

May 1979

This document describes the RSTS/E Task Builder; its switches and options, overlays, resident libraries, and memory allocation.

**RSTS/E Task Builder
Reference Manual**

Order No. AA-5072A-TC
Including AD-5072A-T1

SUPERSESSION/UPDATE INFORMATION: This manual contains information on the RSTS/E Task Builder and includes information on V7.0 resident library capability.

OPERATING SYSTEM AND VERSION: RSTS/E V7.0

SOFTWARE VERSION: TKB V7.0

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

digital equipment corporation · maynard, massachusetts

First Printing, December 1977
Revised: May 1979

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

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

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

Copyright © 1977, 1979 by Digital Equipment Corporation

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

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECTape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	TMS-11
ASSIST-11	RTS-8	ITPS-10

CONTENTS

		Page
PREFACE		xi
CHAPTER 1	INTRODUCTION	
1.1	INTRODUCTION	1-1
1.2	BRIEF DESCRIPTION OF THE TASK BUILDER	1-1
1.3	ORGANIZATION OF THIS MANUAL	1-2
CHAPTER 2	TASK BUILDER COMMANDS	
2.1	INTRODUCTION	2-1
2.2	TASK COMMAND LINE	2-2
2.3	MULTIPLE LINE INPUT	2-4
2.4	OPTIONS	2-5
2.5	MULTIPLE TASK SPECIFICATIONS	2-6
2.6	INDIRECT COMMAND FILES	2-6
2.7	COMMENT LINES	2-9
2.8	THE EXAMPLE PROGRAMS	2-9
2.8.1	Entering the Source Language	2-10
2.8.2	Compiling the Programs	2-13
2.8.3	Task-Building the Programs	2-13
2.9	SYNTAX RULES	2-14
CHAPTER 3	SWITCHES AND OPTIONS	
3.1	SWITCHES	3-1
3.1.1	/CC (Concatenated Object Modules)	3-3
3.1.A	/CM (Compatibility Mode Overlay Structure)	3-3
3.1.2	/DA (Debugging Aid)	3-3
3.1.3	/DL (Default Library)	3-3
3.1.4	/FP (Floating Point)	3-4
3.1.5	/FU (Full Search)	3-4
3.1.B	/HD (Header)	3-5
3.1.6	/LB (Library File)	3-5
3.1.7	/MA (Map Contents of File)	3-6
3.1.8	/MP (Overlay Description)	3-6
3.1.C	/PI (Position Independent)	3-7
3.1.9	/PM (Post-Mortem Dump)	3-7
3.1.D	/RO (Resident Overlay)	3-7
3.1.10	/SH (Short Map)	3-8
3.1.11	/SQ (Sequential)	3-8
3.1.12	/SS (Selective Search)	3-8
3.1.13	/WI (Wide Listing Format)	3-9
3.1.14	/XT (Exit on Diagnostic)	3-9
3.1.15	Conflicting Switches /LB and /CC	3-9
3.2	OPTIONS	3-10

CONTENTS (Cont.)

	Page
3.2.1 Control Option	3-12
3.2.1.1 ABORT (Abort the Building of the Task)	3-12
3.2.2 Identification Options	3-12
3.2.2.1 TASK (Task Name)	3-12
3.2.2.2 PAR (Partition)	3-12
3.2.3 Allocation Options	3-14
3.2.3.1 EXTSC (PSECT Extension)	3-14
3.2.3.2 EXTTSK (Extend Task Memory)	3-14
3.2.3.3 STACK (Stack Size)	3-15
3.2.3.A WNDWS (Number of Address Windows)	3-15
3.2.3.4 Example of Allocation Options	3-15
3.2.4 Storage Sharing Options	3-16
3.2.4.1 HISEG (High Segment)	3-16
3.2.4.2 COMMON (System Common Block) or LIBR (System Resident Library)	3-16
3.2.4.3 RESCOM (Resident Common Block) or RESLIB (Resident Library)	3-17
3.2.4.4 Examples of Resident Library Switches and Options	3-17
3.2.5 Device Specifying Options	3-18
3.2.5.1 UNITS (Logical Unit Usage)	3-18
3.2.5.2 ASG (Device Assignment)	3-19
3.2.5.3 Example of Device Specifying Options	3-19
3.2.6 Storage Altering Options	3-20
3.2.6.1 GBLDEF (Global Symbol Definition)	3-20
3.2.6.2 GBLREF (Global Symbol Reference)	3-20
3.2.6.3 ABSPAT (Absolute Patch)	3-21
3.2.6.4 GBLPAT (Global Relative Patch)	3-21
3.2.6.5 Example of Storage Altering Options	3-22
3.3 ABORTS AND REBUILDING	3-22
3.3.1 Aborting the Task	3-23
 CHAPTER 4 MEMORY ALLOCATION	
4.1 TASK MEMORY STRUCTURE	4-1
4.2 TASK IMAGE MEMORY	4-2
4.2.1 PSECTS	4-2
4.2.2 PSECT Allocation	4-4
4.2.3 PSECT Placement	4-6
4.3 GLOBAL SYMBOL RESOLUTION	4-6
4.4 TASK IMAGE FILE	4-7
4.5 MEMORY ALLOCATION FILE	4-7
4.5.1 Contents of the Memory Allocation File	4-7
4.5.2 Control of Memory Allocation File Contents and Format	4-9
4.6 MEMORY ALLOCATION MAP FOR BASIC-PLUS-2 VERSION OF USER	4-10
4.7 MEMORY ALLOCATION MAP FOR COBOL VERSION OF USER	4-14

CONTENTS (Cont.)

	Page
CHAPTER 5 OVERLAY CAPABILITY	
5.1 OVERLAY DESCRIPTION	5-1
5.1.1 Disk-Resident Overlay Structures	5-2
5.1.2 Overlay Tree	5-4
5.1.2.1 Overlay Loading	5-5
5.1.2.2 Resolving Global Symbols in a Multi-Segment Task	5-6
5.1.2.3 Resolving Global Symbols from the Default Library	5-8
5.1.2.4 Resolving PSECTS in a Multi-Segment Task	5-8
5.1.3 Overlay Description Language (ODL)	5-9
5.1.3.1 .ROOT and .END Directives	5-10
5.1.3.2 .FCTR Directive	5-11
5.1.3.3 .NAME Directive	5-11
5.1.3.4 .PSECT Directive	5-14
5.1.3.5 Indirect Files	5-15
5.1.4 Multiple Tree Structures	5-15
5.1.4.1 Defining a Multiple Tree Structure	5-15
5.1.4.2 Multiple-Tree Example	5-16
5.1.5 Overlay Core Image	5-17
5.1.6 Overlaying Programs Written in a Higher-Level Language	5-18
5.2 USER OVERLAY TREE	5-19
5.2.1 Defining the ODL File	5-19
5.2.2 Building the Task	5-19
5.3 SUBROUTINE COMMUNICATION	5-20
5.4 SUMMARY OF THE OVERLAY DESCRIPTION LANGUAGE	5-40
CHAPTER 6 THE AUTOLOAD MECHANISM	
6.1 AUTOLOAD	6-1
6.1.1 Autoload Indicator	6-1
6.1.2 Path-Loading	6-3
6.1.3 Autoload Vectors	6-4
CHAPTER 7 RESIDENT LIBRARIES	
7.1 INTRODUCTION	7-1
7.1.1 Resident Library Installation	7-3
7.2 CREATING A RESIDENT LIBRARY	7-5
7.2.1 Position Independent and Absolute Libraries	7-5
7.2.2 Resident Libraries With Memory Resident Overlays	7-7
7.2.3 Run-Time System Support for Overlaid Resident Libraries	7-9
7.3 ACCESS TO A RESIDENT LIBRARY	7-10
7.3.1 Referencing a Resident Library	7-11

CONTENTS (Cont.)

	Page	
APPENDIX A	ERROR MESSAGES	
APPENDIX B	OCTAL TO DECIMAL CONVERSION TABLE	
B.1	INTRODUCTION	B-1
B.2	CONVERTING OCTAL NUMBERS RANGING FROM 0 TO 7777 TO DECIMAL NUMBERS	B-2
B.2.1	Converting Octal 43 to Decimal	B-2
B.2.2	Converting Octal 1000 to Decimal	B-3
B.2.3	Converting Octal 7456 to Decimal	B-3
B.3	CONVERTING DECIMAL NUMBERS RANGING FROM 0 TO 4095 TO OCTAL	B-4
B.3.1	Converting Decimal 17 to Octal	B-4
B.3.2	Converting Decimal 870 to Octal	B-5
B.3.3	Converting Decimal 3826 to Octal	B-5
B.4	CONVERTING OCTAL NUMBERS FROM 10000 TO 77777 TO DECIMAL NUMBERS	B-6
B.4.1	Converting Octal 10042 to Decimal	B-6
B.4.2	Converting Octal 67341 to Decimal	B-7
B.4.3	Converting Octal 30000 to Decimal	B-8
B.5	CONVERTING DECIMAL NUMBERS RANGING FROM 4096 TO 32767 TO OCTAL	B-9
B.5.1	Converting Decimal 4787 to Octal	B-9
B.5.2	Converting Decimal 26872 to Octal	B-10
APPENDIX C	TASK BUILDER DATA FORMATS	
C.1	GLOBAL SYMBOL DIRECTORY	C-2
C.1.1	Module Name	C-3
C.1.2	Control Section Name	C-4
C.1.3	Internal Symbol Name	C-4
C.1.4	Transfer Address	C-5
C.1.5	Global Symbol Name	C-5
C.1.6	PSECT Name	C-6
C.1.7	Program Version Identification	C-8
C.2	END OF GLOBAL SYMBOL DIRECTORY	C-9
C.3	TEXT INFORMATION	C-9
C.4	RELOCATION DIRECTORY	C-10
C.4.1	Internal Relocation	C-11
C.4.2	Global Relocation	C-12
C.4.3	Internal Displaced Relocation	C-12
C.4.4	Global Displaced Relocation	C-13
C.4.5	Global Additive Relocation	C-13
C.4.6	Global Additive Displaced Relocation	C-13
C.4.7	Location Counter Definition	C-14
C.4.8	Location Counter Modification	C-14
C.4.9	Program Limits	C-15
C.4.10	PSECT Relocation	C-15
C.4.11	PSECT Displaced Relocation	C-16
C.4.12	PSECT Additive Relocation	C-16
C.4.13	PSECT Additive Displaced Relocation	C-17

CONTENTS (Cont.)

		Page
C.4.14	Complex Relocation	C-18
C.4.15	Additive Relocation	C-19
C.5	INTERNAL SYMBOL DIRECTORY	C-20
C.6	END OF MODULE	C-20
APPENDIX D	TASK IMAGE FILE STRUCTURE	
D.1	LABEL BLOCK GROUP	D-1
D.2	HEADER	D-4
D.2.1	Low Core Context	D-7
D.3	OVERLAY DATA STRUCTURE	D-8
D.3.1	Autoload Vectors	D-9
D.3.2	Segment Descriptor	D-9
D.4	ROOT SEGMENT	D-11
D.5	OVERLAY SEGMENTS	D-11
APPENDIX E	RESERVED SYMBOLS	
APPENDIX F	IMPROVING TASK BUILDER PERFORMANCE	
F.1	EVALUATING AND IMPROVING TASK BUILDER PERFORMANCE	F-1
F.1.1	Task Builder Work File	F-1
F.1.2	Input File Processing	F-3
APPENDIX G	INCLUDING A DEBUGGING AID	
APPENDIX H	GLOSSARY	
INDEX		

FIGURES

FIGURE		
2-1	Indirect File Interaction	2-8
4-1	Task Memory Structure	4-2
4-2	PSECT Allocations Grouped by Access Code	4-5
4-3	Memory Allocation File for BASIC-PLUS-2 Version of User	4-11
4-4	Memory Allocation File for COBOL Version of User	4-15
5-1	TK1 Memory Allocation	5-3
5-2	Allocation for a Multi-Segment Task	5-3
5-3	How to Read a Block Diagram	5-4
5-4	Multi-Level Overlay Tree	5-5
5-5	Global Symbols in a Tree	5-7
5-6	Common Blocks in a Tree	5-9
5-7	A Simple Multi-Level Tree	5-10.1

CONTENTS (Cont.)

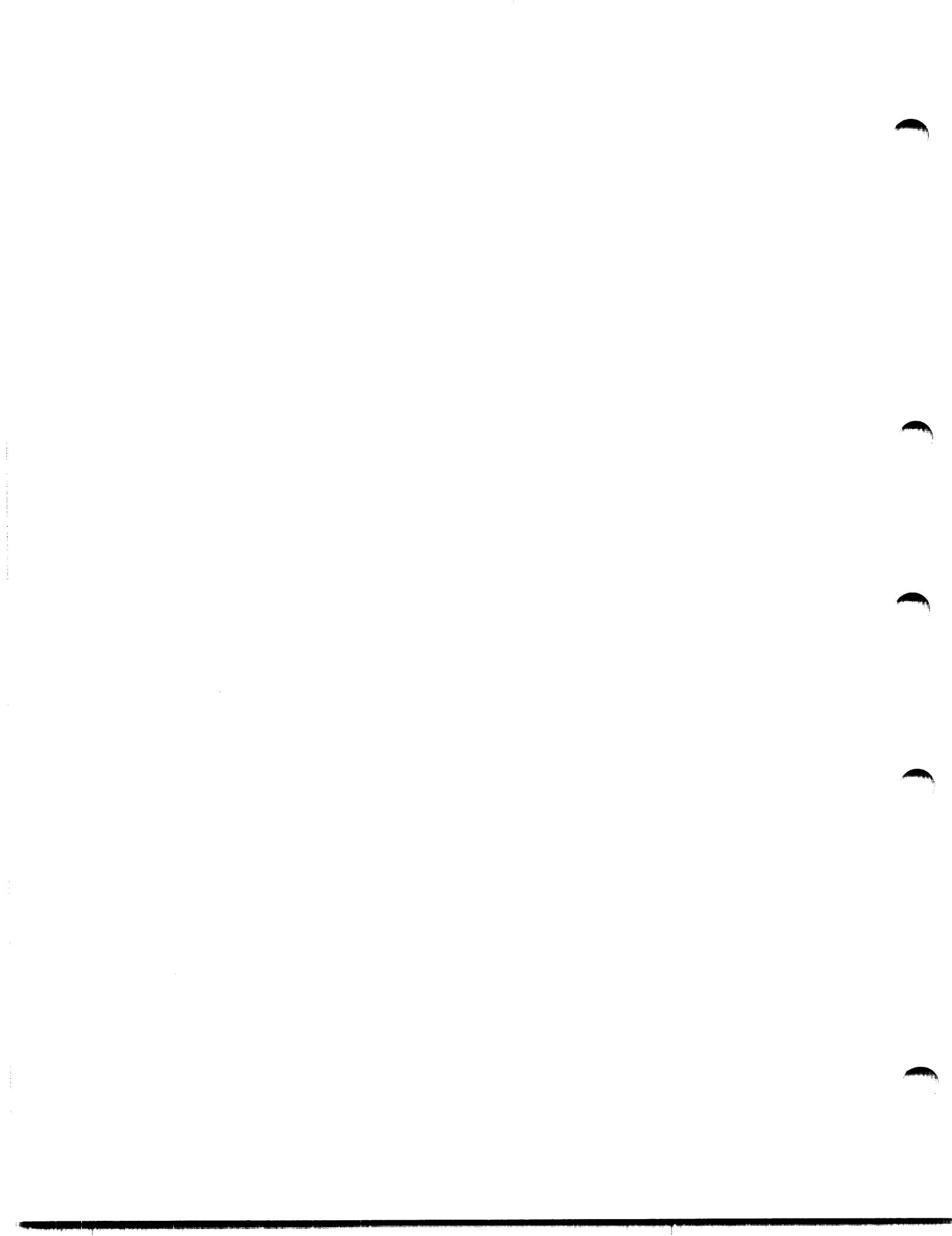
	Page	
5-8	TK1 Modified Tree Using the .NAME Directive	5-13
5-9	Co-Tree	5-17
5-10	Co-Tree Block Diagram	5-17
5-11	User Overlay Tree	5-19
5-12	User Block Diagram	5-20
5-13	BASIC-PLUS-2 User ODL File	5-20
5-14	COBOL User ODL File	5-21
5-15	User COBOL Memory Allocation Map	5-22
5-16	User BASIC-PLUS-2 Memory Allocation Map	5-34
5-17	Simple Tree (Summary Example)	5-41
5-18	Co-Trees (Summary Example)	5-43
6-1	The .FCTR Directive	6-2
7-1	System Memory Usage	7-1
7-2	Shared and Non-Shared Memory	7-2
7-3	Resident Library Access	7-4
B-1	Table B-1, Showing Table Parts for Conversion	B-2
B-2	Steps for Converting Octal 43 to Decimal	B-2
B-3	Steps for Converting Octal 1000 to Decimal	B-3
B-4	Steps for Converting Octal 7456 to Decimal	B-4
B-5	Steps for Converting Decimal 17 to Octal	B-4
B-6	Steps for Converting Decimal 870 to Octal	B-5
B-7	Steps for Converting Decimal 3826 to Octal	B-6
B-8	Steps for Converting Octal 10042 to Decimal	B-7
B-9	Steps for Converting Octal 67341 to Decimal	B-8
B-10	Steps for Converting Octal 30000 to Decimal	B-8
B-11	Steps for Converting Decimal 4787 to Octal	B-9
B-12	Steps for Converting Decimal 26872 to Octal	B-10
C-1	General Object Module Format	C-2
C-2	GSD Record and Entry Format	C-3
C-3	Module Name Entry Format	C-4
C-4	Control Section Name Entry Format	C-4
C-5	Internal Symbol Name Entry Format	C-5
C-6	Transfer Address Entry Format	C-5
C-7	Global Symbol Name Entry Format	C-6
C-8	PSECT Name Entry Format	C-8
C-9	Program Version Identification Entry Format	C-8
C-10	End-of-GSD Record Format	C-9
C-11	Text Information Record Format	C-9
C-12	Relocation Directory Record Format	C-11
C-13	Internal Relocation Entry Format	C-12
C-14	Global Relocation Entry Format	C-12
C-15	Internal Displaced Relocation Entry Format	C-12
C-16	Global Displaced Relocation Entry Format	C-13
C-17	Global Additive Relocation Entry Format	C-13
C-18	Global Additive Displaced Relocation Entry Format	C-14
C-19	Location Counter Definition	C-14
C-20	Location Counter Modification	C-15
C-21	Program Limits Entry Format	C-15
C-22	PSECT Relocation Entry Format	C-16
C-23	PSECT Displaced Relocation Entry Format	C-16

CONTENTS (Cont.)

	Page	
C-24	PSECT Additive Relocation Entry Format	C-17
C-25	PSECT Additive Displaced Relocation	C-17
C-26	Complex Relocation Entry Format	C-19
C-27	Additive Relocation Entry Format	C-19
C-28	Internal Symbol Directory Record Format	C-20
C-29	End-of-Module Record Format	C-20
D-1	Task Image on Disk	D-1
D-2	Label Block Group	D-2
D-3	Task Header Fixed Part	D-5
D-4	Task Header Variable Part	D-6
D-5	Vector Extension Area Format	D-8
D-6	Task-Resident Overlay Data Base	D-8
D-7	Autoload Vector Entry	D-9
D-8	Segment Descriptor	D-9
D-9	Sample Tree	D-10
D-10	Segment Linkage Directives	D-11

TABLES

2-1	Default File Extensions	2-2
2-2	Sample Task Builder Commands	2-3
3-1	Task Builder Switches	3-2
3-2	Task Builder Options	3-11
4-1	PSECT Attributes	4-3
4-2	PSECT Allocation	4-5
4-3	Allocation Totals	4-5
4-4	Global Reference Resolution	4-6
B-1	Octal-Decimal Integer Conversion Table	B-11
E-1	Task Builder Reserved Global Symbols	E-1
E-2	PSECT Names Reserved by the Task Builder	E-2



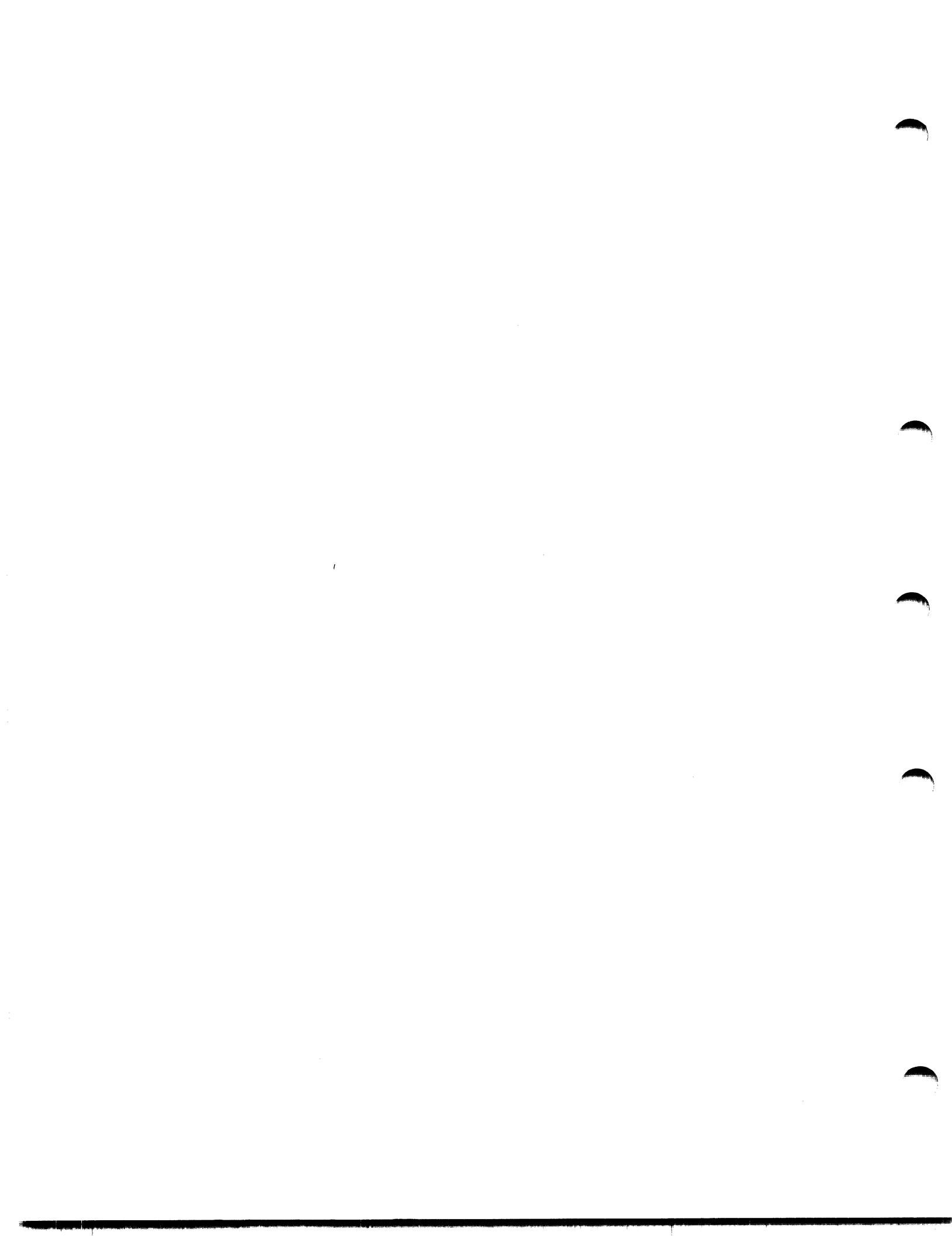
PREFACE

This manual introduces you to the basic concepts and capabilities of the Task Builder. Examples that go from simple to complex introduce and describe Task Builder features. Computer-generated prompts and user-typed responses (in color) are printed in terminal type font.

You will best be able to use this manual if you have compiled several source language programs and are reasonably familiar with your source language. You will find information in this manual that is relevant to users on the systems programmer/analyst level, but is not necessarily relevant for you. Most of this kind of material has been confined to the appendices.

This manual has six chapters. They describe basic Task Builder functions and show you how to use them. The appendices list error messages and give detailed descriptions of the structures the Task Builder uses.

The RSTS/E Documentation Directory tells you what you should know for optimum usage of each manual. Other manuals that may help you if you have a problem are described there.



CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

The Task Builder is a system program that transforms one or more compiled modules into a single, executable image called a task.

Task Builder functions include:

- Linking compiled modules
- Resolving any references to system or user object module libraries
- Allocating required system and task memory
- Producing an optional memory map
- Building an overlaid task according to your overlay description
- Linking the task to run-time systems or resident libraries

An object module is the output from a compiler or assembler and the input to the Task Builder. Object modules cannot be directly executed. They must first be processed by the Task Builder.

1.2 BRIEF DESCRIPTION OF THE TASK BUILDER

The Task Builder connects, or links, object modules by using global symbols to resolve references between modules. Global symbols are labels that are defined in one module and referenced in others. Each global symbol defined in the module can be associated with a unique memory address when the Task Builder assigns a particular memory location to an object module. The Task Builder then uses this address to resolve references to these global symbols in other modules.

Often, the programs you write cannot stand alone. For example, they may need library routines, subprograms, or object modules created by another programming language. The Task Builder can gather the various elements you need and combine them to produce a runnable task, the task image.

The Task Builder makes a set of assumptions, or defaults, about the task image based on typical usage and storage requirements. You can override these defaults by including switches and options when you build the task. This allows you to build a task that is tailored to its own input/output and storage requirements.

INTRODUCTION

The Task Builder also produces, on request, a memory allocation file, or map, that contains information describing storage allocation, the separate modules that comprise the task, and the value of all global symbols. Additionally, you may request that the list of global symbols, accompanied by the name of each referencing module, be appended to the file.

The Task Builder also enables you to build extremely large and complex programs.

- The Task Builder allows you to build overlayable tasks that may be larger in aggregate size than the main memory size limitation.

Run-time systems are segments of code that are shared by numerous tasks. If your task uses resident subroutines to save memory, the Task Builder lets you link to existing run-time systems.

As you read this manual you will encounter two similar sample programs called USER. Written in COBOL and BASIC-PLUS-2, they show you how to coordinate your source language and Task Builder as you build your task.

1.3 ORGANIZATION OF THIS MANUAL

There are six more chapters in this manual:

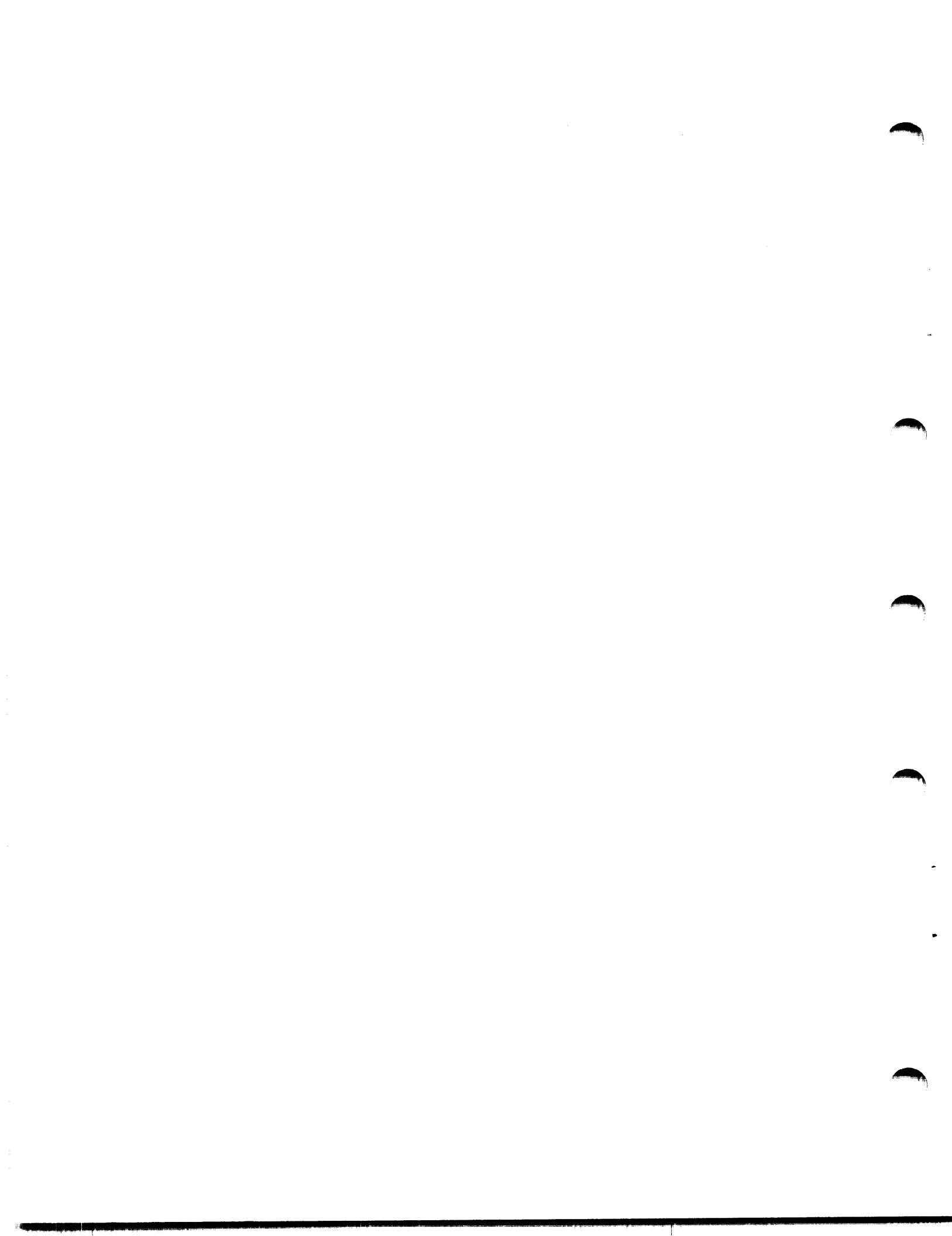
- Chapter 2, TASK BUILDER COMMANDS, discusses the Task Builder command and option modes and how to distinguish between them.
- Chapter 3, SWITCHES AND OPTIONS, covers Task Builder assumptions that you can change.
- Chapter 4, MEMORY ALLOCATION, shows how the Task Builder assigns memory to the various parts of your task.
- Chapter 5, OVERLAY CAPABILITY, discusses overlay design and implementation.
- Chapter 6, THE AUTOLOAD MECHANISM, shows what happens when your program calls for a segment that is not currently in memory.
- Chapter 7, RESIDENT LIBRARIES, describes the creation of resident libraries, their position in memory, and user access to them.

The appendices contain convenient reference material for users on varying levels:

- Appendix A contains Task Builder error messages.
- Appendix B contains an octal-decimal conversion table and instructions for its use.
- Appendix C contains Task Builder data formats.
- Appendix D contains information on task image file structure.

INTRODUCTION

- Appendix E contains symbol names reserved for the Task Builder's use alone.
- Appendix F contains information on how to improve Task Builder performance.
- Appendix G shows how to include a debugging aid in your task.
- Appendix H contains a glossary for your convenience.



CHAPTER 2

TASK BUILDER COMMANDS

2.1 INTRODUCTION

This chapter describes basic command sequences that can be used to build most tasks. The sequences are explained, then shown in an example. A discussion of syntax for the commands ends the chapter. Some examples in this manual begin with the command "TKB". If you have TKB installed as a CCL (Concise Command Language) command, you can invoke the Task Builder with that command. If not, use "RUN \$TKB".

NOTE

The \$ in RUN \$TKB shows that Task Builder is stored under the system library account [1,2]. The filename is TKB.TSK. Task Builder usually runs under the RSX Run-Time System. The CCL command TKB is optional. See your system manager for more information.

Task Builder prompts consist of the characters TKB>. Here is one way to compile, load, and execute a simple task:

1. Write a program (BASIC-PLUS-2 is used here). (The filename is MAIN.B2S).
2. As the system prompts for input, type the following responses:

```
RUN $BASIC2
OLD MAIN.B2S
COMPILE /OBJ
EXIT
TKB
TKB USER.TSK,USER.MAP=MAIN.OBJ
/
HISEG=BASIC2
//
RUN USER.TSK
```

The COBOL equivalent, for the filename MAIN.CBL, is:

```
CBL MAIN.OBJ,MAIN.LST=MAIN.CBL/KER:MA
RUN $CBLMRG
MRGODL.ODL
M
N
```

TASK BUILDER COMMANDS

```
MAIN.ODL  
<CR> or the appropriate PPN  
N  
TKB USER.TSK=MRGODL.ODL/MP  
RUN USER.TSK
```

The COMPILE command causes the BASIC-PLUS-2 compiler to translate the source language in the file MAIN.B2S into the relocatable object module called MAIN.OBJ. The next command (TKB) causes the Task Builder to process the file MAIN.OBJ, producing the task image file USER.TSK. The last command (RUN) causes the task to execute.

The simplest use of the Task Builder is shown in the command:

```
TKB USER.TSK=MAIN.OBJ
```

This command gives the name of a single file as input - MAIN.OBJ - and the name of a single file as output - USER.TSK. Note the filename extensions: they may be default entries, but are specifically listed here. Other forms of task command lines are discussed later in the chapter.

If you do not give extensions for files that are input to or output from the Task Builder, the Task Builder will assign default file extensions. Table 2-1 lists these extensions and the applicable file. If the files you specify do not have the extensions listed as defaults in Table 2-1, you must name the file with its extension to the Task Builder.

Table 2-1
Default File Extensions

Type of File	Extension	File Contents
Input	.OBJ .OLB .ODL .CMD	Object Module Object Library Overlay Description Task Builder Commands
Output	.TSK .MAP .STB	Task Image Memory Allocation Map Symbol Definition Table

2.2 TASK COMMAND LINE

The task command line contains the output file specifications, an equal sign, and then the input file specifications. There can be up to three output files and any number of input files. The task command line has the form:

```
TASK-IMAGE,MAP,SYMBOL-DEFINITION=INPUT-FILE[,INPUT-FILE,...]
```

The output files must be given in a specific order:

1. The task image file
2. The memory allocation file
3. The symbol definition file

TASK BUILDER COMMANDS

The task image file (.TSK) contains the task to be run. The memory allocation file (.MAP) contains information about the size and location of components within the task. The symbol definition file (.STB) contains the global symbol definitions in the task and their virtual or relocatable addresses in a format suitable for reprocessing by the Task Builder. See Section 5.1.2.2 for a discussion of global symbols.

Any output file can be omitted by dropping the filename. Be sure to place commas where necessary so that the Task Builder sees each output file in its correct syntax location.

Please note that the only space in the line should fall immediately after the TKB command and then only if you are using the CCL command TKB to invoke the Task Builder. A space anywhere else in the line terminates the command. The commands in Table 2-2 below illustrate correct comma placement and ways in which the output filenames are interpreted. Note that in all cases, the input file is IN1, the memory allocation file is MP1, the symbol definition file is SF1, and the task image is IMG1. The file extensions have been omitted for easier reading.

Table 2-2
Sample Task Builder Commands

Command	Output Files
TKB IMG1,MP1,SF1=IN1	The task image file is IMG1.TSK, the memory allocation file is MP1.MAP, the symbol definition file is SF1.STB, and the input file is IN1.
TKB IMG1=IN1	The task image file is IMG1.TSK. The other output files are omitted. The first filename the Task Builder encounters is assumed to be that of the task image file unless there are one or two commas preceding the filename. Putting a comma after the last filename, when you are naming less than three files, is unnecessary.
TKB ,MP1=IN1	The memory allocation file is MP1.MAP. The comma preceding "MP1" indicates that the task image file has been omitted. The symbol definition file is also omitted. After picking up the task image filename or recognizing the absence of a task image filename, Task Builder looks for a memory allocation file. The Task Builder assumes that the next filename it encounters is that of the memory allocation file. To tell the Task Builder that there is no memory allocation file you can do one of three things:

(continued on next page)

TASK BUILDER COMMANDS

Table 2-2 (Cont.)
Sample Task Builder Commands

Command	Output Files
TKB ,MP1=IN1 (Cont.)	<ul style="list-style-type: none"> ● Put two commas in front of the symbol definition file, whether you name a task image file or not. ● Name only a task image file. (No commas are necessary in this case.) ● Name no files at all. (See the last example in this series to see how to designate a diagnostic run.)
TKB ,,SF1=IN1	The symbol definition file is SF1.STB. The two preceding commas show that both the task image file and the memory allocation file are omitted.
TKB IMG1,,SF1=IN1	The task image file is IMG1.TSK and the symbol definition file is SF1.STB. Two commas together in this case show that only one file has been omitted - the memory allocation file.
TKB IMG1,MP1=IN1	The absence of a third filename here tells the Task Builder that the symbol definition file is omitted.
TKB =IN1	This is merely a diagnostic run with no output files. The Task Builder assumes this from the fact that an equal sign heads the parameter list.

2.3 MULTIPLE LINE INPUT

When several input files are used in building a task image, a more flexible format is necessary. This multi-line format is also necessary for including options, as discussed in the next section, or for limiting the command line to 80 characters. (Task Builder lines are limited to 80 characters.)

The Task Builder prompts for multi-line format input until it receives a line consisting of only the terminating sequence "///". Here are two ways to do the same thing:

The sequence

```

TKB
TKB>IMG1,MP1=IN1
TKB>IN2,IN3
TKB>///

```

produces the same result as the single line command

```
TKB IMG1,MP1=IN1,IN2,IN3
```

TASK BUILDER COMMANDS

Either sequence produces the task image file IMG1.TSK and the memory allocation file MP1.MAP from the input files IN1.OBJ, IN2.OBJ, and IN3.OBJ.

The output file specifications and the separator "=" must appear on the first TKB command line. Input file specifications can begin or continue on later lines. The terminating pair of slashes directs the Task Builder to stop accepting input, build the task, and return to the system level.

2.4 OPTIONS

You can use options to specify or modify certain features of the task being built. A single slash typed in response to a TKB prompt in command mode¹ directs the Task Builder to request option parameters by displaying "ENTER OPTIONS:" and prompting for input on the next line. The "ENTER OPTIONS:" display notifies you that the Task Builder is now in option mode and is looking for option input. Keep in mind that the Task Builder is still looking for "//" to end the task input and return to the system level. A representative Task Builder command sequence might look like this:

```
TKB
TKB>IMG1,MP1=IN1
TKB>IN2,IN3
TKB>/
ENTER OPTIONS:
TKB>TASK=USER
TKB>HISEG=BASIC2
TKB>//
```

Here, the options TASK=USER and HISEG=BASIC2 appear followed by the double slash terminator, which ends option input. Control then returns to the system level.

The Task Builder provides several options more fully discussed in Chapter 3, but mentioned briefly here. The general form of an option is a keyword followed by an equal sign and an argument list. Two or more options on a single line are separated by exclamation points (!). Arguments, which may be qualified, are separated by commas. Qualifiers are separated from their corresponding arguments by colons (:). The following example shows all three separators in action:

```
TKB>UNITS=6!ASG=TI:5,SY:1
```

This is equivalent to:

```
TKB>UNITS=6
TKB>ASG=TI:5
TKB>ASG=SY:1
```

which does the same thing on three lines instead of one.

¹ You are in command mode when you invoke Task Builder and when you enter the command line(s).

TASK BUILDER COMMANDS

2.5 MULTIPLE TASK SPECIFICATIONS

If you want to build more than one task at a time, type a single slash in answer to a prompt in option mode in the first task. The single slash will direct the Task Builder to build the task you have just finished entering and prompt for the next one. The lines in the following example are numbered for convenient reference:

```
1 TKB
2 TKB>IMG1.TSK=USER.OBJ,INTRO.OBJ           (first task)
3 TKB>CRUNCH.OBJ,CHATR.OBJ
4 TKB>/
5 ENTER OPTIONS:
6 TKB>TASK=USER
7 TKB>HISEG=BASIC2
8 TKB>/
9 TKB>CUSER.TSK,CUSER.MAP=CUSER.ODL/MP      (second task)
10 TKB>//
```

Two tasks are being built here. The names of the task image files are IMG1.TSK and CUSER.TSK (see lines 2 and 9 above). The second one-slash entry, on line 8, is the end of the first task. Task Builder now looks for a new task to build and finds one:

```
TKB>CUSER.TSK,CUSER.MAP=CUSER.ODL/MP
```

As always, the double slash terminator (see line 10 above) ends Task Builder input and directs the building of, in this case, two tasks. Control then returns to the system level.

NOTE

Notice that line 6 above changed the name of the first task (as displayed by SYSTAT) from IMG1 to USER. However, the name of the task image file is still IMG1.TSK. See also Section 3.2.2.1 for more information on the TASK option.

2.6 INDIRECT COMMAND FILES

It is also possible to enter Task Builder commands indirectly. They may be stored on a permanent file for later processing. If you are using a large number of options or switches, using an indirect file could enable you to type those options or switches only once and run with them whenever you wish. The Task Builder goes to the file for information when you type, on a separate line, a commercial "at" character (@) followed by the filename, as in

```
TKB @AFIL
```

If you typed this command and AFIL contained:

```
IMG1,MPL=IN1
IN2,IN3
/
TASK=USER
HISEG=BASIC2
//
```

TASK BUILDER COMMANDS

then the Task Builder would build the same task as that illustrated in the first example in Section 2.3. Note that the contents of AFIL are the same as the entries in that example, but without the TKB prompts.

When the Task Builder encounters a single slash in the indirect file, it does one of two things, depending on which mode it is in at the time:

- In command mode, the Task Builder enters option mode and continues as if it were getting its input from a terminal file.
- In option mode, the Task Builder stops accepting input for the task, builds the task, and enters command mode to look for a command line.

When the Task Builder encounters the double slash terminator in the indirect file (AFIL.CMD, in this case), it:

- stops accepting input from the indirect file,
- builds the task,
- returns to the system level as if the contents of the indirect file had been typed on the terminal.

If an end-of-file condition on the file occurs before the double slash terminator, then the Task Builder returns to the terminal to look for input.

CAUTION

A TKB prompt indicates that the Task Builder has returned to the terminal, but does not indicate whether the Task Builder is looking for commands or options. It is your responsibility to be aware of the contents of an indirect file and what the Task Builder is looking for when it returns to the terminal. One way to solve this problem is to indicate in the filename or extension where the Task Builder returns (with a "C" for command mode and an "O" for option mode, for instance).

The Task Builder permits two levels of indirection, primary and secondary, in file references. That is, an indirect file referenced in a sequence of terminal commands (called a primary indirect file) may contain references to further indirect files (called secondary indirect files). These secondary indirect files may not contain any references to any other indirect files and can only return to the primary indirect file by an end-of-file condition. Also, a secondary indirect file can only return to the system level by the double slash terminator.

In Figure 2-1, the primary indirect file is AFIL.CMD. BITBKT.CMD, BFIL.CMD, and CFIL.CMD are all secondary indirect files.

TASK BUILDER COMMANDS

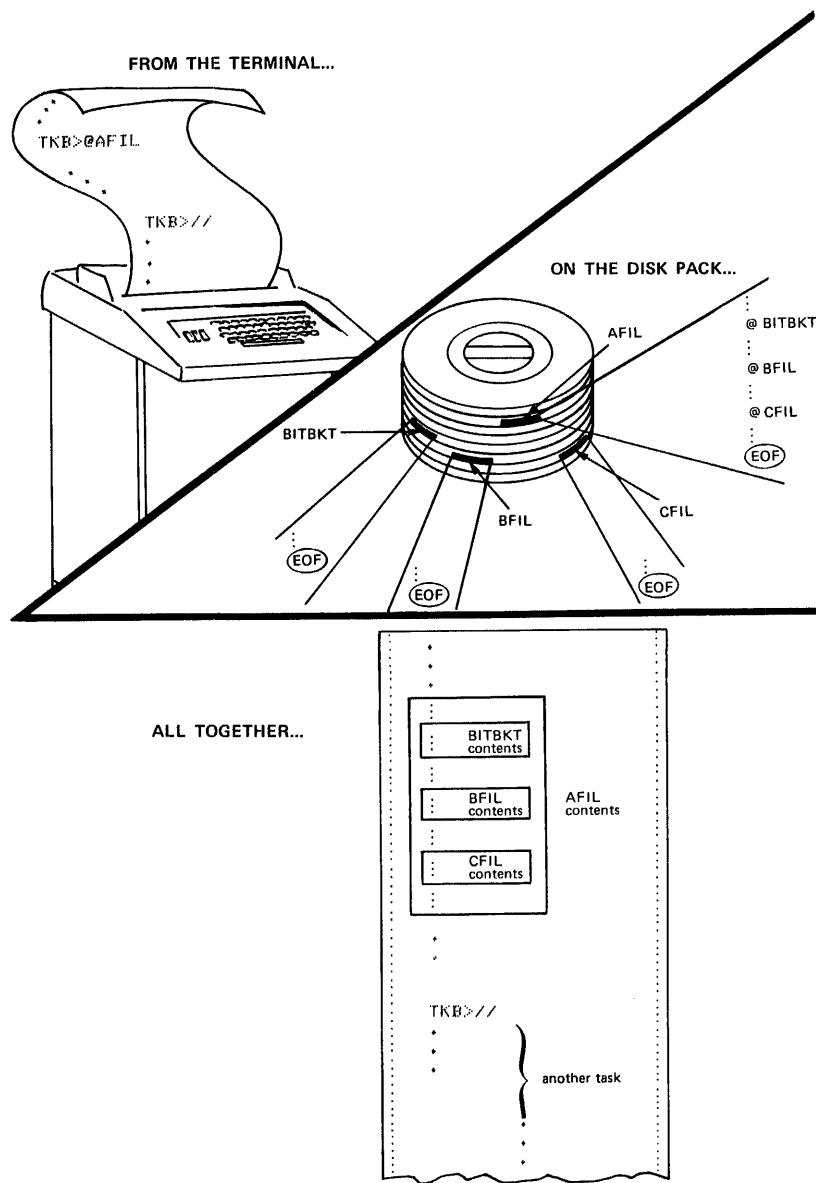


Figure 2-1 Indirect File Interaction

The contents of each secondary indirect file, if any exist, are inserted into the text of the primary indirect file at the point where the secondary indirect file was referenced. The contents of the primary indirect file are inserted into the terminal sequence of commands at the point where the primary indirect file was referenced. If AFIL contains:

```

IMG1,MP1=IN1
IN2,IN3
/
TASK=USER
HISEG=BASIC2
@BFIL
//
```

TASK BUILDER COMMANDS

and BFIL contains:

```
UNITS=12  
ASG=SY:1
```

then the result of the command

```
TKB@AFIL
```

is the same as if you had typed:

```
TKB>IMG1,MP1=IN1  
TKB>IN2,IN3  
TKB>/  
ENTER OPTIONS:  
TKB>TASK=USER  
TKB>HISSEG=BASIC2  
TKB>UNITS=12  
TKB>ASG=SY:1  
TKB>//
```

NOTE

The indirect file reference must appear as a separate line. Otherwise the substitution will not take place and the Task Builder will report the error.

