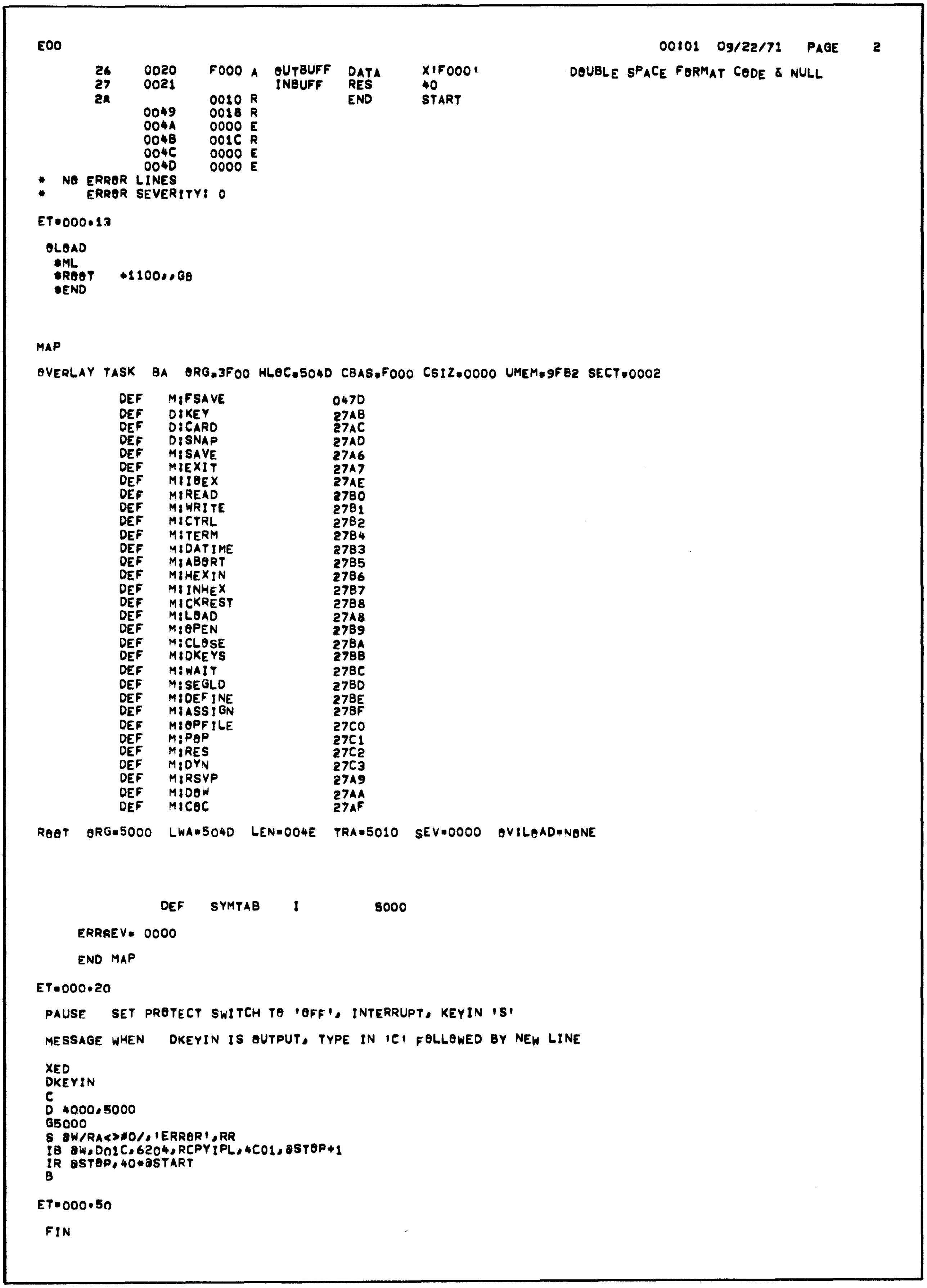
Indicates end of table.