2.7 COMMENT LINES

Comment lines can be included anywhere in the sequence except in lines containing file specifications. A comment begins with a semicolon (;), which may or may not be the first character in the line. A comment ends with a carriage return. Here are some examples:

; <CR>	This is a "null" comment line used for vertical spacing on the page
;THIS COMMENT TAKES A WHOLE LINE. <CR>	
UNITS=12; SET NUMBER OF UNITS <CR>	This comment is used to explain a line of executable TKB command code. Task Builder executes the command and ignores the comment.

2.8 THE EXAMPLE PROGRAMS

The first step in developing your task is to write, compile, and build the basic task. To do this:

- Enter the routines by a text editor or the BASIC-PLUS-2 compiler,
- Have them translated by the appropriate compiler,
- Use the Task Builder to build them into a task.

TASK BUILDER COMMANDS

The routines in the example programs called USER.TSK are:

USER which controls the processing
INTRO which accepts and reformats input data (where necessary)
CRUNCH which performs the computations
CHATR which reports the results

2.8.1 Entering the Source Language

You can enter source lines for the example program by using a text editor. BASIC-PLUS-2 users can also use the BASIC-PLUS-2 compiler. The example programs in BASIC-PLUS-2, then COBOL, are shown below:

```
10      CALL INTRO(A1%,B1%)
20      CALL CRUNCH(A1%,B1%,SUMM%,PRODUCT%,DIFFER%)
30      CALL CHATR(A1%,B1%,SUMM%,PRODUCT%,DIFFER%)
40      END
```

```
10      SUB INTRO(AAZ,BAZ)
20      INPUT "INPUT TWO NUMBERS";AAZ,BAZ
30      SUBEND
```

```
10      SUB CRUNCH(AAZ,BAZ,CAZ,CBZ,CCZ)
20          CAZ = AAZ + BAZ
30          CBZ = AAZ * BAZ
40          CCZ = AAZ - BAZ
50      SUBEND
```

```
10      SUB CHATTR(AAZ%,BAZ%,CAZ%,CBZ%,CCZ%)
20      PRINT "THE SUM OF ";AAZ%;" AND ";BAZ%;" IS ";CAZ%
30      PRINT "THE PRODUCT OF ";AAZ%;" AND ";BAZ%;" IS ";CBZ%
40      PRINT "THE DIFFERENCE OF ";AAZ%;" AND ";BAZ%;" IS ";CCZ%
50      SUBEND
```

TASK BUILDER COMMANDS

```

02 SECOND-NUMBER          PIC S999 VALUE ZERO
      SIGN LEADING SEPARATE CHARACTER.
01 COMPUTATION-AREA.
02 NUMBER-1                PIC S999 USAGE COMP.
02 NUMBER-2                PIC S999 USAGE COMP.
02 SUMM                     PIC S9(6) USAGE COMP.
02 PRODUCT                  PIC S9(6) USAGE COMP.
02 DIFFERENCE               PIC S9(6) USAGE COMP.
01 DISPLAY-AREA.
02 SUM-OUT                  PIC -Z(5)9.
02 PRODUCT-OUT              PIC -Z(5)9.
02 DIFFERENCE-OUT           PIC -Z(5)9.
PROCEDURE DIVISION.
MAIN-MODULE.
CALL "INTRO" USING FIRST-NUMBER, SECOND-NUMBER.
CALL "CRUNCH" USING FIRST-NUMBER, SECOND-NUMBER, SUM-OUT,
      PRODUCT-OUT, DIFFERENCE-OUT.
CALL "CHATR" USING FIRST-NUMBER, SECOND-NUMBER, SUM-OUT,
      PRODUCT-OUT, DIFFERENCE-OUT.
STOP RUN.

```

```

IDENTIFICATION DIVISION.
PROGRAM-ID. INTRO.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. PDP-11.
OBJECT-COMPUTER. PDP-11, SEGMENT-LIMIT IS 3.
DATA DIVISION.
LINKAGE SECTION.
77 FIRST-NUMBER             PIC S999
      SIGN LEADING SEPARATE CHARACTER.
77 SECOND-NUMBER            PIC S999
      SIGN LEADING SEPARATE CHARACTER.
77 NUMBER-1                 PIC S999 USAGE COMP.
77 NUMBER-2                 PIC S999 USAGE COMP.
77 SUMM          PIC S9(6) USAGE COMP.
77 PRODUCT        PIC S9(6) USAGE COMP.
77 DIFFERENCE     PIC S9(6) USAGE COMP.
77 SUM-OUT        PIC -Z(5)9.
77 PRODUCT-OUT    PIC -Z(5)9.
77 DIFFERENCE-OUT PIC -Z(5)9.
PROCEDURE DIVISION USING FIRST-NUMBER, SECOND-NUMBER.
MD1 SECTION 5.
MAIN-MODULE.
DISPLAY "THINK OF TWO NUMBERS THAT ARE SIGNED INTEGERS,".
DISPLAY "      SUCH AS +015 OR -256, THAT ARE LESS THAN 1000",
DISPLAY "      AND GREATER THAN -1000.".
DISPLAY SPACES.
DISPLAY "PLEASE ENTER THE FIRST NUMBER: ".
ACCEPT FIRST-NUMBER.
DISPLAY "AND NOW THE SECOND: ".
ACCEPT SECOND-NUMBER.
EXIT-PARAGRAPH.
      EXIT PROGRAM.
DEFAULT-HALT.
STOP RUN.

```

```

IDENTIFICATION DIVISION.
PROGRAM-ID. CRUNCH.
ENVIRONMENT DIVISION.

```

TASK BUILDER COMMANDS

```
CONFIGURATION SECTION.  
SOURCE-COMPUTER. PDP-11.  
OBJECT-COMPUTER. PDP-11, SEGMENT-LIMIT IS 3.  
DATA DIVISION.  
WORKING-STORAGE SECTION.  
01 COMPUTATION-AREA.  
    02 NUMBER-1          PIC S999 USAGE COMP.  
    02 NUMBER-2          PIC S999 USAGE COMP.  
    02 SUMM              PIC S9(6) USAGE COMP.  
    02 PRODUCT            PIC S9(6) USAGE COMP.  
    02 DIFFERENCE         PIC S9(6) USAGE COMP.  
LINKAGE SECTION.  
77 FIRST-NUMBER          PIC S999  
    SIGN LEADING SEPARATE CHARACTER.  
77 SECOND-NUMBER         PIC S999  
    SIGN LEADING SEPARATE CHARACTER.  
77 SUM-OUT               PIC -Z(5)9.  
77 PRODUCT-OUT            PIC -Z(5)9.  
77 DIFFERENCE-OUT        PIC -Z(5)9.  
PROCEDURE DIVISION USING FIRST-NUMBER, SECOND-NUMBER, SUM-OUT,  
PRODUCT-OUT, DIFFERENCE-OUT.  
MD2 SECTION 5.  
MAIN-MODULE.  
    MOVE FIRST-NUMBER TO NUMBER-1.  
    MOVE SECOND-NUMBER TO NUMBER-2.  
    COMPUTE SUMM = NUMBER-1 + NUMBER-2.  
    COMPUTE PRODUCT = NUMBER-1 * NUMBER-2.  
    COMPUTE DIFFERENCE = NUMBER-1 - NUMBER-2.  
    MOVE SUMM TO SUM-OUT.  
    MOVE PRODUCT TO PRODUCT-OUT.  
    MOVE DIFFERENCE TO DIFFERENCE-OUT.  
EXIT-PARAGRAPH.  
    EXIT PROGRAM.  
DEFAULT-HALT.  
    STOP RUN.
```

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. CHATR.  
ENVIRONMENT DIVISION.  
CONFIGURATION SECTION.  
SOURCE-COMPUTER. PDP-11.  
OBJECT-COMPUTER. PDP-11, SEGMENT-LIMIT IS 3.  
DATA DIVISION.  
LINKAGE SECTION.  
77 FIRST-NUMBER          PIC S999  
    SIGN LEADING SEPARATE CHARACTER.  
77 SECOND-NUMBER         PIC S999  
    SIGN LEADING SEPARATE CHARACTER.  
77 NUMBER-1               PIC S999 USAGE COMP.  
77 NUMBER-2               PIC S999 USAGE COMP.  
77 SUMM                  PIC S9(6) USAGE COMP.  
77 PRODUCT PIC S9(6) USAGE COMP.  
77 DIFFERENCE             PIC S9(6) USAGE COMP.  
77 SUM-OUT PIC -Z(5)9.  
77 PRODUCT-OUT            PIC -Z(5)9.  
77 DIFFERENCE-OUT         PIC -Z(5)9.  
PROCEDURE DIVISION USING FIRST-NUMBER, SECOND-NUMBER, SUM-OUT,  
PRODUCT-OUT, DIFFERENCE-OUT.  
MD3 SECTION 5.  
MAIN-MODULE.
```

TASK BUILDER COMMANDS

```
DISPLAY "THE NUMBERS YOU SELECTED WERE ", FIRST-NUMBER,  
      " AND ", SECOND-NUMBER.  
DISPLAY "THE SUM OF THESE NUMBERS IS ", SUM-OUT.  
DISPLAY "THE PRODUCT OF THESE NUMBERS IS :, PRODUCT-OUT.  
DISPLAY "THE RESULT OF SUBTRACTING THE SECOND NUMBER FROM",  
      " THE FIRST IS ", DIFFERENCE-OUT.  
DISPLAY "USER PROGRAM ENDS.". .  
EXIT-PARAGRAPH.  
EXIT PROGRAM.  
DEFAULT-HALT.  
STOP RUN.
```

2.8.2 Compiling the Programs

The BASIC-PLUS-2 programs are compiled by the following sequence:

```
RUN $BASIC2  
OLD USER.B2S  
COMPILE /OBJ  
OLD INTRO.B2S  
COMPILE /OBJ  
OLD CRUNCH.B2S  
COMPILE /OBJ  
OLD CHATTR.B2S  
COMPILE /OBJ
```

After the call to the BASIC-PLUS-2 compiler, the first command of each pair brings the source code into memory. The second command directs the compiler to translate the source code and place the relocatable object code in the associated object file. The remaining commands perform similar actions for the source files INTRO, CRUNCH, and CHATTR.

The equivalent commands for COBOL users are shown below:

```
CBL CUSER.OBJ,CUSER.LST=CUSER.CBL/KER:CU  
CBL CINTRO.OBJ,CINTRO.LST=CINTRO.CBL/KER:CI  
CBL CCRNCH.OBJ,CCRNCH.LST=CCRNCH.CBL/KER:CR  
CBL CCHATR.OBJ,CCHATR.LST=CCHATR.CBL/KER:CH
```

Note that the letter C was used as a prefix for the filenames and that the spelling of CRUNCH was changed. This is solely for the purpose of differentiating the BASIC-PLUS-2 object module and task image files from their COBOL equivalents in this section.

The /KER:xx switch in the COBOL version must be used if you are linking two or more COBOL object modules. The COBOL compiler cannot generate PSECTS (see Section 4.2.1) with unique PSECT names across two or more object modules because it can only compile one source program at a time. The /KER:xx switch (see your PDP-11 COBOL User's Guide,) lets you specify two characters of the PSECT names that the COBOL compiler creates. If the /KER:xx switch arguments you use are different for each object module, you will avoid multiple definition errors.

2.8.3 Task-Building the Programs

Building your task is the last thing you have to do before trying the first execution. To build the task, you need to create Task Builder commands in one of three ways:

TASK BUILDER COMMANDS

- Through the BASIC-PLUS-2 BUILD command
- Through a text editor
- Through direct instructions to the Task Builder

Task Builder commands for the example program USER were formatted through the creation of an overlay description file (see Chapter 5) that was used as Task Builder input. In the BASIC-PLUS-2 version of USER, the overlay description was created through the BUILD command and adjusted to fit the trident-shaped structure by text editor. In the COBOL version, the overlay description was created by the system program CBLMRG, the MERGE program. The overlay description files are shown in Figures 5-13 and 5-14 respectively.

The commands that actually built the task are

TKB USER,USER=USER/MP
and
TKB CUSER,CUSER=CUSER/MP

for the BASIC-PLUS-2 and COBOL versions, respectively.

2.9 SYNTAX RULES

Here are syntax rules for the interaction between you and the Task Builder. They define in a more formal and concise way the syntax of the commands already described in this chapter.

- Task Builder syntax takes the following forms:
 1. A task-building command can have one of several forms. The first form is a single line:

TKB task-command-line

TASK BUILDER COMMANDS

The second form has additional lines for input file names:

```
TKB (RET)
TKB>task-command-line
TKB>input-line
.
.
.
TKB>terminating-symbol
```

The third form allows the specification of options:

```
TKB (RET)
TKB>task-command-line
TKB>/
ENTER OPTIONS:
TKB>option-line
.
.
.
TKB>terminating-symbol
```

The fourth form has both input lines and option lines:

```
TKB (RET)
TKB>task-command-line
TKB>input-line
.
.
.
TKB>/
ENTER OPTIONS:
TKB>option-line
.
.
.
TKB>terminating-symbol
```

NOTES

1. The terminating symbol can be:

```
/ if more than one task is to be
built, or

// if control is to return to the
system level
```

2. No wild cards are permitted.

2. A task-command-line has one of the three forms:

```
output-file-list=input-file, ...
= input-file, ...
@indirect-file
```

where indirect-file is a file-specification as described in Section 2.5.

TASK BUILDER COMMANDS

3. An output-file-list has one of the three forms:

```
task-file, map-file, symbol-file  
task-file, map-file  
task-file
```

where task-file is the file specification for the task image file; map-file is the file specification for the memory allocation file; and symbol-file is the file specification for the symbol definition file. Any of the specifications can be omitted, so that, for example, the form:

```
task-file,,symbol-file  
is permitted.
```

4. An input-line has either of the forms:

```
input-file,...  
@indirect-file
```

where input-file and indirect-file are file-specifications.

5. An option-line has either of the forms:

```
option!...  
@indirect-file
```

where indirect-file is a file specification.

6. An option has the form:

```
keyword=argument-list,...
```

where the argument-list is

```
arg:...
```

The syntax for each option is given in Chapter 3.

7. A file-specification consists of a filename that conforms to standard RSTS/E conventions, plus optional Task Builder switches. It has the form

```
device:[project,programmer]filename.extension/sw...
```

where everything is optional except the filename. The components are defined as follows:

device is the name of the device that the volume containing the desired file is mounted on. The device name consists of two ASCII characters followed by an optional 1- or 2-digit decimal unit number; for example, LP or DT1. Logical device names of up to six alphanumeric characters may also be used.

TASK BUILDER COMMANDS

[project, programmer] is the project-programmer identification number associated with the file. The default is your own PPN.

filename is the name of the desired file. The file name can contain up to six alphanumeric characters.

extension is the 3-character filename extension. Files having the same name but a different function can be distinguished from one another by the file extension; for example, CALC.TSK and CALC.OBJ.

/sw is a switch specification. More than one switch can be used, each separated from the previous one by its slash (/). The switch name is a 2- to 4-character alphanumeric code that identifies the switch and shows whether or not it is negated. The permissible switches and their syntax are given in Chapter 3.

The device, the PPN, the extension, and the switch specifications are all optional. The following default assumptions apply to missing components of a file-specification:

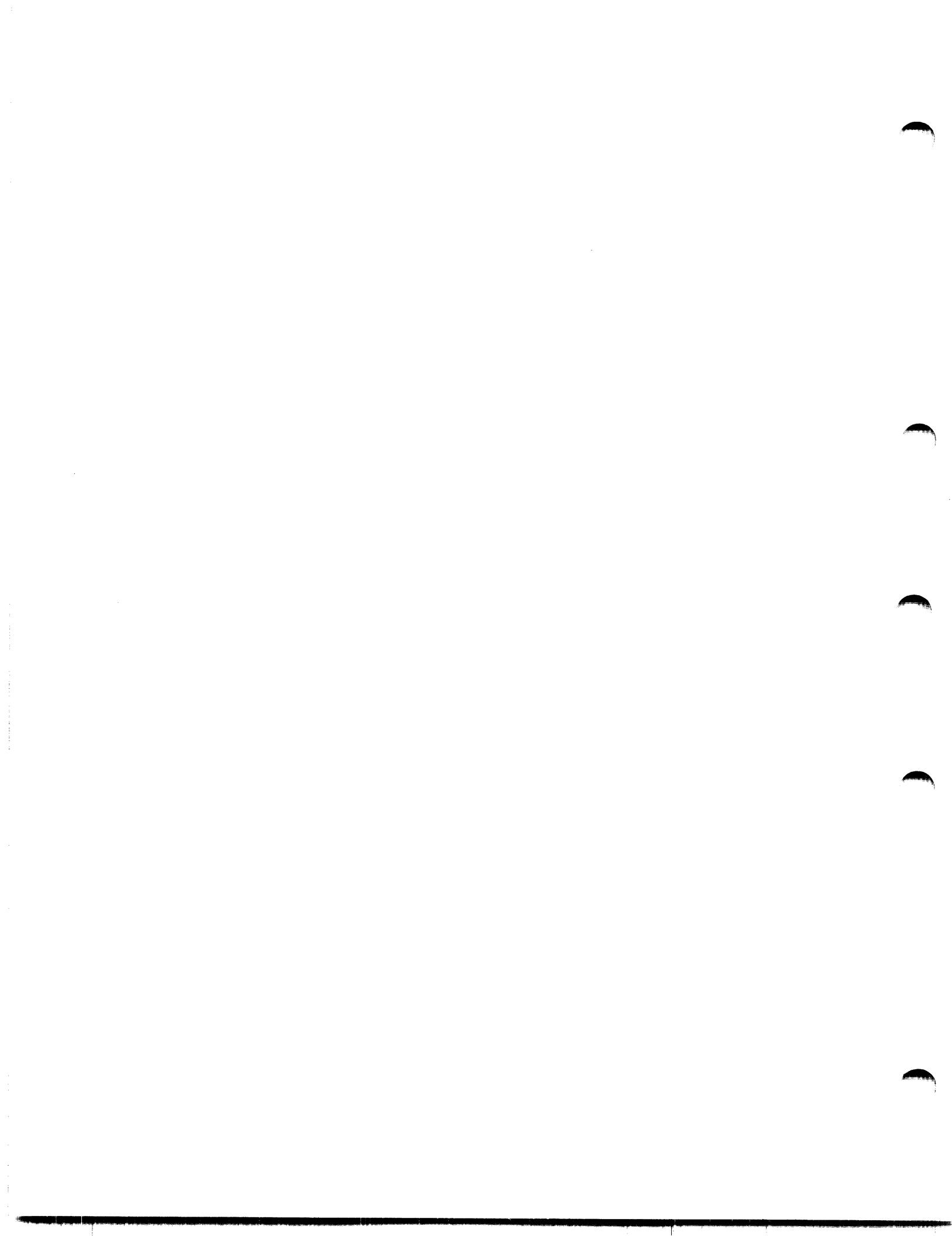
Item	Default
device ¹	The device last specified (SY:, if none)
project-programmer number ¹	the project-programmer number last specified (your own, if there is no previous entry)
extension	(See Table 2-1.)
switch	(See Chapter 3.)

For example:

DK1:IMG1,MPL=IN1,DB0:IN2,IN3

Device	File
DK1	IMG1.TSK
DK1	MPL.MAP
SY	IN1.OBJ
DB0	IN2.OBJ
DB0	IN3.OBJ

¹ When appearing in an overlay description, the default device is always SY:, and the default PPN is your own. (For a discussion of PPNs, consult the RSTS/E System User's Guide. Overlays and overlay descriptions are discussed in Chapter 5 of this manual.)



CHAPTER 3

SWITCHES AND OPTIONS

Switches and options give you more control over the construction of a task image. Many of the functions described here deal with topics discussed more fully later on. These functions are given here to demonstrate the range of functions available and to serve as a reference.

This chapter covers the following major topics.

- Switches
- Options
- Aborts and rebuilding

3.1 SWITCHES

The syntax for a RSTS/E Task Builder file specification is:

device:[project,programmer]filename.extension/sw-1/sw-2.../sw-n

The file specification ends with one or more switches (sw-1, sw-2,...sw-n) from Table 3-1. When a switch is not given in a file specification, the Task Builder uses the default setting for the switch for that file only.

Task Builder recognizes a 2-character alphabetic code that is preceded by a slash as a switch name. If the switch name is preceded by a minus sign (-) or the letters NO, the function of the switch is negated. Either method of negating a switch is acceptable. For example, if the switch is /DA (the task contains a debugging aid), then the switch settings Task Builder recognizes are:

- | | |
|-------|------------------------------------|
| /DA | The task contains a debugging aid. |
| /-DA | The task has no debugging aid. |
| /NODA | The task has no debugging aid. |

The switch codes allowed by the Task Builder are given in alphabetical order in Table 3-1. After the alphabetical listing, a more detailed description is given for each switch.

The following information is given for each switch:

- the switch name and meaning
- the file type(s) to which the switch can be applied
- the default value used if the switch is not present

SWITCHES AND OPTIONS

Table 3-1
Task Builder Switches

SWITCH NAME	MEANING	FILE TYPE ¹	DEFAULT
/CC	Input file consists of concatenated object modules.	I	/CC
/CM	Memory resident overlays are aligned on 256-word boundaries.	T	/-CM
/DA	Task contains a debugging aid	T, I	/-DA
/DL	Specified library file is a replacement for the system object library.	I	/-DL
/FP	Task uses Floating Point Processor.	T	/FP
/FU	All co-tree overlay segments are searched for matching definition or reference when modules from the default object module library are being processed.	T	/-FU
/HD	Task image includes a header.	T, S	/HD
/LB	Input file is a library file.	I	/-LB
/MA	Memory allocation output includes information from the file.	M, I	²
/MP	Input file contains an overlay description	I	/-MP
/PI	Task is position independent.	T, S	/-PI
/PM	Post-mortem dump requested.	T	/-PM
/RO	Memory resident overlay operator (!) is enabled.	T	/RO
/SH	Short memory allocation file is produced	M	/SH
/SQ	Task PSECTS are allocated sequentially.	T	/-SQ
/SS	Selective search for global symbols.	I	/-SS
/WI	Memory allocation file is printed at a width of 132 characters.	M	/WI
/XT:n	Task Builder exits after n diagnostics.	T	/-XT

1 T task image file
 M memory allocation file
 S symbol definition file
 I input file

2 The default is /MA for an input file, and /-MA for system and resident library .STB files.

SWITCHES AND OPTIONS

3.1.1 /CC (Concatenated Object Modules)

file: input

meaning: The file contains more than one object module and the modules are positioned together within the file.

effect: The Task Builder includes in the task image all the modules in the file. If this switch is negated, the Task Builder uses only the first module in the file.

default: /CC

NOTE

Switch /LB overrides this switch. See Section 3.1.15.

3.1.A /CM (Compatibility Mode Overlay Structure)

file: task image

meaning: The task is built in compatibility mode.

effect: The memory resident overlay segments are aligned on 256-word boundaries to ensure compatibility with other implementations of the mapping directives.

default: /-CM

3.1.2 /DA (Debugging aid)

file: task image or input

meaning: The task contains a debugging aid.

effect: The Task Builder performs the special processing described in Appendix G. If this switch is applied to the task image file, the Task Builder automatically includes the system debugging aid SY:[1,1]ODT.OBJ in the task image.

default: /-DA

3.1.3 /DL (Default Library)

file: input

meaning: The file is a replacement for the system object module library.

SWITCHES AND OPTIONS

effect: The specified library replaces the file

SY:[1,1]SYSLIB.OLB

as the library that is searched to resolve undefined global references. This file is referenced only when undefined symbols remain after all other user-specified files have been processed. The DL switch can be applied to only one input file.

default: /-DL

3.1.4 /FP (Floating Point)

file: task image

meaning: The task uses the Floating Point Processor.

effect: This switch directs the RSTS/E monitor to save the state of the Floating Point Processor.

default: /FP

NOTE

Do not negate this switch on systems that have the Floating Point Processor. This switch has no effect on systems without a Floating Point Processor.

3.1.5 /FU (Full Search)

file: task image

meaning: When processing modules from the default object module library, the Task Builder searches all co-tree overlay segments for a matching definition or reference.

effect: If the switch is negated, unintended global references between co-tree overlay segments are eliminated. Definitions of global symbols from the default library are restricted in scope to references in the main root and the current tree. Certain RMS-11 tasks may require you to use this switch.

default: /-FU

SWITCHES AND OPTIONS

3.1.B /HD (Header)

file: task image or symbol definition

meaning: Includes a header in the task. A header is required for executable tasks. You must negate this switch if the task output is to be used as a resident library (see Section 7.2).

effect: The Task Builder constructs a header in the task image.

default: /HD

3.1.6 /LB (Library File)

Alternate Form: /LB:mod-1:mod-2:...mod-8

file: input

meaning: 1. If the switch is applied without arguments, the input file is assumed to be a library file of relocatable object modules. These modules are to be searched for the resolution of undefined global references.

2. If the switch is applied with arguments, the input file is assumed to be a library file of relocatable object modules from which up to eight modules named in the argument list are to be taken for inclusion in the task image.

effect: 1. If no arguments are specified, the Task Builder searches the file to resolve undefined global references. It then takes from the library the modules that contain definitions for these undefined references.

2. If arguments are specified, the Task Builder includes only the named modules in the task image.

NOTES

1. If you want the Task Builder to search a library file both to resolve global references and to select named modules for inclusion in the task image, you must name the library file twice. Name the library file once with arguments for the names of the modules you want included. Name the library file the second time with no arguments but with the /LB switch to get the Task Builder to search the file for undefined global references.
2. You can use the /SS switch (see Section 3.1.12) with the /LB switch to perform a selective search for global definitions.

SWITCHES AND OPTIONS

3. This switch overrides /CC. See Section 3.1.15.

default: /-LB

3.1.7 /MA (Map Contents of File)

file: input or memory allocation

meaning: The Task Builder will include information from the file in the memory allocation output.

effect: Global symbols defined or referenced by the file are displayed in the memory allocation file and global cross-reference. The file is listed in the File Contents section of the memory allocation listing.

When applied to the allocation file, the switch controls the display of information about the default system library or symbol table file that is associated with memory-resident shared regions.

default: for input file, /MA
for system library and resident library .STB files, /-MA

3.1.8 /MP (Overlay Description)

file: input

meaning: The input file describes an overlay structure for the task.

effect: The Task Builder receives all the input file specifications from this file and allocates memory as directed by the overlay description (ODL). It then automatically requests option information by displaying ENTER OPTIONS:. Overlays are discussed in Chapter 5.

NOTES

1. DO NOT type a slash (/) terminator on the line after the **RET** following the /MP switch unless you want to start a new task. The Task Builder automatically prompts for option input after the **RET** following the ODL file specification.

2. When you specify an overlay description file as the input file for a task, it must be the only input file. The Task Builder accepts only one input file in this case.

default: /-MP

SWITCHES AND OPTIONS

3.1.C /PI (Position Independent)

file: task image or symbol definition

meaning: The task or resident library contains only position independent code or data.

effect The Task Builder sets the Position Independent Code (PIC) attribute flag in the task label block flag word. Section 7.2.1 discusses position independent resident libraries.

default: /-PI

3.1.9 /PM (Post Mortem Dump)

file: task image

meaning: If the task terminates abnormally, the system automatically writes the contents of task memory on a disk file created for that purpose. For this file to be read, it must be formatted by the PMDUMP program (refer to the RSTS/E System User's Guide). The name of the file is:

PMDnnn.PMD

where:

nnn is your job number.

effect: The Task Builder sets the post-mortem dump flag in the flag word in the label block group (see Figure D-2, bytes 30 and 31).

default: /-PM

3.1.D /RO (Resident Overlay)

file: task image

meaning: Enables recognition of the memory resident overlay operator (!). See Section 5.1.3.1.

effect: When the memory resident overlay operator is present in the overlay description file (.ODL), the Task Builder uses the operator to construct a task image that contains one or more memory resident overlay segments. If you negate this switch, the Task Builder checks the operator syntactically but does not construct memory resident overlay segments.

default: /RO

SWITCHES AND OPTIONS

3.1.10 /SH (Short Map)

file: memory allocation

meaning: The Task Builder produces a shortened version of the memory allocation file. Chapter 4 describes the memory allocation file.

effect: The Task Builder does not produce the file contents section of the memory allocation file.

default: /SH

3.1.11 /SQ (Sequential)

file: task image

meaning: The Task Builder constructs the task image from the specified PSECTs in the order in which they are accessed.

effect: The Task Builder does not reorder the PSECTs alphabetically. Section 4.2 describes task image allocation.

default: /-SQ

CAUTION

Do not use the /SQ switch on RMS-11 tasks. RMS-11 assumes that PSECTs are arranged alphabetically. See also Section 4.2.3 for other reasons why the use of /SQ is not advised.

3.1.12 /SS (Selective Search)

file: input

meaning: Do not include a global symbol definition from this module unless a previously undefined reference to the global symbol exists.

effect: The Task Builder searches the Global Symbol Table for each global symbol defined in the module. If an undefined reference to a symbol is found, the corresponding definition is included. When applied to a library or a concatenated object file, the switch applies to every module in the file.

default: /-SS

SWITCHES AND OPTIONS

3.1.13 /WI (Wide Listing Format)

file: memory allocation

meaning: Print the memory allocation file in wide (132-character) format.

effect: The listing width is expanded to fill a 132-column hard copy output device. Negating this switch normally produces 80-column hard copy, but see the NOTE below.

default: /WI

NOTE

Some systems are installed so that even if you negate the /WI switch, you will still get 132-column hard copy. See your system manager for details.

3.1.14 /XT[:n] (Exit on Diagnostic)

file: task image

meaning: More than n error diagnostics is not acceptable.

effect: The Task Builder exits after n error diagnostics have been produced. The number of diagnostics can be specified as a decimal or octal number by using the convention:

n. indicates a decimal number (the decimal point must be included)

#n or n indicates an octal number

The default value for n is 1.

default: /-XT

3.1.15 Conflicting Switches /LB and /CC

The /LB and /CC switches conflict when applied to the same file. If you use both switches, the Task Builder applies the overriding switch. Switch /LB overrides switch /CC. A comparison of the functions of the two switches reveals the reason for the override.

In this example:

```
TKB IMG5=IN6,IN5/LB/CC
```

the input file IN5 is assumed to be a library file to be searched for undefined global references, not an input file with several object modules.

SWITCHES AND OPTIONS

3.2 OPTIONS

The Task Builder offers 18 options, which provide information about the task being built. These options can be divided into six categories. Brief descriptions of these categories with their identifying mnemonics are listed below:

1. contr Control options are used to affect Task Builder execution. ABORT is the only member of this category.
2. ident Identification options are used to identify task characteristics to the Task Builder. TASK and PAR are members of this category.
3. alloc Allocation options are used to change the way the task is laid out in memory. You can specify or adjust the size of the stack, the size of the PSECTs in the task, and the number and size of work areas and buffers used by programs written in higher-level languages. EXTSC, EXTTSK, WNDWS, and STACK are members of this category.
4. share Storage sharing options tell the Task Builder that you intend to use a shared run-time system or resident library. HISEG, COMMON, LIBR, RESCOM, and RESLIB are members of this category.
5. device Device-specifying options give the number of units required by the task. These options also assign logical unit numbers to physical devices. ASG and UNITS are members of this category.
6. alter Content-altering options define a global symbol and value or introduce patches in the task image. ABSPAT, GBLDEF, GBLPAT, and GBLREF are members of this category.

Table 3-2 lists all the options alphabetically and gives a brief description of each. The options are then broken down by category and described in more detail in Sections 3.2.1 through 3.2.6.

SWITCHES AND OPTIONS

Table 3-2
Task Builder Options

Option	Meaning	Category
ABORT	Abort the building of the task	contr
ABSPAT	Declare absolute patch value(s)	alter
ASG	Declare device assignment to logical unit(s)	device
COMMON	Declare task's intention to access a memory resident library	share
EXTSCT	Declare extension of a PSECT	alloc
EXTTSK	Declare extension of the amount of memory owned by a task	alloc
GBLDEF	Declare global symbol definition(s)	alter
GBLPAT	Declare patch value(s) relative to a global symbol	alter
GBLREF	Declare global symbol reference(s)	alter
HISEG	Associate the task with a specific high segment (run-time system)	share
LIBR	Declare task's intention to access a memory resident library	share
PAR	Declare partition name and dimensions	ident
RESCOM	Declare task's intention to access a memory resident library	share
RESLIB		
STACK	Declare the size of the stack	alloc
TASK	Declare the name of the task	ident
UNITS	Declare the maximum number of units	device
WNDWS	Declare the number of address windows required by the resident library	alloc

SWITCHES AND OPTIONS

3.2.1 Control Option

3.2.1.1 **ABORT** (Abort the Building of the Task) - ABORT is useful when you discover that an earlier error in the terminal sequence will cause the Task Builder to produce an unusable task image. When the Task Builder recognizes the ABORT command, it stops accepting input for the task being built and prepares to accept input for a new task. You can now, if you wish, rebuild the task you just ABORTed. Section 3.3.1 contains an example of the use of the ABORT option.

syntax: ABORT = n

where n is an integer. (The integer is required to satisfy the general form of an option, but the value is ignored.)

default: (none)

NOTE

Typing a CTRL/Z at any time causes the Task Builder to stop accepting input and start building the task at hand. But ABORT is the only proper way to restart the Task Builder, if you find an error and do not want the resulting Task Builder output.

3.2.2 Identification Options

3.2.2.1 **TASK** (Task Name) - The TASK option specifies the name of the task being built. This name is displayed by the SYSTAT program. You can use this option if you wish to give a name to a task other than the name of the task image file. There is an example of how to use the TASK option in Section 2.4.

syntax: TASK = task-name

where task-name is a 1- to 6-character name from the Radix-50 set identifying the task.

default: the task image filename

3.2.2.2 **PAR** (Partition)

The PAR option identifies the partition (the area of memory within the job's virtual address space) for which the resident library task image is built and allows you to specify a base address and length for the library.

SWITCHES AND OPTIONS

syntax: PAR=pname[:base:length]

where;

pname is the name of the partition.

base is the octal byte address that defines the start of the partition. If the library is position independent (see Section 7.2.1), the base address is zero. If the library is not position independent, the base address is non-zero.

length is the octal number of bytes contained in the partition.

The Task Builder automatically extends the task size of the resident library to make up the difference between the length specified for the partition and the amount of memory required by the task. A length of zero signifies that the task size is to equal the memory required.

If the task size is greater than the partition size, the Task Builder issues the following error message:

%TKB--*DIAG*-TASK HAS ILLEGAL MEMORY LIMITS

You must specify a partition name if the task is to be used as a resident library. If you do not specify a partition base address or length, the library is position independent, and the Task Builder assigns a base of 0 and a length that equals the size of the created task image.

The Task Builder attaches the task to the address defined by the partition base and verifies that the task does not exceed the length specification (if made).

To ensure that a usable task image is produced, the Task Builder must consider two types of task: executable task images and resident libraries. An executable task image must have a header and is capable of direct execution. A resident library must not have a header and is not directly executable. An executable task on RSTS/E cannot be larger than 28K. However, if the task is built under the RSX Run-Time System and executed with a Monitor that has RSX emulation support, you can extend the task up to 31K. The Task Builder enforces address limits according to the type of task, as follows:

Executable Task		Resident Library
base	0	on 4K boundary
length	multiple of 32 words	multiple of 32 words
high address bound	(28K) words	(32K) words

SWITCHES AND OPTIONS

Refer to Section 7.2.1 for a description of the PAR option in command lines that create resident libraries.

3.2.3 Allocation Options

3.2.3.1 EXTSCT (PSECT Extension) - The EXTSCT option declares an extension in size for a PSECT in an input object file or in the overlay description file. PSECTS and their attributes are described in Section 4.2.1.

If the PSECT has the CON (concatenated) attribute, the PSECT is extended by the specified number of bytes. If the PSECT has the OVR (overlay) attribute, the section is extended by the length of the extension if that extension is greater than the previously established length of the PSECT.

syntax: EXTSCT = PSECT-name:extension

where: PSECT-name is the 1- to 6-character name from the Radix-50 set of the PSECT to be extended.

extension is the octal number of bytes by which to extend the PSECT.

default: none

In the following example, PSECT BUFF is initially 200 bytes long:

EXTSCT = BUFF:250

The new size of the PSECT depends on the CON/OVR attribute:

- For CON the extension is an additional 250 bytes for a total of 450 bytes.
- For OVR the extension is an additional 50 bytes for a total of 250 bytes.

3.2.3.2 EXTTSK (Extend Task Memory) - The EXTTSK option declares the amount of additional memory to be allocated to the task up to a maximum of 28K words.

The amount of memory available to the task is the sum of the task size plus the increment specified in the EXTTSK keyword (rounded up to the nearest 32-word boundary).

syntax: EXTTSK = length

where: length is a decimal number specifying the increase in task memory allocation in words.

SWITCHES AND OPTIONS

default: The task is extended to the next multiple of 1K words.

3.2.3.3 **STACK** (Stack Size) - The STACK option declares the maximum size of the stack required by the task.

The stack is an area of memory used for temporary storage, subroutine calls, and other system functions. The stack is referenced by hardware register 6 (the stack pointer).

syntax: STACK = stack-size

where: stack-size is a decimal integer specifying the number of words required for the stack.

default: STACK = 256

CAUTION

Decreasing the size of the stack to less than the default size can lead to unpredictable or fatal results in certain higher level languages.

3.2.3.4 WNDWS (Number of Address Windows)

The WNDWS option declares the number of address windows required by the resident library in addition to those already declared (by default) to map the task image, any mapped array, or resident library.

syntax: WNDWS=n

where:

n is an integer in range of 0 to 7.

If you do not specify a number of address windows, the Task Builder assigns zero to the option. Note that the number of address windows must be equal to the number of simultaneously mapped memory regions that the task will use.

3.2.3.4 Example of Allocation Options - In the following example, the size of PSECT AAAAAA is expanded to 20000 (octal) bytes.

The terminal sequence used to build the task is:

```
TKB
TKB>IMG1,MP1=GRF1
TKB>/
ENTER OPTIONS:
TKB>EXTSCT=AAAAAA:20000
TKB>//
```

SWITCHES AND OPTIONS

3.2.4 Storage Sharing Options

3.2.4.1 HISEG (High Segment) - The HISEG option associates the task image with a high segment, or run-time system, of the name specified. The symbol table of the high segment is automatically included to resolve global references. The .STB file for the named high segment must be in the account specified by the system logical name LB:. If the HISEG option is not specified:

- The high segment associated with the task image is the same as that associated with the Task Builder itself.
- No global references to symbols in that high segment are resolved.

syntax: HISEG = high-segment-name

where: high-segment-name is a 1- to 6-character name from the Radix-50 set specifying the run-time system.

default: the Task Builder high segment

3.2.4.2 COMMON (System Common Block) or LIBR (System Resident Library)

By convention, the COMMON option indicates a resident library that contains data; the LIBR option indicates a resident library that contains code.

These options are identical and each declares a resident library for use by your task.

syntax: COMMON=name:access code[:apr]
LIBR=name:access code[:apr]

where;

name is the 1- to 6-character Radix-50 name (from the Radix-50 character set) of the resident library you wish to attach to your task. The Task Builder expects to find a symbol table file and task image file of the same name (name.STB and name.TSK) under the account specified by the system logical name LB:.

access code is the code RW (for read/write) or RO (for read only) and indicates the type of access required by your task.

SWITCHES AND OPTIONS

apr is an integer in the range of 1 to 7 that specifies the first Active Page Register reserved for the library. The APR can be omitted. It is specified only if the resident library is position independent.

There is no default for this option

3.2.4.3 RESCOM (Resident Common Block) or RESLIB (Resident Library)

By convention, the RESCOM option indicates a resident library that contains data; the RESLIB option indicates a resident library that contains code.

These options are identical and each declares a resident library for use by your task, and unlike COMMON or LIBR, they allow you to include a file specification. Note that comments must not appear on an option line with RESCOM or RESLIB nor can you specify a device name and unit number. However, you can specify an account, filename, and extension on the option line.

syntax: RESCOM=file spec/access code[:apr]
 RESLIB=file spec/access code[:apr]

where;

file spec is the file specification of the resident library.

access code is the code RW (for read/write) or RO (for read only) and indicates the type of access required by your task.

apr is an integer in the range of 1 to 7 that specifies the first Active Page Register to be reserved for the library. The APR can be omitted. It is specified only for position independent libraries.

The Task Builder expects to find a symbol definition file (name.STB) and a task image file (name.TSK) of the same name as the specified resident library on the public disk structure in the account you specify in the file specification. If you do not specify an account, the Task Builder searches the account associated with your task. That is, the Task Builder assigns the current account on the public structure, and a file extension of .TSK as defaults for the file specification.

3.2.4.4 Examples of Resident Library Switches and Options

In the following example, the task is composed of the MACRO-11 programs TST1 and TST. The task accesses the resident library, DTST, which contains data, and the resident library, STST, which contains code.

SWITCHES AND OPTIONS

The Task Builder command lines used to build the task are as follows:

```
RUN $TKE
TKE>TST,TST=TST1,TST2
TKE>/
ENTER OPTIONS:
TKE>COMMON:D:TST:RW
TKE>LIBR:S:TST:RO
TKE>/
```

In a similar fashion, the RESLIB and RESCOM options can be used to link the task to user-created resident libraries. For example:

```
RUN $TKB
TKB>TST,TST=TST1,TST2
TKB>/
ENTER OPTIONS:
TKB>RESCOM=[20,20]DTST/RW
TKB>RESLIB=[20,20]STST/RO
TKB>/
```

3.2.5 Device Specifying Options

The two options in this category are UNITS and ASG. The UNITS option declares the maximum number of input/output units that the task uses. The ASG option declares the devices that are assigned to these units.

A logical unit number, or LUN, is assigned to each file or device used by the task. (Each LUN corresponds to a channel number in BASIC-PLUS and BASIC-PLUS-2 terminology.) The LUN provides the link between the filename and the channel the file is associated with.

The number of logical units and the highest unit number assigned must be compatible. An attempt to assign a physical device to a logical unit number that is larger than the total number of units declared is an error. Conversely, the number of units declared cannot be fewer than the highest-numbered logical unit assigned.

NOTE

To increase the number of units and assign devices to these units, you should enter the UNITS option first and then the ASG option. Because the options are processed as they are encountered, entering the options in the reverse order can produce an error message.

3.2.5.1 UNITS (Logical Unit Usage) - The UNITS option declares the number of logical units that are used by the task.

SWITCHES AND OPTIONS

syntax: UNITS = max-units

where: max-units is a decimal integer from 0 to 14 specifying the maximum number of logical units.

default: UNITS = 6

NOTE

BASIC-PLUS-2 programmers should always set max-units to 12.

3.2.5.2 ASG (Device Assignment) - The ASG option declares the physical device that is assigned to one or more units.

syntax: ASG = device-name:unit-num-1:unit-num-2...:unit-num-8

where: device-name is a 2-character alphabetic device name followed by an optional 1- or 2-digit decimal unit number.

unit-num-1: are decimal integers indicating the Logical
unit-num-2: Unit Numbers, or LUNs.

.

.

.

unit-num-8

default: ASG = SY:1:2:3:4, TI:5, CL:6

3.2.5.3 Example of Device Specifying Options - In the following example, the BASIC-PLUS-2 programs specified in the file GRP1 require a maximum of nine logical units. The device assignments for units 1-6 agree with the default assumptions. Logical units 7 and 8 are assigned to magtape 1 (MT1) and unit 9 is assigned to magtape 2. The terminal sequence of the example in Section 3.2.3.4 is changed to include device assignment options, as follows:

```
RUN $TKB
TKB>IMG1,MP1=GRP1
TKB>/
ENTER OPTIONS:
TKB>UNITS=9
TKB>ASG=MT1:7:8
TKB>ASG=MT2:9
TKB>EXTSCT=AAAAAA:20000
TKB>//
```

SWITCHES AND OPTIONS

3.2.6 Storage Altering Options

Four options alter the task image:

- GBLDEF (global symbol definition)
- GBLREF (global symbol reference)
- ABSPAT (absolute patch)
- GBLPAT (global relative patch)

The GBLDEF option declares a global symbol and value; GBLREF declares a global symbol reference. The options ABSPAT and GBLPAT introduce patches into the task image.

CAUTION

The options in this section are for use by the experienced programmer or analyst only.

3.2.6.1 GBLDEF (Global Symbol Definition) - The GBLDEF option declares the definition of a global symbol. A global symbol is a label for a data item that is defined in one module and referenced in others.

The symbol definition is considered absolute.

syntax: GBLDEF = symbol-name:symbol-value

where: symbol-name is the 1- to 6-character name from the Radix-50 set of the defined symbol.

symbol-value is an octal number in the range 0-177777 assigned to the defined symbol.

default: none

3.2.6.2 GBLREF (Global Symbol Reference) - The GBLREF option declares a global symbol reference. The reference originates in the root segment of the task. A global symbol is a label for a data item that is defined in one module and referenced in others.

syntax: GBLREF = symbol-name

where: symbol name is the 1- to 6-character name from the Radix-50 set of the global symbol reference.

default: none

SWITCHES AND OPTIONS

3.2.6.3 ABSPAT (Absolute Patch) - The ABSPAT option declares a series of patches starting at the specified base address within the specified segment. Up to eight patch values can be given.

syntax: ABSPAT = seg-name:address:val-1:val-2...:val-8

where: seg-name is the 1- to 6-character name from the Radix-50 set of the segment.

address is the octal address of the first patch. The address must be on a word boundary. Two bytes are always modified for each patch.

val-1 is an octal number in the range 0 - 177777 to be stored at address.

val-2 is an octal number in the range 0 - 177777 to be stored at address + 2 bytes.

.

.

.

val-8 is an octal number in the range 0 - 177777 to be stored at address + 16(octal) bytes.

default: none

NOTE

All patches must be within the segment address limits or a fatal error is generated.

3.2.6.4 GBLPAT (Global Relative Patch) - The GBLPAT option declares a series of patch values starting at an offset relative to a global symbol. Up to eight patch values can be given.

syntax: GBLPAT=seg-name:sym-name[+/-offset]:val-1:val-2 ...:val-8

where: seg-name is the 1- to 6-character name from the Radix-50 set of the segment.

sym-name is the 1- to 6-character name from the Radix-50 set specifying the global symbol.

offset is an octal number specifying the offset from the global symbol.

val-1 is an octal number in the range 0 - 177777 to be stored at the octal address of the first patch.

SWITCHES AND OPTIONS

```
val-2      is an octal number in the range 0 - 177777 to
           be stored at the first address + 2 bytes.
.
.
.
val-8      is an octal number in the range 0 - 177777 to
           be stored at the first address + 16(octal)
           bytes.

default: none
```

NOTE

All patches must be within the segment address limits, or a fatal error is generated.

3.2.6.5 Example of Storage Altering Options - In the following example, GAMMA is a referenced symbol whose value is specified as 25 (octal) when the task is built. Ten patch values relative to the global symbol DELTA are also introduced.

The terminal command sequence looks like this:

```
TKB>CHK,CHK=TST1,TST2
TKB>/
ENTER OPTIONS:
TKB>GBLDEF=GAMMA:25
TKB>GBLPAT=TST1:DELTA:1:5:10:15:20:25:30:35
TKB>GBLPAT=TST1:DELTA+20:40:45
TKB>//
```

3.3 ABORTS AND REBUILDING

The first execution of a task may have yielded several logical errors. After correcting the program, you are now ready to make some changes. You may also decide to make adjustments in the task image file. These adjustments are based on the information obtained about the size of the task in the first task-build.

To make the needed changes in the task image:

1. Change the text file for the program
2. Recompile (and remerge, if you are using COBOL)
3. Rebuild the task

SWITCHES AND OPTIONS

3.3.1 Aborting the Task

Rather than continue to build a task that you know will crash, you may decide that you want to abort and start over. You may also decide to abort the task when you discover that you forgot something.

Here is an example of such a situation:

```
1   TKB
2   TKB>CALC,=RDIN,PROC1,RPRT
3   TKB>/
4   ENTER OPTIONS:
5   TKB>UNITS=12
6   TKB>ABORT=1
7   TKB -- *FATAL* - TASK-BUILD ABORTED VIA REQUEST
8   TKB>CALC,CALC/SH=RDIN,PROC1,RPRT
9   TKB>/
10  ENTER OPTIONS:
11  TKB>UNITS=12
12  TKB>HISEG=BASIC2
13  TKB>/
14  TKB>                                (command mode)
```

Notice lines 6 and 7 above. The ABORT option was used to end the task-build in this example because the memory allocation file was omitted. After printing the abort message, Task Builder prompts in command mode (line 8).



CHAPTER 4

MEMORY ALLOCATION

The Task Builder allocates the physical memory and virtual address space required by a task. This allocation can consist of two parts:

1. A region containing the task itself
2. Memory that is not physically a part of the task image, but contains subroutines shared by several tasks

This chapter covers the following major topics.

- Task Memory Structure
- Task Image Memory
- Global Symbol Resolution
- Task Image File
- Memory Allocation File
- Memory Allocation Map for BASIC-PLUS-2 Version of USER
- Memory Allocation Map for COBOL Version of USER

4.1 TASK MEMORY STRUCTURE

Task memory structure (see Figure 4-1) is divided into two physically contiguous areas containing:

1. The task image
2. Additional memory allocated while the task is running. (You can allocate additional memory by using the Extend Task system directive. If you use the Task Builder EXTTSK option, RSTS/E can also extend memory.)

MEMORY ALLOCATION

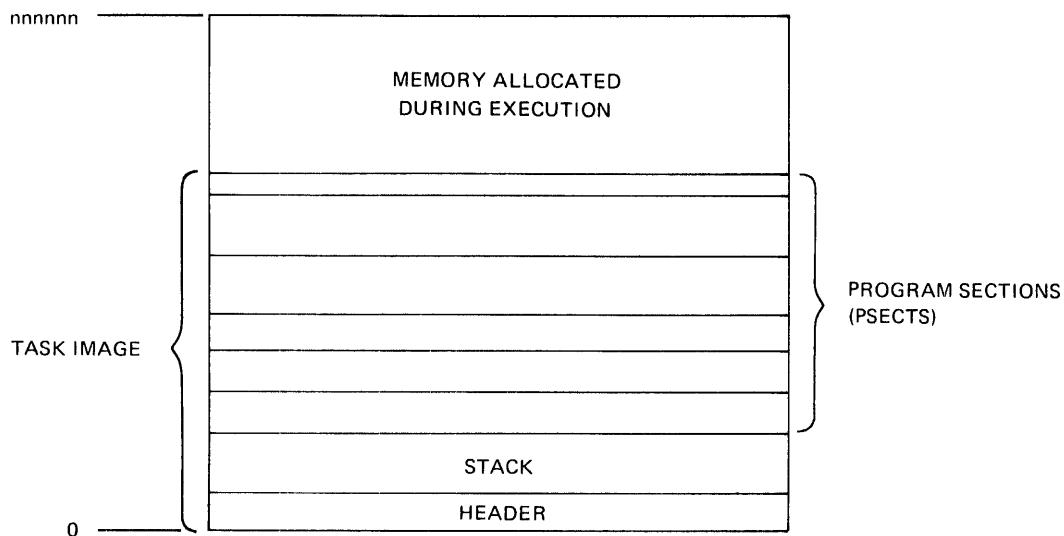


Figure 4-1 Task Memory Structure

4.2 TASK IMAGE MEMORY

The area of memory allocated for task image storage contains

1. A header,
2. A stack, and
3. A group of PSECTs (see Section 4.2.1).

The header contains task parameters and data required by RSTS/E and provides a storage area for saving the task's context. The contents of the header are described in detail in Section D.2.

Note that a header is required only for executable tasks. If you are creating a task for use as a resident library, you must omit the header from the task image. That is, you must specify a /-HD switch (negate the header) in the TKB command line to create a resident library.

The stack is an area that can be used for temporary storage and subroutine linkage. It is referenced by general register 6, the stack pointer. You can change the size of the stack by using the STACK option, as described in Section 3.2.3.3.

4.2.1 PSECTs

A program section or PSECT, of variable size, is the basic unit of task memory that contains code or data and can be referenced by name. Associated with each PSECT is a set of attributes that controls the allocation and placement of the PSECT within the task image. A PSECT is composed of the following elements:

- A name by which it is referenced
- A set of attributes that define its contents, mode of access, size allocation, and placement in memory
- A length that determines how much storage is to be reserved for the PSECT

MEMORY ALLOCATION

PSECTS are created or referenced in either of the following ways:

- Language processors automatically include PSECTS in the object module to reserve storage for code or data.
- You can explicitly create PSECTS by using facilities present in the language processors or Task Builder.

PSECTS are created through the COMMON and MAP statements in BASIC-PLUS-2, or through the association of a segment number with a section name in COBOL.

The Task Builder overlay processor allows PSECTS to be created and inserted at specific points in the overlay structure. This facility is described in Chapter 5.

As noted above, each reference to a PSECT is accompanied by a length and set of attributes that describe memory allocation to that PSECT. Task Builder collects scattered references to the PSECT in a single area of task memory. The attributes, listed in Table 4-1, control the way the Task Builder collects and places PSECT storage and determine the contents of the PSECT name entry flag byte (see Section C.1.6).

Table 4-1
PSECT Attributes

Attribute	Value	Meaning
Access code	RW	Read/Write - Data can be read from and written into the PSECT.
	RO	Read Only - Data can be read from but cannot be written into the PSECT.
Allocation code	CON	Concatenate - All references to a given PSECT name are concatenated. The total allocation is the sum of the individual allocations.
	OVR	Overlay - All references to a given PSECT name overlay each other. The total allocation is the length of the longest individual allocation.
Relocation code	REL	Relocatable - The base address of the PSECT is relocated relative to the virtual base address of the task.
	ABS	Absolute - The base address of the PSECT is not relocated. It is always zero.
Scope code	GBL	Global - The PSECT name is recognized across overlay segment boundaries. The Task Builder allocates storage for the PSECT from references outside the defining overlay segment.

(continued on next page)

MEMORY ALLOCATION

Table 4-1 (Cont.)
PSECT Attributes

Attribute	Value	Meaning
Scope code (Cont.)	LCL	Local - The PSECT name is recognized only within the defining overlay segment. The Task Builder allocates storage for the PSECT from references within the defining overlay segment only.
Type code ¹	D	Data - The PSECT contains data.
	I	Instruction - The PSECT contains either instructions, or data and instructions.

¹These codes should not be confused with the I and D space hardware codes on PDP-11 systems.

The Task Builder uses the access code and allocation code to determine the size of the PSECT and its placement in task memory.

The Task Builder divides storage into read/write and read-only memory, and places PSECTS in the appropriate area according to access code. Memory allocated to read-only PSECTS is not hardware protected.

The allocation code is used to determine the starting address and length of memory allocated by modules that reference a common PSECT. If the allocation code indicates that such a PSECT is to be overlaid, the Task Builder places the allocation from each module at the same location in task memory, and determines the total size from the length of the longest reference to the PSECT. If the allocation code indicates that a PSECT is to be concatenated, the Task Builder places the allocation of each of the modules one after the other in task memory, and determines the total allocation from the sum of the lengths of each reference.

The allocation of memory for a PSECT always begins on a word boundary. If the PSECT has the D (data) and CON (concatenate) attributes, all storage contributed by subsequent modules within that PSECT is appended to the last byte of the previous allocation. This occurs regardless of whether or not that byte is on a word boundary. Thus, the first allocation in the PSECT begins on a word boundary, but the remaining allocations may not. For a PSECT with the I (instruction) and CON attributes, however, all storage contributed by subsequent modules begins at the nearest following word boundary.

The scope code and type code are meaningful only when an overlay structure is defined for the task. The scope code is described in Chapter 5, in the context of PSECT resolution. The type code is described in Chapter 6, in the context of autoload vector generation.

4.2.2 PSECT Allocation

Here is an example of PSECT allocation:

```
TKB IMG1,MP1/SH/MA=IN1,IN2,IN3,LBRL/LB
```

MEMORY ALLOCATION

This command directs the Task Builder to build a task image file, IMG1.TSK, and a memory allocation file, MP1.MAP, from the input files IN1.OBJ, IN2.OBJ, and IN3.OBJ. It also initiates a search of the library file LBR1.OLB for any undefined global references. The input files are composed of PSECTs with access codes, allocation codes, and sizes as illustrated in Table 4-2:

Table 4-2
PSECT Allocation

Filename	PSECT Name	Access Code	Allocation Code	Size (octal)
IN1	B	RW	CON	100
	A	RW	OVR	300
	C	RO	CON	150
IN2	A	RW	OVR	250
	B	RW	CON	120
IN3	C	RO	CON	50

In Table 4-2, there are two occurrences of the PSECT named B with attributes RW and CON. The total allocation for B is the sum of the lengths of the references; that is, $100 + 120 = 220$ blocks. If the OVR attributes had been used instead of CON, as in PSECT A, the total allocation would have been 120 blocks, which is the largest allocation for PSECT B. The total allocation for each uniquely named PSECT is shown in Table 4-3.

Table 4-3
Allocation Totals

PSECT Name	Total Allocation
B	220
A	300
C	200

The Task Builder then groups the PSECTS according to their access-codes, and alphabetizes each group. Figure 4-2 shows the results:

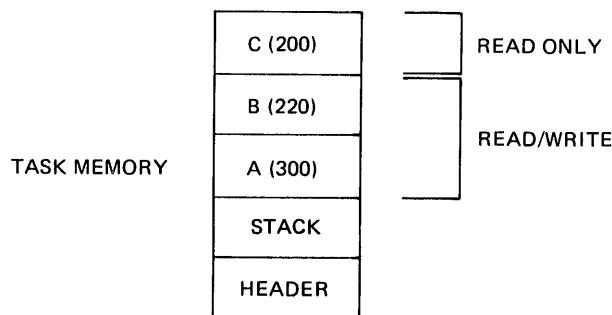


Figure 4-2 PSECT Allocations Grouped by Access Code

MEMORY ALLOCATION

4.2.3 PSECT Placement

The placement of PSECTS in task memory is affected by the /SQ (sequential) switch. References to a given PSECT from object modules are collected as described. All PSECTS are then grouped according to access-code and, within these groups, are placed in memory in the order they were input, rather than alphabetically.

The /SQ switch was intended to satisfy adjacency requirements of existing code that was previously written for another PDP-11 operating system. Using this feature is otherwise discouraged for the following reasons:

- Standard library routines will not work properly.
- Sequential allocation can result in errors if the order in which modules are linked is altered.
- RMS-11, BASIC-PLUS-2, and COBOL assume that PSECTS are arranged in alphabetical sequence.

You can place PSECTS together by selecting names alphabetically to correspond to the desired order.

4.3 GLOBAL SYMBOL RESOLUTION

When creating the task image file IMG1.TSK in the command in Section 4.2.3, the Task Builder resolves the global references shown in Table 4-4 in the following manner.

Table 4-4
Global Reference Resolution

File Name	PSECT Name	Global Definition	Global Reference
IN1	B A C	B1 B2	A1 L1 C1 XXX
IN2	A B	A1 B1	B2
IN3	C		B1

In processing the first file, IN1.OBJ, the Task Builder finds definitions for B1 and B2 and references to A1, L1, C1, and XXX. Because no definition exists for these references in IN1.OBJ, the Task Builder defers the resolution of these global symbols. In processing the next file, IN2.OBJ, the Task Builder finds a definition for A1, which resolves the previous reference, and a reference to B2, which can be immediately resolved. The third file IN3.OBJ contains a reference to B1. (The last paragraph in this section shows how references to B1 are resolved.)

MEMORY ALLOCATION

When all the object files have been processed, the Task Builder has three unresolved global references -- C1, L1, and XXX. A search of the library file LBRI resolves L1, and the Task Builder includes the defining module in the task image. A search of the System Library resolves XXX. The global symbol C1 remains unresolved and is listed as an undefined global symbol.

The relocatable global symbol B1 is defined in two different modules and is listed as a multiply defined global symbol on the terminal. The first definition of a multiply defined symbol is the one used by the Task Builder. An absolute global symbol can be defined more than once without being listed as multiply defined as long as all occurrences of the symbol have the same value.

4.4 TASK IMAGE FILE

The task image file contains a copy of the task that can be read into memory and started with little system overhead. The Task Builder does all linking, memory allocation, and address resolution. The system loads the task image and transfers control to it.

The task image file also contains a label block group. The label block group contains data that is used by the system loader when the task is run. The label block group is described in detail in Section D.1.

4.5 MEMORY ALLOCATION FILE

The memory allocation file lists information about the allocation of task memory and the resolution of global symbols.

4.5.1 Contents of the Memory Allocation File

The memory allocation file contains the following items:

- Page Header
- Task Attributes
- Overlay Description (if applicable)
- Segment Description
- Memory Allocation Synopsis
- Global Symbols
- File Contents
- Summary of Undefined Global Symbols
- Task Builder Statistics

Sample memory allocation files are shown in Figures 4-3 of Section 4.6 and 4-4 of Section 4.7, where each item is identified. The following paragraphs discuss the map items in greater detail.

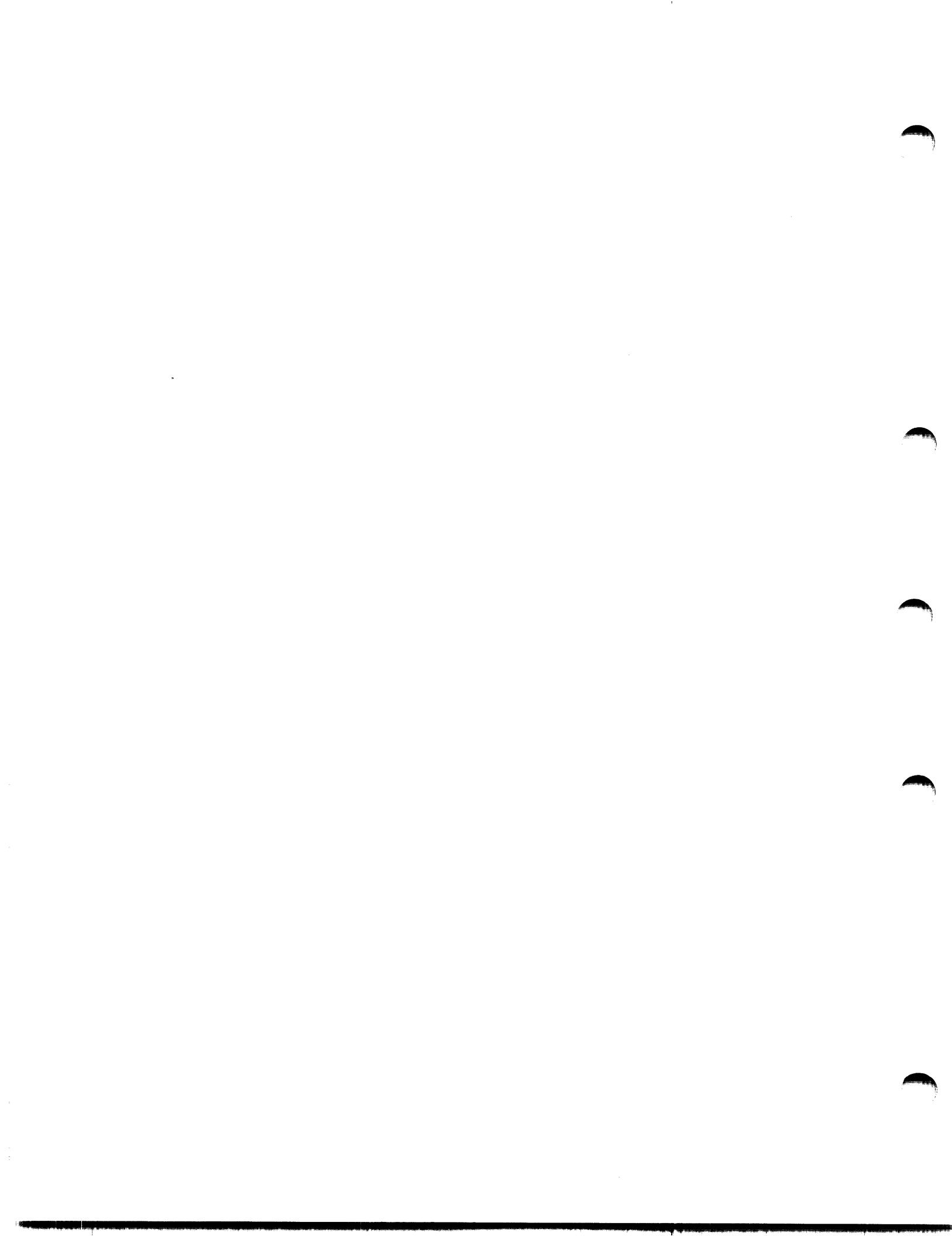
MEMORY ALLOCATION

1. The page header shows the name of the task image file and the overlay segment name, along with the date, time, and version of the Task Builder that was used.
 2. The task attribute section may contain the following information, some of which does not appear in Figure 4-3 or 4-4:
 - a. Task name
 - b. Task partition (always GEN)
 - c. Identification (task version)
 - d. Task UIC (PPN)
 - e. Stack limits -- consisting of the low and high addresses, followed by the length in octal and decimal bytes
 - f. ODT transfer address -- starting address of the debugging aid
 - g. Program transfer address
 - h. Task attributes -- shown only if you specify a switch (see Table 3-1) that differs from the default. For example, one or more of the following can be displayed:

DA	task contains debugging aid
-FP	task does not use floating-point processor
PM	task requests post-mortem dump
-HD	task does not contain a header (resident library)
PI	task contains only position independent code
CM	task built in compatibility mode
- i. Total address windows -- the number of address windows allocated to the task
 - j. Task extension -- the increment of physical memory allocated through the EXTTSK keyword
 - k. Task image -- the amount of memory required to contain task code
 - l. Total task size -- the amount of memory allocated to task extension and task image listed above
 - m. Task address limits -- the lowest and highest virtual addresses allocated to the task

MEMORY ALLOCATION

3. The overlay description shows the address limits, length, and name of each overlay segment. Indenting is used to illustrate the overlay structure. The overlay description is printed only when a multi-segment task is created.
4. The segment description gives the name of the segment, along with the segment address and disk space limits.
5. The memory allocation synopsis gives information about the PSECTs that make up the memory allocated to each overlay segment. The information shown consists of the PSECT name, attributes, starting address, and length in bytes, followed by a list of modules that contributed storage to the section. The entry for each module shows the starting address and length of the allocation, the module name, module identification, and file name.



MEMORY ALLOCATION

If the /SQ switch is applied, the PSECTs are listed in the order of input; otherwise they appear in alphabetical order.

The following PSECT information is omitted:

- a. The absolute section . ABS. is not shown because it appears in every module and always has a length of 0.
- b. The unnamed relocatable section . BLK. is not displayed if its length is 0, because it appears in every module.
6. Global symbols that are defined in the segment are listed along with their octal values. The code -R is appended to the value if the symbol is relocatable. The list is alphabetized in columns.
7. The file contents section lists the module name, the filename, and any PSECTs, and global definitions occurring in the module.
8. A summary of undefined global references is printed after the listing of file contents.
9. The display of Task Builder statistics lists the following information, which may be used to evaluate Task Builder performance.
 - Work File References -- The number of times that the Task Builder accessed data stored in its work file.
 - Work File Reads -- The number of times that the work file device was accessed to read work file data.
 - Work File Writes -- The number of times that the work file device was accessed to write work file data.
 - Size of Core Pool -- The amount of memory that was available for work file data and table storage.
 - Size of Work File -- The amount of device storage that was required to contain the work file.
 - Elapsed Time -- The amount of wall-clock time required to construct the task image and produce the memory allocation file. Elapsed time is measured from the completion of option input to the completion of map output. This value excludes the time required to process the overlay description, parse the list of input file names, and create the cross-reference listing (if specified).

Section F.1.1 contains a more detailed discussion of the work file and its relationship to task performance.

4.5.2 Control of Memory Allocation File Contents and Format

By using the memory allocation and input file switches described below, you can:

1. Eliminate nonessential information from the output
2. Improve Task Builder throughput

MEMORY ALLOCATION

3. Obtain output in a format that is more compatible with the hard copy device

The /SH (short map) and /MA (map wanted) switches control the amount of information presented in the memory allocation file. When the /SH switch is included in the map file specification, the Task Builder eliminates:

1. the file contents section of the allocation listing
2. the list of global definitions by module
3. the list of unresolved global references within each module

All other contents can be found elsewhere in the memory allocation file.

In general, the short format gives enough information for debugging, yet reduces the task-building time considerably. You can get listings that contain a full description of the file contents at less frequent intervals and keep them for later reference.

You can keep the contents of individual input files out of the listing by negating the /MA switch (/NOMA or /-MA). For each file so treated, the following information is omitted:

1. PSECT contributions as shown in the memory allocation synopsis
2. global symbol definitions
3. file contents
4. global definitions or references, and module names as shown in the cross-reference listing

To disable map output for individual files, include /NOMA in the appropriate input file specification. To disable such output for the default system object module library and all memory-resident library files, include /NOMA in the memory allocation file specification.

The width of the listing is controlled by the /WI (wide) switch. This switch is included in the map file specification to increase the listing format from 80 to 132 columns. The global symbols, overlay description, and cross-reference output are expanded to fill the additional space. Some systems are installed so that /-WI gives you 132-column output anyway. Check with your system management to be sure.

4.6 MEMORY ALLOCATION MAP FOR BASIC-PLUS-2 VERSION OF USER

The first run of the BASIC-PLUS-2 version of USER, discussed in Section 2.8, produces the memory allocation file shown in Figure 4-3. The memory map shown results from a task containing no overlays. That is, all four segments are in memory at all times. The overlaid version of the memory map is shown in Section 5.3.

The task attributes section lists the principal characteristics of interest, such as task size in words, and task address limits. Items such as task attributes, that are not specified or that do not differ from the default, have been omitted.

USER.TSK MEMORY ALLOCATION MAP TKB M26
16-JUL-77 22:44 PAGE 1

PARTITION NAME : GEN
IDENTIFICATION : V01X03
TASK VIC : [1,13]
STACK LIMITS: 001000 001777 0010000 00512.
PRG XFR ADDRESS: 002674
TOTAL ADDRESS WINDOWS: 2.
TASK IMAGE SIZE : 1568. WORDS
TASK ADDRESS LIMITS: 000000 006027

*** ROOT SEGMENT: USER

R/W MEM LIMITS: 000000 006027 006030 03096.
DISK BLK LIMITS: 000002 000010 000007 00007.

MEMORY ALLOCATION SYNOPSIS:

SECTION	TITLE	IDENT	FILE
-----	-----	-----	-----
• BLK.: (RW,I,LCL,REL,CON)	002000 000674 00444.	\$JPADD 01CM	BASIC2.0LB
	002000 000046 00038.	\$CALLS 02CM	BASIC2.0LB
	002046 000464 00308.	\$JPMOV 02CM	BASIC2.0LB
	002532 000074 00060.	\$JPSSUB 01CM	BASIC2.0LB
\$ARRAY: (RW,D,LCL,REL,CON)	002626 000046 00038.		
	002674 000000 00000.		
	002674 000000 00000.	USER V01X03	USER.OBJ
	002674 000000 00000.	INTRO V01X03	INTRO.OBJ
\$CODE : (RW,I,LCL,REL,CON)	002674 000674 00444.	CRUNCH V01X03	CRUNCH.OBJ
	002674 000344 00228.	CHATR V01X03	CHATR.OBJ
	003240 000110 00072.		
	003350 000110 00072.		
\$FLAGR: (RW,D,GBL,REL,CON)	003460 000110 00072.		
	003570 000000 00000.	USER V01X03	USER.OBJ
	003570 000000 00000.	INTRO V01X03	INTRO.OBJ
	003570 000000 00000.	CRUNCH V01X03	CRUNCH.OBJ
	003570 000000 00000.	CHATR V01X03	CHATR.OBJ

Figure 4-3 Memory Allocation File for BASIC-PLUS-2 Version of USER

MEMORY ALLOCATION

```

$FLAGS: (RW,D,GBL,REL,CON) 003570 000010 00008. USER VOLX03 USER.OBJ
003570 00002 00002. INTRO VOLX03 INTRO.OBJ
003572 00002 00002. CRUNCH VOLX03 CRUNCH.OBJ
003574 00002 00002. CHATR VOLX03 CHATR.OBJ
003576 00002 00002. USER VOLX03 USER.OBJ
$FLAGT: (RW,D,GBL,REL,CON) 003600 00000 00000. INTRO VOLX03 INTRO.OBJ
003600 00000 00000. CRUNCH VOLX03 CRUNCH.OBJ
003600 00000 00000. CHATR VOLX03 CHATR.J
$ICIO1: (RW,D,GBL,REL,OVR) 003600 000200 00128. USER VOLX03 USER.OBJ
003600 000200 00128. INTRO VOLX03 INTRO.OBJ
003600 00000 00000. CRUNCH VOLX03 CRUNCH.OBJ
003600 00000 00000. CHATR VOLX03 CHATR.J
$IDATA: (RW,D,LCL,REL,CON) 004000 001504 00836. USER VOLX03 USER.OBJ
004000 001454 00812. INTRO VOLX03 INTRO.OBJ
005454 000010 00008. CRUNCH VOLX03 CRUNCH.OBJ
005464 000010 00008. CHATR VOLX03 CHATR.OBJ
005474 000010 00008. USER VOLX03 USER.OBJ
$PDATA: (RW,D,LCL,REL,CON) 005504 000250 00168. INTRO VOLX03 INTRO.OBJ
005504 001154 00108. CRUNCH VOLX03 CRUNCH.OBJ
005660 000024 00020. CHATR VOLX03 CHATR.OBJ
005704 000024 00020. USER VOLX03 USER.OBJ
005730 000024 00020. INTRO VOLX03 INTRO.OBJ
005754 000002 00002. CRUNCH VOLX03 CRUNCH.OBJ
005754 000002 00002. CHATR VOLX03 CHATR.OBJ
$SAVSP: (RW,D,LCL,REL,CON) 005756 000000 00000. USER VOLX03 USER.OBJ
$STRNG: (RW,D,LCL,REL,CON) 005756 000000 00000. INTRO VOLX03 INTRO.OBJ
005756 000000 00000. CRUNCH VOLX03 CRUNCH.OBJ
005756 000000 00000. CHATR VOLX03 CHATR.OBJ
$TDATA: (RW,D,LCL,REL,CON) 005756 000000 00000. USER VOLX03 USER.OBJ
005756 000000 00000. INTRO VOLX03 INTRO.OBJ
005756 000000 00000. CRUNCH VOLX03 CRUNCH.OBJ
005756 000000 00000. CHATR VOLX03 CHATR.OBJ
$$ALER: (RW,I,LCL,REL,CON) 005756 000024 00020.
$$MRKS: (RO,I,LCL,REL,OVR) 006026 00000 00000.
$$OVR: (RW,D,LCL,REL,OVR) 006002 000020 00016.
$$RDSSG: (RO,I,LCL,REL,OVR) 006026 00000 00000.
$$RTS: (RW,I,GBL,REL,OVR) 006022 00002 00002.
$$SGDU: (RW,D,LCL,REL,OVR) 006024 00000 00000.
$$SGD2: (RW,D,LCL,REL,OVR) 006024 000002 00002.

```

Figure 4-3 (Cont.) Memory Allocation File for BASIC-PLUS-2 Version of USER

MEMORY ALLOCATION

GLOBAL SYMBOLS:

ADISIP 002000-R	ADISPS 002024-R	MOISIP 002532-R	MOISPS 002556-R	INTRO 003240-R	SUISPA 002666-R	\$INITS 002046-R
ADISMP 002014-R	ADISSP 002002-R	MOISMP 002546-R	MOISSP 002534-R	CRUNCH 003350-R	SUISPM 002660-R	\$OTSSVA 004050-R
ADISPA 002040-R	CAL\$ 002214-R	NOISPA 002664-R	NOISP 002610-R	CHATR 003460-R	SUI\$PP 002636-R	
ADISPM 002032-R	CBR\$ 002510-R	MOISPM 002572-R	ONISP 002600-R	SUI\$IP 002626-R	SUI\$PS 002652-R	
ADISPP 002010-R	CLI\$P 002620-R	MOISPP 002542-R	SBE\$ 002310-R	SUI\$MP 002642-R	SUI\$SP 002630-R	

*** TASK BUILDER STATISTICS:

TOTAL WORK FILE REFERENCES: 90135.

WORK FILE READS: 0.

WORK FILE WRITES: 0.

SIZE OF CORE POOL: 8548. WORDS (33. PAGES)

SIZE OF WORK FILE: 7168. WORDS (28. PAGES)

ELAPSED TIME: 00:00:14

Figure 4-3 (Cont.) Memory Allocation File for BASIC-PLUS-2 Version of USER

MEMORY ALLOCATION

4.7 MEMORY ALLOCATION MAP FOR COBOL VERSION OF USER

Figure 4-4 shows the memory allocation map for the COBOL version of USER.

NOTE

A single-segment task, such as that illustrated by the memory allocation map in Figure 4-4, does not require use of the /KER:xx switch. Compare the PSECT names in this map that begin with "\$C\$" (along the left margin) with their equivalents in Figure 5-15. Note that in many cases "\$C\$" has been replaced by "\$xx" where xx represents kernel characters for each module.

TSK.TSK MEMORY ALLOCATION MAP TKB M26
16-JUL-77 10:51 PAGE 1

PARTITION NAME : GEN
IDENTIFICATION : 182089
TASK UIC : [120,80]
STACK LIMITS: 001000 001777 001000 00512.
PRG XFR ADDRESS: 024570
TOTAL ADDRESS WINDOWS: 2.
TASK IMAGE SIZE : 8032. WORDS
TASK ADDRESS LIMITS: 000000 037203

*** ROOT SEGMENT: USER

R/W MEM LIMITS: 000000 037203 037204 16004.
DISK BLK LIMITS: 000002 000041 000040 00032.

MEMORY ALLOCATION SYNOPSIS:

SECTION		TITLE	IDENT	FILE
• BLK.: (RW,I,LCL,REL,CON)	002000 000422 00274.			
ACDDAT: (RW,D,GBL,REL,OVR)	002422 000244 00164.			
ACDINS: (RW,I,GBL,REL,OVR)	002422 000244 00164.	ACDQIO	1A.17	COBLIB.OLB
ARITH : (RW,I,GBL,REL,OVR)	002666 001356 00750.			
EDITI : (RW,I,GBL,REL,OVR)	002666 001356 00750.	ACDQIO	1A.17	COBLIB.OLB
EDITTI : (RW,I,GBL,REL,OVR)	004244 005262 02738.			
EDITTI : (RW,I,GBL,REL,OVR)	004244 005262 02738.	ADDSUB	1A.07	COBLIB.OLB
EDITX : (RW,D,GBL,REL,OVR)	011526 001522 00850.			
EDITXD: (RW,D,GBL,REL,OVR)	011526 001522 00850.	EDITX	1A.3	COBLIB.OLB
EXECD1: (RW,D,GBL,REL,OVR)	013250 000026 00022.			
EXECD1: (RW,D,GBL,REL,OVR)	013276 000362 00242.	EDITX	1A.3	COBLIB.OLB
EXECI2: (RW,I,GBL,REL,OVR)	013276 000362 00242.			
EXECI2: (RW,I,GBL,REL,OVR)	013660 006414 03340.	EXEC	1A.24	COBLIB.OLB
MSGPSD: (RW,D,GBL,REL,OVR)	013660 006414 03340.			
MSGPSR: (RO,I,GBL,REL,OVR)	022274 000152 00106.	MSGPSR	1A.09	COBLIB.OLB
MSGPSR: (RO,I,GBL,REL,OVR)	031426 004420 02320.			
MSGPSR: (RO,I,GBL,REL,OVR)	031426 004420 02320.	MSGPSR	1A.09	COBLIB.OLB

Figure 4-4 Memory Allocation File for COBOL Version of USER

MEMORY ALLOCATION

```

UTIL : (RW, I, GBL, REL, OVR) 022446 000214 00140. UTIL 1A. 4 COBLIB.OLB
UTILD : (RW, D, GBL, REL, OVR) 022662 00012 0010. UTIL 1A. 4 COBLIB.OLB
$CBBD0: (RW, I, GBL, REL, OVR) 022674 000000 00000. TASKCA 1A. 04 COBLIB.OLB
$CBBD1: (RW, I, GBL, REL, CON) 022674 000000 00000. USER 182089 USER.OBJ
022674 000000 00000. INTRO 182088 INTRO.OBJ
022674 000000 00000. CRUNCH 182089 CRUNCH.OBJ
022674 000000 00000. CHATR 182089 CHATR.OBJ

$CBBD2: (RW, I, GBL, REL, OVR) 022674 000000 00000. TASKCA 1A. 04 COBLIB.OLB
$CBFA0: (RW, I, GBL, REL, OVR) 022674 000000 00000. TASKCA 1A. 04 COBLIB.OLB
$CBFA1: (RW, I, GBL, REL, OVR) 022674 000000 00000. USER 182089 USER.OBJ
022674 000000 00000. INTRO 182088 INTRO.OBJ
022674 000000 00000. CRUNCH 182089 CRUNCH.OBJ
022674 000000 00000. CHATR 182089 CHATR.OBJ

$CBFD0: (RW, I, GBL, REL, OVR) 022674 000000 00000. TASKCA 1A. 04 COBLIB.OLB
$CBFD1: (RW, I, GBL, REL, CON) 022674 000000 00000. USER 182089 USER.OBJ
022674 000000 00000. INTRO 182088 INTRO.OBJ
022674 000000 00000. CRUNCH 182089 CRUNCH.OBJ
022674 000000 00000. CHATR 182089 CHATR.OBJ

$CBFD2: (RW, I, GBL, REL, OVR) 022674 000002 00002. TASKCA 1A. 04 COBLIB.OLB
$CBIFO: (RW, I, GBL, REL, OVR) 022676 000000 00000. TASKCA 1A. 04 COBLIB.OLB
$CBIFF1: (RW, I, GBL, REL, CON) 022676 000000 00000. USER 182089 USER.OBJ
022676 000000 00000. INTRO 182088 INTRO.OBJ
022676 000000 00000. CRUNCH 182089 CRUNCH.OBJ
022676 000000 00000. CHATR 182089 CHATR.OBJ

$CBIFF2: (RW, I, GBL, REL, OVR) 022676 000000 00000. TASKCA 1A. 04 COBLIB.OLB
$CBIOT: (RW, I, GBL, REL, OVR) 022676 000132 00090. USER 182089 USER.OBJ
022676 000132 00090. INTRO 182088 INTRO.OBJ
022676 000132 00090. CRUNCH 182089 CRUNCH.OBJ
022676 000132 00090. CHATR 182089 CHATR.OBJ