The Debug G command is used to define the start of the symbol table. The listing in Figure 63 is an example of using @NAME with a background program.



```
E00                                                                 00:01  09/22/71   PAGE    1

                            DEF      SYMTAB
                            REF      M:READ,M:WRITE,M:TERM
      0000    E2E3 A SYMTAB  TEXT     'START   '
      0001    C1D9 A
      0002    E340 A
      0003    4040 A
      0004    0010 R         DATA     START
      0005    E640 A         TEXT     'W       '
      0006    4040 A
      0007    4040 A
      0008    4040 A
      0009    0013 R         DATA     WRITE
      000A    E2E3 A         TEXT     'STOP    '
      000B    D6D7 A
      000C    4040 A
      000D    4040 A
      000E    0016 R         DATA     STOP
      000F    0000 A         DATA     0
      0010    C839 A START   LDX      *INLIST              READ
      0011    75A1 A         RCPYI    1,2                   FROM
      0012    4C38 A         B        M:READ                 OC
      0013    C838 A WRITE   LDX      *OUTLIST             WRITE
      0014    75A1 A         RCPYI    1,2                   ON
      0015    4C37 A         B        M:WRITE                OC
      0016    75A1 A STOP    RCPYI    1,2                  TERMINATE
      0017    4C36 A         B        M:TERM                PROGRAM
      0018    3006 A INLIST  DATA     X'3006'              READ AUTO, WAIT, STD ERR RECOVERY
      0019    D6C3 A         DATA     'OC'                 OPERATIONAL LABEL
      001A    0021 R         DATA     INBUFF               BUFFER ADDRESS
      001B    0050 A         DATA     80                   BYTE COUNT
      001C    3005 A OUTLIST DATA     X'3005'              WRITE EBCDIC, WAIT, STD ERR RECOVERY
      001D    D6C3 A         DATA     'OC'
      001E    0020 R         DATA     OUTBUFF
      001F    0050 A         DATA     80
```

Figure 63. Background Debug Example Using @NAME



```
EOO                                                                                   00:01  09/22/71   PAGE     2
       26     0020     F000 A  ØUTBUFF  DATA     X'F000'            DØUBLE SPACE FØRMAT CØDE & NULL
       27     0021             INBUFF   RES      40
       28              0010 R           END      START
              0049     0018 R
              004A     0000 E
              004B     001C R
              004C     0000 E
              004D     0000 E
*  NØ ERRØR LINES
*     ERRØR SEVERITY: 0

ET=000.13

 ØLØAD
  $ML
  $RØØT    +1100,,GØ
  $END



MAP

ØVERLAY TASK  BA  ØRG=3F00 HLØC=504D CBAS=F000 CSIZ=0000 UMEM=9FB2 SECT=0002

            DEF    M:FSAVE              047D
            DEF    D:KEY                27AB
            DEF    D:CARD               27AC
            DEF    D:SNAP               27AD
            DEF    M:SAVE               27A6
            DEF    M:EXIT               27A7
            DEF    M:IØEX               27AE
            DEF    M:READ               27B0
            DEF    M:WRITE              27B1
            DEF    M:CTRL               27B2
            DEF    M:TERM               27B4
            DEF    M:DATIME             27B3
            DEF    M:ABØRT              27B5
            DEF    M:HEXIN              27B6
            DEF    M:INHEX              27B7
            DEF    M:CKREST             27B8
            DEF    M:LØAD               27A8
            DEF    M:ØPEN               27B9
            DEF    M:CLØSE              27BA
            DEF    M:DKEYS              27BB
            DEF    M:WAIT               27BC
            DEF    M:SEGLD              27BD
            DEF    M:DEFINE             27BE
            DEF    M:ASSIGN             27BF
            DEF    M:ØPFILE             27C0
            DEF    M:PØP                27C1
            DEF    M:RES                27C2
            DEF    M:DYN                27C3
            DEF    M:RSVP               27A9
            DEF    M:DØW                27AA
            DEF    M:CØC                27AF

RØØT  ØRG=5000  LWA=504D  LEN=004E  TRA=5010  SEV=0000  ØV:LØAD=NØNE



                  DEF    SYMTAB    I          5000

     ERRSEV= 0000

     END MAP

ET=000.20

 PAUSE   SET PRØTECT SWITCH TØ 'ØFF', INTERRUPT, KEYIN 'S'

 MESSAGE WHEN   DKEYIN IS ØUTPUT, TYPE IN 'C' FØLLØWED BY NEW LINE

 XED
 DKEYIN
 C
 D 4000,5000
 G5000
 S @W/RA<>#0/,'ERRØR',RR
 IB @W,D01C,6204,RCPYIPL,4C01,@STØP+1
 IR @STØP,40+@START
 B

ET=000.50

 FIN
```

Figure 63. Background Debug Example Using @NAME (cont.)