$CBIRO: (RW, I, GBL, REL, OVR) 023030 000000 00000. TASKCA 1A. 04 COBLIB.OLB
023030 000000 00000. TASKCA 1A. 04 COBLIB.OLB

```

Figure 4-4 (Cont.) Memory Allocation File for COBOL Version of USER

MEMORY ALLOCATION

SCBIR1: (RW, I, GBL, REL, CON)	023030 000000 000000.	USER	182089	USER.OBJ
	023030 000000 000000.	INTRO	182088	INTRO.OBJ
	023030 000000 000000.	CRUNCH	182089	CRUNCH.OBJ
	023030 000000 000000.	CHATR	182089	CHATR.OBJ
SCBIR2: (RW, I, GBL, REL, OVR)	023030 000000 000000.	TASKCA	1A. 04	COBLIB.OLB
SCBKBO: (RW, I, GBL, REL, OVR)	023030 000000 000000.	TASKCA	1A. 04	COBLIB.OLB
SCBKB1: (RW, I, GBL, REL, CON)	023030 000000 000000.	TASKCA	1A. 04	COBLIB.OLB
	023030 000000 000000.	USER	182089	USER.OBJ
	023030 000000 000000.	INTRO	182088	INTRO.OBJ
	023030 000000 000000.	CRUNCH	182089	CRUNCH.OBJ
	023030 000000 000000.	CHATR	182089	CHATR.OBJ
SCBKB2: (RW, I, GBL, REL, OVR)	023030 000000 000000.	TASKCA	1A. 04	COBLIB.OLB
SCBKDO: (RW, I, GBL, REL, OVR)	023030 000000 000000.	TASKCA	1A. 04	COBLIB.OLB
SCBKD1: (RW, I, GBL, REL, CON)	023030 000000 000000.	TASKCA	1A. 04	COBLIB.OLB
	023030 000000 000000.	USER	182089	USER.OBJ
	023030 000000 000000.	INTRO	182088	INTRO.OBJ
	023030 000000 000000.	CRUNCH	182089	CRUNCH.OBJ
	023030 000000 000000.	CHATR	182089	CHATR.OBJ
SCBKD2: (RW, I, GBL, REL, OVR)	023030 000000 000000.	TASKCA	1A. 04	COBLIB.OLB
SCBSWT: (RW, I, GBL, REL, OVR)	023030 000002 00002.	USER	182089	USER.OBJ
	023030 000002 00002.	INTRO	182088	INTRO.OBJ
	023030 000002 00002.	CRUNCH	182089	CRUNCH.OBJ
	023030 000002 00002.	CHATR	182089	CHATR.OBJ
SCBTSK: (RW, I, GBL, REL, OVR)	023032 000040 00032.	TASKCA	1A. 04	COBLIB.OLB
	023032 000040 00032.	USER	182089	USER.OBJ
	023032 000040 00032.	INTRO	182088	INTRO.OBJ
	023032 000040 00032.	CRUNCH	182089	CRUNCH.OBJ
	023032 000040 00032.	CHATR	182089	CHATR.OBJ
	023032 000040 00032.	TASKCA	1A. 04	COBLIB.OLB
SCBXA0: (RW, I, GBL, REL, OVR)	023072 000000 000000.	USER	182089	USER.OBJ
SCBXA1: (RW, I, GBL, REL, OVR)	023072 000000 000000.	INTRO	182088	INTRO.OBJ
	023072 000000 000000.	CRUNCH	182089	CRUNCH.OBJ
	023072 000000 000000.	CHATR	182089	CHATR.OBJ
SCBXA2: (RW, I, GBL, REL, OVR)	023072 000000 000000.	TASKCA	1A. 04	COBLIB.OLB
SC\$ADT: (RW, I, GBL, REL, CON)	023072 000000 000000.	USER	182089	USER.OBJ
	023072 000000 000000.	INTRO	182088	INTRO.OBJ

Figure 4-4 (Cont.) Memory Allocation File for COBOL version of USER

MEMORY ALLOCATION

\$C\$ARG: (RW, I, GBL, REL, CON)	023072 000000 00000. CRUNCH 182089 CRUNCH.OBJ 023072 000000 00000. CHATR 182089 CHATR.OBJ
\$C\$DAT: (RW, I, GBL, REL, CON)	023072 000110 00072. USER 182089 USER.OBJ 023106 000036 00030. CRUNCH 182089 CRUNCH.OBJ 023072 000014 00012. INTRO 182088 INTRO.OBJ 023144 000036 00030. CHATR 182089 CHATR.OBJ
\$C\$DDD: (RW, I, GBL, REL, CON)	023202 001250 00680. USER 182089 USER.OBJ 023514 000224 00148. INTRO 182088 INTRO.OBJ 023740 000266 00182. CRUNCH 182089 CRUNCH.OBJ 024226 000224 00148. CHATR 182089 CHATR.OBJ
\$C\$SENT: (RW, I, GBL, REL, CON)	024452 000116 00078. USER 182089 USER.OBJ 024452 000036 00030. INTRO 182088 INTRO.OBJ 024510 000000 00000. CRUNCH 182089 CRUNCH.OBJ 024510 000060 00048. CHATR 182089 CHATR.OBJ
\$C\$TOB: (RW, I, GBL, REL, CON)	024570 000152 00106. USER 182089 USER.OBJ 024570 000020 00016. INTRO 182088 INTRO.OBJ 024610 000036 00030. CRUNCH 182089 CRUNCH.OBJ 024646 000036 00030. CHATR 182089 CHATR.OBJ
\$C\$SLT: (RW, I, GBL, REL, CON)	024742 000000 00000. USER 182089 USER.OBJ 024742 000000 00000. INTRO 182088 INTRO.OBJ 024742 000000 00000. CRUNCH 182089 CRUNCH.OBJ 024742 000000 00000. CHATR 182089 CHATR.OBJ
\$C\$LTD: (RW, I, GBL, REL, CON)	024742 000014 000420. USER 182089 USER.OBJ 024756 000030 00216. INTRO 182088 INTRO.OBJ 024756 000016 00014. CRUNCH 182089 CRUNCH.OBJ 025324 000262 00178. CHATR 182089 CHATR.OBJ
\$C\$PDT: (RW, I, GBL, REL, CON)	025606 000022 00018. USER 182089 USER.OBJ 025606 000014 00012. INTRO 182088 INTRO.OBJ 025622 000002 00002. CRUNCH 182089 CRUNCH.OBJ 025624 000002 00002. CHATR 182089 CHATR.OBJ
\$C\$SLTD: (RW, I, GBL, REL, CON)	025626 000002 00002. USER 182089 USER.OBJ 025630 000204 00132. INTRO 182088 INTRO.OBJ 025630 000014 00012. USER 182089 USER.OBJ 025644 000066 00054. INTRO 182088 INTRO.OBJ 025644 000002 00002. CRUNCH 182089 CRUNCH.OBJ 025732 000022 00018. CHATR 182089 CHATR.OBJ
\$C\$PDT: (RW, I, GBL, REL, CON)	025754 000060 00048. USER 182089 USER.OBJ 026034 000034 00028. INTRO 182088 INTRO.OBJ 026034 000004 00004. USER 182089 USER.OBJ 026040 000010 00008. INTRO 182088 INTRO.OBJ

Figure 4-4 (Cont.) Memory Allocation File for COBOL Version of USER

MEMORY ALLOCATION

SC\$PPM: (RW, I, GBL, REL, CON)	026050 000010 00008. CRUNCH 182089 CRUNCH.OBJ 026060 000010 00008. CHATR 182089 CHATR.OBJ 026070 001060 00560. USER 182089 USER.OBJ 026070 000214 00140. INTRO 182088 INTRO.OBJ 026304 000214 00140. CRUNCH 182089 CRUNCH.OBJ 026520 000214 00140. CHATR 182089 CHATR.OBJ
SC\$SDT: (RW, I, GBL, REL, CON)	026734 000214 00140. CHATR 182089 CHATR.OBJ
SC\$USE: (RW, I, GBL, REL, CON)	027150 000006 00006. USER 182089 USER.OBJ 027156 000000 00000. INTRO 182088 INTRO.OBJ 027156 000000 00000. CRUNCH 182089 CRUNCH.OBJ 027156 000000 00000. CHATR 182089 CHATR.OBJ
SC\$WRK: (RW, I, GBL, REL, CON)	027156 000140 00096. USER 182089 USER.OBJ 027156 000030 00024. INTRO 182088 INTRO.OBJ 027206 000030 00024. CRUNCH 182089 CRUNCH.OBJ 027236 000030 00024. CHATR 182089 CHATR.OBJ
SC\$WRL: (RW, I, GBL, REL, CON)	027266 000030 00024. CHATR 182089 CHATR.OBJ
SC\$WRT: (RW, I, GBL, REL, CON)	027316 000410 00264. USER 182089 USER.OBJ 027316 000102 00066. INTRO 182088 INTRO.OBJ 027420 000102 00066. CRUNCH 182089 CRUNCH.OBJ 027522 000102 00066. CHATR 182089 CHATR.OBJ
SC\$WU1: (RW, I, GBL, REL, CON)	027726 001114 00588. USER 182089 USER.OBJ 027726 000130 00088. INTRO 182088 INTRO.OBJ 030056 000256 00174. CRUNCH 182089 CRUNCH.OBJ 027522 000102 00066. CHATR 182089 CHATR.OBJ
SC\$WU2: (RW, I, GBL, REL, CON)	027726 001114 00588. USER 182089 USER.OBJ 027726 000130 00088. INTRO 182088 INTRO.OBJ 030334 000272 00186. CRUNCH 182089 CRUNCH.OBJ 030626 000214 00140. CHATR 182089 CHATR.OBJ
SXABRT: (RO, I, GBL, REL, CON)	031042 000314 00204. XGO 1A.21 COBLIB.OLB
SXALT : (RO, I, GBL, REL, CON)	031042 000060 00048. INTRO 182088 INTRO.OBJ 031122 000116 00078. CRUNCH 182089 CRUNCH.OBJ 031240 000116 00078. CHATR 182089 CHATR.OBJ
SXCALL: (RO, I, GBL, REL, CON)	036046 000022 00018. XCALL 1A.08 COBLIB.OLB
SXDDDI: (RO, I, GBL, REL, CON)	036046 000022 00018. XGO 1A.21 COBLIB.OLB 036070 000014 00012. XGO 1A.21 COBLIB.OLB
SXENDP: (RO, I, GBL, REL, CON)	036070 000014 00012. XCALL 1A.08 COBLIB.OLB
SXERR : (RO, I, GBL, REL, CON)	036276 000152 00106. XGO 1A.21 COBLIB.OLB
SXEXIT: (RO, I, GBL, REL, CON)	036450 000020 00016. XGO 1A.21 COBLIB.OLB 036470 000014 00012. XCALL 1A.08 COBLIB.OLB

Figure 4-4 (Cont.) Memory Allocation File for COBOL Version of USER

MEMORY ALLOCATION

GLOBAL SYMBOLS:									
\$XGO : (RO, I, GBL, REL, CON)	036504	000120	0008U.	XGO	1A.21	COBLIB.OLB			
\$XGOD : (RO, I, GBL, REL, CON)	036624	000120	00080.	XGO	1A.21	COBLIB.OLB			
\$XGOUN: (RO, I, GBL, REL, CON)	036624	000052	00042.	XGO	1A.21	COBLIB.OLB			
\$XINIT: (RO, I, GBL, REL, CON)	036676	000032	00026.	XGO	1A.21	COBLIB.OLB			
\$XSTOP: (RO, I, GBL, REL, CON)	036676	000032	00026.	XGO	1A.21	COBLIB.OLB			
\$XSTPR: (RO, I, GBL, REL, CON)	037020	000016	00014.	XGO	1A.21	COBLIB.OLB			
\$XSUBK: (RO, I, GBL, REL, CON)	037036	000046	00038.	XGO	1A.21	COBLIB.OLB			
\$SSALER: (RW, I, LCL, REL, CON)	031356	000024	00020.						
\$SMRKS: (RO, I, LCL, REL, OVR)	037202	000000	00000.						
\$SOVRS: (RW, D, LCL, REL, OVR)	031402	000020	00016.						
\$SRDSG: (RO, I, LCL, REL, OVR)	037202	000000	00000.						
\$SRTS : (RW, I, GBL, REL, OVR)	031422	000002	00002.						
\$SSGDD0: (RW, D, LCL, REL, OVR)	031424	000000	00000.						
\$SSGDD2: (RW, D, LCL, REL, OVR)	031424	000002	00002.						
ACCBUF	002476-R	C3F	013724-R	INTEG	035140-R	RSCRY	013572-R	USSGN	006354-R
ACCCQIO	003606-R	DCMLPT	022662-R	INTRO	024610-R	RSTRN	022642-R	WFBI	013402-R
ADDET	003554-R	DISQIO	003726-R	IxDDD	013604-R	SAVE	022622-R	WFB2	013422-R
ADDEV	002460-R	DSETNG	006514-R	LDD	013372-R	SEPSGN	006474-R	WFB3	013442-R
ADLUN	002456-R	DSETPS	006476-R	LNHLD	013570-R	SIGNF	013566-R	WFB4	013454-R
ADSET	003154-R	EASTP	006736-R	MASKPT	020360-R	SIZFLG	013600-R	WFB6	013524-R
ASLUN	003442-R	EMAIL	016724-R	MSG	002471-R	SSTBL	036026-R	WFID1	013534-R
AWFB1	013412-R	EMBB	015340-R	MSGPTL	034330-R	SZEEFLG	013602-R	SCBSWT	023030-R
AWFB2	013432-R	EMBX	015302-R	MSGPTW	034322-R	TTBYUF	002466-R	SCBTSK	023032-R
AWFB3	013452-R	EMCC	016656-R	MSGRTL	034004-R	UBADD	007032-R	SBART	036470-R
AWFB4	013464-R	EMCE	013052-R	MSGRTN	033776-R	UCFV	006554-R	SBACCS	002666-R
AWFBD	013516-R	EMDD	017056-R	M.DPID	011454-R	UDCS	005440-R	SDADBB	006442-R
AWFDF1	013544-R	EMDE	011556-R	M.DPID	011106-R	UDEA	014112-R	SDADBB	006442-R
BAN	004404-R	EMJR	015206-R	NEQOAD	016220-R	UDXP	015012-R	SDADBB	004444-R
BAR	004374-R	EMXB	015256-R	NGFLDO	011072-R	UGBR	010200-R	SDADDR	004436-R
BININS	014004-R	ESGNP	006450-R	NGFLD1	011030-R	UGD	005212-R	SXALT	036070-R
CDN	013374-R	FCP1	013302-R	OCPRY	013400-R	UGRCI	005736-R	SXCALL	021104-R
CHATTR	024704-R	FCD2	013304-R	PARAM	013650-R	UGRCIV	005722-R	SXCALL	036104-R
CONDTR	021257-R	FCD3	013306-R	PARAM1	013650-R	UGRCIV1	0056312-R	SXCALLS	021476-R

GLOBAL SYMBOLS:

Figure A-4 (Cont.) Memory Allocation File for COBOL Version of USER

MEMORY ALLOCATION

CRLFQI	003332-R	FCS	013310-R	PARAM2	013652-R	UMB	016752-R	\$XCBB	021072-R	\$XINIT	036730-R	\$XSUBR	004736-R
CRUNCH	024646-R	FIRLUN	013300-R	PARAM3	013654-R	UMLQ	010070-R	\$XCCC	020674-R	\$XIABY	015106-R	\$XSWT	020442-R
CTEST	013606-R	GETSWT	004066-R	PARAM4	013656-R	UMND	020172-R	\$XCCCS	020660-R	\$XIACP	015116-R	\$XSZEC	020276-R
C1FC	013660-R	GETUB	014646-R	PUTCMG	022446-R	USD	005302-R	\$XCDD	021356-R	\$XMAL	016712-R		
C2FC	013676-R	HDL	013370-R	QIODPB	002422-R	USER	024570-R	\$XCHD	020310-R	\$XMA LD	013212-R		
C2FL	013742-R	IDXMS1	022664-R	RNDFL	013376-R	USFV	006564-R	\$XCHDR	020304-R	\$XMBB	015326-R		

*** TASK BUILDER STATISTICS:

TOTAL WORK FILE REFERENCES: 47354.

WORK FILE READS: 0.

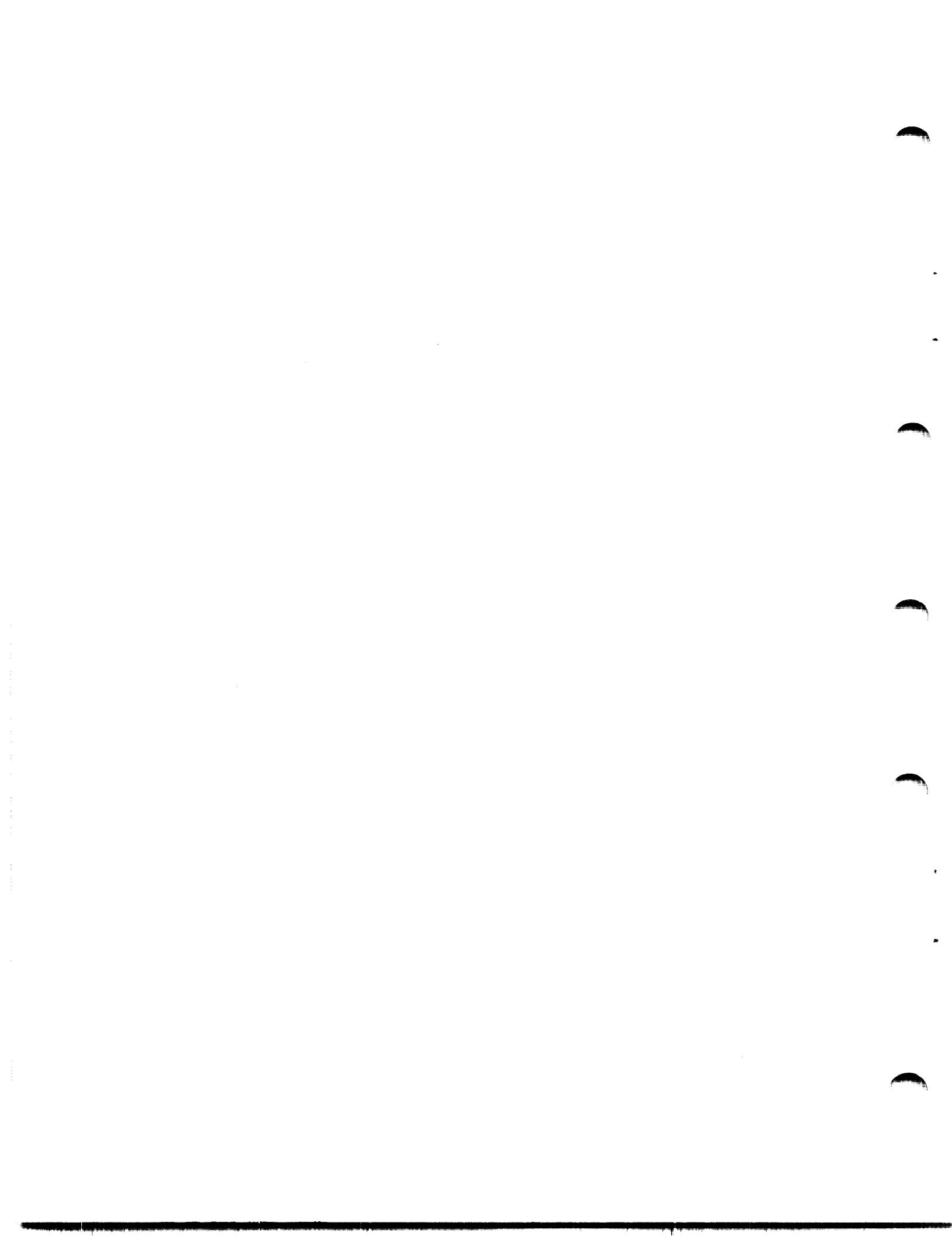
WORK FILE WRITES: 0.

SIZE OF CORE POOL: 6436. WORDS (25. PAGES)

SIZE OF WORK FILE: 4608. WORDS (18. PAGES)

ELAPSED TIME: 00:00:33

Figure 4-4 (Cont.) Memory Allocation File for COBOL version of USER



CHAPTER 5

OVERLAY CAPABILITY

The Task Builder gives you the means to reduce the memory and/or virtual address space requirements of a task. Tree-like overlay structures created with the aid of the Overlay Description Language (ODL) enable you to have only the operating portion of your task in memory at any given time.

This chapter covers the following major topics:

- Overlay Description
- USER Overlay Tree
- Subroutine Communication
- Summary of Overlay Description

5.1 OVERLAY DESCRIPTION

To create an overlay structure, divide a task into a series of segments as follows:

- A single, controlling, root segment (always in memory)
- Any number of overlay segments (residing on disk and sharing virtual address space and memory with one another according to your overlay structure)

A segment is a set of modules and PSECTS. Segments that overlay each other must be logically independent; that is, the components of one segment cannot reference components of any segment with which it shares virtual address space.

Consider also the general flow of control within the task. There are several large classes of tasks that can be handled effectively by an overlay structure. For example, one that moves sequentially through a set of modules is well-suited to an overlay structure. Another that selects one of several modules according to the value of an item of input data is also well-suited, if speed of execution is not critical. Tasks having several distinct functions are overlay candidates, too.

You must decide what kind of overlay segment to have at a given position in the structure and how to construct it. Dividing a task into disk-resident overlays saves physical space, but introduces the overhead activity of loading these segments each time they are needed and not present in memory.

OVERLAY CAPABILITY

5.1.1 Disk-Resident Overlay Structures

Disk-resident overlays conserve memory by sharing it. Segments that are logically independent need not be present in memory at the same time. Therefore, they can be allocated a common physical area in memory for use by each as needed.

The example task TK1 shows the use of disk-resident overlays. TK1 consists of four input files. Each input file contains a single module having the same name as the file. The task is built by the command:

```
TKB TK1=CNTRL,A,B,C
```

where the file extensions conform to the defaults listed in Table 2-1. The complete filenames and extensions are:

- TK1.TSK
- CNTRL.OBJ
- A.OBJ
- B.OBJ
- C.OBJ

Here, the modules A, B, and C are logically independent, so:

- A does not call B or C and does not use the data of B or C.
- B does not call A or C and does not use the data of A or C.
- C does not call A or B and does not use the data of A or B.

You can define a disk-resident overlay structure in which A, B, and C are overlay segments that occupy the same storage area in memory. The flow of control for the task is as follows:

- TK1 starts in the segment CNTRL.
- CNTRL calls A and A returns to CNTRL.
- CNTRL calls B and B returns to CNTRL.
- CNTRL calls C and C returns to CNTRL.
- CNTRL calls A again and A returns to CNTRL.
- TK1 ends in the segment CNTRL.

In this example, overlay loading occurs only four times during the execution of the task. So you can reduce the memory requirements of a similar task without unduly increasing the overhead activity.

The effect of an overlay structure on memory allocation for the task is discussed in the following paragraphs.

The lengths of the modules (expressed in octal) are:

Module	Length in Bytes
CNTRL	10000
A	6000
B	5000
C	1200

The memory allocation produced when you build the task as a single segment is shown in Figure 5-1.

OVERLAY CAPABILITY

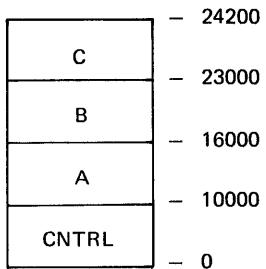


Figure 5-1 TK1 Memory Allocation

Figure 5-2 shows the memory allocation produced when you use the overlay capability and build a multi-segment task.

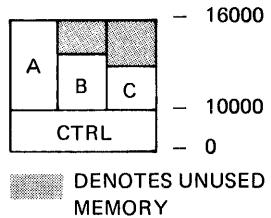


Figure 5-2 Allocation for a Multi-Segment Task

The memory allocation for a single-segment task requires 24200 (octal) bytes, and the multi-segment task requires 16000 (octal) bytes resulting in a net saving of 6200 (octal) bytes. In addition to the module storage, storage is required for overhead in handling the overlay structure. This overhead is described further on and illustrated in the examples.

NOTE

Module lengths are given in octal and module length calculations are done using octal arithmetic. See Appendix B for an octal-to-decimal conversion table and instructions.

You can determine the amount of storage required for the task by adding the length of the root segment and the length of the longest overlay segment. Overlay segments A and B in Figure 5-2 are much longer than overlay segment C.

If you can divide A and B into sets of logically independent modules, you can reduce task storage requirements even more. As shown in Figure 5-3, A can be divided into a control program (A0) and two overlays (A1 and A2). A2 is then divided into a control module (A2) and two overlays (A21 and A22). Similarly, the B overlay can be divided into a control module (B0) and two overlays (B1 and B2). The unlabelled portions of the block diagram represent unused memory space.

OVERLAY CAPABILITY

The memory allocation for the task produced by the additional overlays defined for A and B is shown in Figure 5-3 below. The left side of the figure is unmarked for clarity. The paragraphs following the figure discuss the method of reading a block diagram.

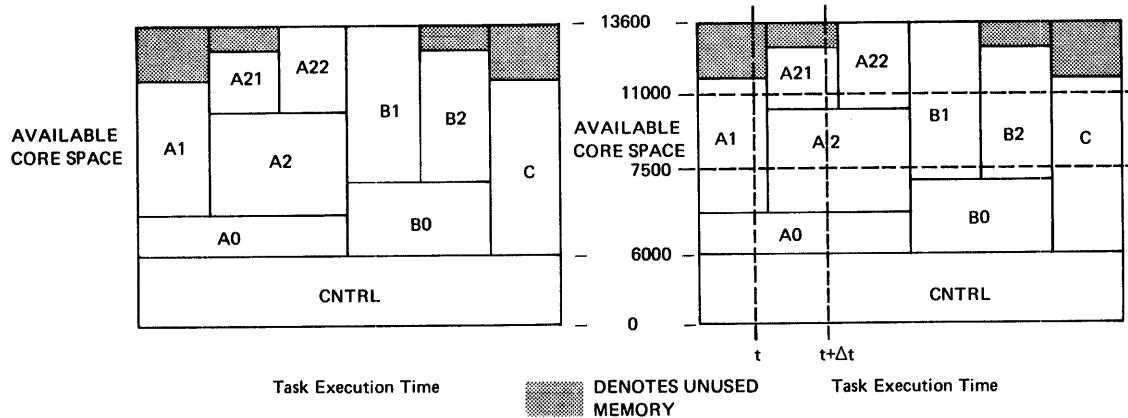


Figure 5-3 How to Read a Block Diagram

A vertical line can be drawn through a memory diagram to show which modules are in memory at a given time. On the right side of Figure 5-3 the line at t shows memory when CNTRL, A0, and A1 are loaded. The line at $t+\Delta t$ shows memory when CNTRL, A0, A2, and A21 are loaded, and so on.

A horizontal line can be drawn through a memory diagram to show which segments share the same storage. The line at 11000 passes through A1, A21, A22, B1, B2, and C, all of which can use the same memory. The line at 7500 passes through A1, A2, B1, B2, and C, all of which can use the same memory.

5.1.2 Overlay Tree

The arrangement of overlay segments in a task can also be represented schematically as a tree-like structure. Each branch in the tree represents a segment. Parallel branches rising from the same horizontal bar denote segments that overlay one another; these segments must be logically independent. Branches connected end-to-end represent segments that do not share virtual or physical memory with each other; these segments need not be logically independent. The topmost segments, which contain no subroutine calls, are leaves.

The Task Builder provides a language for representing an overlay structure consisting of one or more trees (described in Section 5.1.4).

The single overlay tree shown in Figure 5-4 below represents the overlay structure for the block diagram in Figure 5-3.

OVERLAY CAPABILITY

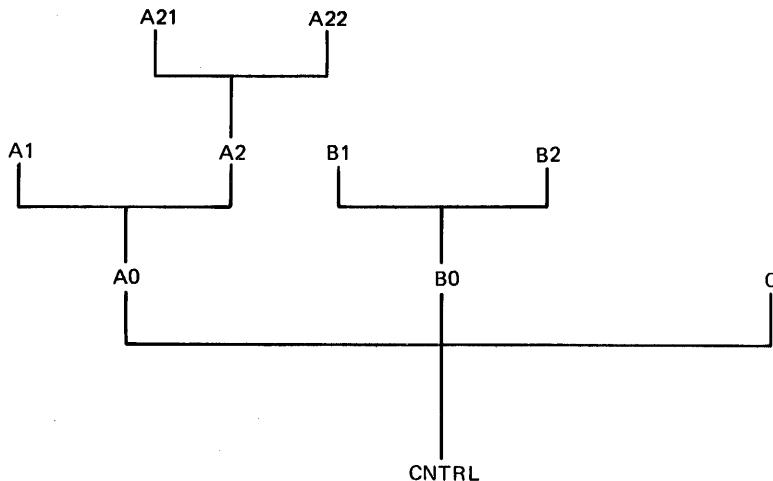


Figure 5-4 Multi-level Overlay Tree

The tree in Figure 5-4 has a root (the main module, or driver, CNTRL) and two main branches (the major subprograms A0 and B0). It also has five leaves (the minor subroutines A1, A21, A22, B1, and B2). Subprogram C, which calls no other routines, can also be considered to be a leaf.

Relationships between the modules in an overlay tree are expressed as paths. Paths show the flow of control between modules in a tree, and show how to access a given module. The tree has as many paths as it has leaves. The path down is defined from the leaf to the root. For example in Figure 5-4:

A21-A2-A0-CNTRL

The path up is defined from the root to the leaf:

CNTRL-B0-B1

If you know the properties of the tree and its paths, you will better understand overlay loading and global symbol resolution (see also Section 4.3).

5.1.2.1 Overlay Loading - Modules can call other modules that exist on the same path. Look at the tree in Figure 5-4. Module CNTRL is common to every path of the tree and therefore can call and be called by every module in the tree. Module A2 can call the modules A21, A22, A0, and CNTRL because these modules are on the same paths as A2. But A2 cannot call A1, B1, B2, B0 or C because these modules are on different paths from A2.

When a module in one overlay segment calls a module in another overlay segment, the called segment must be in memory or must be loaded. The autoload mechanism, which handles all high-level language loading, is described in Chapter 6.

OVERLAY CAPABILITY

5.1.2.2 Resolving Global Symbols in a Multi-segment Task - In resolving global symbols for a multi-segment task, the Task Builder performs the same activities as for a single-segment task. The rules defined in Section 4.3 for the resolution of global symbols in a single-segment task also apply in this case, but the scope of the global symbols is restricted by the overlay structure.

In a single-segment task, any module can reference any global symbol. In a multi-segment task, however, a module can reference only global symbols that are defined on the same path.

The following points, illustrated in the tree shown in Figure 5-5, describe the two distinct cases of multiply-defined symbols, and ambiguously-defined symbols.

In a single segment task, if two global symbols with the same name are defined, the symbols are considered multiply-defined and an error message is produced.

In a multi-segment task:

- Two global symbols with the same name can be legally defined if they are on separate paths and are not referenced from a segment common to both.
- A global symbol defined more than once on separate paths, but referenced from a segment that is common to both, is ambiguously defined.
- A global symbol defined more than once on a single path is multiply defined.

The procedure for resolving global symbols can be summarized as follows:

1. The Task Builder selects an overlay segment for processing.
2. Each module in the segment is scanned for global definitions and references.
3. If the symbol is a definition, the Task Builder searches all segments on paths that pass through the segment being processed, and looks for references that must be resolved.
4. If the symbol is a reference, the Task Builder performs the tree search as described in step 3, looking for an existing definition.
5. If the symbol is new, it is entered in a list of global symbols associated with the segment.

Overlay segments are selected for processing in an order corresponding to their distance from the root. The Task Builder considers a branch farther away from the root or a leaf before processing an adjoining branch.

When a segment is being processed, the search for global symbols proceeds in the following order:

- the segment being processed
- all segments toward the root
- all segments away from the root
- all co-trees (see Section 5.1.4)

OVERLAY CAPABILITY

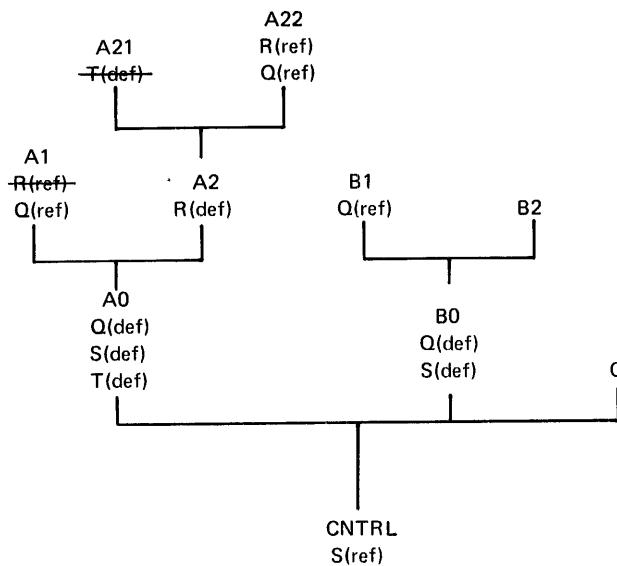


Figure 5-5 Global Symbols in a Tree

The following notes apply to the use of the symbols Q, R, S, and T, shown in the tree structure in Figure 5-5 above:

1. The global symbol Q is defined in the segment A0 and B0. The reference to Q in segments A22 and A1 are resolved by the definition in A0. The reference to Q in B1 is resolved by the definition in B0. The two definitions of Q are distinct in all respects and occupy different overlay paths (CNTRL-A0-A2-A22 and CNTRL-B0-B1, respectively).
2. The global symbol R is defined in the segment A2. The reference to R in A22 is resolved by the definition in A2 because there is a path to the reference from the definition (CNTRL-A0-A2-A22). The reference to R in A1, however, is undefined because there is no definition for R on the path through A1 (CNTRL-A0-A1). To correct this situation, move the definition of R to A0.
3. The global symbol S is defined in A0 and B0. References to S from A1, A21, or A22 are resolved by the definition in A0, and references to S in B1 and B2 are resolved by the definition in B0. However, the reference to S in CNTRL cannot be resolved because there are two definitions of S on separate paths through CNTRL (CNTRL-A0 and CNTRL-B0). S is ambiguously defined. To correct this situation, move the definition of S to CNTRL.
4. The global symbol T is defined in A21 and A0. Because there is a single path through the two definitions (CNTRL-A0-A2-A21), the global symbol T is multiply defined. To correct this situation, remove the erroneous definition and, preferably, place the correct definition in A0.

OVERLAY CAPABILITY

5.1.2.3 Resolving Global Symbols from the Default Library - The process of resolving global symbols may require two passes over the tree structure. The global symbols discussed in the previous section are included in user-specified input modules that the Task Builder scans on the first pass. If any undefined symbols remain, the Task Builder makes a second pass over the structure to try to resolve such symbols by searching the default object module library (normally SY:[1,1]SYSLIB.OLB). Any undefined symbols remaining after the second pass are reported to you at the terminal.

When you define multiple tree structures (see Section 5.1.4), you run the risk of multiply or ambiguously defining global symbols. This can occur when the Task Builder tries to resolve global symbols during its second pass over the co-tree structures. Multiple or ambiguous definitions of global symbols can cause overlay segments to be inadvertently displaced from memory by the overlay loading routines, thereby causing run-time failures to occur. To eliminate these conditions, the tree search on the second pass is restricted to:

- The segment in which the undefined reference has occurred
- All segments in the current tree that are on a path through the segment
- The root segment

When the current segment is the main root, the tree search is extended to all segments. You can extend the tree search to all segments for the entire tree by including the /FU (full search) switch in the task image file specification for the entire tree.

5.1.2.4 Resolving PSECTS in a Multi-segment Task - A PSECT has an attribute that indicates whether the PSECT is local (LCL) to the segment in which it is defined or is global (GBL).

Local PSECTS with the same name can appear in any number of segments. (Thus, an often-used routine can be called with minimum system overhead from many places in your task.) Storage is allocated for each local PSECT in the segment in which it is declared. Global PSECTS that have the same name, however, must be resolved by the Task Builder.

When a global PSECT is defined in several overlay segments along a common path, the Task Builder allocates all storage for the PSECT in the overlay segment closest to the root.

BASIC-PLUS-2 COMMON and MAP blocks are translated into global PSECTS and given the overlay attribute. In the tree shown in Figure 5-6 the common block COMA is defined in modules A2 and A21. The Task Builder allocates the storage for COMA in A2, because that segment is closer to the root than the segment that contains A21.

If the programs A0 and B0 use a common block COMAB, however, the Task Builder allocates the storage for COMAB in both the segment that contains A0 and the segment that contains B0. A0 and B0 cannot communicate through COMAB. When the overlay segment containing B0 is loaded, any data stored in COMAB by A0 is lost.

OVERLAY CAPABILITY

Figure 5-6 shows the tree for task TK1, including the allocation of the common blocks COMA and COMAB.

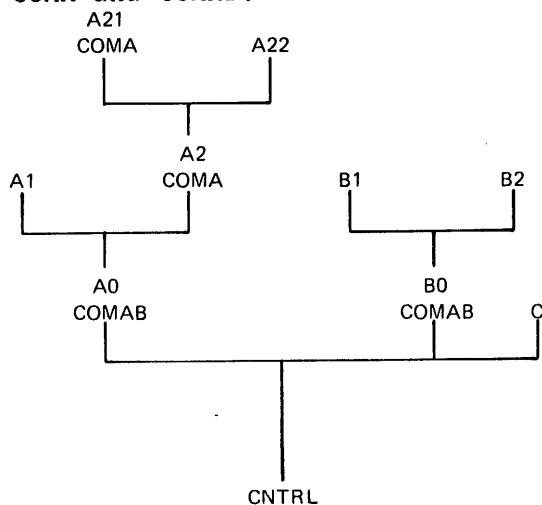


Figure 5-6 Common Blocks in a Tree

You can specify PSECT allocation. If A0 and B0 need to share the contents of COMAB, you can force the allocation of COMAB into the root segment by using the .PSECT directive, described in Section 5.1.3.4.

5.1.3 Overlay Description Language (ODL)

The Task Builder provides a language that lets you describe the overlay structure of a task. An overlay description is a text file consisting of a series of ODL directives, one directive per line. This file is entered in a Task Builder command line, and is identified as an ODL file by the presence of the /MP switch (see Section 3.1.7) after the filename. If an overlay description text file is entered, it must be the only input file specified.

The format for an ODL line is:

```
label: directive argument-list;comment
```

The label is a necessary part of the .FCTR directive only (see Section 5.1.3.2).

Directives act upon argument lists:

- Named input files
- Overlay segments
- PSECTS
- Lines in the ODL file itself

The hyphen, exclamation point, and comma operators, described in Section 5.1.3.1, group these named task elements, or attach attributes to them.

If the name belongs to a file, a complete file specification can be given. Defaults for omitted parts of the file specification are as described in Chapters 2 and 3, except that the default device is always SY, and the default PPN is your own.

OVERLAY CAPABILITY

In addition, the following restrictions apply to argument-lists:

- The dot character (.) can only be used in a filename.
- Comments cannot appear on a line ending with a filename (see Section 2.6).

5.1.3.1 .ROOT and .END Directives - There must be one .ROOT directive and one .END directive in your ODL file. The .ROOT directive tells the Task Builder where to start building the tree, and the .END directive tells Task Builder where the input ends.

The arguments of the .ROOT directive use four operators to express concatenation, overlaying, memory and library residency:

- A pair of parentheses delimits a group of segments that start at the same virtual address and thus share storage. The number of nested parenthetical groups cannot exceed 16.
- The hyphen operator (-) indicates the concatenation of storage. For example, X-Y means that sufficient memory will be allocated to contain X and Y simultaneously. X and Y are allocated in sequence.
- The exclamation point operator (!) allows the specification of resident library overlay segments that will permanently reside in memory rather than on disk. The use of the operator for executable task images is not supported. Memory residency is specified by placing an exclamation point immediately before the left parentheses (in the .ODL file) that enclose the desired segments. For example:

```
.ROOT A-!(B,C)
```

In this example, segments B and C are declared resident in separate areas of memory. The single starting virtual address for both B and C is determined by the Task Builder. The Task Builder rounds the octal length of segment A up to the next 4K boundary. It then determines the physical memory allocated to segments B and C by rounding the actual length of each segment to the next 32-word boundary (256-word boundary if the /CM switch is in effect; see Section 3.1.A), and adding the determined value to the total memory required by the task.

The exclamation point operator applies only to segments at the first level inside a pair of parentheses; segments of the ODL that are nested within the first level are not affected.

- The comma operator (,) appearing within parentheses indicates a virtual memory overlay involving the two modules that are separated by the comma. For example, Y,Z means that virtual memory can contain either Y or Z.

OVERLAY CAPABILITY

The comma operator is also used to define multiple tree structures, as described in Section 5.1.4, when it separates two structures as in TFIL.ODL below.

The directives:

```
.ROOT X-(Y,Z-(Z1,Z2))
.END
```

describe the tree and corresponding memory diagram in Figure 5-7:

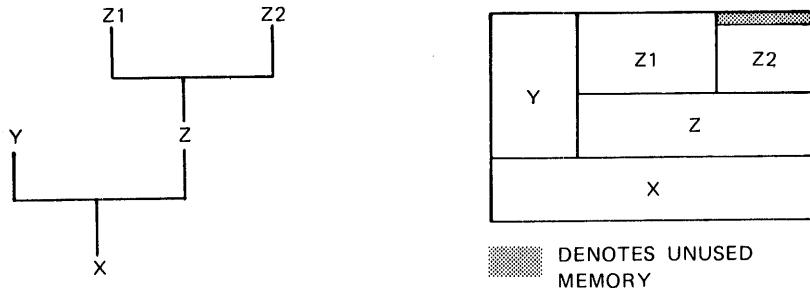
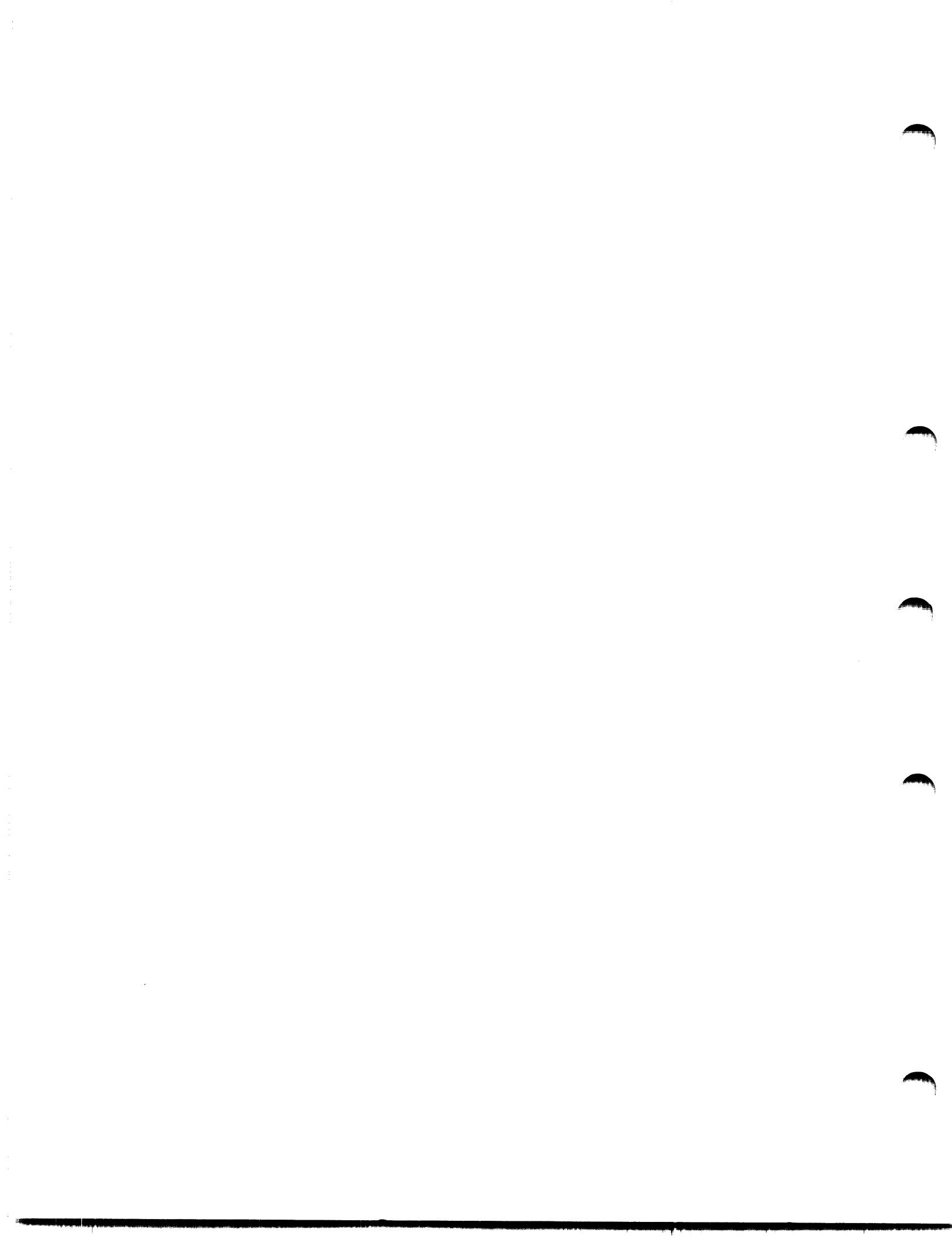


Figure 5-7 A Simple Multi-level Tree

The overlay description for the task TK1 described in Section 5.1.1, contains the directives:

```
.ROOT CNTRL-(A0-(A1,A2-(A21,A22)),B0-(B1,B2),C)
.END
```



OVERLAY CAPABILITY

Assuming that this ODL description appears in a file named TFIL.ODL, you can build the required structure with the 1-line command:

```
TKB TK1.IMG=TFIL.ODL/MP
```

The switch /MP tells the Task Builder that there is only one input file, TFIL.ODL, and that this file contains an overlay description for the task.

5.1.3.2 .FCTR Directive - The Overlay Description Language includes another directive, .FCTR, to help you build large, complex trees and represent them more clearly.

The .FCTR directive has a label in the left margin that is referenced in a .ROOT or another .FCTR statement. The .FCTR directive lets you extend the tree description beyond a single line. (There can be only one .ROOT directive.)

To simplify the tree in TFIL.ODL, you can introduce the .FCTR directive into the overlay description:

```
.ROOT CNTRL-(AFCTR,BFCTR,C)
AFCTR:   .FCTR A0-(A1,A2-(A21,A22))
BFCTR:   .FCTR B0-(B1,B2)
.END
```

The label AFCTR designates the structure A0-(A1,A2-(A21,A22)), as shown in the .FCTR directive on the next line.

The label BFCTR designates the structure B0-(B1,B2). The resulting overlay description is easier to interpret than the original description. The tree consists of a root, CNTRL, and three main branches. Two of the main branches have sub-branches.

The .FCTR directive can be nested to 16 levels. You can change TFIL to read as follows:

```
.ROOT CNTRL-(AFCTR,BFCTR,C)
AFCTR:   .FCTR A0-(A1,A2FCTR)
A2FCTR:  .FCTR A2-(A21,A22)
BFCTR:   .FCTR B0-(B1,B2)
.END
```

NOTE

The order in which .FCTR and .NAME lines appear is irrelevant.

5.1.3.3 .NAME Directive - The .NAME directive lets you specify a name for a segment and to attach desired attributes to the segment. The name must be unique with respect to filenames, PSECT names, .FCTR labels, and other segment names that are used in the overlay description.

The chief uses of the .NAME directive are:

1. to uniquely name a segment

OVERLAY CAPABILITY

2. to permit a segment that does not contain executable code to be loaded

The format of the .NAME directive is

```
.NAME segname[,attr][,attr]
```

where:

segname is a 1- to 6-character name from the Radix-50 characters A - Z, 0 - 9, and \$

brackets ([]) denote optional attributes

attr represents one of the following attributes:

NOTE

Attributes are not attached to a segment until the name is used in a .ROOT or .FCTR statement that defines an overlay segment. When multiple segment names are applied to a segment, the attributes of the latest name given go into effect.

GBL The name is entered in the segment's global symbol table.

The GBL attribute make possible the loading of non-executable overlay segments by means of the autoload mechanism (see Chapter 6).

NOGBL The name is not entered in the segment's global symbol table.

NOTE

If the GBL attribute is not present, NOGBL is assumed.

DSK Disk storage is allocated to the named segment.

NODSK No disk space is allocated to the named segment.

If a data overlay segment has no initial values, but will have its contents established by the running task, no space for the task image on disk need be reserved in advance. If the NODSK attribute has been specified, an attempt to initialize the segment with data at task-build time results in a fatal error.

NOTE

If the NODSK attribute is not present, DSK is assumed.

OVERLAY CAPABILITY

In Figure 5-8, a modified tree for TK1, the three main branches, A0, B0, and C, are named by specifying the names in the .NAME directive, and using them in the .ROOT directive. The default attributes NOGBL and DSK are in effect for BRNCH1 and BRNCH3. But BRNCH2 has the complementary attributes (GBL and NODSK) that cause the name BRNCH2 to be entered into its segment's global symbol table, and the allocation of disk space for the segment to be suppressed. BRNCH2 contains uninitialized storage to be used at run-time.

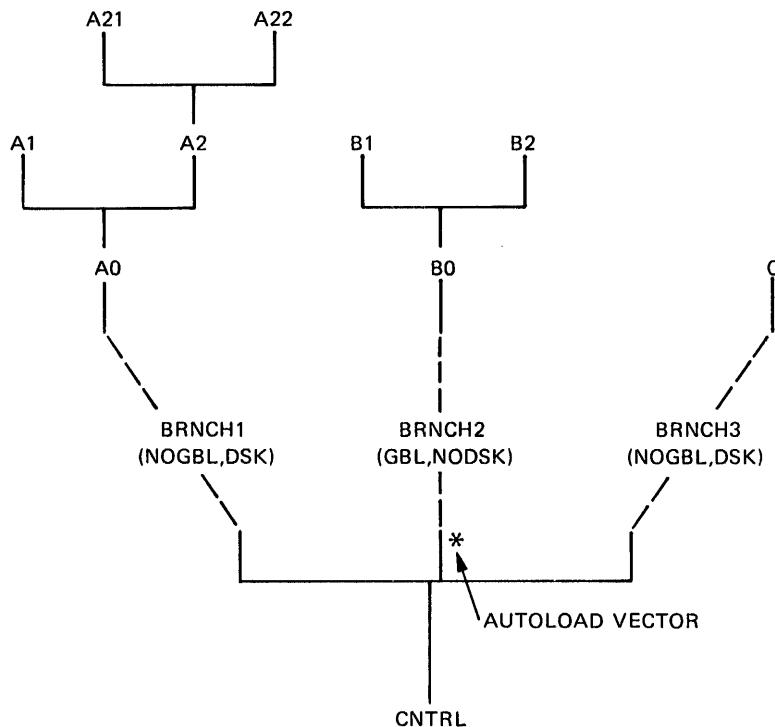


Figure 5-8 TK1 Modified Tree Using the .NAME Directive

```

.NAME BRNCH1
.NAME BRNCH2,GBL,NODSK
.NAME BRNCH3
.ROOT CNTRL-(BRNCH1-AFCTR,*BRNCH2-BFCTR,BRNCH3-C)
AFCTR:   .FCTR A0-(A1,A2-(A21,A22))
BFCTR:   .FCTR B0-*(B1,B2)
.END
  
```

(*, in the statement labelled BFCTR:, is the autoload indicator. It is discussed in Section 6.1.1.)

Global segment names allow segments containing only data, such as message text, constant values, etc. to be autoloaded. Such data segments must have the autoload indicator applied. See Section 6.1.1 for more information about the autoload indicator.

BRNCH2 above does have the autoload indicator applied so it can be loaded by the following statement in the CNTRL program.

CALL BRNCH2

OVERLAY CAPABILITY

This action is immediately followed by an automatic return to the next instruction in the CNTRL program.

You can also use segment names to make patches with the options ABSPAT and GBLPAT (described in Sections 3.2.6.3 and 3.2.6.4).

NOTE

If there is no unique .NAME specification, the Task Builder establishes a segment name, using the first .PSECT, file, or library module name occurring in the segment.

5.1.3.4 .PSECT Directive - The .PSECT directive lets you direct the placement of a global PSECT in an overlay structure. The name of the PSECT (a 1- to 6-character name composed from the set A-Z, 0-9, and \$) and its attributes are given in the .PSECT directive. This allows use of the name to indicate which segment the PSECT will be allocated to. An example of the use of .PSECT is given in the modified version of task TK1 shown below.

Be careful about logical independence of the modules in the overlay segment, but do not forget to take into account the requirement for logical independence in multiple executions of the same overlay segment. In other words, if you call a segment twice, be sure you do not change the flow of control between the first call and the second. (COBOL programmers should particularly avoid the ALTER statement.)

The flow of task TK1 (described in Section 5.1.1) can be summarized this way. CNTRL calls each of the overlay segments in the order A, B, C, A and each overlay segment returns to CNTRL. Module A is executed twice. The overlay segment containing A must be reloaded for the second execution because it was overlaid when B was loaded.

Module A uses the common block DATA3. The Task Builder allocates DATA3 to the overlay segment containing A. The first execution of A stores some results in DATA3. The second execution of A requires these values. In this disk-resident overlay structure, however, the values calculated by the first execution of A are overlaid. When the segment containing A is read in for the second execution, the common block is in its initial state.

To permit the data in DATA3 to be accessed on the second execution of A, use a .PSECT directive to force the allocation of DATA3 into the root. One way to do this is to replace the last four statements of the previous overlay description of TK1 (starting with the .ROOT statement) with the following:

```
.PSECT DATA3,RW,GBL,REL,OVR
.ROOT CNTRL-DATA3-(AFCTR,BFCTR,C)
AFCTR:   .FCTR A0-(A1,A2-(A21,A22))
BFCTR:   .FCTR B0-(B1,B2)
.END
```

OVERLAY CAPABILITY

5.1.3.5 Indirect Files - The Overlay Description Language processor can accept ODL text from an indirect file, if the text is included in a file specified in the proper format. If a commercial "at" (@) is the first character in an ODL line, processor reads text from the file specified immediately after the "@". It accepts the ODL text from the file as input at the point in the overlay description where the file is specified. Two levels of indirection are allowed.

For example, if the file BIND.ODL contains

```
B: .FCTR B1-(B2,B3)
```

then this text can be replaced by a line beginning with @BIND, at the position where the text would have appeared:

	Direct	Indirect
C:	.ROOT A-(B,C) .FCTR C1-(C2,C3)	.ROOT A-(B,C) C: .FCTR C1-(C2,C3)

Note that the extension of the filename BIND is assumed to be .ODL. If the file you are using does not have the .ODL extension, you must specify the extension to the Task Builder.

5.1.4 Multiple Tree Structures

The Task Builder lets you define more than one tree in an overlay structure. A multiple tree structure contains one main tree and one or more co-trees. RSTS/E loads the root segment at the start of the task. Segments of the co-tree(s) are loaded by the Overlay Run-time System as they are called.

Except for this distinction, all overlay trees have identical characteristics; a root segment that resides in memory, and, usually, two or more overlay segments. The main property of a structure containing more than one tree is that storage is not shared among trees. Any segment in a tree can be referenced from another tree without displacing segments from the calling tree. Routines that are called from several main tree overlay segments, for example, can overlay one another in a co-tree.

The next two sections describe the procedure for specifying multiple trees in the Overlay Description Language, and illustrate the use of co-trees to produce the memory allocation best suited to the needs of the task.

5.1.4.1 Defining a Multiple-Tree Structure - The comma, when included within parentheses, defines a pair of segments that share storage. The comma outside all parentheses delimits overlay trees. The first overlay tree so defined is the main tree. Other trees in the same ODL file are co-trees. Here is an ODL description of a main tree and a co-tree:

```
X: .ROOT      X,Y  
    .FCTR      X0-(X1,X2,X3)  
Y: .FCTR      Y0-(Y1,Y2)  
    .END
```

OVERLAY CAPABILITY

You define co-trees in the .ROOT directive by placing the comma operator outside all parentheses and immediately in front of the co-tree root (Y, in the example above). Any number of co-trees can be defined. Co-trees can access any module in the main tree or any other co-tree. In the example above, there are two overlay trees. The main tree X contains the root segment X0 and three overlay segments. The co-tree Y contains the root segment Y0 and two overlay segments. RSTS/E loads segment X0 into memory when the task starts. The Overlay Run-time System then loads the remaining segments as they are called.

A co-tree must have a root segment to establish linkage with its own overlay segments. But co-tree root segments need not contain code or data. A segment of this type, called a null segment, can be created using the .NAME directive. The previous example is modified as shown below, to move file Y0.OBJ to the root, and include a null segment.

```
.ROOT      X,Y
X:        .FCTR      X0-Y0-(X1,X2,X3)
          .NAME      YNUL
Y:        .FCTR      YNUL-(Y1,Y2)
          .END
```

The .NAME directive creates the null segment YNUL which replaces the co-tree root that formerly contained Y0.OBJ.

5.1.4.2 Multiple-Tree Example - You can use multiple trees to reduce the size of a task.

In the example below, CNTRLX and CNTRLY are logically independent of each other and must be accessed from modules on all the paths of the main tree. A co-tree for CNTRLX and CNTRLY that names a root segment (CNTRL2) satisfies these requirements and reduces the amount of storage required by the task. The overlay description looks like this:

```
.NAME CNTRL2
.ROOT CNTRL-(AFCTR,BFCTR,C),CNTRL2-(CNTRLX,CNTRLY)
.
.
.
.END
```

The tree for the task TK1 is shown in Figure 5-9.

OVERLAY CAPABILITY

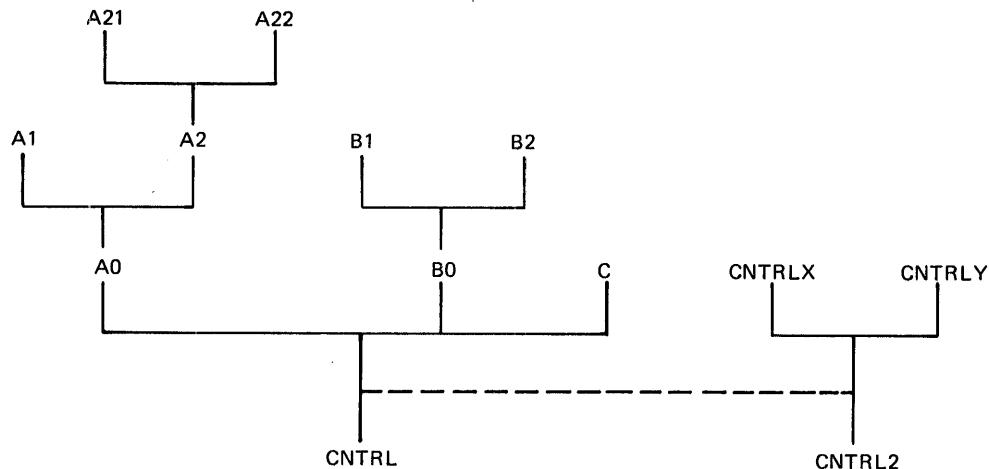


Figure 5-9 Co-tree

The corresponding memory diagram is shown in Figure 5-10.

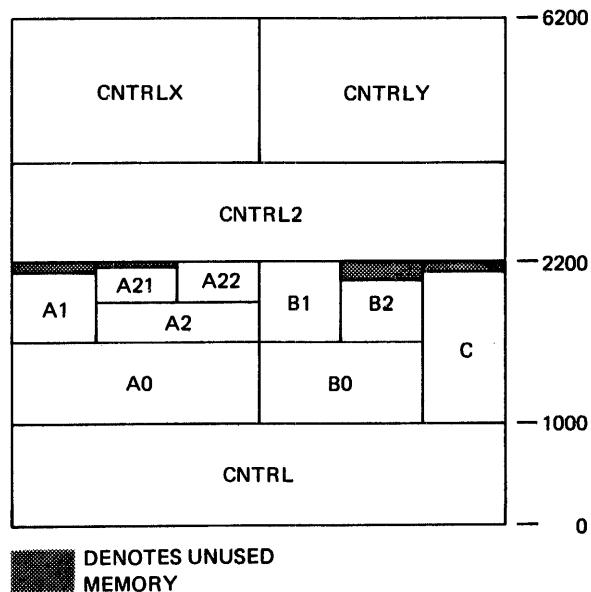


Figure 5-10 Co-tree Block Diagram

Specifying a co-tree decreases the storage allocation by 4000 bytes. CNTRLX and CNTRLY can be accessed by all modules in the main tree. The co-tree only requires that CNTRLX and CNTRLY be independent.

5.1.5 Overlay Core Image

The contents of the core image for a task with an overlay structure are discussed briefly in this section. (The header and stack are described in Section 4.2.)

OVERLAY CAPABILITY

The root segment of the main tree contains:

- modules that are resident in memory throughout task execution
- segment tables and autoload vectors that are required by the overlay loading routines

Segment tables contain a descriptor for every segment in the task. The segment descriptor contains information about the load address, the length of the segment, and the tree linkage. The segment table is described in detail in Appendix D.

Autoload vectors appear in every segment that calls modules in another segment located farther away from the root of the tree. The autoload mechanism is described in Chapter 6. The detailed composition of the autoload vector is given in Section D.3.1.

The main tree overlay region consists of memory allocated for the overlay segments of the main tree. The overlays are read into this area of memory as they are needed.

The co-tree overlay region consists of memory allocated for co-tree overlay segments. The co-tree root segment contains modules that, once loaded, must remain resident in memory. The first co-tree is loaded above the main tree overlay region. Other co-trees are loaded above the overlay region of the preceding co-tree in the same fashion. Figure 5-10 shows the block diagram for the main tree and the co-tree of TK1.

5.1.6 Overlaying Programs Written in a Higher-level Language

Programs that are written in a higher-level language usually require a large number of library routines in order to execute. Unless care is taken when overlaying such programs, these problems can occur:

1. Task Builder throughput may be drastically reduced because of the number of library references in each overlay segment.
2. Default object module library references resolved across tree boundaries can cause unintentional displacement of segments from memory at run-time.
3. Attempts to task-build such programs can result in multiple and ambiguous symbol definitions when a co-tree structure is defined.

Effective procedures for solving these problems are:

1. Linking commonly used library routines into the main root segment. Task Builder throughput can thereby be increased.
2. Using the /-FU switch (the default) to restrict the scope of the default library search. Ambiguous and multiple definitions, and cross-tree references can thereby be eliminated.

You can force library modules into the root by preparing a list of the appropriate global references and linking the object module containing them into the root segment.

OVERLAY CAPABILITY

The User's Guide for the language you are using contains other ways to reduce the size of your task.

NOTE (COBOL USERS ONLY)

COBOL overlay procedures require that you use the /KER:xx switch at compile-time to generate unique names for certain compiler-generated PSECTS. See the PDP-11 COBOL User's Guide for an explanation of this switch. Be sure your kernel characters are unique within the ODL file.

5.2 USER OVERLAY TREE

Figure 5-11 shows the overlay tree for USER.

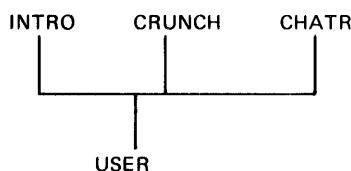


Figure 5-11 USER Overlay Tree

5.2.1 Defining the ODL File

After you determine how the final structure is to operate, create ODL directives to represent the overlay tree, such as the following:

```
.ROOT TREE
TREE: .FCTR USER-LIBR-*(INTRO-LIBR,CRUNCH-LIBR,CHATR-LIBR)
LIBR: .FCTR [1,1]BASIC2/LB
.END
```

(The * in the ODL description is the autoload indicator. It is described in Section 6.1.1.)

This section applies only to BASIC-PLUS-2 users.

5.2.2 Building the Task

You can build the task with the same options used in the example in Section 3.3.1. Here, the names of the input files are replaced by a single filename that designates the file containing the overlay description:

```
TKB (RET)
TKB>USER,USER=NEWODL/MF
ENTER OPTIONS:
TKB>MISSEG=BASIC2
TKB>//
```

OVERLAY CAPABILITY

Note that the ODL file specification automatically terminates command input and the Task Builder automatically prompts for options.

The memory diagram for the COBOL and BASIC-PLUS-2 versions of USER is shown in Figure 5-12 below:

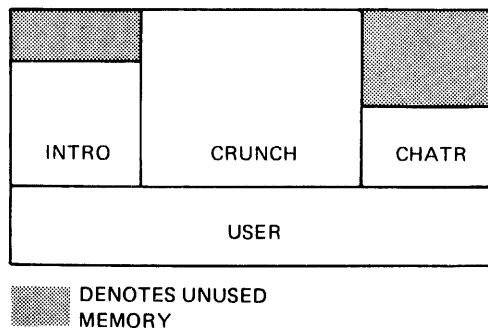


Figure 5-12 USER Block Diagram

5.3 SUBROUTINE COMMUNICATION

The three subroutines of the example program USER (INTRO, CRUNCH, and CHATR) cannot transfer data among themselves because of the trident-shaped tree structure. Any data stored in one module's copy of a storage area is lost when another module, with its unaltered copy of the original storage area is loaded. To transfer data, common storage areas must be forced to the root or to a segment accessible by all three calling segments. If your common storage area is not in the root segment, you run the risk of losing your data when the non-root segment containing your data is overlaid.

A .PSECT directive added to the overlay description forces common storage areas to the root of the tree. The actual allocation is made by using the PSECT name in the .ROOT directive so that the three modules can communicate with one another. An overlay description solving this problem might look like this:

```
.ROOT RDIN-RPRT-ADTA-*(PROC1,PROC2,P3FCTR)
P3FCTR:   .FCTR PROC3-(SUB1,SUB2)
          .PSECT ADTA,RW,GBL,REL,OVR,D
          .END
```

Figures 5-13 and 5-14 contain the ODL files for the BASIC-PLUS-2 and COBOL versions of USER, respectively. The reason for the more complex COBOL ODL file is that COBOL deals with PSECTS where BASIC-PLUS-2 deals with modules. Consult the PDP-11 COBOL User's Guide for additional information on COBOL-generated ODL files. The COBOL ODL file in Figure 5-14 was generated by the system program CBLMRG (the COBOL Merge program).

```
LIBR:     .ROOT USER-LIBR-*(INTRO-LIBR,CRUNCH-LIBR,CHATR-LIBR)
          .FCTR [1,1]BASIC2/LB
          .END
```

Figure 5-13 BASIC-PLUS-2 USER ODL File

OVERLAY CAPABILITY

```
!MERGED ODL FILE CREATED ON 20-JUL-77 AT 14:11:29
!COBOL STANDARD ODL FILE GENERATED ON: 19-JUL-77    08:56:43
!COBOBJ=USER.OBJ
!COBMAIN
!COBOL STANDARD ODL FILE GENERATED ON: 14-JUL-77    11:10:12
!COBOBJ=INTRO.OBJ
!COBKER=IN
      .NAME IN$003,GBL
      .PSECT $IN002,GBL,I,RW,CON
IN003$: .FCTR *IN$003-$IN002
      .NAME IN$005,GBL
      .PSECT $IN001,GBL,I,RW,CON
IN005$: .FCTR *IN$005-$IN001
IN0VR$: .FCTR      IN003$,IN005$
!COBOL STANDARD ODL FILE GENERATED ON: 14-JUL-77    15:26:19
!COBOBJ=CRUNCH.OBJ
!COBKER=CR
      .NAME CR$003,GBL
      .PSECT $CRO02,GBL,I,RW,CON
CR003$: .FCTR *CR$003-$CRO02
      .NAME CR$005,GBL
      .PSECT $CRO01,GBL,I,RW,CON
CR005$: .FCTR *CR$005-$CRO01
CR0VR$: .FCTR      CR003$,CR005$
!COBOL STANDARD ODL FILE GENERATED ON: 14-JUL-77    15:27:19
!COBOBJ=CHATR.OBJ
!COBKER=CH
      .NAME CH$003,GBL
      .PSECT $CHO02,GBL,I,RW,CON
CH003$: .FCTR *CH$003-$CHO02
      .NAME CH$005,GBL
      .PSECT $CHO01,GBL,I,RW,CON
CH005$: .FCTR *CH$005-$CHO01
CH0VR$: .FCTR      CH003$,CH005$
CBOBJ$: .FCTR USER.OBJ-INTRO.OBJ-CRUNCH.OBJ-CHATR.OBJ
CBOVR$: .FCTR IN0VR$,CR0VR$,CH0VR$
CBOTS$: .FCTR E1,1)COBLIB/LB
RMS$: .FCTR E1,1)RMSLIB/LB
OBJRT$: .FCTR CB0BJ$-CBOTS$-RMS$
      .ROOT OBJRT$-(CBOVR$)
      .END
```

Figure 5-14 COBOL USER ODL File

Figures 5-15 and 5-16 show the memory allocation maps that correspond to these ODL files.

OVERLAY CAPABILITY

CUSER.TSK MEMORY ALLOCATION MAP TKB M26
 28-JUL-77 17:01 PAGE 1

PARTITION NAME :	GEN
IDENTIFICATION :	209164
TASK UIC :	[200,47]
STACK LIMITS:	001000 001777 001000 00512.
PRG XFR ADDRESS:	027210
TOTAL ADDRESS WINDOWS:	1.
TASK IMAGE SIZE :	7936 WORDS
TASK ADDRESS LIMITS:	000000 036737

CUSER.TSK OVERLAY DESCRIPTION:

BASE	TOP	LENGTH	
-----	-----	-----	-----
000000	036443	036444	15652.
036444	036523	000060	00048.
036444	036723	000260	00176.
036444	036563	00012U	00080.
036444	036737	000274	00188.
036444	036563	00012U	00080.
036444	036657	000214	00140.

CUSER.TSK MEMORY ALLOCATION MAP TKB M26
 28-JUL-77 17:01 PAGE 2
 MD0

*** ROOT SEGMENT: MD0

R/W MEM LIMITS: 000000 036443 036444 15652.
 DISK BLK LIMITS: 000002 000040 000037 00031.

Figure 5-15 User COBOL Memory Allocation Map

OVERLAY CAPABILITY

MEMORY ALLOCATION SYNOPSIS:

SECTION		TITLE	IDENT	FILE
• BLK.: (RW, I, ICL, REL, CON)	002000 000440 00288.			
ACDDAT: (RW, D, GBL, REL, OVR)	002440 000244 01164.			
ACDINS: (RW, I, GBL, REL, OVR)	002440 000244 01164.	ACDDQIO	1A.17	COBLIB.OLB
ARITH : (RW, I, GBL, REL, OVR)	002704 001356 00750.	ACDQIO	1A.17	COBLIB.OLB
EDITI : (RW, I, GBL, REL, OVR)	002704 001356 00750.	ACDQIO	1A.17	COBLIB.OLB
EDITXD: (RW, D, GBL, REL, OVR)	004262 005262 02738.	ADDSSUB	1A.07	COBLIB.OLB
EXECDL: (RW, D, GBL, REL, OVR)	004262 005262 02738.	ADDSSUB	1A.07	COBLIB.OLB
EXEC12: (RW, I, GBL, REL, OVR)	011544 001522 00850.	EDITX	1A.3	COBLIB.OLB
EXEC12: (RW, I, GBL, REL, OVR)	013266 000026 00022.	EDITX	1A.3	COBLIB.OLB
MSGPSD: (RW, D, GBL, REL, OVR)	013266 000026 00022.	EDITX	1A.3	COBLIB.OLB
MSGPSR: (RO, I, GBL, REL, OVR)	013314 000362 00242.	EXEC	1A.24	COBLIB.OLB
EXEC12: (RW, I, GBL, REL, OVR)	013314 000362 00242.	EXEC	1A.24	COBLIB.OLB
EXEC12: (RW, I, GBL, REL, OVR)	013676 006414 03340.	EXEC	1A.24	COBLIB.OLB
MSGPSR: (RW, D, GBL, REL, OVR)	022312 000152 00106.	MSGPSR	1A.09	COBLIB.OLB
MSGPSR: (RO, I, GBL, REL, OVR)	022312 000152 00106.	MSGPSR	1A.09	COBLIB.OLB
UTIL : (RW, I, GBL, REL, OVR)	030422 004420 02320.	MSGPSR	1A.09	COBLIB.OLB
UTIL : (RW, I, GBL, REL, OVR)	022464 000214 00140.	UTIL	1A.4	COBLIB.OLB
UTILD : (RW, D, GBL, REL, OVR)	022464 000214 00140.	UTIL	1A.4	COBLIB.OLB
SCBBDO: (RW, I, GBL, REL, OVR)	022700 000012 00010.	UTIL	1A.4	COBLIB.OLB
SCBBD1: (RW, I, GBL, REL, CON)	022700 000012 00010.	UTIL	1A.4	COBLIB.OLB
SCBBDO: (RW, I, GBL, REL, OVR)	022712 000000 00000.	TASKCA	1A.04	COBLIB.OLB
SCBBD1: (RW, I, GBL, REL, CON)	022712 000000 00000.	TASKCA	1A.04	COBLIB.OLB
SCBBD2: (RW, I, GBL, REL, OVR)	022712 000000 00000.	USER	209164	MD0.OBJ
SCBFA0: (RW, I, GBL, REL, OVR)	022712 000000 00000.	INTRO	209169	MD1.OBJ
SCBFA1: (RW, I, GBL, REL, OVR)	022712 000000 00000.	CRUNCH	209169	MD2.OBJ
SCBFD0: (RW, I, GBL, REL, OVR)	022712 000000 00000.	CHATR	209169	MD3.OBJ

Figure 5-15 (Cont.) User COBOL Memory Allocation Map

OVERLAY CAPABILITY

SCBF01: (RW, I, GBL, REL, CON)	022712 000000 00000.	TASKCA 1A. 04 COBLIB.OLB
	022712 000000 00000.	USER 209164 MD0.OBJ
	022712 000000 00000.	INTRO 209169 MD1.OBJ
	022712 000000 00000.	CRUNCH 209169 MD2.OBJ
	022712 000000 00000.	CHATR 209169 MD3.OBJ
SCBF02: (RW, I, GBL, REL, OVR)	022712 000002 00002.	
	022712 000002 00002.	TASKCA 1A. 04 COBLIB.OLB
SCBIF0: (RW, I, GBL, REL, OVR)	022714 000000 00000.	
	022714 000000 00000.	TASKCA 1A. 04 COBLIB.OLB
SCBIF1: (RW, I, GBL, REL, CON)	022714 000000 00000.	
	022714 000000 00000.	USER 209164 MD0.OBJ
	022714 000000 00000.	INTRO 209169 MD1.OBJ
	022714 000000 00000.	CRUNCH 209169 MD2.OBJ
	022714 000000 00000.	CHATR 209169 MD3.OBJ
SCBIF2: (RW, I, GBL, REL, OVR)	022714 000000 00000.	
	022714 000000 00000.	TASKCA 1A. 04 COBLIB.OLB
SCBIOT: (RW, I, GBL, REL, OVR)	022714 000132 00090.	
	022714 000132 00090.	USER 209164 MD0.OBJ
	022714 000132 00090.	INTRO 209169 MD1.OBJ
	022714 000132 00090.	CRUNCH 209169 MD2.OBJ
	022714 000132 00090.	CHATR 209169 MD3.OBJ
SCBIR0: (RW, I, GBL, REL, OVR)	023046 000000 00000.	
	023046 000000 00000.	TASKCA 1A. 04 COBLIB.OLB
SCBIR1: (RW, I, GBL, REL, CON)	023046 000000 00000.	
	023046 000000 00000.	USER 209164 MD0.OBJ
	023046 000000 00000.	INTRO 209169 MD1.OBJ
	023046 000000 00000.	CRUNCH 209169 MD2.OBJ
	023046 000000 00000.	CHATR 209169 MD3.OBJ
SCBIR2: (RW, I, GBL, REL, OVR)	023046 000000 00000.	
	023046 000000 00000.	TASKCA 1A. 04 COBLIB.OLB
SCBKBU: (RW, I, GBL, REL, OVR)	023046 000000 00000.	
	023046 000000 00000.	TASKCA 1A. 04 COBLIB.OLB
SCBKBL: (RW, I, GBL, REL, CON)	023046 000000 00000.	
	023046 000000 00000.	USER 209164 MD0.OBJ
	023046 000000 00000.	INTRO 209169 MD1.OBJ
	023046 000000 00000.	CRUNCH 209169 MD2.OBJ
	023046 000000 00000.	CHATR 209169 MD3.OBJ
SCBKBL2: (RW, I, GBL, REL, OVR)	023046 000000 00000.	
	023046 000000 00000.	TASKCA 1A. 04 COBLIB.OLB
SCBKD0: (RW, I, GBL, REL, OVR)	023046 000000 00000.	
	023046 000000 00000.	TASKCA 1A. 04 COBLIB.OLB
SCBKD1: (RW, I, GBL, REL, CON)	023046 000000 00000.	
	023046 000000 00000.	USER 209164 MD0.OBJ
	023046 000000 00000.	INTRO 209169 MD1.OBJ

Figure 5-15 (Cont.) User COBOL Memory Allocation Map

OVERLAY CAPABILITY

```

023046 000000 00000. CRUNCH 209169 MD2.OBJ
023046 000000 00000. CHATTR 209169 MD3.OBJ
$CBKD2: (RW, I, GBL, REL, OVR) 023046 000000 00000. TASKCA 1A. 04 COBLIB.OLB

$CBSWT: (RW, I, GBL, REL, OVR) 023046 000002 00002. USER 209164 MD0.OBJ
023046 000002 00002. INTRO 209169 MD1.OBJ
023046 000002 00002. CRUNCH 209169 MD2.OBJ
023046 000002 00002. CHATTR 209169 MD3.OBJ
023046 000000 00000. TASKCA 1A. 04 COBLIB.OLB

$CBTSK: (RW, I, GBL, REL, OVR) 023050 000040 00032. USER 209164 MD0.OBJ
023050 000040 00032. INTRO 209169 MD1.OBJ
023050 000040 00032. CRUNCH 209169 MD2.OBJ
023050 000040 00032. CHATTR 209169 MD3.OBJ
023110 000000 00000. TASKCA 1A. 04 COBLIB.OLB

$CBXA0: (RW, I, GBL, REL, OVR) 023110 000000 00000. USER 209164 MD0.OBJ
023110 000000 00000. INTRO 209169 MD1.OBJ
023110 000000 00000. CRUNCH 209169 MD2.OBJ
023110 000000 00000. CHATTR 209169 MD3.OBJ
023110 000000 00000. TASKCA 1A. 04 COBLIB.OLB

$CBXA1: (RW, I, GBL, REL, OVR) 023110 000000 00000. USER 209164 MD0.OBJ
023110 000000 00000. INTRO 209169 MD1.OBJ
023110 000000 00000. CRUNCH 209169 MD2.OBJ
023110 000000 00000. CHATTR 209169 MD3.OBJ
023110 000000 00000. TASKCA 1A. 04 COBLIB.OLB

$CBXA2: (RW, I, GBL, REL, OVR) 023110 000000 00000. USER 209164 MD0.OBJ
023110 000000 00000. INTRO 209169 MD1.OBJ
023110 000000 00000. CRUNCH 209169 MD2.OBJ
023110 000000 00000. CHATTR 209169 MD3.OBJ
023110 000000 00000. TASKCA 1A. 04 COBLIB.OLB

$CHADT: (RW, I, GBL, REL, CON) 023110 000000 00000. USER 209164 MD0.OBJ
023110 000000 00000. INTRO 209169 MD1.OBJ
023110 000000 00000. CRUNCH 209169 MD2.OBJ
023110 000000 00000. CHATTR 209169 MD3.OBJ
023110 000000 00000. TASKCA 1A. 04 COBLIB.OLB

$CHARG: (RW, I, GBL, REL, CON) 023110 000036 00030. USER 209164 MD0.OBJ
023110 000036 00030. INTRO 209169 MD1.OBJ
023110 000036 00030. CRUNCH 209169 MD2.OBJ
023110 000036 00030. CHATTR 209169 MD3.OBJ
023110 000000 00000. TASKCA 1A. 04 COBLIB.OLB

$CHDAT: (RW, I, GBL, REL, CON) 023146 000224 00148. USER 209164 MD0.OBJ
023146 000224 00148. INTRO 209169 MD1.OBJ
023146 000224 00148. CRUNCH 209169 MD2.OBJ
023146 000224 00148. CHATTR 209169 MD3.OBJ
023146 000000 00000. TASKCA 1A. 04 COBLIB.OLB

$CHDDD: (RW, I, GBL, REL, CON) 023372 000000 00000. USER 209164 MD0.OBJ
023372 000000 00000. INTRO 209169 MD1.OBJ
023372 000000 00000. CRUNCH 209169 MD2.OBJ
023372 000000 00000. CHATTR 209169 MD3.OBJ
023372 000000 00000. TASKCA 1A. 04 COBLIB.OLB

$CHENT: (RW, I, GBL, REL, CON) 023372 000036 00030. USER 209164 MD0.OBJ
023372 000036 00030. INTRO 209169 MD1.OBJ
023372 000036 00030. CRUNCH 209169 MD2.OBJ
023372 000036 00030. CHATTR 209169 MD3.OBJ
023372 000000 00000. TASKCA 1A. 04 COBLIB.OLB

$CHIOB: (RW, I, GBL, REL, CON) 023430 000000 00000. USER 209164 MD0.OBJ
023430 000000 00000. INTRO 209169 MD1.OBJ
023430 000000 00000. CRUNCH 209169 MD2.OBJ
023430 000000 00000. CHATTR 209169 MD3.OBJ
023430 000000 00000. TASKCA 1A. 04 COBLIB.OLB

$CHLIT: (RW, I, GBL, REL, CON) 023430 000262 00178. USER 209164 MD0.OBJ
023430 000262 00178. INTRO 209169 MD1.OBJ
023430 000262 00178. CRUNCH 209169 MD2.OBJ
023430 000262 00178. CHATTR 209169 MD3.OBJ
023430 000000 00000. TASKCA 1A. 04 COBLIB.OLB

$CHLST: (RW, I, GBL, REL, CON) 023712 000002 00002. USER 209164 MD0.OBJ
023712 000002 00002. INTRO 209169 MD1.OBJ
023712 000002 00002. CRUNCH 209169 MD2.OBJ
023712 000002 00002. CHATTR 209169 MD3.OBJ
023712 000000 00000. TASKCA 1A. 04 COBLIB.OLB

$CHLTD: (RW, I, GBL, REL, CON) 023714 000060 00048. USER 209164 MD0.OBJ
023714 000060 00048. INTRO 209169 MD1.OBJ
023714 000060 00048. CRUNCH 209169 MD2.OBJ
023714 000060 00048. CHATTR 209169 MD3.OBJ
023714 000000 00000. TASKCA 1A. 04 COBLIB.OLB

$CHPDT: (RW, I, GBL, REL, CON) 023774 000010 00008. USER 209164 MD0.OBJ
023774 000010 00008. INTRO 209169 MD1.OBJ
023774 000010 00008. CRUNCH 209169 MD2.OBJ
023774 000010 00008. CHATTR 209169 MD3.OBJ
023774 000000 00000. TASKCA 1A. 04 COBLIB.OLB

$CHPPM: (RW, I, GBL, REL, CON) 024004 000214 00140. USER 209164 MD0.OBJ
024004 000214 00140. INTRO 209169 MD1.OBJ
024004 000214 00140. CRUNCH 209169 MD2.OBJ
024004 000214 00140. CHATTR 209169 MD3.OBJ
024004 000000 00000. TASKCA 1A. 04 COBLIB.OLB

$CHSDT: (RW, I, GBL, REL, CON) 024220 000000 00000. USER 209164 MD0.OBJ
024220 000000 00000. INTRO 209169 MD1.OBJ
024220 000000 00000. CRUNCH 209169 MD2.OBJ
024220 000000 00000. CHATTR 209169 MD3.OBJ
024220 000000 00000. TASKCA 1A. 04 COBLIB.OLB

```

Figure 5-15 (Cont.) User COBOL Memory Allocation Map

OVERLAY CAPABILITY

```

$SCHUSE: (RW, I, GBL, REL, CON) 024220 000030 00024. CHATR 209169 MD3.OBJ
$SCHWRK: (RW, I, GBL, REL, CON) 024220 000030 00024. CHATR 209169 MD3.OBJ
$CH001: (RW, I, GBL, REL, CON) 024250 000102 00066. CHATR 209169 MD3.OBJ
$CH002: (RW, I, GBL, REL, CON) 036444 000214 00140. CHATR 209169 MD3.OBJ
$CRADT: (RW, I, GBL, REL, CON) 036444 000214 00140. CHATR 209169 MD3.OBJ
$CRARG: (RW, I, GBL, REL, CON) 036444 000116 00078. CHATR 209169 MD3.OBJ
$CRDAT: (RW, I, GBL, REL, CON) 036444 000116 00078. CHATR 209169 MD3.OBJ
$CRDST: (RW, I, GBL, REL, CON) 024352 000000 00000. CRUNCH 209169 MD2.OBJ
$CRDDD: (RW, I, GBL, REL, CON) 024352 000036 00030. CRUNCH 209169 MD2.OBJ
$CRENT: (RW, I, GBL, REL, CON) 024352 000036 00030. CRUNCH 209169 MD2.OBJ
$CRIOB: (RW, I, GBL, REL, CON) 024410 000266 00182. CRUNCH 209169 MD2.OBJ
$CRDDO: (RW, I, GBL, REL, CON) 024410 000060 00048. CRUNCH 209169 MD2.OBJ
$CRSLIT: (RW, I, GBL, REL, CON) 024476 000060 00048. CRUNCH 209169 MD2.OBJ
$CRSLST: (RW, I, GBL, REL, CON) 024476 000036 00030. CRUNCH 209169 MD2.OBJ
$CRSLTD: (RW, I, GBL, REL, CON) 025014 000000 00000. CRUNCH 209169 MD2.OBJ
$CRSPDT: (RW, I, GBL, REL, CON) 025014 000012 00010. CRUNCH 209169 MD2.OBJ
$CRSPFM: (RW, I, GBL, REL, CON) 025026 000002 00002. CRUNCH 209169 MD2.OBJ
$CRSLTD: (RW, I, GBL, REL, CON) 025030 000014 00012. CRUNCH 209169 MD2.OBJ
$CRSPDT: (RW, I, GBL, REL, CON) 025044 000010 00008. CRUNCH 209169 MD2.OBJ
$CRUSE: (RW, I, GBL, REL, CON) 025054 000214 00140. CRUNCH 209169 MD2.OBJ
$CRWRK: (RW, I, GBL, REL, CON) 025270 000000 00000. CRUNCH 209169 MD2.OBJ
$CR001: (RW, I, GBL, REL, CON) 025270 000030 00024. CRUNCH 209169 MD2.OBJ
$CR002: (RW, I, GBL, REL, CON) 025320 000102 00066. CRUNCH 209169 MD2.OBJ
$INADT: (RW, I, GBL, REL, CON) 025444 000272 00186. CRUNCH 209169 MD2.OBJ
$INARG: (RW, I, GBL, REL, CON) 025444 000116 00078. CRUNCH 209169 MD2.OBJ
$INADT: (RW, I, GBL, REL, CON) 025442 000000 00000. INTRO 209169 MD1.OBJ
$INARG: (RW, I, GBL, REL, CON) 025442 000014 00012. INTRO 209169 MD1.OBJ

```

Figure 5-15 (Cont.) User COBOL Memory Allocation Map

OVERLAY CAPABILITY

```

$INDAT: (RW ,I ,GBL ,REL ,CON) 025436 000224 00148.
$INDDD: (RW ,I ,GBL ,REL ,CON) 025436 000224 00148. INTRO 209169 MD1.OBJ
$INENT: (RW ,I ,GBL ,REL ,CON) 025662 000000 00000. INTRO 209169 MD1.OBJ
$INIOB: (RW ,I ,GBL ,REL ,CON) 025662 000000 00000. INTRO 209169 MD1.OBJ
$INLIT: (RW ,I ,GBL ,REL ,CON) 025720 000000 00000. INTRO 209169 MD1.OBJ
$INLIST: (RW ,I ,GBL ,REL ,CON) 025720 000314 00204. INTRO 209169 MD1.OBJ
$INLTD: (RW ,I ,GBL ,REL ,CON) 026234 000002 00002. INTRO 209169 MD1.OBJ
$INPDT: (RW ,I ,GBL ,REL ,CON) 026234 000002 00002. INTRO 209169 MD1.OBJ
$INPPM: (RW ,I ,GBL ,REL ,CON) 026236 000052 00042. INTRO 209169 MD1.OBJ
$INSDT: (RW ,I ,GBL ,REL ,CON) 026310 000010 00008. INTRO 209169 MD1.OBJ
$INUSE: (RW ,I ,GBL ,REL ,CON) 026310 000010 00008. INTRO 209169 MD1.OBJ
$INWRK: (RW ,I ,GBL ,REL ,CON) 026320 000214 00140. INTRO 209169 MD1.OBJ
$INSDT: (RW ,I ,GBL ,REL ,CON) 026534 000000 00000. INTRO 209169 MD1.OBJ
$INUSE: (RW ,I ,GBL ,REL ,CON) 026534 000000 00024. INTRO 209169 MD1.OBJ
$INWRK: (RW ,I ,GBL ,REL ,CON) 026564 000102 00066. INTRO 209169 MD1.OBJ
$IN001: (RW ,I ,GBL ,REL ,CON) 036444 000256 00174. INTRO 209169 MD1.OBJ
$IN002: (RW ,I ,GBL ,REL ,CON) 036444 000030 00024. INTRO 209169 MD1.OBJ
$IN002: (RW ,I ,GBL ,REL ,CON) 026564 000102 00066. INTRO 209169 MD1.OBJ
$IN002: (RW ,I ,GBL ,REL ,CON) 036444 000060 00048. INTRO 209169 MD1.OBJ
$USADT: (RW ,I ,GBL ,REL ,CON) 0266666 000000 00000. USER 209164 MD0.OBJ
$USARG: (RW ,I ,GBL ,REL ,CON) 0266666 000000 00000. USER 209164 MD0.OBJ
$USDAT: (RW ,I ,GBL ,REL ,CON) 0266666 000264 00180. USER 209164 MD0.OBJ
$USDDD: (RW ,I ,GBL ,REL ,CON) 0266666 000264 00180. USER 209164 MD0.OBJ
$USENT: (RW ,I ,GBL ,REL ,CON) 0271152 000036 00030. USER 209164 MD0.OBJ
$USIOB: (RW ,I ,GBL ,REL ,CON) 0271152 000036 00030. USER 209164 MD0.OBJ
$USLIT: (RW ,I ,GBL ,REL ,CON) 027230 000010 00016. USER 209164 MD0.OBJ
$USLST: (RW ,I ,GBL ,REL ,CON) 027230 000010 00008. USER 209164 MD0.OBJ
$USLTD: (RW ,I ,GBL ,REL ,CON) 027240 000014 00012. USER 209164 MD0.OBJ
$USLTD: (RW ,I ,GBL ,REL ,CON) 027254 000006 00006.

```

Figure 5-15 (Cont.) User COBOL Memory Allocation Map

OVERLAY CAPABILITY

§USPDT: (RW, I, GBL, REL, CON)	027254	000006	00006.	USER	209164	MDO.OBJ
	027262	000004	00004.		209164	
§USPFM: (RW, I, GBL, REL, CON)	027266	000214	00140.	USER	209164	MDO.OBJ
	027266	000214	00140.		209164	
§USSDT: (RW, I, GBL, REL, CON)	027502	000006	00006.	USER	209164	MDO.OBJ
	027502	000006	00006.		209164	
§USUSE: (RW, I, GBL, REL, CON)	027510	000030	00024.	USER	209164	MDO.OBJ
	027510	000030	00024.		209164	
§USWRK: (RW, I, GBL, REL, CON)	027540	000102	00066.	USER	209164	MDO.OBJ
	027540	000102	00066.		209164	
§US001: (RW, I, GBL, REL, CON)	027642	000130	00088.	USER	209164	MDO.OBJ
	027642	000130	00088.		209164	
§XABRT: (RO, I, GBL, REL, CON)	035042	000022	00018.	XGO	1A.21	COBLIB.OLB
	035042	000022	00018.		1A.21	
§XALT : (RO, I, GBL, REL, CON)	035064	000014	00012.	XGO	1A.21	COBLIB.OLB
	035064	000014	00012.		1A.21	
§XCALL: (RO, I, GBL, REL, CON)	035100	000130	00088.	XCALL	1A.08	COBLIB.OLB
	035100	000130	00088.		1A.08	
§XDDDI: (RO, I, GBL, REL, CON)	035230	000042	00034.	XCALL	1A.08	COBLIB.OLB
	035230	000042	00034.		1A.08	
§XENDP: (RO, I, GBL, REL, CON)	035272	000152	00106.	XGO	1A.21	COBLIB.OLB
	035272	000152	00106.		1A.21	
§XERR : (RO, I, GBL, REL, CON)	035444	000020	00016.	XGO	1A.21	COBLIB.OLB
	035444	000020	00016.		1A.21	
§XEXIT: (RO, I, GBL, REL, CON)	035464	000014	00012.	XCALL	1A.08	COBLIB.OLB
	035464	000014	00012.		1A.08	
§XGO : (RO, I, GBL, REL, CON)	035500	000120	00080.	XGO	1A.21	COBLIB.OLB
	035500	000120	00080.		1A.21	
§XGOD : (RO, I, GBL, REL, CON)	035620	000052	00042.	XGO	1A.21	COBLIB.OLB
	035620	000052	00042.		1A.21	
§XGOUN: (RO, I, GBL, REL, CON)	035672	000032	00026.	XGO	1A.21	COBLIB.OLB
	035672	000032	00026.		1A.21	
§XINIT: (RO, I, GBL, REL, CON)	035724	000070	00056.	XGO	1A.21	COBLIB.OLB
	035724	000070	00056.		1A.21	
§XSTOP: (RO, I, GBL, REL, CON)	036014	000016	00014.	XGO	1A.21	COBLIB.OLB
	036014	000016	00014.		1A.21	
§XSTPR: (RO, I, GBL, REL, CON)	036032	000046	00038.	XGO	1A.21	COBLIB.OLB
	036032	000046	00038.		1A.21	
§XSUBK: (RO, I, GBL, REL, CON)	036100	000076	00062.	XCALL	1A.08	COBLIB.OLB
	036100	000076	00062.		1A.08	
§SALER: (RW, I, LCL, REL, CON)	027772	000024	00020.			
	027772	000024	00020.			
§SALYC: (RW, D, LCL, REL, CON)	030016	000060	00048.			
	030016	000060	00048.			
§SAUTO: (RW, I, LCL, REL, CON)	030076	000130	00088.			
	030076	000130	00088.			
§\$MRKS: (RO, I, LCL, REL, QVR)	036176	000076	00062.			
	036176	000076	00062.			

Figure 5-15 (Cont.) User COBOL Memory Allocation Map

OVERLAY CAPABILITY

GLOBAL SYMBOLS:											
\$SOVDT: (RW, I, LCL, ABS, CON)	0000000	0000000	0000000	0000000	0000000	0000000	0000000	0000000	0000000	0000000	0000000
\$SOVRS: (RW, D, LCL, REL, OVR)	030226	000020	00016								
\$SRDSSG: (RO, I, LCL, REL, OVR)	036274	000150	00104								
\$SPRESL: (RW, I, LCL, REL, CON)	030246	000024	00020								
\$SSRGDS: (RW, D, LCL, REL, CON)	030272	000000	00000								
\$SPRTS : (RW, I, GBL, REL, OVR)	030272	000002	00002								
\$SSGD0: (RW, D, LCL, REL, OVR)	030274	000000	00000								
\$SSGD1: (RW, D, LCL, REL, CON)	030274	000124	00084								
\$SSGD2: (RW, D, LCL, REL, OVR)	030420	000002	00002								
\$SWNDS: (RW, D, LCL, REL, CON)	030422	000000	00000								
ACCBUF 002514-R	C1FC	013676-R	HDL	013406-R	PUTCMG	022464-R	USER	027210-R	\$XCHDR	020322-R	\$XMBD
ACCCQIO 003624-R	C2FC	013714-R	IDXMS1	022702-R	QIOPB	002440-R	SXFV	006602-R	\$XCNDT	020346-R	\$XMCC
ADDET 003572-R	C2FL	013760-R	INTEG	034134-R	RNDFL	013414-R	USSGN	006372-R	\$XDDDI	035230-R	\$XMDB
ADDEV 002476-R	C3F	013742-R	INTRO	025662-R	RSCRY	013610-R	WFB1	013420-R	\$XDIVB	010324-R	\$XMDD
ADLUN 002474-R	DCMPLPT	022700-R	IN\$003	030026-R	RSTRN	022660-R	WFB2	013440-R	\$XDIVR	010316-R	\$XMED
ADSET 003172-R	DISQIO	003744-R	IN\$005	030016-R	SAVE	022640-R	WFB3	013460-R	\$XEACC	003010-R	\$XMJR
ASLUN 003460-R	DSETNG	006532-R	IXDDD	013622-R	SEPSGN	006512-R	WFB4	013472-R	\$XEDIS	003252-R	\$XMANA
AWFB1 013430-R	DSETPS	006514-R	LDL	013410-R	SIGNF	013604-R	WFB6	013542-R	\$XENDP	035272-R	\$XMANAE
AWFB2 013450-R	EASTP	006754-R	LNHLD	013606-R	SIZFLG	013616-R	WFD1	013552-R	\$XERR	035444-R	\$XMULB
AWFB3 013470-R	EMAIL	016742-R	MASKPT	020376-R	SSTBSWT	035022-R	SCBTSK	023046-R	\$XEXIT	035464-R	\$XMULR
AWFB4 013502-R	EMBB	015356-R	MSG	002507-R	SIZEFLG	013620-R	SCBTSK	023050-R	\$XGO	035500-R	\$XNGAT
AWFB6 013532-R	EMBX	015320-R	MSGPTL	033324-R	TTYBUF	002504-R	SXABRT	035042-R	\$XGOD	035620-R	\$XPTPR
AWFD1 013562-R	EMCC	016674-R	MSGPTW	033316-R	UBADD	007050-R	\$XACCS	002704-R	\$XGOFL	035574-R	\$XSUBB
BAN 004422-R	EMCE	013070-R	MSGRTL	033000-R	UCFV	006572-R	\$XADBR	006652-R	\$XGOR5	035500-R	\$XSUSIZ
BAR 004412-R	EMDD	017074-R	MSGRTN	032772-R	UDCS	005456-R	\$XADDB	006660-R	\$XGOSP	035602-R	\$XSTOP
BININS 014022-R	EMDE	011574-R	M.DPID	011472-R	UDEA	014130-R	\$XADD	004462-R	\$XGOTR	035564-R	
CDN 013412-R	EMJR	015224-R	M4DPID	011124-R	UDXP	015030-R	\$XADDR	004454-R	\$XGOUN	035672-R	\$XSUBB
CHATR 023372-R	EMXB	015274-R	NEGQAD	016236-R	UGBR	010216-R	\$XALDT	035064-R	\$XIFNA	022164-R	\$XSUBD
CH\$003 030066-R	ESGNP	006466-R	NGFLD0	011110-R	UGD	005230-R	\$XCAL	021530-R	\$XIFSN	021664-R	\$XSUBK
CH\$005 030056-R	FCP1	013320-R	NGFLD1	011046-R	UGFC	005754-R	\$XCALL	035100-R	\$XIFUN	022052-R	\$XSUBR
CONDRI 013614-R	FCP2	013322-R	OPCRY	013416-R	UGFC1	005740-R	\$XCALLS	021514-R	\$XINIT	035724-R	
CRFLQI 003350-R	FCP3	013324-R	PARAM	013626-R	UGREV1	006330-R	\$XCBB	021110-R	\$XIXBY	015124-R	\$XSZEC
CRUNCH 024756-R	FCS	013326-R	PARAM1	013666-R	UMB	016770-R	\$XCCC	020712-R	\$XIXCP	015134-R	
CR\$003 030046-R	FIRLUN	013316-R	PARAM2	013670-R	UMLQ	010106-R	\$XCCCS	020676-R	\$XMAL	016730-R	
CR\$005 030036-R	GETSWT	004104-R	PARAM3	013672-R	UMND	020210-R	\$XCCDD	021374-R	\$XMALD	013230-R	
CTEST 013624-R	GETUB	014664-R	PARAM4	013674-R	USD	005320-R	\$XCHD	020326-R	\$XMBB	015344-R	

Figure 5-15 (Cont.) User COBOL Memory Allocation Map

OVERLAY CAPABILITY

CUSER.TSK MEMORY ALLOCATION MAP TKB M26
IN\$003 28-JUL-77 17:01 PAGE 8

*** SEGMENT: IN\$003

R/W MEM LIMITS: 036444 036523 000060 00048.
DISK BLK LIMITS: 000041 000041 000001 00001.

MEMORY ALLOCATION SYNOPSIS:

SECTION	TITLE	IDENT	FILE
\$IN002: (RW,I,GBL,REL,CON)	035444	000060	00048.
\$\$ALVC: (RW,D,LCL,REL,CON)	036524	000000	00000.
\$SRTS : (RW,I,GBL,REL,OVR)	030272	000002	00002.

GLOBAL SYMBOLS:

IN\$003 030272-R

CUSER.TSK MEMORY ALLOCATION MAP TKB M26
IN\$005 28-JUL-77 17:01 PAGE 9

*** SEGMENT: IN\$005

R/W MEM LIMITS: 036444 036723 000260 00176.
DISK BLK LIMITS: 000042 000042 000001 00001.

MEMORY ALLOCATION SYNOPSIS:

SECTION	TITLE	IDENT	FILE
\$IN001: (RW,I,GBL,REL,CON)	036444	000256	00174.
\$\$ALVC: (RW,D,LCL,REL,CON)	036722	000000	00000.
\$SRTS : (RW,I,GBL,REL,OVR)	030272	000002	00002.

Figure 5-15 (Cont.) User COBOL Memory Allocation Map

OVERLAY CAPABILITY

GLOBAL SYMBOLS:
IN\$005 030272-R

CUSER.TSK MEMORY ALLOCATION MAP TKB M26
CR\$003 28-JUL-77 17:01 PAGE 10

*** SEGMENT: CR\$003

R/W MEM LIMITS: 036444 036563 000120 00080.
DISK BLK LIMITS: 000043 000043 000001 00001.

MEMORY ALLOCATION SYNOPSIS:

SECTION	TITLE	IDENT	FILE
-----	-----	-----	-----
SCR002: (RW,I,GBL,REL,CON)	036444	000116	00078.
\$\$ALVC: (RW,D,LCL,REL,CON)	036562	000000	00000.
\$\$RTS : (RW,I,GBL,REL,OVR)	030272	000002	00002.

GLOBAL SYMBOLS:
CR\$003 030272-R

CUSER.TSK MEMORY ALLOCATION MAP TKB M26
CR\$005 28-JUL-77 17:01 PAGE 11

*** SEGMENT: CR\$005

R/W MEM LIMITS: 036444 036737 000274 00188.
DISK BLK LIMITS: 000044 000044 000001 00001.

Figure 5-15 (Cont.) User COBOL Memory Allocation Map

OVERLAY CAPABILITY

MEMORY ALLOCATION SYNOPSIS:

SECTION	TITLE	IDENT	FILE
SCRU01: (RW, I, GBL, REL, CON)	036444	000272	00186.
SSALVC: (RW, D, LCL, REL, CON)	036736	000000	00000.
SSRTS : (RW, I, GBL, REL, OVR)	030272	000002	00002.

GLOBAL SYMBOLS:

CR\$005 030272-R

CUSER.TSK MEMORY ALLOCATION MAP TKB M26
CH\$003 28-JUL-77 17:01

*** SEGMENT: CH\$003

R/W MEM LIMITS: 036444 036563 000120 00080.
DISK BLK LIMITS: 000045 000045 000001 00001.

MEMORY ALLOCATION SYNOPSIS:

SECTION	TITLE	IDENT	FILE
SCH002: (RW, I, GBL, REL, CON)	036444	000116	00078.
SSALVC: (RW, D, LCL, REL, CON)	036562	000000	00000.
SSRTS : (RW, I, GBL, REL, OVR)	030272	000002	00002.

GLOBAL SYMBOLS:

CH\$003: 030272-R

Figure 5-15 (Cont.) User COBOL Memory Allocation Map

OVERLAY CAPABILITY

CUSER.TSK MEMORY ALLOCATION MAP TKB M26
CH\$005 28-JUL-77 17:01 PAGE 13

*** SEGMENT: CH\$005

R/W MEM LIMITS: 036444 036657 000214 00140.
DISK BLK LIMITS: 000046 000046 000001 000001.

MEMORY ALLOCATION SYNOPSIS:

SECTION	
SCH001:	(RW, I, GBL, REL, CON)
SSALVC:	(RW, D, LCL, REL, CON)
SSRTS :	(RW, I, GBL, REL, OVR)

MEMORY ALLOCATION SYNOPSIS:

GLOBAL SYMBOLS:

*** TASK BILL DEB STATISTICS:

TOTAL WORK FILE REFERENCES: 69063.
WORK FILE READS: 0.
WORK FILE WRITES: 0.
SIZE OF CORE POOL: 8548. WORDS (33. PAGES)
SIZE OF WORK FILE: 6656. WORDS (26. PAGES)

Figure 5-15 (Cont.) User COBOL Memory Allocation Map

USER.TSK MEMORY ALLOCATION MAP TKB M26 PAGE 1
18-JUL-77 11:41