19. HOW TO ASSIGN AND USE DEVICE OPERATIONAL LABELS

Physical devices are normally assigned at SYSGEN to device file numbers (DFNs). However, at installations where relatively large numbers of personnel submit jobs on a somewhat irregular basis, it is highly useful to permanently assign device mnemonic operational labels to hard-to-remember DFNs. This is particularly true when large numbers of Utility jobs are submitted, since the Utility processor works only with operational labels.

For instance, a nonprofessional programmer would find it much easier to use

```
!ASSIGN  BO=M0
```

instead of the standard DFN assignment of

```
!ASSIGN BO=10
```

for a temporary assignment of binary output to a magnetic tape unit.

Or again, the nonprofessional programmer submitting a Utility job could assign (for instance)

```
!ASSIGN UI=CR
```

instead of a "normal" DFN assignment of

```
!ASSIGN UI=3
```

to read in his input, with much less possibility of an incorrect assignment.

When permanently assigning DFNs to operational labels at SYSGEN, the oplabels should convey as much mnemonic information as possible. The following list, however, is suggestive only:

Typical DFN	Physical Device	Suggested Mnemonic Oplabel
1	Keyboard/Printer	KP
2	Line Printer	LP
3	Card Reader	CR
4	Card Punch	CP
5	Paper Tape Reader	PR
6	Paper Tape Reader	PP
10	Magnetic Tape Unit 0	M0
1n	Magnetic Tape Unit n	Mn

The assignment of DFNs to device mnemonic operational labels takes place during the SYSGEN assignment of the background operational labels (see "BCKG. OP. LBL" output message in the SYSGEN Input Options and Parameters table in the System Generation chapter of the RBM Reference Manual).

Assuming that the card reader is assigned DFN3, the permanent SYSGEN assignments would appear as

SI = 3

UI = 3

CR = 3

with corresponding device mnemonic operational labels for other DFN assignments.

20. HOW TO PATCH RBM

RBM can be patched either temporarily or permanently through use of the RBM Hex Corrector. Whether the patch is temporary or permanent is determined by the manner in which the Hex Corrector is activated.

A temporary patch means that the copy of RBM located on the RAD is not altered, and this is achieved by activating the Hex corrector via a !HEX control command or an "H" operator key-in.

A permanent patch means that the RAD copy of RBM is altered, and therefore the changes will remain in effect for all future boots of the system from the RAD. Permanent changes are effected through activating the Hex Corrector by setting DATA switch 1 when RBM is booted in. By using the two methods in conjunction, you can check out patches on a temporary basis and when satisfied that they are correct, make the patches permanent.

When the Hex Corrector has been activated by either one of the two methods described above, it will read records from the CC operational label and write records on the DO operational label. The records read in are either bias or corrector records.

Bias records have the form

$$+\left\{\begin{matrix}\text{bbbb}\\ \text{ID}\left\{\begin{matrix}\text{PA}\\ \text{XX}\end{matrix}\right\}\end{matrix}\right\}[\text{*Comments}]$$

where

bbbb is a hexadecimal number.

PA represents the RBM Patch area defined at SYSGEN.

XX is an RBM overlay identifier (for example, 41 is the Hex Corrector).

Corrector records have the form

aaaa $cccc_0$ $cccc_1$ ... $cccc_i$... $cccc_n$ [*comments]

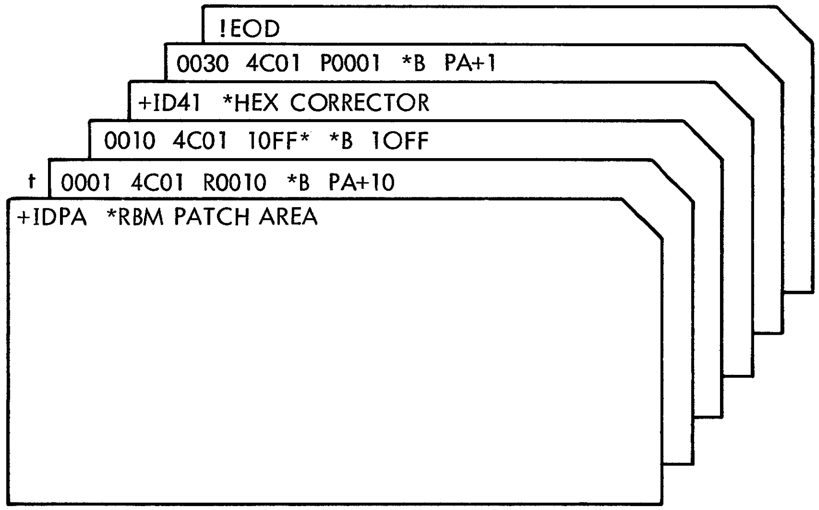
where

aaaa is the hex location where the corrections will go. (If a bias card has been encountered, aaaa will be added to it to determine the location of the patches.)

$cccc_i$ is the hex correction to be inserted at the location aaaa + bias + i. The hex correction $cccc_i$ may also also be of the form R$cccc_i$ which means the value to be stored is $cccc_i$ + bias, or it may be of the form P$cccc_i$ which means the value to be stored is $cccc_i$ + bias of the RBM Patch area.

An !EOD terminates the Hex Corrector's input.

Figure 64 shows sample input to the Hex Corrector.



```
        !EOD
      0030  4C01  P0001  *B  PA+1
    +ID41  *HEX CORRECTOR
  0010  4C01  10FF*  *B  1OFF
† 0001  4C01  R0010  *B  PA+10
+IDPA  *RBM PATCH AREA
```

[†] The first and last cells of the RBM Patch area should not be used for corrections, since the first contains the length of the Patch area and the last contains the number of temporary RBM overlay patches. Each temporary overlay patch takes three Patch area words (taken from the top of the Patch area down).

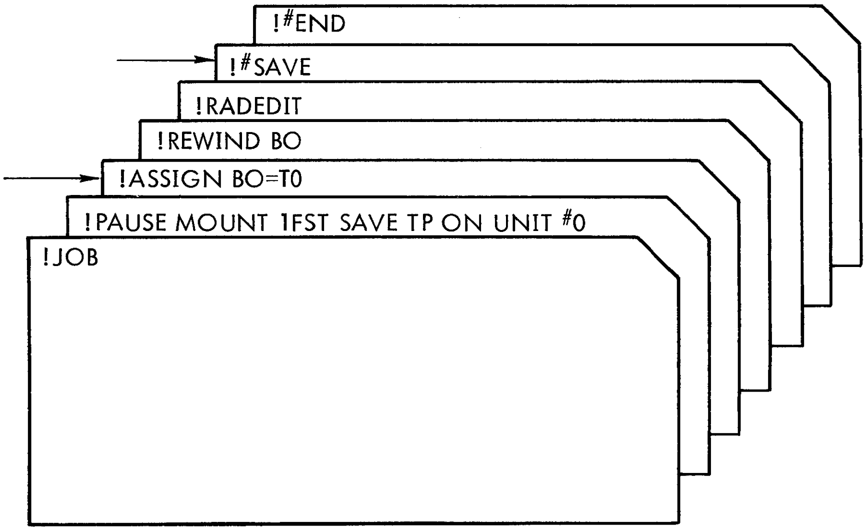
Figure 64. Hex Correction Input Example

21. HOW TO SAVE AND RESTORE AN RBM SYSTEM

The RAD Editor can be used to save and restore an RBM system without the necessity of going through a complete SYSGEN. Two methods are available for saving the RBM system files: rebootable save and file save.

HOW TO CREATE A REBOOTABLE SAVE TAPE

The following control command sequence



```
!#END
!#SAVE
!RADEDIT
!REWIND BO
!ASSIGN BO=T0
!PAUSE MOUNT 1FST SAVE TP ON UNIT #0
!JOB
```

will generate a rebootable save tape on magnetic tape that contains the entire [†]RBM system.

Note that the "T0" device operational label used on the !ASSIGN command instead of a standard device file number (DFN) is an option that must be defined at SYSGEN (see Chapter 19).

The RBM areas contained on the rebootable save tape may be restored in their entirety by performing a bootstrap operation with the magnetic tape, or may be selectively restored via the RAD Editor !#RESTORE command.

BOOTING AN RBM SAVE TAPE

When an RBM save tape is bootstrap loaded via the hardware load procedure, the message

```
RESTORING VERSION XX OF mm/DD/yy  HRMN
```

is output on the keyboard/printer.

where

XX   is the version of the RBM system.

mm/DD/yy HRMN   is the date and time the save tape was created.

[†] If no parameter follows !#SAVE, the RAD Editor will save all areas currently known to RBM except CP (Checkpoint) and BT (Background Temp).

As each new area is encountered on the save tape, the message

```
RESTORING AR TO DN
```

is output on the keyboard printer

where

AR is the area mnemonic.

DN is the device number to contain the area.

If an area is being restored to a disk pack, the first occurrence of such an area will cause the following message to be output:

```
IDLE, RUN TO WRITE
```

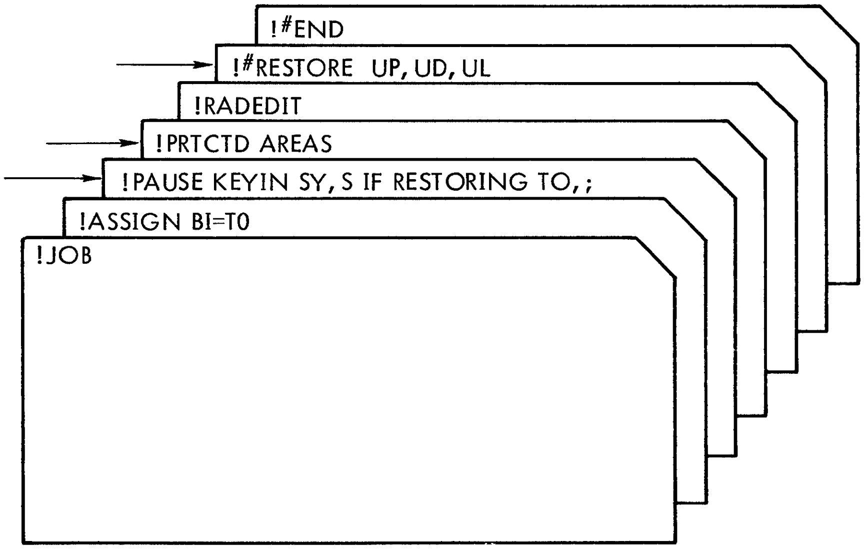
If it is permissible to write on the indicated device, move the COMPUTE switch to IDLE and then back to RUN. This measure is intended to prevent inadvertent destruction of information on disk packs.

If an I/O error occurs, the program will output an appropriate message and retry the operation. If the error condition persists, the operator may abort the restoration of the area currently being restored by pressing the Control Panel INTERRUPT switch. This will cause the program to skip to the next area on the tape.

When all areas have been restored (or the logical end of tape is encountered), the program will unload the input tape and execute the RBM bootstrap.

SELECTIVELY RESTORING AREAS FROM A REBOOTABLE SAVE TAPE

The control command sequence in the example



```
!#END
!#RESTORE  UP, UD, UL
!RADEDIT
!PRTCTD AREAS
!PAUSE KEYIN SY, S IF RESTORING TO, ;
!ASSIGN BI=T0
!JOB
```

will restore the User Processor, User Data, and User Library areas from a magnetic tape that was generated as described previously.

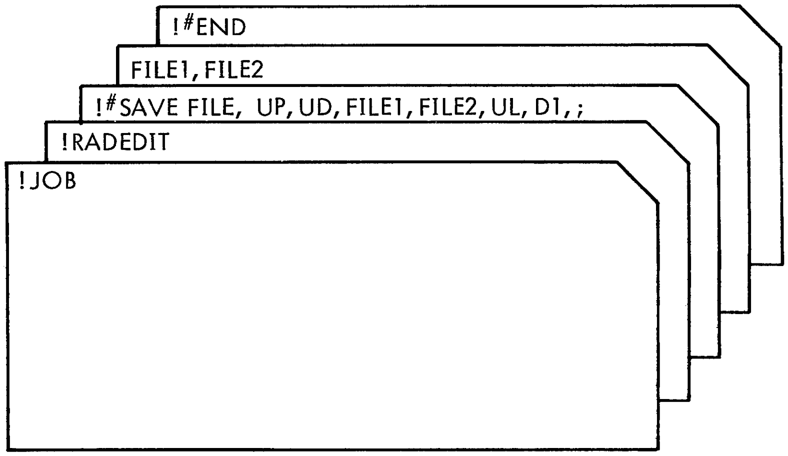
The RAD Editor restores the selected areas to their currently allocated regions, which must be on the same device as they were at the time of the save. However, the areas being restored need not be to the same physical region of the device. If the BOT of the area being restored is different from the BOT of the current allocation for that area, the restore will proceed normally and the area file directory will be updated to reflect the new file positions. If the area being restored contains nonzero data past the EOT of the current allocation, an EOT message will be output and the area will be truncated to fit the current allocation. In this case, the updated file directory may contain files that appear beyond the area EOT; these files should be deleted.

Note that selective restoration may not be used to restore the SP or SD areas. The bootstrap operation must be used if the SP or SD areas are to be restored.

HOW TO SAVE RBM SYSTEM FILES

If the RAD Editor !#SAVE command is used with the keyword "FILE", the files indicated by the remainder of the parameters are saved on the BO device, which may be a magnetic tape, a paper tape punch, or a card punch.

Each file is identified by its area mnemonic and file name, together with sufficient information to restore the file to the area from which it was saved. The example



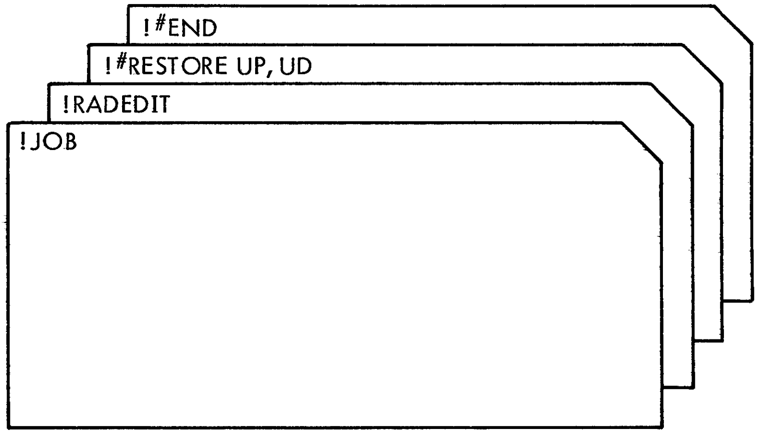
```
!#END
FILE1, FILE2
!#SAVE FILE, UP, UD, FILE1, FILE2, UL, D1, ;
!RADEDIT
!JOB
```

will cause the following files to be saved on the current BO device assignment:

Area	Files
UP	all
UD	FILE1,FILE2
UL	all
D1	FILE1,FILE2

RESTORING RBM SYSTEM FILES

The RAD Editor !#RESTORE command is used to restore files previously saved via a !#SAVE command. For example, if the commands



```
!#END
!#RESTORE UP, UD
!RADEDIT
!JOB
```

were given with the output from the previous !#SAVE example being read from the BI device, the following files would be restored:

Area	Files
UP	All that existed at the time of the save would be added to current area contents.
UD	FILE1 and FILE2 would be added to current area contents.

In this method of restoring files, the file name is added to the area if it does not already exist; otherwise, the current content of the file is replaced by that from BI.

Xerox Data Systems

READER COMMENT FORM

XEROX®

We would appreciate your comments and suggestions for improving this publication.

Publication No.	Rev. Letter	Title	Current Date

How did you use this publication?

□ Learning □ Installing □ Operating
□ Reference □ Maintaining □ Sales

Is the material presented effectively?

□ Fully covered □ Well illustrated
□ Clear □ Well organized

What is your overall rating of this publication?

□ Very good □ Fair □ Very poor
□ Good □ Poor

What is your occupation?

Your other comments may be entered here. Please be specific and give page, column, and line number references where applicable. To report errors, please use the XDS Software Improvement or Difficulty Report (1188) instead of this form.

Thank you for your interest.

Fold and fasten as shown on back.
No postage needed if mailed in U.S.A.

Your name and return address.

STAPLE STAPLE

FOLD

FIRST CLASS
PERMIT NO. 229
EL SEGUNDO, CALIF.

BUSINESS REPLY MAIL

NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY

Xerox Data Systems

701 South Aviation Boulevard
El Segundo, California 90245

ATTN: PROGRAMMING PUBLICATIONS

CUT ALONG LINE

FOLD

Xerox Data Systems

701 South Aviation Boulevard
El Segundo, California 90245
213 679-4511

XEROX