```

PARTITION NAME : GEN
IDENTIFICATION : V01X03
TASK UIC : [200,47]
STACK LIMITS: 001000 001777 001000 00512.
PRG XFR ADDRESS: 002464
TOTAL ADDRESS WINDOWS: 2.
TASK IMAGE SIZE : 1600. WORDS
TASK ADDRESS LIMITS: 000000 006177

```

USER.TSK OVERLAY DESCRIPTION:

BASE	TOP	LENGTH	USER	INTRO	CRUNCH	CHATR
---	---	-----				
000000	005357	005360	02800.			
005360	005567	000210	00136.			
005360	005777	000420	00272.			
005360	006177	000620	00400.			

USER.TSK MEMORY ALLOCATION MAP TKB M26 PAGE 2
USER 18-JUL-77 11:41

*** ROOT SEGMENT: IISER

```
R/W MEM LIMITS: 000000 005357 005360 02800-
DISK BLK LIMITS: 000002 000007 000006 00006-
```

MEMORY ALLOCATION SYNOPSIS:

Figure 5-16 User BASIC-PLUS-2 Memory Allocation Map

OVERLAY CAPABILITY

SECTION		TITLE	IDENT	FILE
BLK. : (RW , I , LCL , REL , CON)	002000	000464	00308.	
BLK. : (RW , I , LCL , REL , CON)	002000	000464	00308.	\$CALLS 02CM BASIC2.OLB
SARRAY: (RW , D , LCL , REL , CON)	002464	000000	00000.	USER V01X03 USER. OBJ
SCODE : (RW , I , LCL , REL , CON)	002464	000000	00000.	
SFLAGR: (RW , D , GBL , REL , CON)	002464	000204	00132.	USER V01X03 USER. OBJ
SFLAGS: (RW , D , GBL , REL , CON)	002670	000000	00000.	
SFLAGS: (RW , D , GBL , REL , CON)	002670	000010	00008.	USER V01X03 USER. OBJ
SFLAGT: (RW , D , GBL , REL , CON)	002670	000002	00002.	USER V01X03 USER. OBJ
SIC101: (RW , D , GBL , REL , OVR)	002700	000000	00000.	USER V01X03 USER. OBJ
SIDATA: (RW , D , LCL , REL , CON)	002700	000200	00128.	USER V01X03 USER. OBJ
SIDATA: (RW , D , LCL , REL , CON)	002700	000200	00128.	
SPDATA: (RW , D , LCL , REL , CON)	003100	001454	00812.	USER V01X03 USER. OBJ
SPDATA: (RW , D , LCL , REL , CON)	003100	001454	00812.	
SSAVSP: (RW , D , LCL , REL , CON)	004554	000024	00020.	USER V01X03 USER. OBJ
SSAVSP: (RW , D , LCL , REL , CON)	004554	000024	00020.	
SSTRNG: (RW , D , LCL , REL , CON)	004600	000002	00002.	USER V01X03 USER. OBJ
SSTRNG: (RW , D , LCL , REL , CON)	004602	000000	00000.	
STDATA: (RW , D , LCL , REL , CON)	004602	000000	00000.	USER V01X03 USER. OBJ
STDATA: (RW , D , LCL , REL , CON)	004602	000000	00000.	
S\$ALER: (RW , I , LCL , REL , CON)	004602	000024	00020.	USER V01X03 USER. OBJ
S\$ALYC: (RW , D , LCL , REL , CON)	004626	000030	00024.	
S\$ALTO: (RW , I , LCL , REL , CON)	004656	000130	00088.	
S\$MARKS: (RO , I , LCL , REL , OVR)	005112	000076	00062.	
S\$COVDT: (RW , I , LCL , ABS , CON)	000000	000000	00000.	
S\$OVRSS: (RW , D , LCL , REL , OVR)	005006	000020	00016.	
S\$RDSSG: (RO , I , LCL , REL , OVR)	005210	000150	00104.	
S\$RGDS: (RW , D , LCL , REL , CON)	005026	000000	00000.	
S\$RTTS : (RW , I , GBL , REL , OVR)	005026	000002	00002.	
S\$SGDU: (RW , D , LCL , REL , OVR)	005030	000000	00000.	
S\$SGD1: (RW , D , LCL , REL , CON)	005030	000060	00048.	
S\$SGD2: (RW , D , LCL , REL , OVR)	005110	000002	00002.	
S\$WNDS: (RW , D , LCL , REL , CON)	005112	000000	00000.	

GLOBAL SYMBOLS:

Figure 5-16 (Cont.) User BASIC-PLUS-2 Memory Allocation Map

OVERLAY CAPABILITY

```
CALS 002146-R CBR$ 002442-R CHATR 004646-R CRUNCH 004636-R INTRO 004626-R SBE$ 002242-R $INITS 002000-R
$OTSVA 003150-R
```

```
USER-TSK MEMORY ALLOCATION MAP TKB M26
INTRO 18-JUL-77 11:41 PAGE 4
```

*** SEGMENT: INTRO

```
R/W MEM LIMITS: 005360 005567 000210 00136.
DISK BLK LIMITS: 000010 000010 000001 00001.
```

MEMORY ALLOCATION SYNOPSIS:

SECTION		TITLE	IDENT	FILE
SARRAY: (RW,D,LCL,REL,CON)	005360 000000 000000.	INTRO	V01X03	INTRO.OBJ
\$CODE : (RW,I,LCL,REL,CON)	005360 000126 00086.	INTRO	V01X03	INTRO.OBJ
\$FLAGR: (RW,D,GBL,REL,CON)	005360 000126 00086.	INTRO	V01X03	INTRO.OBJ
\$FLAGS: (RW,D,GBL,REL,CON)	002670 000000 00000.	INTRO	V01X03	INTRO.OBJ
\$FLAGT: (RW,D,GBL,REL,CON)	002670 000010 00008.	INTRO	V01X03	INTRO.OBJ
\$IDATA: (RW,D,LCL,REL,CON)	002672 000002 00002.	INTRO	V01X03	INTRO.OBJ
\$PDATA: (RW,D,LCL,REL,CON)	002700 000000 00000.	INTRO	V01X03	INTRO.OBJ
\$STRNG: (RW,D,LCL,REL,CON)	002700 000000 00000.	INTRO	V01X03	INTRO.OBJ
\$LDATA: (RW,D,LCL,REL,CON)	005506 000006 00006.	INTRO	V01X03	INTRO.OBJ
\$PDATA: (RW,D,LCL,REL,CON)	005514 000052 00042.	INTRO	V01X03	INTRO.OBJ
\$STRNG: (RW,D,LCL,REL,CON)	005514 000052 00042.	INTRO	V01X03	INTRO.OBJ
\$DATA: (RW,D,LCL,REL,CON)	005566 000000 00000.	INTRO	V01X03	INTRO.OBJ
\$SALVC: (RW,D,LCL,REL,CON)	005566 000000 00000.	INTRO	V01X03	INTRO.OBJ
\$SRTS : (RW,I,GBL,REL,OVR)	005026 000002 00002.			

GLOBAL SYMBOLS:

Figure 5-16 (Cont.) User BASIC-PLUS-2 Memory Allocation Map

INTRO 005360-R

USER.TSK MEMORY ALLOCATION MAP TKB M26
CRUNCH PAGE 5
18-JUL-77 11:41

*** SEGMENT: CRUNCH

R/W MEM LIMITS: 005360 005777 000420 00272.
DISK BLK LIMITS: 000011 000011 000001 00001.

MEMORY ALLOCATION SYNOPSIS:

SECTION	TITLE	IDENT	FILE
• BLK.: (RW, I, LCL, REL, CON)	005360	000210	00136.
	005360	000046	\$JPADD 01CM BASIC2.OLB
	005426	000074	\$JPMOV 02CM BASIC2.OLB
	005522	000046	\$JPSUB 01CM BASIC2.OLB
\$ARRAY: (RW, D, LCL, REL, CON)	005570	000000	00000.
\$CODE : (RW, I, LCL, REL, CON)	005570	000000	00000. CRUNCH V01X03 CRUNCH.OBJ
	005570	000150	00104.
\$FLAGR: (RW, D, GBL, REL, CON)	002670	000000	00000. CRUNCH V01X03 CRUNCH.OBJ
\$FLAGS: (RW, D, GBL, REL, CON)	002670	000000	00000. CRUNCH V01X03 CRUNCH.OBJ
	002674	000002	00002. CRUNCH V01X03 CRUNCH.OBJ
\$FLAGT: (RW, D, GBL, REL, CON)	002700	000000	00000.
\$IDATA: (RW, D, LCL, REL, CON)	005740	000014	00012. CRUNCH V01X03 CRUNCH.OBJ
	005740	000014	00012. CRUNCH V01X03 CRUNCH.OBJ
\$PDATA: (RW, D, LCL, REL, CON)	005754	000024	00020.
\$STRNG: (RW, D, LCL, REL, CON)	005754	000024	00020. CRUNCH V01X03 CRUNCH.OBJ
\$TDATA: (RW, D, LCL, REL, CON)	006000	000000	00000. CRUNCH V01X03 CRUNCH.OBJ
\$\$ALVC: (RW, D, LCL, REL, CON)	006000	000000	00000. CRUNCH V01X03 CRUNCH.OBJ
\$\$RTS : (RW, I, GBL, REL, OVR)	005026	000002	00002.

Figure 5-16 (Cont.) User BASIC-PLUS-2 Memory Allocation Map

OVERLAY CAPABILITY

GLOBAL SYMBOLS:

ADISPP	005360-R	ADI\$PP	005370-R	CRUNCH	005570-R	MOI\$PM	005466-R	NOI\$P	005504-R	SUI\$PA	005562-R	SUI\$SP 005524-R
ADISIP	005374-R	ADI\$PS	005404-R	MOI\$IP	005426-R	MOI\$PP	005436-R	ONI\$P	005474-R	SUI\$PM	005554-R	
ADISPA	005420-R	ADI\$P	005362-R	MOI\$MP	005442-R	MOI\$PS	005452-R	SUI\$IP	005522-R	SUI\$PP	005532-R	
ADISPM	005412-R	CLISP	005514-R	MOI\$PA	005460-R	MOI\$SP	005430-R	SUI\$MP	005536-R	SUI\$PS	005546-R	

USER.TSK MEMORY ALLOCATION MAP TKB M26 PAGE 6
CHATR 18-JUL-77 11:41

*** SEGMENT: CHATTR

R/W MEM LIMITS: 005360 006177 000620 00400.
DISK BLK LIMITS: 000012 000012 000001 000001.

MEMORY ALLOCATION SYNOPSIS:

SECTION				TITLE	IDENT	FILE
BLK. :	(RW, I, LCL, REL, CON)	005360	000074	00060.	-----	-----
SARRAY :	(RW, D, LCL, REL, CON)	005360	000074	00060.	\$JPMOV	02CM
SCODE :	(RW, I, LCL, REL, CON)	005454	000000	00000.	CHATR	V01X03 CHATTR.OBJ
\$FLAGR:	(RW, D, GBL, REL, CON)	005454	000000	00000.	CHATR	V01X03 CHATTR.OBJ
\$FLAGS:	(RW, D, GBL, REL, CON)	002670	000000	00000.	CHATR	V01X03 CHATTR.OBJ
\$FLAGT:	(RW, D, GBL, REL, CON)	002670	000010	00008.	CHATR	V01X03 CHATTR.OBJ
\$IDATA:	(RW, D, LCL, REL, CON)	006022	000014	00012.	CHATR	V01X03 CHATTR.OBJ
\$PDATA:	(RW, D, LCL, REL, CON)	006036	000140	00096.	CHATR	V01X03 CHATTR.OBJ
\$STRNG:	(RW, D, LCL, REL, CON)	006176	000000	00000.	CHATR	V01X03 CHATTR.OBJ

Figure 5-16 (Cont.) User BASIC-PL/SS-2 Memory Allocation Map

OVERLAY CAPABILITY

```
$TDATA: (RW,D,LCL,REL,CON) 006176 000000 00000.  
      006176 000000 00000. CHATR  VOLX03 CHATR.OBJ  
$SALVC: (RW,D,LCL,REL,CON) 006176 000000 00000.  
$SRTS : (RW,I,GBL,REL,OVR) 005026 000002 00000.  
  
GLOBAL SYMBOLS:  
CHATR 005454-R MOIS$P 005360-R MOIS$PA 005412-R MOIS$PP 005370-R MOIS$SP 005362-R ONI$P 005426-R  
CLI$P 005446-R MOIS$MP 005374-R MOIS$PM 005420-R MOIS$PS 005404-R MOIS$P 005436-R  
  
*** TASK BUILDER STATISTICS:  
TOTAL WORK FILE REFERENCES: 93702.  
WORK FILE READS: 0.  
WORK FILE WRITES: 0.  
SIZE OF CORE POOL: 18052. WORDS (70. PAGES)  
SIZE OF WORK FILE: 16128. WORDS (63. PAGES)  
ELAPSED TIME:00:01:08
```

Figure 5-16 (Cont.) User BASIC-PLUS-2 Memory Allocation Map

OVERLAY CAPABILITY

5.4 SUMMARY OF THE OVERLAY DESCRIPTION LANGUAGE

1. An overlay structure consists of one or more trees. Each tree contains at least one segment. A segment is a set of modules and PSECTS that can be loaded by a single disk access. A tree can have only one root segment, but it can have any number of overlay segments or co-trees.
2. An overlay description is a text file consisting of a series of ODL directives, one directive per line. This file is entered in a Task Builder command line and is identified as an ODL file by the presence of the /MP switch after the filename. An overlay description text file, if entered, must be the only input file specified.
3. The overlay description language provides five directives for specifying the tree representation of the overlay structure, namely:

- .ROOT
- .END
- .PSECT
- .FCTR
- .NAME

These directives can appear in any order in the overlay description subject to the following restrictions:

- a. There can be only one .ROOT and one .END directive.
- b. The .END directive must be the last directive, because it terminates input.
4. The tree structure is defined by the operators hyphen (-), exclamation point (!), and comma (,) and by the use of parentheses.
 - The hyphen operator indicates that the arguments preceding and following it are concatenated and thus coexist in memory.
 - The exclamation point operator is only used for resident libraries and allows you to specify library overlay segments that will permanently reside in memory.
 - The comma operator within parentheses indicates that its arguments are to overlay each other physically.
 - The comma operator outside parentheses delimits overlay trees.
 - Parentheses group segments that begin at the same address in memory.

CAUTION

DO NOT treat parentheses in ODL like parentheses in English or mathematics. Putting parentheses around a series of segments for grouping purposes changes the meaning of any commas within the parentheses.

For example, the directives

```
.ROOT A-B-(C,D-(E,F))
.END
```

OVERLAY CAPABILITY

define an overlay structure with a root segment consisting of the modules A and B. In this structure, there are four overlay segments: C, D, E, and F. The outer pair of parentheses indicates that the overlay segments C and D start at the same location in memory; and similarly, the inner parentheses indicate that E and F start at their own shared address. The resulting tree, assuming that the next line is a .END directive, looks like Figure 5-17.

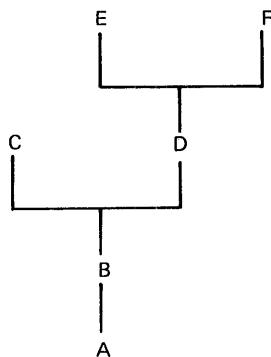


Figure 5-17 Simple Tree (Summary Example)

5. The .ROOT directive defines the overlay structure. The arguments of the .ROOT directive are one or more of the following:
 - File specifications as described in Section 2.8, item 7
 - Factor labels, which match the labels of .FCTR directives in the same structure
 - Segment names
 - PSECT names, which match the labels of .PSECT directives in the same structure
6. The .END directive terminates input.
7. The .FCTR directive provides a means for replacing text by a symbolic reference (the factor label). This replacement is useful for two reasons:
 - a. The .FCTR directive extends the text of the .ROOT directive to more than one line and thus allows complex trees to be represented.
 - b. The .FCTR directive allows the overlay description to be written in a form that makes the structure of the tree more apparent.

For example:

```
.ROOT A-(B-(C,D),E-(F,G),H)  
.END
```

can be expressed, using the .FCTR directive, as follows:

OVERLAY CAPABILITY

```
.ROOT A-(F1,F2,H)
F1:    .FCTR B-(C,D)
F2:    .FCTR E-(F,G)
      .END
```

The second representation shows more clearly that the tree has three main branches and that branches B and E each have two leaves.

8. The .PSECT directive provides a means for directly specifying the segment in which a PSECT is placed. It accepts the name of the PSECT and its attributes. For example:

```
.PSECT ALPHA,CON,GBL,RW,I,REL
```

ALPHA is the PSECT name and the remaining arguments are attributes. PSECT attributes are described in Table 4-1.

The PSECT name (composed from the characters A-Z, 0-9, and \$) must appear first in the .PSECT directive, but the attributes can appear in any order or be omitted. If an attribute is omitted, a default condition is assumed. The defaults for PSECT attributes are RW, I, LCL, REL, and CON.

NOTE

The use of the dollar sign (\$) in PSECT names is customarily reserved for system software. While \$ is a valid character in a PSECT name, its use in user-generated PSECTs is not recommended.

As in the example above, therefore, specify only those attributes that do not correspond to the defaults:

```
.PSECT ALPHA,GBL
```

9. The .NAME directive provides a means for designating a segment name for use in the overlay description, and for specifying segment attributes. This directive is useful for creating a null segment, naming a segment differently from the name of the first module, or naming a non-executable segment that is to be autoloadable. If the .NAME directive is not used, the name of the first file, PSECTs, or library module is used to identify the segment.

The .NAME directive creates a segment name as follows:

```
.NAME segname,attr,attr
```

where segname is the designated name (composed from the character set A-Z, 0-9, and \$), and attr is an optional attribute taken from the following: GBL, NODSK, NOGBL, DSK. The defaults are NOGBL and DSK. The defined name must be unique with respect to the names of PSECT, segments, files, and factor labels referenced in the ODL file.

10. A co-tree can be defined by specifying an additional tree structure in the .ROOT directive. The first overlay tree description in the .ROOT directive is the main tree. Subsequent overlay descriptions are co-trees. For example:

```
.ROOT A-B-(C,D-(E,F)),X-(Y,Z),Q-(R,S,T)
```

OVERLAY CAPABILITY

The main tree in this example has the root segment consisting of files A.OBJ and B.OBJ; two co-trees are defined; the first co-tree has the root segment X and the second co-tree has the root segment Q. The tree structure looks like Figure 5-18.

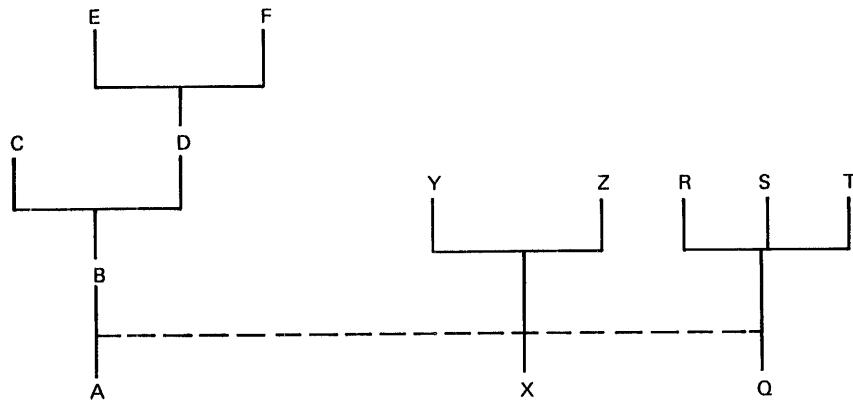


Figure 5-18 Co-trees (Summary Example)

If the preceding overlay description were written using the .FCTR directive, it might look like this:

```
.ROOT ATREE,XTREE,QTREE
ATREE .FCTR A-B-(C,D-(E,F))
XTREE .FCTR X-(Y,Z)
QTREE .FCTR Q-(R,S,T)
.END
```

It is now clear that there are three trees involved in this structure and that ATREE is the main tree (because it was the first tree mentioned in the .ROOT directive).

Contrast this ODL description with the one preceding. Notice how difficult it is to see at first glance in the first description that there are three trees.



CHAPTER 6

THE AUTOLOAD MECHANISM

The autoload mechanism is a method for loading disk-resident overlays. In the autoload method, the Overlay Run-time System handles loading and error recovery. Overlays are automatically loaded when referenced through a transfer-of-control instruction in the calling segment (CALL, PERFORM, GO TO, or GOSUB). No specific calls to the Overlay Run-time System are needed.

This section discusses the following topics.

- Autoload Indicator
- Path-Loading
- Autoload Vectors
- Autoloadable Data Segments

6.1 AUTOLOAD

All loading in higher level languages is done for you by the autoload method. Once loaded, the root segment of a co-tree remains in memory throughout the execution process. The execution of a transfer-of-control instruction to an autoloadable segment farther away from the root automatically initiates the autoload process.

6.1.1 Autoload Indicator

You can assist the autoload method by putting asterisks (*) in the ODL description of the task at the points where autoloading should take place. The autoload indicator can be applied to the following elements:

- Filenames - Make all the components of the file autoloadable.
- Portions of ODL tree descriptions enclosed in parentheses - Make all the elements within the parentheses autoloadable. This includes elements within any nested parentheses.
- PSECT names - Make the PSECT autoloadable. The PSECT must have the I (instruction) attribute.
- Segment names defined by the .NAME directive - Make all components of the segment autoloadable.

THE AUTOLOAD MECHANISM

- .FCTR label names - Make the first component of the factor autoloadable. All elements of the .FCTR are autoloadable if they are enclosed in parentheses.

If the autoload indicator is applied to an ODL statement enclosed in parentheses, then every task element named within the parentheses is marked as autoloadable. Applying the autoload indicator at the outermost parenthesis level of the ODL tree description marks every module in the overlay segments as autoloadable.

In Figure 5-8, if segment C consisted of a set of modules C1, C2, C3, C4, and C5, the tree diagram would resemble Figure 6-1.

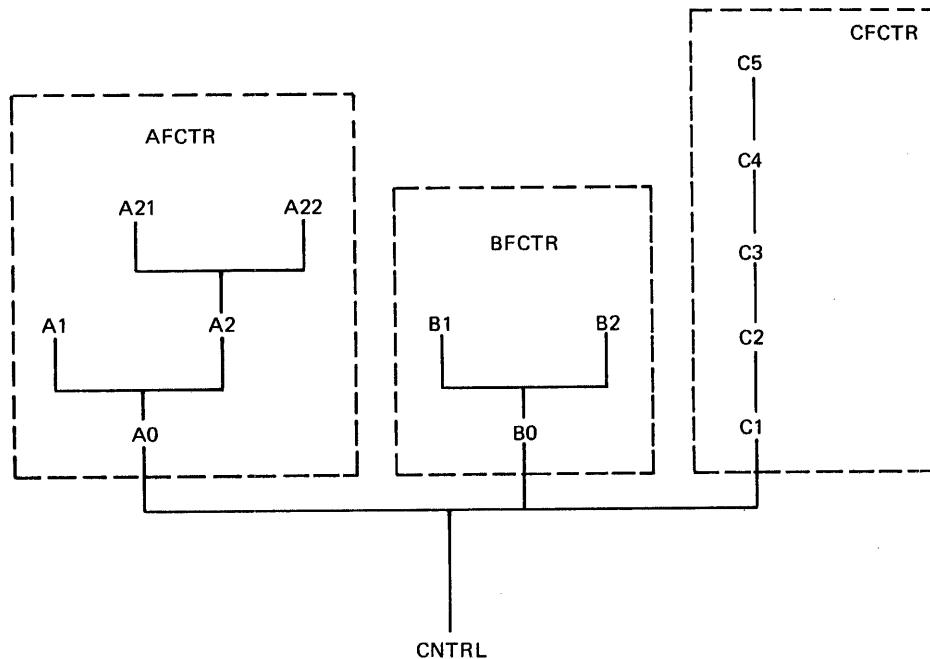


Figure 6-1 The .FCTR Directive

Placing the autoload indicator at the outermost parenthesis level assures that, regardless of the flow of control within the task, a module will be properly loaded when called. The ODL description for the task with this provision is:

```
.ROOT CNTRL-* (AFCTR,BFCTR,CFCTR)
AFCTR:   .FCTR A0-(A1,A2-(A21,A22))
BFCTR:   .FCTR B0-(B1,B2)
CFCTR:   .FCTR C1-C2-C3-C4-C5
        .END
```

To be sure that all modules of a co-tree are properly loaded, you must mark its root segment as well as its outermost parenthesis level:

```
.ROOT CNTRL-* (AFCTR,BFCTR,CFCTR),*CNTRL2-* (CNTRLX,CNTRLY)
```

The example above assumes that one or more modules containing executable code reside in CNTRL2.

THE AUTOLOAD MECHANISM

Note in the following example, how two .PSECT directives, a .NAME directive, and five autoload indicators are used:

```
        .ROOT CNTRL-(*AFCTR,*BFCTR,*CFCTR)
AFCTR:   .FCTR A0-*ASUBL-ASUB2-*(A1,A2-(A21,A22))
BFCTR:   .FCTR (B0-(B1,B2))
CFCTR:   .FCTR CNAM-C1-C2-C3-C4-C5
          .NAME CNAM, GBL
          .PSECT ASUBL,I, GBL, OVR
          .PSECT ASUB2,I, GBL, OVR
          .END
```

The autoload indicators function as follows:

(*AFCTR,*BFCTR,*CFCTR)

The autoload indicator is applied to each factor name.

- *AFCTR = *A0
- *BFCTR = *(B0-(B1-B2))
- *CFCTR = *CNAM

CNAM is an element defined by a .NAME directive. Therefore, all the components of the segment to which the name applies are autoloadable: C1, C2, C3, C4, and C5.

*ASUBL The autoload indicator is applied to the name of a PSECT having the I (Instruction) attribute, so the PSECT ASUBL is autoloadable.

*(A1,A2-(A21,A22))

The autoload indicator is applied to a portion of the ODL description enclosed in parentheses, so every element within the parentheses is autoloadable: A1, A2, A21, and A22.

The net effect of this ODL description is to make every element autoloadable except ASUB2. The others are all accounted for.

6.1.2 Path-Loading

Autoload uses the technique of path-loading. In the path-loading method, all the segments on the path from the calling segment to the called segment are brought into physical memory and mapped if they are not already there. Path-loading is confined to the tree in which the called segment resides. A call from a segment in one tree to a segment in another causes all unloaded segments in the second tree on the path from the root to the called module to be loaded.

Look at Figure 6-1. If CNTRL calls A21, then all the modules between the calling module CNTRL and the called module A21 are loaded. In this case, modules A0 and A2 are loaded. This permits the loading of A21 and the call can now be made.

The Overlay Run-time System keeps track of which segments are loaded and mapped, and issues load requests only for segments that are not in memory and mapped. (If CNTRL calls A2 after calling A1, A0 is not loaded again. It is already in memory and mapped).

THE AUTOLOAD MECHANISM

A reference from one segment to another segment closer to the root and on the same path is resolved directly. For example, A2 can immediately access A0 because A0 was path-loaded when A2 was called.

6.1.3 Autoload Vectors

When the Task Builder sees a reference to a global symbol in an autoloadable segment farther up the tree, it generates an autoload vector for the referenced global symbol. The reference is changed to a definition that points to an autoload vector entry. A transfer-of-control instruction to the global symbol executes the call to the autoload routine, \$AUTO. But references from a segment to a global symbol up-tree in a PSECT with the D attribute (see Table 4-1) are resolved directly.

The Task Builder often generates more autoload vectors than are necessary, because it knows very little about the flow of control in the task. You can tell the Task Builder more about the path-loading necessary and the flow of control. If you put autoload indicators only where they are needed, you can minimize the number of autoload vectors generated.

Assume that the root segment CNTRL has the following contents:

```
PROGRAM CNTRL
CALL A1
CALL A21
CALL A2
CALL A0
CALL A22
CALL B0
CALL B1
CALL B2
CALL C1
CALL C2
CALL C3
CALL C4
CALL C5
END
```

Note that all calls to the overlays come from the root segment. If you put the autoload indicator at the outermost parenthesis level, thirteen autoload vectors are generated for this task; one for each segment.

You can eliminate the unnecessary autoload vectors by placing the autoload indicator only where explicit loading is required:

```
.ROOT CNTRL-(AFCTR,*BFCTR,CFCTR)
AFCTR:   .FCTR A0-(*A1,A2-*(A21,A22))
BFCTR:   .FCTR (B0-(B1,B2))
CFCTR:   .FCTR *C1-C2-C3-C4-C5
.END
```

With this ODL description, the Task Builder generates only seven autoload vectors which act on the following modules: A1, A21 and A22, B0, B1, and B2, and C1. A0 is path-loaded when A1 is called. Likewise, A2 is path-loaded when A21 is called. Autoload vectors for A0 and A2 are therefore unnecessary. All modules of BFCTR are autoloaded, because BFCTR is autoloaded and its entire contents are within parentheses in the .FCTR statement. Therefore, autoload vectors for B0, B1, and B2 are unnecessary. The call to C1 loads the segment that contains C2, C3, C4, and C5. Therefore, autoload vectors for these modules are unnecessary.

CHAPTER 7

RESIDENT LIBRARIES

7.1 INTRODUCTION

A resident library is a block of data or code that resides in memory and can be used by any number of tasks. These libraries are useful because they make efficient use of memory:

1. By providing a way in which two or more tasks can communicate, and
2. By providing a way in which a single copy of a date base or commonly used subroutine can be shared by several tasks.

The first case is illustrated by Figure 7-1.

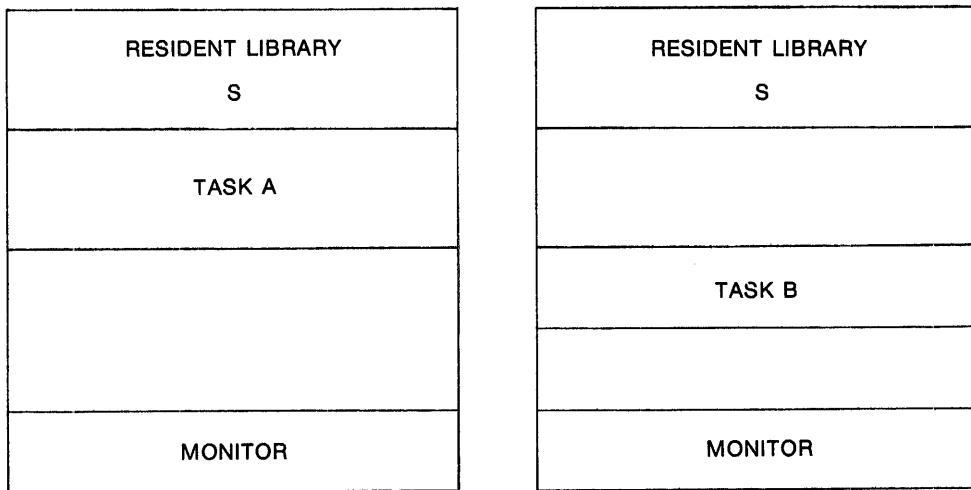


Figure 7-1 System Memory Usage

In Figure 7-1, task A stores some result in library S and task B retrieves the data from the library at a later time.

In the second case, common reentrant subroutines are not included in each task image. Rather, a single copy is shared by all tasks, as shown in Figure 7-2.

RESIDENT LIBRARIES

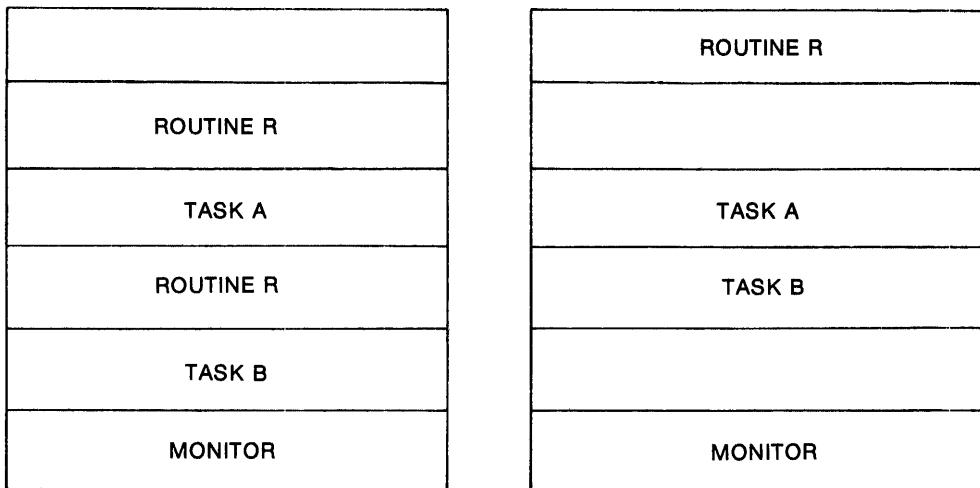


Figure 7-2 Shared and Non-Shared Memory

As an example of the usefulness of resident libraries, consider the current mechanism for access to RMS (Record Management Services) code. Every task image that references RMS code must currently have that code linked into the task by means of the Task Builder. This process has a number of effects:

1. The Task Builder must resolve RMS global references each time RMS code is linked to a task.
2. The size of the task image on disk increases as a result of linking to the RMS code.
3. RMS code is duplicated in physical memory whenever two or more executing tasks use RMS.

If you create an RMS resident library, the Task Builder makes only one resolution of RMS global symbols. Also, because your task references memory resident overlays in the library and not RMS disk overlays, disk I/O is minimized, task build time for other tasks that access RMS can be reduced, and execution speed for those tasks can be increased.

Note that a resident library is contained in a contiguous portion of physical memory, called a region. A region is resident in physical memory only when a task references it. If no tasks are referencing the region, that portion of physical memory is available for user jobs.

RESIDENT LIBRARIES

7.1.1 Resident Library Installation

A resident library is a collection of reentrant, shareable routines or data that the Task Builder links together into a task image file on disk. The MAKSIL program (see the RSTS/E Programmer's Utilities Manual) is used to format this disk file into Save Image Library (SIL) format. The UTILITY system program ADD LIBRARY command is then used to assign the task image portion of the SIL file to a contiguous region of physical memory. Note that you can use a Monitor SYS call (-18) to assign the task image portion of the SIL file to memory. Once the body of shareable routines or data is linked, formatted, and assigned to memory, it becomes a resident library that is accessible to user tasks through "windows" in their virtual address space.

The creation, installation, and maintenance of resident libraries are tasks that require a variety of distinct operations. The nature of these operations causes them to be described in a number of different manuals in the RSTS/E document set. The following list highlights the operations involved and the manuals which contain the pertinent information:

1. The task building of user-created library code to produce a symbol table and task image. This information is contained in Section 7.2.
2. Using the MAKSIL utility to format task builder output to produce suitable Monitor input. This information is contained in the RSTS/E Programmer's Utilities Manual.
3. The loading of MAKSIL output into memory for use by system users. This operation can be performed in two ways. The Monitor SYS calls to add, remove, load, and unload resident libraries (SYS -18) are described in the RSTS/E Programming Manual. The use of the UTILITY system program to add, remove, load, and unload resident libraries is described in the RSTS/E System Manager's Guide.

Consider the processes illustrated in Figure 7-3.

RESIDENT LIBRARIES

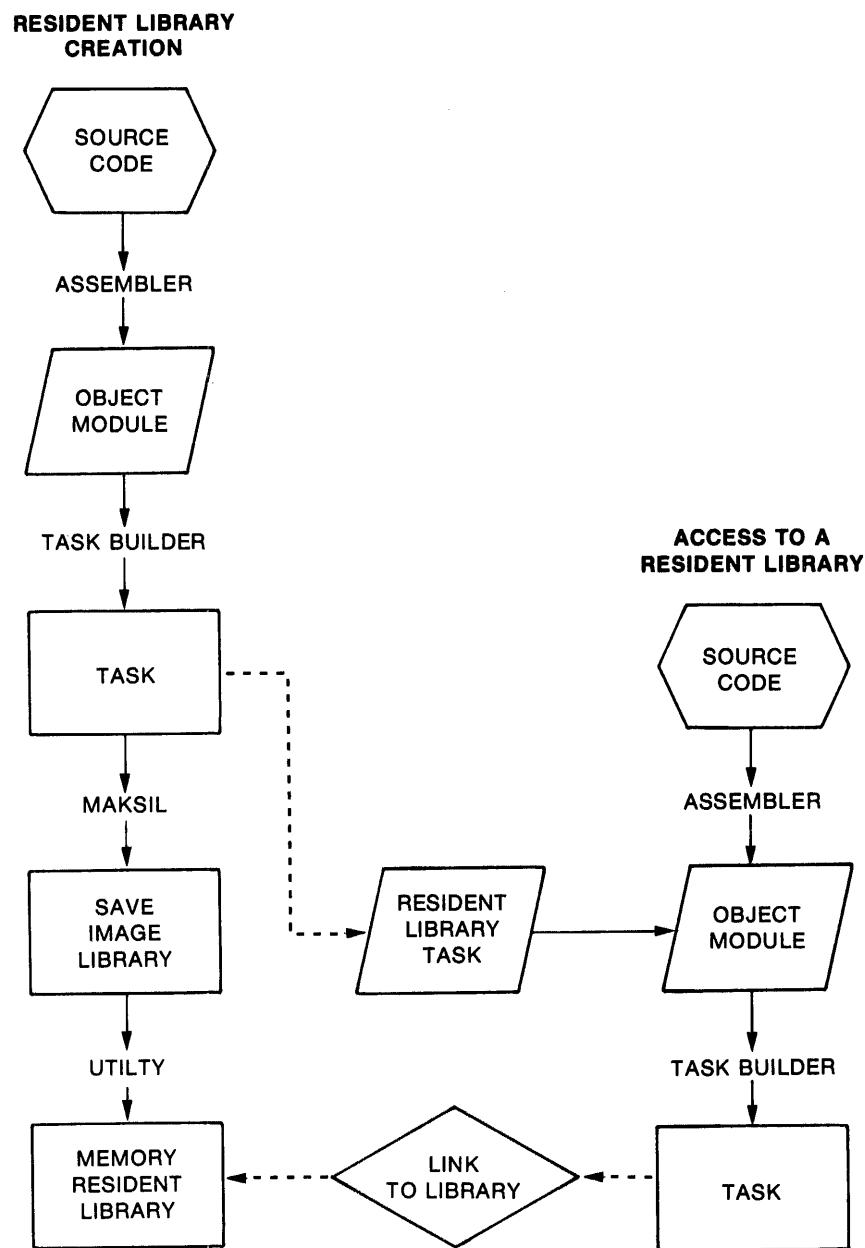


Figure 7-3 Resident Library Access

RESIDENT LIBRARIES

A task can link to as many as five resident libraries. Virtual address space must be allocated in 4K-word increments to map the resident library. To conserve address space, you can define a memory resident overlay structure for the library. In such a case, the entire library is resident in memory, but the task is mapped into only part of it at any one time.

A resident library has a task image file and a symbol definition file associated with it. When a task links to a resident library, the Task Builder uses the symbol definition file of the resident library to link the task to storage and entry points within the library.

7.2 CREATING A RESIDENT LIBRARY

Use the /-HD switch (see Section 3.1.B) to signal creation of a resident library. For example, when you specify this switch with the task image file specification to the Task Builder command line:

```
TKBD> ZETA/-HD,ZETA,ZETA=Z1,Z2,Z3
```

it causes the Task Builder to create a task image file (ZETA.TSK) with no header. Because the task has no header, it is not executable. Thus, other users can link against this file to reference the code or data in the object modules (Z1, Z2, Z3) that compose the task. Note that the symbol table and task image that the Task Builder generates for resident library usage must be in an account that is accessible to users who attempt to link against that library.

To create a resident library, request a task image output file and a symbol definition file (containing a list of global symbols and p-sections that reside within that task) from the Task Builder. Note that the symbol definition file is the vehicle whereby the Task Builder links a referencing task to specific data items and entry points within the resident library and, thus, is required for resident library usage.

7.2.1 Position Independent and Absolute Libraries

A resident library can be either position independent or absolute. Position independent libraries can be placed anywhere in the task's virtual address space. Absolute libraries are fixed in the virtual address space.

Declaring a library to be position independent causes the Task Builder to:

1. Include definitions for each root segment p-section in the symbol definition (.STB) file. A task can later reference this shared storage by p-section name.

RESIDENT LIBRARIES

2. Automatically select the set of virtual addresses in the referencing task that the resident library will occupy. You can suppress automatic selection with an APR specification (that maps the library). See Sections 3.2.4.2 and 3.2.4.3.

If the resident library is not position independent, only an absolute section (.ABS.) is included in the symbol definition file. All references to code or data in such a library must be by global symbol name.

You should declare a resident library to be position independent if:

1. The library contains code that executes correctly regardless of its location in the address space of the referencing task.
2. The library contains data that is not address dependent.
3. The library contains data that is referenced by a program (such data must reside in a named common block).

Because the p-section name is preserved in a position independent library, you should observe the following precautions when building and referencing such a library:

1. No code or data in the library should be included in the blank (unnamed) p-section.
2. No code or data in a referencing task should appear in a p-section of the same name as a section in the library.
3. The order in which memory is allocated to p-sections (alphabetic or sequential) must be the same for the library and its referencing task.

When a task references a position independent resident library, the Task Builder automatically positions the library in the task's virtual address space. If a reference is being made from a program but the data is not position independent, you must suppress automatic positioning by means of an APR (Active Page Register) specification (as described in Sections 3.2.4.2 and 3.2.4.3). The Task Builder uses the APR specification to map the library.

Consider the following example:

```
TKB> ZETA/-HD,ZETA,ZETA=Z1,Z2,Z3
```

In this example, the code contained in the task image file (ZETA.TSK) is referenced absolute. If the object module code (Z1, Z2, Z3) is position independent, you can specify the /PI switch (see Section 3.1.C) to the task image file and cause the Task Builder to produce a position independent task image. Without the /PI switch, you must use the PAR option (see Section 3.2.2.2) to position absolute code at the desired virtual base address. For example:

```
TKB> ZETA/-HD,ZETA,ZETA=Z1,Z2,Z3
TKB> /
ENTER OPTIONS:
TKB> PAR=ZETA:140000:20000
TKB> //
```

RESIDENT LIBRARIES

where the base address of the resident library will be 140000 (octal) in every task that references this library. Note that the partition base address (the PAR option argument) should be set as high as possible to ensure that enough address space is left for the task.

A task can now link to the library (ZETA); however, before the task can run, the library must be formatted (using MAKSIL) and made resident in memory (using SYS call -18 or UTILITY). These procedures are described in Section 7.1.1.

7.2.2 Resident Libraries with Memory Resident Overlays

If the resident library is to contain memory resident overlays, you must define the overlay structure in an ODL (Overlay Description Language) file. The Task Builder does not include the overlay data base (segment descriptions, autoload vectors, etc.) or the overlay run-time system in the resident library task image. Rather, the data base is made part of the symbol definition file that the Task Builder links to the referencing task. Note that the overlay run-time system is the autoload mechanism that the Task Builder links into each task which references overlays.

When you task build the referencing task, the following is automatically included in the task's root segment:

1. The data base.
2. Global references to overlay support routines that reside in the system object module library.

Each overlay segment in the resident library is marked with the NODSK attribute (see Section 5.1.3.3) to suppress overlay load requests.

The symbol table file contains global definitions for only those symbols that are defined or referenced in the root segment of the library. Such symbols consist of the following:

1. Actual entry points to routines and data elements that are in the root.
2. Autoload vector addresses that point to real definitions within a memory resident overlay (see Section 6.1.3).
3. Actual definitions of symbols defined in a memory resident overlay and referenced in the root.

You can force the inclusion of a global reference in the root segment of the resident library by means of the GBLREF option. Thus, the necessary autoload vectors and definitions can be generated without explicitly including such references in an object module. The syntax for the GBLREF option is as follows:

```
GBLREF=name
```

RESIDENT LIBRARIES

where name is a 1- to 6-character name from the Radix-50 character set. If the definition resides within an autoloadable segment, the Task Builder creates an autoload vector and includes it in the symbol table file. If the definition is not autoloadable, the real value is obtained and defined in the root segment.

No global symbol appears in the symbol table file unless:

1. It is defined in the root segment, or
2. It is referenced in the root segment and defined elsewhere in the overlay structure.

The procedure used to create an overlaid resident library is as follows:

1. Define an overlay structure that contains only memory resident overlays.
2. Include in the GBLREF option, or provide in the root segment, a module that contains the global references for defining entry points within the overlay segments. The Task Builder generates autoload vectors and global definitions for the entry points.

This procedure is illustrated in the following example. The resident library being constructed consists of reentrant code that resides within the overlay structure defined as follows:

```
*ROOT A---!(*B,C---*D)
*NAME A
*END
```

Root segment A contains no code or data and has a length of zero. All executable code exists within memory resident overlay segments composed of the files B.OBJ, C.OBJ, and D.OBJ. These object modules contain global entry points for segments B, C, and D respectively.

The task image, map, and symbol table files are generated with the following Task Builder commands:

```
TKB>A/-HD,A,A=A/MP
TKB>/
ENTER OPTIONS:
TKB>GBLREF=B,C,D
TKB>PAR=A:140000:20000
TKB>STACK=0
TKB>//
```

NOTE

The partition, task, and symbol table file (STB) names must be identical when creating a resident library.

RESIDENT LIBRARIES

References to entry points B, C, and D are inserted in the root segment by the Task Builder and subsequently appear in the symbol table file as definitions.

The definition for symbol C is resolved directly to the actual entry point. The definitions for symbols B and D are resolved to autoload vectors that are included in each referencing task. Unlike overlays that reside in the task image, each autoload vector in the resident library is included in each referencing task, regardless of whether the entry points are called during task execution. Only those global symbols defined or referenced in the root segment of the library appear in the symbol table file.

The symbol table file also contains the data base required by the overlay run-time system, in relocatable object module format. The data base contains the following:

1. All autoload vectors.
2. Segment tables linked to the task.
3. Address window descriptors.
4. Memory region descriptor.

The overlay structure, as reflected in the symbol table linkage, is preserved and conveyed to the referencing task by the STB file. Thus, path loading for the resident library can occur exactly as it does within a task. Aside from address space restrictions, there is no limit on the overlay structures that can be defined for a resident library.

7.2.3 Run-Time System Support for Overlaid Resident Libraries

Memory resident overlays within a resident library require additional support from the overlay run-time system of the task image. The resident library overlay data base that is linked within the image of the referencing task has a structure that is identical to the data created for an overlaid task. The only additional processing required of the overlay run-time system is to attach the library and obtain its identification for use by the mapping directives.

Consider the following:

1. A resident library cannot use the autoload facility to reference memory resident overlays within itself or any other resident library.
2. Named p-sections in a resident library overlay segment cannot be referenced by the task. If reference to the storage is required, such sections must be included in the root segment of the library (this results in a loss of virtual address space).

RESIDENT LIBRARIES

3. The number of autoload vectors is independent of the entry points actually referenced. The maximum number of vectors will be allocated within each referencing task. In some cases, the size of the allocation will be large.
4. There is an overhead of six instructions for each autoload call even when the segment is mapped.

As implied in the previous list, you must exercise care to ensure that an efficient memory resident overlay is implemented.

7.3 ACCESS TO A RESIDENT LIBRARY

In order to access a resident library, your task must first attach to the memory region that contains the library. This ensures that the library's own access requirements (protection code, for example) are fulfilled and that, once attached, the library will not be removed from memory while a task is accessing it. That is, a resident library need not be physically in memory when not in use (it can be marked as non-resident and read into memory when needed) but it cannot be removed from memory until all attached tasks are detached.

After your task has attached to the resident library, a virtual address window must be created. The virtual address window defines the portion of your task's virtual address space that is used to access the code or data in the resident library.

Finally, your task must map all or a portion of the created virtual address window into all or a portion of the resident library.

These operations: attaching your task to a region, creating a virtual address window, and mapping the window into the library can be accomplished in one of two ways. The easiest (and recommended) method is to include one or more options (COMMON, LIBR, RESCOM, or RESLIB) in the Task Builder command line when you create the task. The options (described in Sections 3.2.4.2 and 3.2.4.3) cause the Task Builder to include the code to perform the attach, create, and map operations when your task references the code or data in the resident library. Alternatively, you can use Monitor directives (PLAS functions) to perform these operations. The use of these directives is described in the RSTS/E System Directives Manual.

Once the attach, create, and map operations are completed, your task can directly access the code or data contained in the resident library.

RESIDENT LIBRARIES

7.3.1 Referencing a Resident Library

When you construct the Task Builder command lines to link your task, you indicate in the ENTER OPTIONS: portion of the command that a resident library will be referenced. The options you use to generate the reference are as follows:

1. RESLIB (Resident Library) or RESCOM (Resident Common Block)

RESLIB and RESCOM accept a file specification as one of their arguments, thus allowing you to specify an account, filename, and extension for the memory image and, by implication, the symbol table files. Note that device and unit number specifications are not allowed.

2. LIBR (System Resident Library) or COMMON (System Common Block)

LISR and COMMON accept a 1- to 6-character name (from the Radix-50 character set) of a resident library; the library memory image and symbol table file must reside under the account specified by the system logical name LB:.

These four options (described in Sections 3.2.4.2 and 3.2.4.3) accept two additional arguments:

1. The type of access required (RO - read only or RW - read/write).
2. The first Active Page Register (APR) that is used to map the library within the task's virtual address space (valid only when the library is position independent).

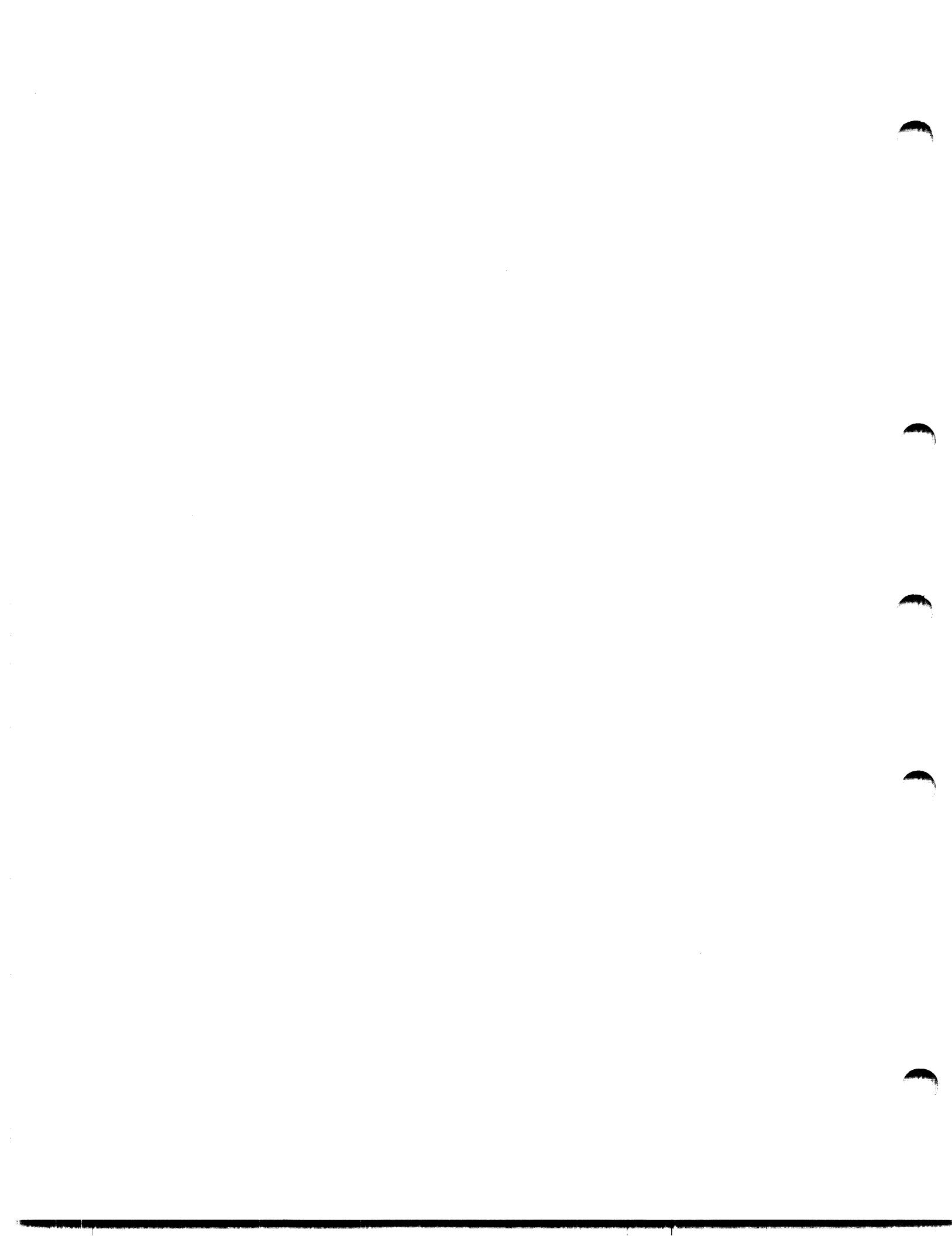
A symbol table file of the same name as the resident library (but with an .STB extension) must reside on the same device and under the same account as the resident library memory image file (.TSK extension). Consider the following example:

```
TKB>TASK,MAP,SYMBOL=INPUT  
TKB>/  
ENTER OPTIONS:  
TKB>COMMON=A:RO  
TKB>//
```

The Task Builder expects to find files A.TSK and A.STB under account LB:.

If the task is to reference a private resident library, the following command series might be used:

```
TKB>TASK,MAP,SYMBOL=INPUT  
TKB>/  
ENTER OPTIONS:  
TKB>RESLIB=C21,131A/RO  
TKB>//
```



APPENDIX A

ERROR MESSAGES

This appendix lists the error messages the Task Builder produces. Most of the messages are self-explanatory. In some cases, the line in which the error occurred is printed. Task Builder produces diagnostic and fatal error messages. Error messages are printed in two forms:

- TKB -- *DIAG*-error-message
- TKB -- *FATAL*-error-message

Some errors are correctable when command input comes from a terminal. With these errors, a diagnostic error message is printed: correct the error, and continue the task building sequence. If the same error occurs in an indirect file you cannot correct it on the terminal and proceed, so the task-build is aborted. You have to correct the indirect file in which the error occurred and rerun from the beginning.

Some diagnostic error messages merely tell you about an unusual condition. If you consider the condition to be something you can live with, or to be normal to your task, you can go ahead and run the task image.

If the explanation accompanying your error message refers to a system error, please send a Software Performance Report (SPR) to DIGITAL.

ALLOCATION FAILURE ON FILE file-name

The Task Builder could not find enough disk space to store the task image file, or did not have write access to the UFD or volume that was to contain the file.

BLANK PSECT NAME IS ILLEGAL overlay-description-line

The overlay description line printed contains a .PSECT directive that does not have a PSECT name.

COMMAND I/O ERROR

An I/O error occurred on command input device. (The device may not be on line, or there may be a possible hardware error.)

COMMAND SYNTAX ERROR command-line

The command line printed has incorrect syntax.

ERROR MESSAGES

COMPLEX RELOCATION ERROR - DIVIDE BY ZERO: MODULE module-name

A divisor having the value zero was detected in a complex expression. The result of the division was set to zero. (A probable cause is division by a global symbol whose value is undefined.)

FILE filename HAS ILLEGAL FORMAT

The file filename contains an object module in an invalid format.

ILLEGAL DEFAULT PRIORITY SPECIFIED option-line

The option line printed contains a priority greater than 250.

ILLEGAL ERROR-SEVERITY CODE octal-list

System error (no recovery). Please send DIGITAL an SPR with a copy of the message containing the octal-list as printed.

ILLEGAL FILENAME invalid-line

The invalid line printed contains a wild card (*) in a file specification. The use of wild cards is prohibited.

ILLEGAL GET COMMAND LINE ERROR CODE

System error (no recovery). Please send an SPR to DIGITAL.

ILLEGAL LOGICAL UNIT NUMBER invalid-line

The invalid line printed contains a device assignment to a unit number larger than the number of logical units specified by the UNITS keyword or assumed by default if the UNITS keyword is not used.

ILLEGAL MULTIPLE PARAMETER SETS invalid-line

The invalid line printed contains multiple sets of parameters for a keyword that allows only a single parameter set.

ILLEGAL NUMBER OF LOGICAL UNITS invalid-line

The invalid-line printed contains a logical unit number greater than 14.

ILLEGAL ODT OR TASK VECTOR SIZE

The ODT or SST vector size specified is greater than 32 words.

ILLEGAL OVERLAY DESCRIPTION OPERATOR invalid-line

The invalid line printed contains an unrecognizable operator in an overlay description. This error occurs if the first character in a PSECT or segment name is a dot (.).

ERROR MESSAGES

ILLEGAL OVERLAY DIRECTIVE invalid-line

The invalid line printed contains an unrecognizable overlay directive.

ILLEGAL PARTITION/COMMON BLOCK SPECIFIED invalid-line

The invalid line printed contains a partition or a common block that does not lie on a 32-word boundary.

ILLEGAL PSECT/SEGMENT ATTRIBUTE invalid-line

The invalid line printed contains a PSECT or segment attribute that is not recognized.

ILLEGAL REFERENCE TO LIBRARY PSECT PSECT-name

The task has attempted to reference a PSECT name existing in a shared run-time system but has not named the run-time system in a keyword.

ILLEGAL SWITCH file-specification

The file specification printed contains an illegal switch or switch value.

INCOMPATIBLE REFERENCE TO LIBRARY PSECT PSECT-name

The task has attempted to reference more storage in a run-time system than exists in the run-time system definition.

INCORRECT LIBRARY MODULE SPECIFICATION invalid-line

The invalid line contains a module name with a non-Radix-50 character.

INDIRECT COMMAND SYNTAX ERROR invalid-line

The invalid line printed contains a syntactically incorrect indirect file specification.

INDIRECT FILE OPEN FAILURE invalid-line

The invalid line contains a reference to a command input file that could not be located.

INSUFFICIENT PARAMETERS invalid-line

The invalid line contains a keyword with too few number of parameters to complete its meaning.

INVALID KEYWORD IDENTIFIER invalid-line

The invalid line printed contains an unrecognizable keyword.

ERROR MESSAGES

INVALID PARTITION/COMMON BLOCK SPECIFIED invalid-line

The invalid line contains a partition or common block that is invalid for one of the following reasons:

1. The base address of the partition is not on a 4K boundary or is not 0.
2. The memory bounds for the partition overlap a run-time system.

I/O ERROR LIBRARY IMAGE FILE

An I/O error has occurred during an attempt to open or read the .STB file of a Run-Time System.

I/O ERROR ON INPUT FILE file-name

I/O ERROR ON OUTPUT FILE file-name

LABEL OR NAME IS MULTIPLY DEFINED invalid-line

The invalid line printed defines a name that has already appeared as a .FCTR, .NAME, or .PSECT directive.

LIBRARY FILE filename HAS INCORRECT FORMAT

A module has been requested from a library file that has an empty module name table.

LOAD ADDR OUT OF RANGE IN MODULE module-name

An attempt has been made to store data in the task image outside the address limits of the segment. This problem is usually caused by one of the following:

1. an attempt to initialize a PSECT contained in a run-time system
2. an attempt to initialize an absolute location outside the limits of the segment or in the task header
3. a patch outside the limits of the segment it applies to
4. an attempt to initialize a segment having the NODSK attribute

LOOKUP FAILURE ON FILE filename invalid-line

The invalid line printed contains a filename that cannot be located in the directory.

LOOKUP FAILURE ON SYSTEM LIBRARY FILE

The Task Builder cannot find the system library (SY:[1,1]SYSLIB.OLB) file to resolve undefined symbols.

ERROR MESSAGES

LOOKUP FAILURE RESIDENT LIBRARY FILE
invalid-line

No symbol table (.STB) file or task image file (.TSK) can be found in account [1,1] for the run-time system.

MAXIMUM INDIRECT FILE DEPTH EXCEEDED
invalid-line

The invalid line printed gives the file reference that exceeded the permissible indirect file depth (2).

MODULE module-name AMBIGUOUSLY DEFINES PSECT PSECT-name

The PSECT named has been defined in two modules not on a common path and referenced by a segment that is common to both paths.

MODULE module-name AMBIGUOUSLY DEFINES SYMBOL sym-name

The module named references or defines a symbol whose definition exists on two different paths but is referenced by a segment that is common to both paths.

MODULE module-name ILLEGALLY DEFINES XFR ADDRESS PSECT-name addr

1. The start address printed is odd.
2. The module named is in an overlay segment and has a start address. The start address must be in the root segment of the main tree.
3. The address is in a PSECT that has not yet been defined. Please send an SPR to DIGITAL if this is caused by DIGITAL-supplied software.

MODULE module-name MULTIPLY DEFINES PSECT PSECT-name

1. The PSECT named has been defined more than once in the same segment with different attributes.
2. A global PSECT has been defined more than once with different attributes in more than one segment along a common path.

MODULE module-name MULTIPLY DEFINES SYMBOL sym-name

1. Two definitions for the relocatable symbol sym-name have occurred on a common path.
2. Two definitions for an absolute symbol with the same name but different values have occurred.

MODULE module-name MULTIPLY DEFINES XFR ADDR IN SEG
segment-name

More than one module making up the root has a start address.

MODULE module-name NOT IN LIBRARY

The Task Builder could not find the module named on the /LB switch in the library specified.

NO DYNAMIC STORAGE AVAILABLE

The Task Builder needs additional symbol table storage and cannot find it.

ERROR MESSAGES

NO MEMORY AVAILABLE FOR LIBRARY library-name

The Task Builder could not find enough free virtual memory to map the specified Run-Time System.

NO ROOT SEGMENT SPECIFIED

The overlay description did not contain a .ROOT directive.

NO VIRTUAL MEMORY STORAGE AVAILABLE

The maximum permissible size of the Task Builder work file was exceeded. Consult Appendix F for suggestions on reducing the size of the work file.

OPEN FAILURE ON FILE file-name

OPTION SYNTAX ERROR

invalid-line

The invalid line printed contains unrecognizable syntax.

OVERLAY DIRECTIVE HAS NO OPERANDS

invalid-line

All overlay directives except .END require operands.

OVERLAY DIRECTIVE SYNTAX ERROR

invalid-line

The invalid line printed contains a syntax error.

PASS CONTROL OVERFLOW AT SEGMENT segment-name

System error. Please send an SPR to DIGITAL with a copy of the ODL file associated with the error.

PSECT PSECT-name HAS OVERFLOWED

You have tried to create a PSECT larger than 28K words.

REQUIRED INPUT FILE MISSING

At least one input file is required for a task-build.

ROOT SEGMENT IS MULTIPLY DEFINED

invalid-line

The invalid line printed contains the second .ROOT directive encountered. Only one .ROOT directive is allowed. (Check Section 5.1.4.1 to see how to correctly specify co-trees.)

SEGMENT seg-name HAS ADDR OVERFLOW: ALLOCATION DELETED

Within a segment, the program has attempted to allocate more than 28K words. A map file is produced, but no task image file is produced.

TASK HAS ILLEGAL MEMORY LIMITS

An attempt has been made to build a task whose size exceeds the partition boundary. If a task image file was produced, it should be deleted.

ERROR MESSAGES

TASK-BUILD ABORTED VIA REQUEST option-line

The option line contains your request to abort the task-build.
Retype your commands and correct the error to rerun.

TOO MANY NESTED .ROOT/.FCTR DIRECTIVES invalid-line

The invalid line printed contains a .FCTR directive that exceeds the maximum nesting level (16).

TOO MANY PARAMETERS invalid-line

The invalid line printed contains a keyword with more parameters than required.

TOO MANY PARENTHESES LEVELS invalid-line

The invalid line printed contains a parenthesis that exceeds the maximum nesting level (16).

TRUNCATION ERROR IN MODULE module-name

You tried to load a global value greater than +127 or less than -128 into a byte. Only the low-order eight bits are loaded.

UNABLE TO OPEN WORK FILE

The work file device is not mounted.

UNBALANCED PARENTHESES invalid-line

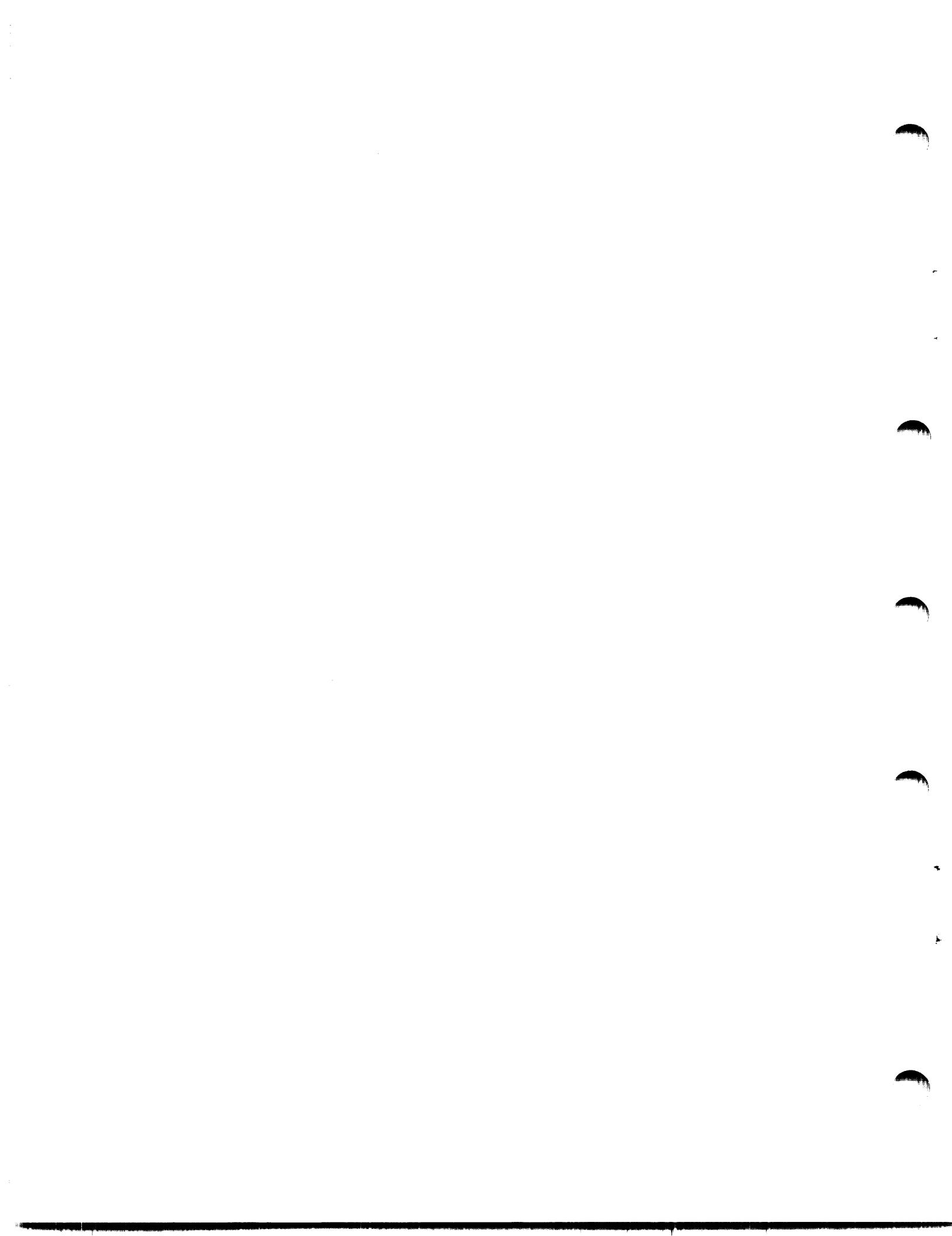
The invalid line printed contains unbalanced parentheses.

n UNDEFINED SYMBOLS SEGMENT seg-name

The segment named contains n undefined symbols. If no memory allocation is requested, the symbols are printed on the terminal.

WORK FILE I/O ERROR

An I/O error occurred during an attempt to reference data stored by the Task Builder in its work file.



APPENDIX B

OCTAL TO DECIMAL CONVERSION TABLE

B.1 INTRODUCTION

Table B-1 (listed on the last four pages of this appendix) is the octal-decimal integer conversion table. It directly converts octal numbers ranging from 0 to 7777 to decimal numbers, and decimal numbers ranging from 0 to 4095 to octal numbers. In addition it can be used to convert octal numbers up to 77777 to decimal and decimal numbers up to 32767 to octal. Figure B-1 shows a portion of one page of the table.

As shown in this figure, a group of numbers in the margin of each page (1) shows the range of octal and decimal numbers covered by that page. Use this group to locate the page containing the number you wish to convert. Also located in the margin are two columns (2). One column is labeled "OCTAL" and the other "DECIMAL". Use these columns to convert octal numbers ranging from 10000 to 77777 to decimal and decimal numbers from 4096 to 32767 to octal.

The left-most column and the top row of the table (3) contain the octal numbers. The top row contains the least significant digit of the octal number. The remaining columns contain the decimal numbers to be converted. Use these columns and rows to convert octal numbers to decimal and decimal numbers to octal.

NOTE

The left-most column and top row are shaded for easy identification.

This appendix illustrates how to perform these conversion processes as follows:

- Converting octal numbers ranging from 0 to 7777 to decimal numbers
- Converting decimal numbers ranging from 0 to 4095 to octal numbers
- Converting octal numbers ranging from 10000 to 77777 to decimal numbers
- Converting decimal numbers ranging from 4096 to 32767 to octal numbers

OCTAL TO DECIMAL CONVERSION TABLE

Table B-1
Octal-Decimal Integer Conversion Table

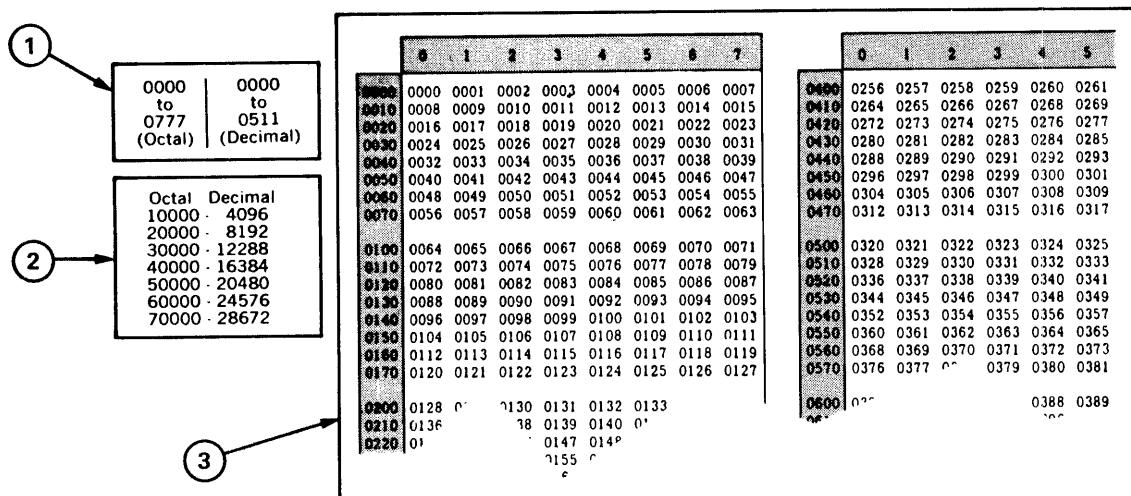


Figure B-1 Table B-1, Showing Table Parts for Conversion

B.2 CONVERTING OCTAL NUMBERS RANGING FROM 0 TO 7777 TO DECIMAL NUMBERS

Three examples follow in Sections B.2.1 through B.2.3. The procedures outlined in Section B.2.1 apply to Sections B.2.2 and B.2.3.

B.2.1 Converting Octal 43 to Decimal

Refer to Figure B-2. The numbers listed in the figure correspond to the steps presented below.

Table B-1
Octal-Decimal Integer Conversion Table

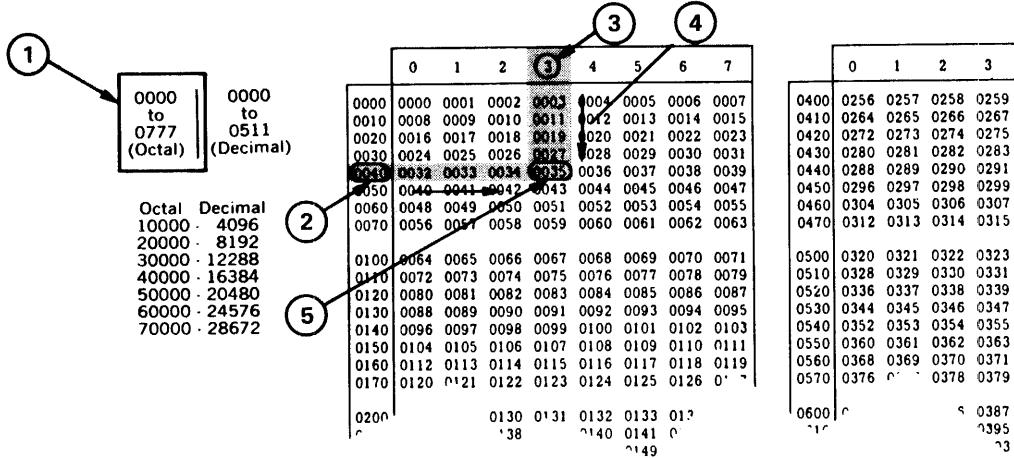


Figure B-2 Steps for Converting Octal 43 to Decimal

OCTAL TO DECIMAL CONVERSION TABLE

1. Locate the page having the range of octal numbers that includes 43 (0000 to 0777).
2. Locate the row containing 0040 in the left-most column of Table B-1.
3. Locate the vertical column containing 3 (the least significant digit).
4. Read down the vertical column until you intersect the horizontal row for 0040.
5. The number 35 where the column and row intersect is your answer (35 is the decimal equivalent of octal 43).

B.2.2 Converting Octal 1000 to Decimal

Figure B-3 illustrates the steps for converting octal 1000 to decimal 512.

0300	0192	0193	0194	0195	0196	0197	0198	0199
0310	0200	0201	0202	0203	0204	0205	0206	0207
0320	0200	0209	0210	0211	0212	0213	0214	0215
0330	0216	0217	0218	0219	0220	0221	0222	0223
0340	0224	0225	0226	0227	0228	0229	0230	0231
0350	0232	0233	0234	0235	0236	0237	0238	0239
0360	0240	0241	0242	0243	0244	0245	0246	0247
0370	0248	0249	0250	0251	0252	0253	0254	0255

0700	0448	0449	0450	0451
0710	0456	0457	0458	0459
0720	0464	0465	0466	0467
0730	0472	0473	0474	0475
0740	0480	0481	0482	0483
0750	0488	0489	0490	0491
0760	0496	0497	0498	0499
0770	0504	0505	0506	0507

0	1	2	3	4	5	6	7	
1000	0512	0513	0514	0515	0516	0517	0518	0519
1010	0520	0521	0523	0524	0525	0526	0527	
1020	0528	0529	0530	0531	0532	0533	0534	0535
1030	0536	0537	0538	0539	0540	0541	0542	0543
1040	0544	0545	0546	0547	0548	0549	0550	0551
1050	0552	0553	0554	0555	0556	0557	0558	0559
1060	0560	0561	0562	0563	0564	0565	0566	0567
1070	0568	0569	0570	0571	0572	0573	0574	0575
1100	0576	0577	0578	0579	0580	0581	0582	0583
1110	0584	0585	0586	0587	0588	0589	0590	0591
1120	0592	0593	0594	0595	0596	0597	0598	0599
1130	0600	0601	0602	0603	0604	0605	0606	0607
1140	0608	0609	0610	0611	0612	0613	0614	0615

0	1	2	3	
1400	0768	0769	0770	0771
1410	0776	0777	0778	0779
1420	0784	0785	0786	0787
1430	0792	0793	0794	0795
1440	0800	0801	0802	0803
1450	0808	0809	0810	0811
1460	0816	0817	0818	0819
1470	0824	0825	0826	0827
1500	0832	0833	0834	0835
1510	0840	0841	0842	0843
1520	0848	0849	0850	0851
1530	0856	0857	0858	0859
1540	0864	0865	0866	0867

Figure B-3 Steps for Converting Octal 1000 to Decimal

B.2.3 Converting Octal 7456 to Decimal

Figure B-4 illustrates the steps for converting octal 7456 to decimal 3886.

OCTAL TO DECIMAL CONVERSION TABLE

3292	3293	3294	3295	6730	3544	3545	3546	3547	3548	3549	3550	3551
3300	3301	3302	3303	6740	3552	3553	3554	3555	3556	3557	3558	3559
3308	3309	3310	3311	6750	3560	3561	3562	3563	3564	3565	3566	3567
3316	3317	3318	3319	6760	3568	3569	3570	3571	3572	3573	3574	3575
3324	3325	3326	3327	6770	3576	3577	3578	3579	3580	3581	3582	3583
4	5	6	7		0	1	2	3	4	5	6	7
3588	3589	3590	3591	7400	3840	3841	3842	3843	3844	3845	3846	3847
3596	3597	3598	3599	7410	3848	3849	3850	3851	3852	3853	3854	3855
3604	3605	3606	3607	7420	3856	3857	3858	3859	3860	3861	3862	3863
3612	3613	3614	3615	7430	3864	3865	3866	3867	3868	3869	3870	3871
3620	3621	3622	3623	7440	3872	3873	3874	3875	3876	3877	3878	3879
3628	3629	3630	3631	7450	3880	3881	3882	3883	3884	3885	3886	3887
3636	3637	3638	3639	7460	3888	3889	3890	3891	3892	3893	3894	3895
3644	3645	3646	3647	7470	3896	3897	3898	3899	3900	3901	3902	3903
3652	3653	3654	3655	7500	3904	3905	3906	3907	3908	3909	3910	3911
3660	3661	3662	3663	7510	3912	3913	3914	3915	3916	3917	3918	3919
3668	3669	3670	3671	7520	3920	3921	3922	3923	3924	3925	3926	3927
3676	3677	3678	3679	7530	3928	3929	3930	3931	3932	3933	3934	3935
3685	3686	3687		7540	3936	3937	3938	3939	3940	3941	3942	3943
3693	3694	3695		7550	3944	3945	3946	3947	3948	3949	3950	3951
3701	3702			7560	3952	3953	3954	3955	3956	3957	3958	3959
3709	3710			7570	3960	3961	3962	3963	3964	3965	3966	3967

Figure B-4 Steps for Converting Octal 7456 to Decimal

B-3 CONVERTING DECIMAL NUMBERS RANGING FROM 0 TO 4095 TO OCTAL

Three examples in Sections B.3.1 through B.3.3 follow; all conform to the procedures outlined in Section B.3.1.

B.3.1 Converting Decimal 17 to Octal

Refer to Figure B-5. The numbers listed in the figure correspond to the steps listed below.

Figure B-5 Steps for Converting Decimal 17 to Octal

OCTAL TO DECIMAL CONVERSION TABLE

1. Locate the range of decimal numbers in the margin of table B-1 containing 17 (0000 to 0511 decimal).
2. Locate the decimal number 0017 within the table.
3. Reading left from 0017, locate 0020 in the left-hand column of the table.
4. Reading up from 0017, locate 1 at the top of the column. (The "1" is the least significant digit of the octal number.)
5. Add 1 to 0020. The sum (0021) is your answer (21 is the octal equivalent of 17).

B.3.2 Converting Decimal 870 to Octal

Figure B-6 illustrates the steps for converting decimal 870 to octal 1546.

The diagram shows a conversion table for octal to decimal conversion. The table has two main sections: a margin section on the left and a body section on the right. The margin section contains rows of decimal numbers (0340 to 0770) and their corresponding octal equivalents. The body section contains rows of octal numbers (0 to 7) and columns for each decimal value. A box labeled '1' points to the decimal number 0512 (1023 decimal). A box labeled '2' points to the octal row 1546. A box labeled '3' points to the column for the least significant digit. A box labeled '4' points to the top of the column for reading up.

Margin (Decimal to Octal)								Body (Octal to Decimal)										
0340	0224	0225	0226	0227	0228	0229	0230	0231	0740	0480	0481	0482	0483	0484	0485	0486	0487	
0350	0232	0233	0234	0235	0236	0237	0238	0239	0750	0488	0489	0490	0491	0492	0493	0494	0495	
0360	0240	0241	0242	0243	0244	0245	0246	0247	0760	0496	0497	0498	0499	0500	0501	0502	0503	
0370	0248	0249	0250	0251	0252	0253	0254	0255	0770	0504	0505	0506	0507	0508	0509	0510	0511	
1000 to 1777 (Octal)	0512 to 1023 (Decimal)	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	
1000	0512	0513	0514	0515	0516	0517	0518	0519		1400	0768	0769	0770	0771	0772	0773	0774	0775
1010	0520	0521	0522	0523	0524	0525	0526	0527		1410	0776	0777	0778	0779	0780	0781	0782	0783
1020	0528	0529	0530	0531	0532	0533	0534	0535		1420	0784	0785	0786	0787	0788	0789	0790	0791
1030	0536	0537	0538	0539	0540	0541	0542	0543		1430	0792	0793	0794	0795	0796	0797	0798	0799
1040	0544	0545	0546	0547	0548	0549	0550	0551		1440	0800	0801	0802	0803	0804	0805	0806	0807
1050	0552	0553	0554	0555	0556	0557	0558	0559		1450	0808	0809	0810	0811	0812	0813	0814	0815
1060	0560	0561	0562	0563	0564	0565	0566	0567		1460	0816	0817	0818	0819	0820	0821	0822	0823
1070	0568	0569	0570	0571	0572	0573	0574	0575		1470	0824	0825	0826	0827	0828	0829	0830	0831
1100	0576	0577	0578	0579	0580	0581	0582	0583		1500	0832	0833	0834	0835	0836	0837	0838	0839
1110	0584	0585	0586	0587	0588	0589	0590	0591		1510	0840	0841	0842	0843	0844	0845	0846	0847
1120	0592	0593	0594	0595	0596	0597	0598	0599		1520	0848	0849	0850	0851	0852	0853	0854	0855
1130	0600	0601	0602	0603	0604	0605	0606	0607		1530	0856	0857	0858	0859	0860	0861	0862	0863
1140	0608	0609	0610	0611	0612	0613	0614	0615		1540	0864	0865	0866	0867	0868	0869	0870	0871
1150	0616	0617	0618	0619	0620	0621	0622	0623		1550	0872	0873	0874	0875	0876	0877	0878	0879
1160	0624	0625	0626	0627	0628	0629	0630	0631		1560	0880	0881	0882	0883	0884	0885	0886	0887
1170	0632	0633	0634	0635	0636	0637	0638	0639		1570	0888	0889	0890	0891	0892	0893	0894	0895
1200	0640	0641	0642	0643	0644	0645	0646	0647		1600	0896	0897	0898	0899	0900	0901	0902	0903
1210	0648	0649	0650	0651	0652	0653	0654	0655		1610	0904	0905	0906	0907	0908	0909	0910	0911
1220	0656	0657	0658	0659	0660	0661	0662	0663		1620	0912	0913	0914	0915	0916	0917	0918	0919
1230	0664	0665	0666	0667	0668	0669	0670	0671		1630	0920	0921	0922	0923	0924	0925	0926	0927
1240	0673	0674	0675	0676	0677	0678	0679			1640	0928	0929	0930	0931	0932	0933	0934	0935
1250	0681	0682	0683	0684	0685	0686	0687			1650	0937	0938	0939	0940	0941	0942	0943	
1260	0692	0693	0694	0695	0696	0697	0698			1660	0945	0946	0947	0948	0949	0950	0951	
1270	0699	0700	0701	0702	0703	0704	0705			1670	0954	0955	0956	0957	0958			

Figure B-6 Steps for Converting Decimal 870 to Octal

B.3.3 Converting Decimal 3826 to Octal

Figure B-7 illustrates the steps for converting decimal 3826 to octal 7382.

OCTAL TO DECIMAL CONVERSION TABLE

6360	3312	3313	3314	3315	3316	3317	3318	3319		6760	3568	3569	3570	3571	3572	3573	3574	3575
6370	3320	3321	3322	3323	3324	3325	3326	3327		6770	3576	3577	3578	3579	3580	3581	3582	3583
7000	3584	3585	3586	3587	3588	3589	3590	3591		7400	3840	3841	3842	3843	3844	3845	3846	3847
7010	3592	3593	3594	3595	3596	3597	3598	3599		7410	3848	3849	3850	3851	3852	3853	3854	3855
7020	3600	3601	3602	3603	3604	3605	3606	3607		7420	3856	3857	3858	3859	3860	3861	3862	3863
7030	3608	3609	3610	3611	3612	3613	3614	3615		7430	3864	3865	3866	3867	3868	3869	3870	3871
7040	3616	3617	3618	3619	3620	3621	3622	3623		7440	3872	3873	3874	3875	3876	3877	3878	3879
7050	3624	3625	3626	3627	3628	3629	3630	3631		7450	3880	3881	3882	3883	3884	3885	3886	3887
7060	3632	3633	3634	3635	3636	3637	3638	3639		7460	3888	3889	3890	3891	3892	3893	3894	3895
7070	3640	3641	3642	3643	3644	3645	3646	3647		7470	3896	3897	3898	3899	3900	3901	3902	3903
7100	3648	3649	3650	3651	3652	3653	3654	3655		7500	3904	3905	3906	3907	3908	3909	3910	3911
7110	3656	3657	3658	3659	3660	3661	3662	3663		7510	3912	3913	3914	3915	3916	3917	3918	3919
7120	3664	3665	3666	3667	3668	3669	3670	3671		7520	3920	3921	3922	3923	3924	3925	3926	3927
7130	3672	3673	3674	3675	3676	3677	3678	3679		7530	3928	3929	3930	3931	3932	3933	3934	3935
7140	3680	3681	3682	3683	3684	3685	3686	3687		7540	3936	3937	3938	3939	3940	3941	3942	3943
7150	3688	3689	3690	3691	3692	3693	3694	3695		7550	3944	3945	3946	3947	3948	3949	3950	3951
7160	3696	3697	3698	3699	3700	3701	3702	3703		7560	3952	3953	3954	3955	3956	3957	3958	3959
7170	3704	3705	3706	3707	3708	3709	3710	3711		7570	3960	3961	3962	3963	3964	3965	3966	3967
7200	3712	3713	3714	3715	3716	3717	3718	3719		7600	3968	3969	3970	3971	3972	3973	3974	3975
7210	3720	3721	3722	3723	3724	3725	3726	3727		7610	3976	3977	3978	3979	3980	3981	3982	3983
7220	3728	3729	3730	3731	3732	3733	3734	3735		7620	3984	3985	3986	3987	3988	3989	3990	3991
7230	3736	3737	3738	3739	3740	3741	3742	3743		7630	3992	3993	3994	3995	3996	3997	3998	3999
7240	3744	3745	3746	3747	3748	3749	3750	3751		7640	4000	4001	4002	4003	4004	4005	4006	4007
7250	3752	3753	3754	3755	3756	3757	3758	3759		7650	4008	4009	4010	4011	4012	4013	4014	4015
7260	3760	3761	3762	3763	3764	3765	3766	3767		7660	4016	4017	4018	4019	4020	4021	4022	4023
7270	3768	3769	3770	3771	3772	3773	3774	3775		7670	4024	4025	4026	4027	4028	4029	4030	4031
7300	3776	3777	3778	3779	3780	3781	3782	3783		7700	4032	4033	4034	4035	4036	4037	4038	4039
7310	3784	3785	3786	3787	3788	3789	3790	3791		7710	4040	4041	4042	4043	4044	4045	4046	4047
7320	3792	3793	3794	3795	3796	3797	3798	3799		7720	4048	4049	4050	4051	4052	4053	4054	4055
7330	3800	3801	3802	3803	3804	3805	3806	3807		7730	4056	4057	4058	4059	4060	4061	4062	4063
7340	3808	3809	3810	3811	3812	3813	3814	3815		7740	4064	4065	4066	4067	4068	4069	4070	4071
7352	3816	3817	3818	3819	3820	3821	3822	3823		7750	4072	4073	4074	4075	4076	4077	4078	4079
7362	3824	3825	3826	3827	3828	3829	3830	3831		7760	4080	4081	4082	4083	4084	4085	4086	4087
7370	3833	3834	3835	3836	3837	3838	3839	3840		7770	4088	4089	4090	4091	4092	4093	4094	4095

Figure B-7 Steps for Converting Decimal 3826 to Octal

B.4 CONVERTING OCTAL NUMBERS FROM 10000 TO 77777 TO DECIMAL NUMBERS

Three examples follow. The procedures outlined in Section B.4.1 apply to Sections B.4.2 and B.4.3.

B.4.1 Converting Octal 10042 to Decimal

Figure B-8 illustrates the steps to convert octal 10042 to decimal 4130.

OCTAL TO DECIMAL CONVERSION TABLE

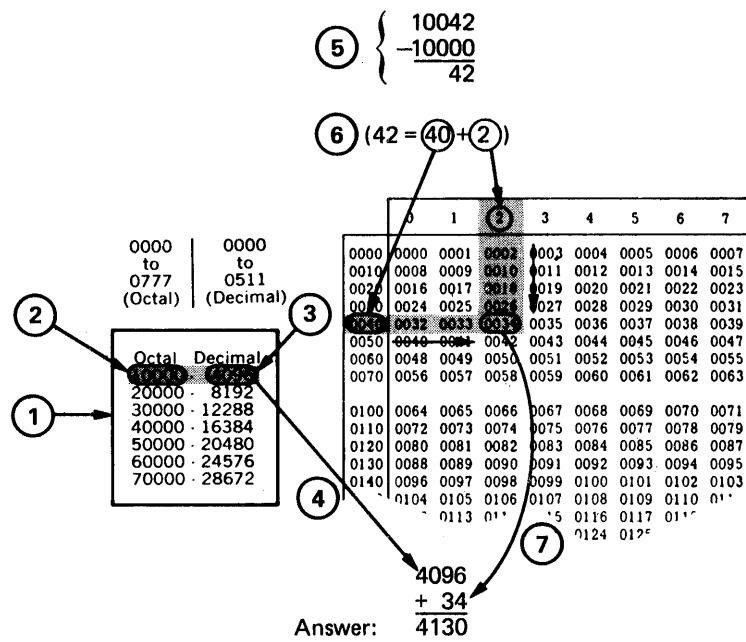


Figure B-8 Steps for Converting Octal 10042 to Decimal

1. Locate the columns labeled "OCTAL" and "DECIMAL" in the margin of Table B-1.
2. Find 10000 under the "OCTAL" column. (This is the largest octal number listed in the column less than 10042.)
3. Locate the decimal equivalent of 10000 under the "DECIMAL" column (4096).
4. Record this number.
5. Subtract octal 10000 from octal 10042 ($10042 - 10000 = 42$).
6. Take the difference (42) obtained in step 5 and use it to locate its decimal equivalent in Table B-1 as described in Section B.2.1. (The decimal equivalent of octal 42 is 34.)
7. Add 34 to 4096. The sum (4130) is your answer (4130 is the decimal equivalent of octal 10042).

B.4.2 Converting Octal 67341 to Decimal

Figure B-9 illustrates the steps for converting octal 67341 to decimal 28385.

OCTAL TO DECIMAL CONVERSION TABLE

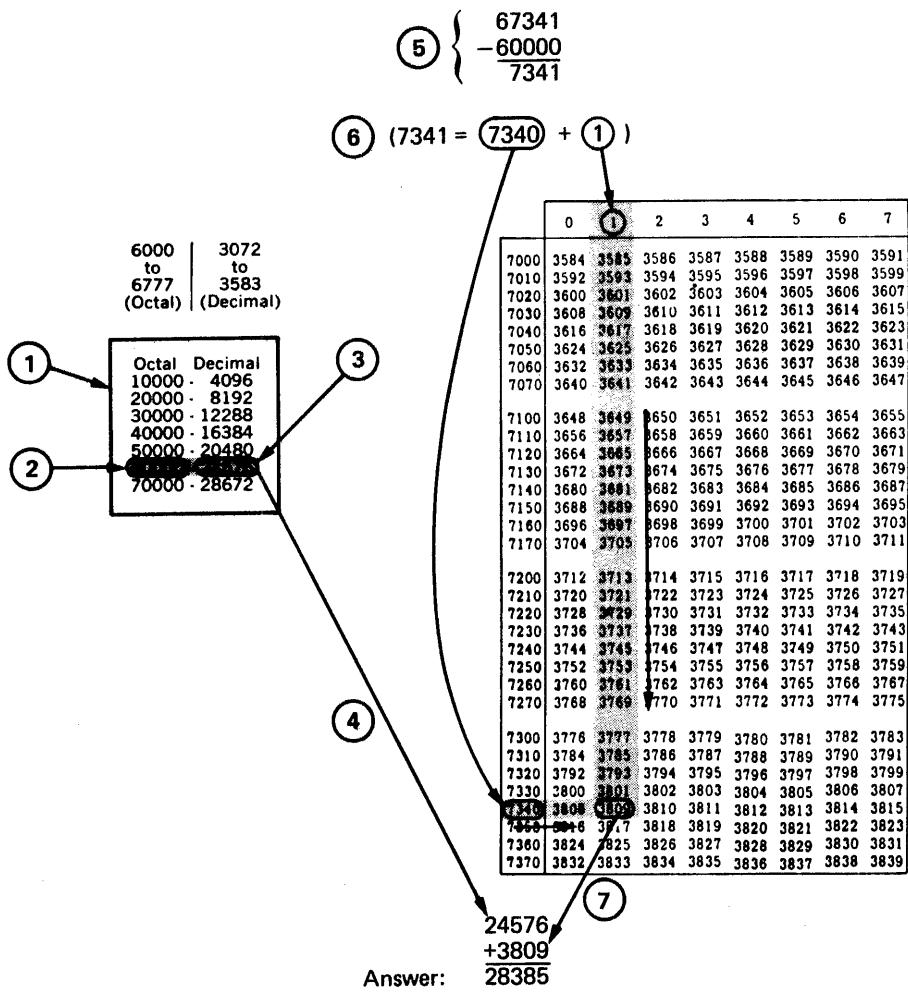


Figure B-9 Steps for Converting Octal 67341 to Decimal

B.4.3 Converting Octal 30000 to Decimal

Figure B-10 illustrates the steps for converting octal 30000 to decimal 12288.

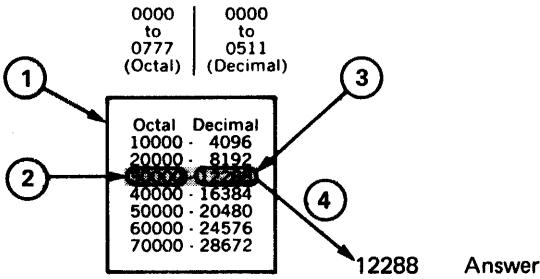


Figure B-10 Steps for Converting Octal 30000 to Decimal

OCTAL TO DECIMAL CONVERSION TABLE

B.5 CONVERTING DECIMAL NUMBERS RANGING FROM 4096 TO 32767 TO OCTAL

Two examples follow. The procedures outlined in Section B.5.1 apply to Section B.5.2.

B.5.1 Converting Decimal 4787 to Octal

Refer to Figure B-11. The numbers listed in the table correspond to the steps presented below.

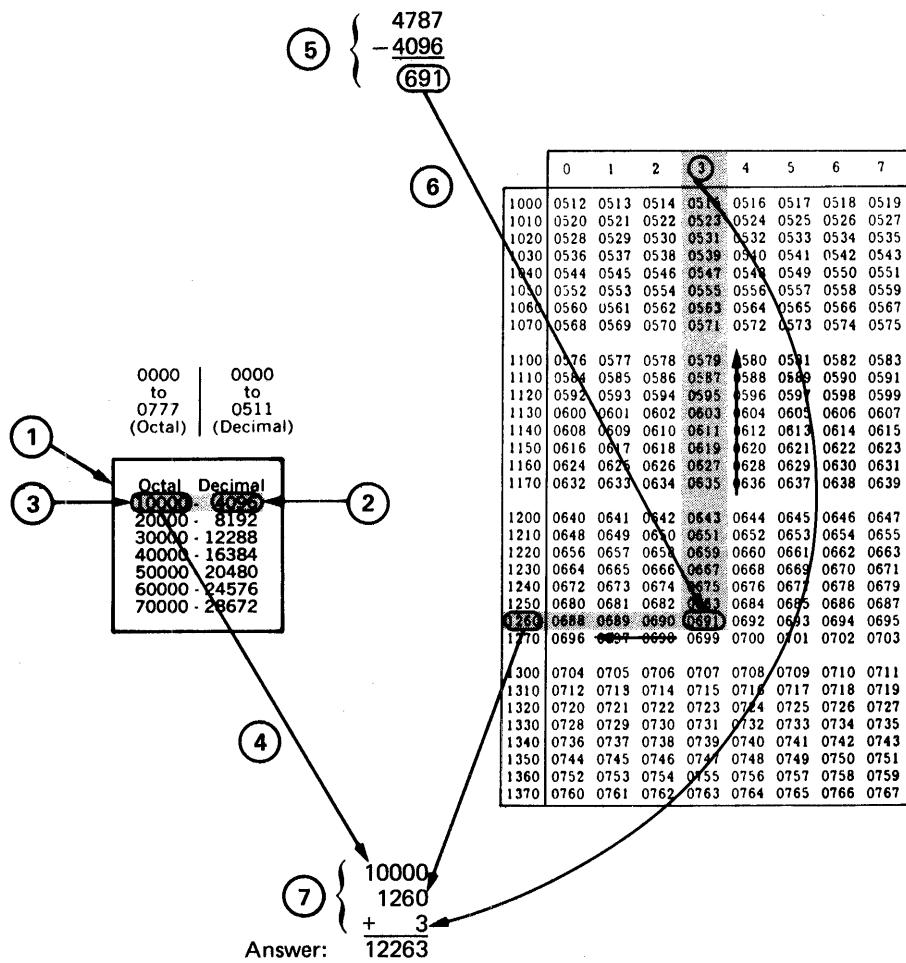


Figure B-11 Steps for Converting Decimal 4787 to Octal

1. Locate the columns labeled "OCTAL" and "DECIMAL" in the margin of Table B-1.
2. Find 4096 under the "DECIMAL" column. (This is the largest decimal number listed in the column less than 4787.)
3. Locate the octal equivalent of 4096 under the "OCTAL" column (10000).
4. Record this number.

OCTAL TO DECIMAL CONVERSION TABLE

5. Subtract 4096 from 4787. ($4787 - 4096 = 691$)
6. Take the difference (691) obtained in step 5 and use it to locate the octal equivalent as described in Section B.3.1. The octal equivalent of 691 is 1263.)
7. Add 1263 to 10000. The sum is your answer ($10000 + 1263 = 11263$. 11263 is the octal equivalent of decimal 4787.)

B.5.2 Converting Decimal 26872 to Octal

Figure B-12 illustrates the steps for converting decimal 26872 to octal 64370.

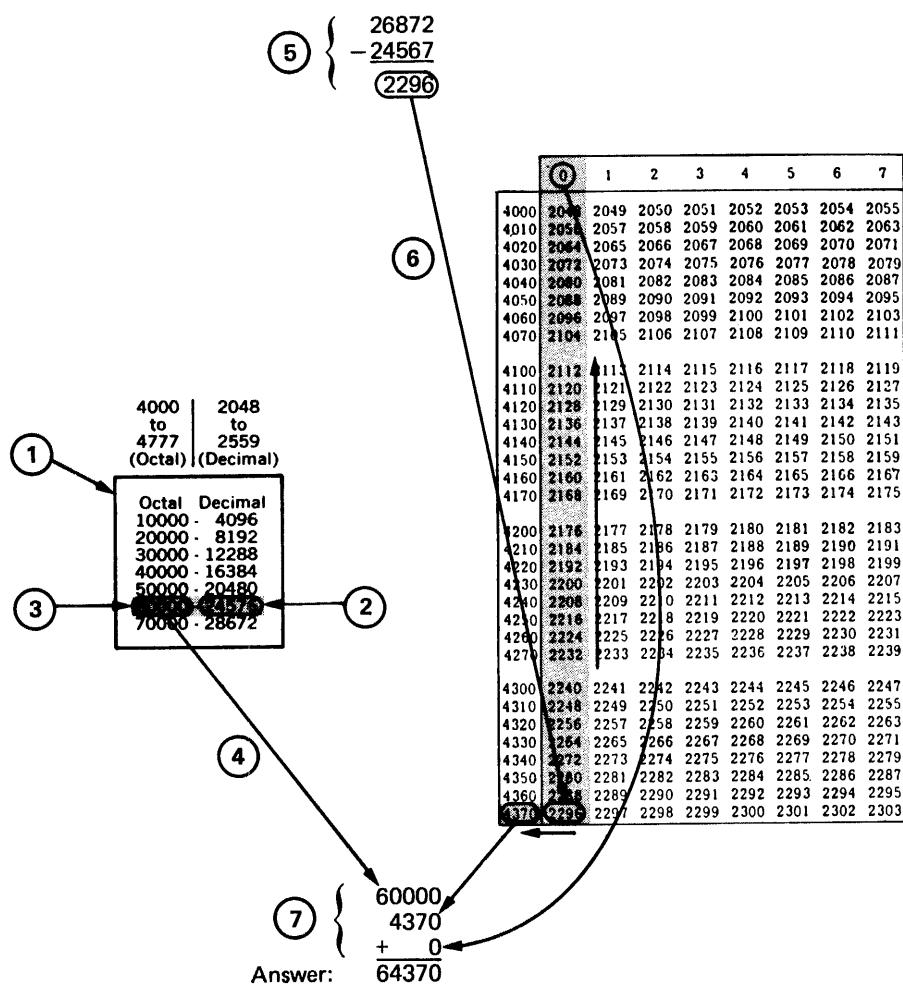


Figure B-12 Steps for Converting Decimal 26872 to Octal

OCTAL TO DECIMAL CONVERSION TABLE

Table B-1
Octal-Decimal Integer Conversion

Octal								Decimal									
0		1		2		3		4		5		6		7			
0000	0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	0010	0011	0012	0013	0014	0015	
0010	0016	0017	0018	0019	0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	0030	0031	
0030	0032	0033	0034	0035	0036	0037	0038	0039	0040	0041	0042	0043	0044	0045	0046	0047	
0050	0056	0057	0058	0059	0060	0061	0062	0063	0064	0065	0066	0067	0068	0069	0070	0071	
0100	0112	0117	0122	0123	0124	0125	0126	0127	0130	0137	0138	0139	0140	0141	0142	0143	
0120	0144	0145	0146	0147	0148	0149	0150	0151	0152	0153	0154	0155	0156	0157	0158	0159	
0140	0160	0161	0162	0163	0164	0165	0166	0167	0168	0169	0170	0171	0172	0173	0174	0175	
0170	0184	0185	0186	0187	0188	0189	0190	0191	0192	0193	0194	0195	0196	0197	0198	0199	
0200	0200	0201	0202	0203	0204	0205	0206	0207	0208	0209	0210	0211	0212	0213	0214	0215	
0230	0216	0217	0218	0219	0220	0221	0222	0223	0240	0225	0226	0227	0228	0229	0230	0231	
0250	0232	0233	0234	0235	0236	0237	0238	0239	0270	0240	0241	0242	0243	0244	0245	0246	
0270	0248	0249	0250	0251	0252	0253	0254	0255	0300	0312	0313	0314	0315	0316	0317	0318	
0310	0320	0321	0322	0323	0324	0325	0326	0327	0350	0360	0361	0362	0363	0364	0365	0366	
0320	0366	0337	0338	0339	0340	0341	0342	0343	0370	0368	0369	0370	0371	0372	0373	0374	
0330	0376	0377	0378	0379	0380	0381	0382	0383	0400	0384	0385	0386	0387	0388	0389	0390	
0350	0400	0401	0402	0403	0404	0405	0406	0407	0420	0408	0409	0410	0411	0412	0413	0414	
0370	0440	0441	0442	0443	0444	0445	0446	0447	0460	0446	0457	0458	0459	0460	0461	0462	
0400	0463	0464	0465	0466	0467	0468	0469	0470	0471	0472	0473	0474	0475	0476	0477	0478	
0420	0479	0480	0481	0482	0483	0484	0485	0486	0487	0488	0489	0490	0491	0492	0493	0494	
0440	0495	0497	0498	0499	0500	0501	0502	0503	0504	0505	0506	0507	0508	0509	0510	0511	
Octal								Decimal									
0		1		2		3		4		5		6		7			
1000	0512	0513	0514	0515	0516	0517	0518	0519	0520	0521	0522	0523	0524	0525	0526	0527	
1010	0528	0529	0530	0531	0532	0533	0534	0535	0536	0537	0538	0539	0540	0541	0542	0543	
1030	0544	0545	0546	0547	0548	0549	0550	0551	0552	0553	0554	0555	0556	0557	0558	0559	
1050	0560	0561	0562	0563	0564	0565	0566	0567	0568	0569	0570	0571	0572	0573	0574	0575	
1100	0576	0577	0578	0579	0580	0581	0582	0583	0584	0585	0586	0587	0588	0589	0590	0591	
1110	0592	0593	0594	0595	0596	0597	0598	0599	0600	0601	0602	0603	0604	0605	0606	0607	
1130	0608	0609	0610	0611	0612	0613	0614	0615	0616	0617	0618	0619	0620	0621	0622	0623	
1150	0624	0625	0626	0627	0628	0629	0630	0631	0632	0633	0634	0635	0636	0637	0638	0639	
1200	0640	0641	0642	0643	0644	0645	0646	0647	0648	0649	0650	0651	0652	0653	0654	0655	
1210	0656	0657	0658	0659	0660	0661	0662	0663	0664	0665	0666	0667	0668	0669	0670	0671	
1220	0672	0673	0674	0675	0676	0677	0678	0679	0680	0681	0682	0683	0684	0685	0686	0687	
1240	0688	0689	0690	0691	0692	0693	0694	0695	0696	0697	0698	0699	0700	0701	0702	0703	
1300	0704	0705	0706	0707	0708	0709	0710	0711	0712	0713	0714	0715	0716	0717	0718	0719	
1310	0720	0721	0722	0723	0724	0725	0726	0727	0728	0729	0730	0731	0732	0733	0734	0735	
1330	0736	0737	0738	0739	0740	0741	0742	0743	0744	0745	0746	0747	0748	0749	0750	0751	
1350	0752	0753	0754	0755	0756	0757	0758	0759	0760	0761	0762	0763	0764	0765	0766	0767	
1370	0760	0761	0762	0763	0764	0765	0766	0767	0768	0769	0770	0771	0772	0773	0774	0775	
1400	0776	0777	0778	0779	0780	0781	0782	0783	0784	0785	0786	0787	0788	0789	0790	0791	
1420	0792	0793	0794	0795	0796	0797	0798	0799	0800	0801	0802	0803	0804	0805	0806	0807	
1440	0808	0809	0810	0811	0812	0813	0814	0815	0816	0817	0818	0819	0820	0821	0822	0823	
1460	0824	0825	0826	0827	0828	0829	0830	0831	0832	0833	0834	0835	0836	0837	0838	0839	
1500	0832	0833	0834	0835	0836	0837	0838	0839	0840	0841	0842	0843	0844	0845	0846	0847	
1510	0848	0849	0850	0851	0852	0853	0854	0855	0856	0857	0858	0859	0860	0861	0862	0863	
1530	0864	0865	0866	0867	0868	0869	0870	0871	0872	0873	0874	0875	0876	0877	0878	0879	
1550	0878	0879	0880	0881	0882	0883	0884	0885	0886	0887	0888	0889	0890	0891	0892	0893	
1570	0888	0889	0890	0891	0892	0893	0894	0895	0896	0897	0898	0899	0900	0901	0902	0903	
1600	0896	0897	0898	0899	0900	0901	0902	0903	0904	0905	0906	0907	0908	0909	0910	0911	
1620	0912	0913	0914	0915	0916	0917	0918	0919	0920	0921	0922	0923	0924	0925	0926	0927	
1640	0928	0929	0930	0931	0932	0933	0934	0935	0936	0937	0938	0939	0940	0941	0942	0943	
1660	0944	0945	0946	0947	0948	0949	0950	0951	0952	0953	0954	0955	0956	0957	0958	0959	
1700	0960	0961	0962	0963	0964	0965	0966	0967	0968	0969	0970	0971	0972	0973	0974	0975	
1720	0976	0977	0978	0979	0980	0981	0982	0983	0984	0985	0986	0987	0988	0989	0990	0991	
1730	0984	0985	0986	0987	0988	0989	0990	0991	0992	0993	0994	0995	0996	0997	0998	0999	
1740	0992	0993	0994	0995	0996	0997	0998	0999	1000	1001	1002	1003	1004	1005	1006	1007	
1750	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023	

(Continued on next page)

OCTAL TO DECIMAL CONVERSION TABLE

Table B-1 (Cont.)
Octal-Decimal Integer Conversion

		0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7		
2000 1024 to to 2777 1535 (Octal) (Decimal)		2000	1024	1025	1026	1027	1028	1029	1030	1031	2400	1280	1281	1282	1283	1284	1285	1286	1287
		2010	1032	1033	1034	1035	1036	1037	1038	1039	2410	1288	1289	1290	1291	1292	1293	1294	1295
		2020	1040	1041	1042	1043	1044	1045	1046	1047	2420	1296	1297	1298	1299	1300	1301	1302	1303
		2030	1048	1049	1050	1051	1052	1053	1054	1055	2430	1304	1305	1306	1307	1308	1309	1310	1311
		2040	1056	1057	1058	1059	1060	1061	1062	1063	2440	1312	1313	1314	1315	1316	1317	1318	1319
		2050	1064	1065	1066	1067	1068	1069	1070	1071	2450	1320	1321	1322	1323	1324	1325	1326	1327
		2060	1072	1073	1074	1075	1076	1077	1078	1079	2460	1328	1329	1330	1331	1332	1333	1334	1335
		2070	1080	1081	1082	1083	1084	1085	1086	1087	2470	1336	1337	1338	1339	1340	1341	1342	1343
Octal Decimal											2500	1344	1345	1346	1347	1348	1349	1350	1351
10000 - 4096											2510	1352	1353	1354	1355	1356	1357	1358	1359
20000 - 8192											2520	1360	1361	1362	1363	1364	1365	1366	1367
30000 - 12288											2530	1368	1369	1370	1371	1372	1373	1374	1375
40000 - 16384											2540	1376	1377	1378	1379	1380	1381	1382	1383
50000 - 20480											2550	1384	1385	1386	1387	1388	1389	1390	1391
60000 - 24576											2560	1392	1393	1394	1395	1396	1397	1398	1399
70000 - 28672											2570	1400	1401	1402	1403	1404	1405	1406	1407
											2600	1408	1409	1410	1411	1412	1413	1414	1415
											2610	1416	1417	1418	1419	1420	1421	1422	1423
											2620	1424	1425	1426	1427	1428	1429	1430	1431
											2630	1432	1433	1434	1435	1436	1437	1438	1439
											2640	1440	1441	1442	1443	1444	1445	1446	1447
											2650	1448	1449	1450	1451	1452	1453	1454	1455
											2660	1456	1457	1458	1459	1460	1461	1462	1463
											2670	1464	1465	1466	1467	1468	1469	1470	1471
											2700	1472	1473	1474	1475	1476	1477	1478	1479
											2710	1480	1481	1482	1483	1484	1485	1486	1487
											2720	1488	1489	1490	1491	1492	1493	1494	1495
											2730	1496	1497	1498	1499	1500	1501	1502	1503
											2740	1504	1505	1506	1507	1508	1509	1510	1511
											2750	1512	1513	1514	1515	1516	1517	1518	1519
											2760	1520	1521	1522	1523	1524	1525	1526	1527
											2770	1528	1529	1530	1531	1532	1533	1534	1535
3000 1536 to to 3777 2047 (Octal) (Decimal)		3000	1536	1537	1538	1539	1540	1541	1542	1543	3400	1792	1793	1794	1795	1796	1797	1798	1799
		3010	1544	1545	1546	1547	1548	1549	1550	1551	3410	1800	1801	1802	1803	1804	1805	1806	1807
		3020	1552	1553	1554	1555	1556	1557	1558	1559	3420	1808	1809	1810	1811	1812	1813	1814	1815
		3030	1560	1561	1562	1563	1564	1565	1566	1567	3430	1816	1817	1818	1819	1820	1821	1822	1823
		3040	1568	1569	1570	1571	1572	1573	1574	1575	3440	1824	1825	1826	1827	1828	1829	1830	1831
		3050	1576	1577	1578	1579	1580	1581	1582	1583	3450	1832	1833	1834	1835	1836	1837	1838	1839
		3060	1584	1585	1586	1587	1588	1589	1590	1591	3460	1840	1841	1842	1843	1844	1845	1846	1847
		3070	1592	1593	1594	1595	1596	1597	1598	1599	3470	1848	1849	1850	1851	1852	1853	1854	1855
										3500	1856	1857	1858	1859	1860	1861	1862	1863	
										3510	1864	1865	1866	1867	1868	1869	1870	1871	
										3520	1872	1873	1874	1875	1876	1877	1878	1879	
										3530	1880	1881	1882	1883	1884	1885	1886	1887	
										3540	1888	1889	1890	1891	1892	1893	1894	1895	
										3550	1896	1897	1898	1899	1900	1901	1902	1903	
										3560	1904	1905	1906	1907	1908	1909	1910	1911	
										3570	1912	1913	1914	1915	1916	1917	1918	1919	
										3600	1920	1921	1922	1923	1924	1925	1926	1927	
										3610	1928	1929	1930	1931	1932	1933	1934	1935	
										3620	1936	1937	1938	1939	1940	1941	1942	1943	
										3630	1944	1945	1946	1947	1948	1949	1950	1951	
										3640	1952	1953	1954	1955	1956	1957	1958	1959	
										3650	1960	1961	1962	1963	1964	1965	1966	1967	
										3660	1968	1969	1970	1971	1972	1973	1974	1975	
										3670	1976	1977	1978	1979	1980	1981	1982	1983	
										3700	1984	1985	1986	1987	1988	1989	1990	1991	
										3710	1992	1993	1994	1995	1996	1997	1998	1999	
										3720	2000	2001	2002	2003	2004	2005	2006	2007	
										3730	2008	2009	2010	2011	2012	2013	2014	2015	
										3740	2016	2017	2018	2019	2020	2021	2022	2023	
										3750	2024	2025	2026	2027	2028	2029	2030	2031	
										3760	2040	2041	2042	2043	2044	2045	2046	2047	
										3770									

(Continued on next page)

OCTAL TO DECIMAL CONVERSION TABLE

Table B-1 (Cont.)
Octal-Decimal Integer Conversion

	0	1	2	3	4	5	6	7		0	1	2	3	4	5	6	7		0	1	2	3	4	5	6	7		
4000	2048	2049	2050	2051	2052	2053	2054	2055		4400	2304	2305	2306	2307	2308	2309	2310	2311	4000	2048								
4010	2056	2057	2058	2059	2060	2061	2062	2063		4410	2312	2313	2314	2315	2316	2317	2318	2319	to	4777	to							
4020	2064	2065	2066	2067	2068	2069	2070	2071		4420	2320	2321	2322	2323	2324	2325	2326	2327		(Octal)	(Decimal)							
4030	2072	2073	2074	2075	2076	2077	2078	2079		4430	2328	2329	2330	2331	2332	2333	2334	2335										
4040	2080	2081	2082	2083	2084	2085	2086	2087		4440	2336	2337	2338	2339	2340	2341	2342	2343										
4050	2088	2089	2090	2091	2092	2093	2094	2095		4450	2344	2345	2346	2347	2348	2349	2350	2351										
4060	2096	2097	2098	2099	2100	2101	2102	2103		4460	2352	2353	2354	2355	2356	2357	2358	2359										
4070	2104	2105	2106	2107	2108	2109	2110	2111		4470	2360	2361	2362	2363	2364	2365	2366	2367										
4100	2112	2113	2114	2115	2116	2117	2118	2119		4500	2368	2369	2370	2371	2372	2373	2374	2375										
4110	2120	2121	2122	2123	2124	2125	2126	2127		4510	2376	2377	2378	2379	2380	2381	2382	2383										
4120	2128	2129	2130	2131	2132	2133	2134	2135		4520	2384	2385	2386	2387	2388	2389	2390	2391										
4130	2136	2137	2138	2139	2140	2141	2142	2143		4530	2392	2393	2394	2395	2396	2397	2398	2399										
4140	2144	2145	2146	2147	2148	2149	2150	2151		4540	2400	2401	2402	2403	2404	2405	2406	2407										
4150	2152	2153	2154	2155	2156	2157	2158	2159		4550	2408	2409	2410	2411	2412	2413	2414	2415										
4160	2160	2161	2162	2163	2164	2165	2166	2167		4560	2416	2417	2418	2419	2420	2421	2422	2423										
4170	2168	2169	2170	2171	2172	2173	2174	2175		4570	2424	2425	2426	2427	2428	2429	2430	2431										
4200	2176	2177	2178	2179	2180	2181	2182	2183		4600	2432	2433	2434	2435	2436	2437	2438	2439										
4210	2184	2185	2186	2187	2188	2189	2190	2191		4610	2440	2441	2442	2443	2444	2445	2446	2447										
4220	2192	2193	2194	2195	2196	2197	2198	2199		4620	2448	2449	2450	2451	2452	2453	2454	2455										
4230	2200	2201	2202	2203	2204	2205	2206	2207		4630	2456	2457	2458	2459	2460	2461	2462	2463										
4240	2208	2209	2210	2211	2212	2213	2214	2215		4640	2464	2465	2466	2467	2468	2469	2470	2471										
4250	2216	2217	2218	2219	2220	2221	2222	2223		4650	2472	2473	2474	2475	2476	2477	2478	2479										
4260	2224	2225	2226	2227	2228	2229	2230	2231		4660	2480	2481	2482	2483	2484	2485	2486	2487										
4270	2232	2233	2234	2235	2236	2237	2238	2239		4670	2488	2489	2490	2491	2492	2493	2494	2495										
4300	2240	2241	2242	2243	2244	2245	2246	2247		4700	2496	2497	2498	2499	2500	2501	2502	2503										
4310	2248	2249	2250	2251	2252	2253	2254	2255		4710	2504	2505	2506	2507	2508	2509	2510	2511										
4320	2256	2257	2258	2259	2260	2261	2262	2263		4720	2512	2513	2514	2515	2516	2517	2518	2519										
4330	2264	2265	2266	2267	2268	2269	2270	2271		4730	2520	2521	2522	2523	2524	2525	2526	2527										
4340	2272	2273	2274	2275	2276	2277	2278	2279		4740	2528	2529	2530	2531	2532	2533	2534	2535										
4350	2280	2281	2282	2283	2284	2285	2286	2287		4750	2536	2537	2538	2539	2540	2541	2542	2543										
4360	2288	2289	2290	2291	2292	2293	2294	2295		4760	2544	2545	2546	2547	2548	2549	2550	2551										
4370	2296	2297	2298	2299	2300	2301	2302	2303		4770	2552	2553	2554	2555	2556	2557	2558	2559										
	0	1	2	3	4	5	6	7		0	1	2	3	4	5	6	7		0	1	2	3	4	5	6	7		
5000	2560	2561	2562	2563	2564	2565	2566	2567		5400	2816	2817	2818	2819	2820	2821	2822	2823		5000	2560							
5010	2568	2569	2570	2571	2572	2573	2574	2575		5410	2824	2825	2826	2827	2828	2829	2830	2831		to	5777	to						
5020	2576	2577	2578	2579	2580	2581	2582	2583		5420	2832	2833	2834	2835	2836	2837	2838	2839										
5030	2584	2585	2586	2587	2588	2589	2590	2591		5430	2840	2841	2842	2843	2844	2845	2846	2847										
5040	2592	2593	2594	2595	2596	2597	2598	2599		5440	2848	2849	2850	2851	2852	2853	2854	2855										
5050	2600	2601	2602	2603	2604	2605	2606	2607		5450	2856	2857	2858	2859	2860	2861	2862	2863										
5060	2608	2609	2610	2611	2612	2613	2614	2615		5460	2864	2865	2866	2867	2868	2869	2870	2871										
5070	2616	2617	2618	2619	2620	2621	2622	2623		5470	2872	2873	2874	2875	2876	2877	2878	2879										
5100	2624	2625	2626	2627	2628	2629	2630	2631		5500	2880	2881	2882	2883	2884	2885	2886	2887										
5110	2632	2633	2634	2635	2636	2637	2638	2639		5510	2888	2889	2890	2891	2892	2893	2894	2895										
5120	2640	2641	2642	2643	2644	2645	2646	2647		5520	2896	2897	2898	2899	2900	2901	2902	2903										
5130	2648	2649	2650	2651	2652	2653	2654	2655		5530	2904	2905	2906	2907	2908	2909	2910	2911										
5140	2656	2657	2658	2659	2660	2661	2662	2663		5540	2912	2913	2914	2915	2916	2917	2918	2919										
5150	2664	2665	2666	2667	2668	2669	2670	2671		5550	2920	2921	2922	2923	2924	2925	2926	2927										
5160	2672	2673	2674	2675	2676	2677	2678	2679		5560	2928	2929	2930	2931	2932	2933	2934	2935										
5170	2680	2681	2682	2683	2684	2685	2686	2687		5570	2936	2937	2938	2939	2940	2941	2942	2943										
5200	2688	2689	2690	2691	2692	2693	2694	2695		5600	2944	2945	2946	2947	2948	2949	2950	2951										
5210	2696	2697	2698	2699	2700	2701	2702	2703		5610	2952	2953	2954	2955	2956	2957	2958	2959										
5220	2704	2705	2706	2707	2708	2709	2710	2711		5620	2960	2961	2962	2963	2964	2965	2966	2967										

OCTAL TO DECIMAL CONVERSION TABLE

Table B-1 (Cont.)
Octal-Decimal Integer Conversion

		0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7		
6000	3072	6000	3072	3073	3074	3075	3076	3077	3078	3079	6400	3328	3329	3330	3331	3332	3333	3334	3335
to	to	6010	3080	3081	3082	3083	3084	3085	3086	3087	6410	3336	3337	3338	3339	3340	3341	3342	3343
6777	3583	6020	3088	3089	3090	3091	3092	3093	3094	3095	6420	3344	3345	3346	3347	3348	3349	3350	3351
(Octal)	(Decimal)	6030	3096	3097	3098	3099	3100	3101	3102	3103	6430	3352	3353	3354	3355	3356	3357	3358	3359
Octal	Decimal	6040	3104	3105	3106	3107	3108	3109	3110	3111	6440	3360	3361	3362	3363	3364	3365	3366	3367
10000	4096	6050	3112	3113	3114	3115	3116	3117	3118	3119	6450	3368	3369	3370	3371	3372	3373	3374	3375
20000	8192	6060	3120	3121	3122	3123	3124	3125	3126	3127	6460	3376	3377	3378	3379	3380	3381	3382	3383
30000	12288	6070	3128	3129	3130	3131	3132	3133	3134	3135	6470	3384	3385	3386	3387	3388	3389	3390	3391
40000	16384	6100	3136	3137	3138	3139	3140	3141	3142	3143	6500	3392	3393	3394	3395	3396	3397	3398	3399
50000	20480	6110	3144	3145	3146	3147	3148	3149	3150	3151	6510	3400	3401	3402	3403	3404	3405	3406	3407
60000	24576	6120	3152	3153	3154	3155	3156	3157	3158	3159	6520	3408	3409	3410	3411	3412	3413	3414	3415
70000	28672	6130	3160	3161	3162	3163	3164	3165	3166	3167	6530	3416	3417	3418	3419	3420	3421	3422	3423
		6140	3168	3169	3170	3171	3172	3173	3174	3175	6540	3424	3425	3426	3427	3428	3429	3430	3431
		6150	3176	3177	3178	3179	3180	3181	3182	3183	6550	3432	3433	3434	3435	3436	3437	3438	3439
		6160	3184	3185	3186	3187	3188	3189	3190	3191	6560	3440	3441	3442	3443	3444	3445	3446	3447
		6170	3192	3193	3194	3195	3196	3197	3198	3199	6570	3448	3449	3450	3451	3452	3453	3454	3455
		6200	3200	3201	3202	3203	3204	3205	3206	3207	6600	3456	3457	3458	3459	3460	3461	3462	3463
		6210	3208	3209	3210	3211	3212	3213	3214	3215	6610	3464	3465	3466	3467	3468	3469	3470	3471
		6220	3216	3217	3218	3219	3220	3221	3222	3223	6620	3472	3473	3474	3475	3476	3477	3478	3479
		6230	3224	3225	3226	3227	3228	3229	3230	3231	6630	3480	3481	3482	3483	3484	3485	3486	3487
		6240	3232	3233	3234	3235	3236	3237	3238	3239	6640	3488	3489	3490	3491	3492	3493	3494	3495
		6250	3240	3241	3242	3243	3244	3245	3246	3247	6650	3496	3497	3498	3499	3500	3501	3502	3503
		6260	3248	3249	3250	3251	3252	3253	3254	3255	6660	3504	3505	3506	3507	3508	3509	3510	3511
		6270	3256	3257	3258	3259	3260	3261	3262	3263	6670	3512	3513	3514	3515	3516	3517	3518	3519
		6300	3264	3265	3266	3267	3268	3269	3270	3271	6700	3520	3521	3522	3523	3524	3525	3526	3527
		6310	3272	3273	3274	3275	3276	3277	3278	3279	6710	3528	3529	3530	3531	3532	3533	3534	3535
		6320	3280	3281	3282	3283	3284	3285	3286	3287	6720	3536	3537	3538	3539	3540	3541	3542	3543
		6330	3288	3289	3290	3291	3292	3293	3294	3295	6730	3544	3545	3546	3547	3548	3549	3550	3551
		6340	3296	3297	3298	3299	3300	3301	3302	3303	6740	3552	3553	3554	3555	3556	3557	3558	3559
		6350	3304	3305	3306	3307	3308	3309	3310	3311	6750	3560	3561	3562	3563	3564	3565	3566	3567
		6360	3312	3313	3314	3315	3316	3317	3318	3319	6760	3568	3569	3570	3571	3572	3573	3574	3575
		6370	3320	3321	3322	3323	3324	3325	3326	3327	6770	3576	3577	3578	3579	3580	3581	3582	3583
7000	3584	7000	3584	3585	3586	3587	3588	3589	3590	3591	7400	3840	3841	3842	3843	3844	3845	3846	3847
to	to	7010	3592	3593	3594	3595	3596	3597	3598	3599	7410	3848	3849	3850	3851	3852	3853	3854	3855
7777	4095	7020	3600	3601	3602	3603	3604	3605	3606	3607	7420	3856	3857	3858	3859	3860	3861	3862	3863
(Octal)	(Decimal)	7030	3608	3609	3610	3611	3612	3613	3614	3615	7430	3864	3865	3866	3867	3868	3869	3870	3871
7040	3616	3617	3618	3619	3620	3621	3622	3623	3624	7440	3872	3873	3874	3875	3876	3877	3878	3879	
7050	3624	3625	3626	3627	3628	3629	3630	3631	3632	7450	3880	3881	3882	3883	3884	3885	3886	3887	
7060	3632	3633	3634	3635	3636	3637	3638	3639	3640	7460	3888	3889	3890	3891	3892	3893	3894	3895	
7070	3640	3641	3642	3643	3644	3645	3646	3647	3648	7470	3896	3897	3898	3899	3900	3901	3902	3903	
7100	3648	3649	3650	3651	3652	3653	3654	3655	3656	7500	3904	3905	3906	3907	3908	3909	3910	3911	
7110	3656	3657	3658	3659	3660	3661	3662	3663	3664	7510	3912	3913	3914	3915	3916	3917	3918	3919	
7120	3664	3665	3666	3667	3668	3669	3670	3671	3672	7520	3920	3921	3922	3923	3924	3925	3926	3927	
7130	3672	3673	3674	3675	3676	3677	3678	3679	3680	7530	3928	3929	3930	3931	3932	3933	3934	3935	
7140	3680	3681	3682	3683	3684	3685	3686	3687	3688	7540	3936	3937	3938	3939	3940	3941	3942	3943	
7150	3688	3689	3690	3691	3692	3693	3694	3695	3696	7550	3944	3945	3946	3947	3948	3949	3950	3951	
7160	3696	3697	3698	3699	3700	3701	3702	3703	3704	7560	3952	3953	3954	3955	3956	3957	3958	3959	
7170	3704	3705	3706	3707	3708	3709	3710	3711	3712	7570	3960	3961	3962	3963	3964	3965	3966	3967	
7200	3712	3713	3714	3715	3716	3717	3718	3719	3720	7600	3968	3969	3970	3971	3972	3973	3974	3975	
7210	3720	3721	3722	3723	3724	3725	3726	3727	3728	7610	3976	3977	3978	3979	3980	3981	3982	3983	
7220	3728	3729	3730	3731	3732	3733	3734	3735	3736	7620	3984	3985	3986	3987	3988	3989	3990	3991	
7230	3736	3737	3738	3739	3740	3741	3742	3743	3744	7630	3992	3993	3994	3995	3996	3997	3998	3999	
7240	3744	3745	3746	3747	3748	3749	3750	3751	3752	7640	4000	4001	4002	4003	4004	4005	4006	4007	
7250	3752	3753	3754	3755	3756	3757	3758	3759	3760	7650	4008	4009	4010	4011	4012	4013	4014	4015	
7260	3760	3761	3762	3763	3764	3765	3766	3767	3768	7660	4016	4017	4018	4019	4020	4021	4022	4023	
7270	3768	3769	3770	3771	3772	3773	3774	3775	3776	7670	4024	4025	4026	4027	4028	4029	4030	4031	
7300	3776	3777	3778	3779	3780	3781	3782	3783	3784	7700	4032	4033	4034	4035	4036	4037	4038	4039	
7310	3784	3785	3786	3787	3788	3789	3790	3791	3792	7710	4040	4041	4042	4043	4044	4045	4046	4047	
7320	3792	3793	3794	3795	3796	3797	3798	3799	3800	7720	4048	4049	4050	4051	4052	4053	4054	4055	
7330	3800	3801	3802	3803	3804	3805	3806	3807	3808	7730	4056	4057	4058	4059	4060	4061	4062	4063	
7340	3808	3809	3810	3811	3812	3813	3814	3815	3816	7740	4064	4065	4066	4067	4068	4069	4070	4071	
7350	3816	3817	38																

APPENDIX C

TASK BUILDER DATA FORMATS

An object module consists of variable length records of information that describe the contents of the module. Six record (or block) types are included in the object language. These records guide the Task Builder in the translation of the object language into a task image.

The six record types are:

- Type 1 - Declare Global Symbol Directory (GSD)
- Type 2 - End of Global Symbol Directory
- Type 3 - Text Information (TXT)
- Type 4 - Relocation Directory (RLD)
- Type 5 - Internal Symbol Directory (ISD)
- Type 6 - End of Module

Each object module must consist of at least five of the record types. The only record type that is not mandatory is the internal symbol directory. The appearance of the various record types in an object module follows a defined format. See Figure C-1.

An object module must begin with a GSD record and end with an end-of-module record. Additional GSD records can occur anywhere in the file but must appear before an end-of-GSD record. An end-of-GSD record must appear before the end-of-module record, and at least one relocation directory record (RLD) must appear before the first text information record (TXT). Additional RLDs and TXTs can appear anywhere in the file. The internal symbol directory records (ISDs) can appear anywhere in the file between the initial GSD and end-of-module records.

Object module records are of variable length, and are identified by a record type code in the first byte of the record. The format of additional information in the record depends on the record type.

TASK BUILDER DATA FORMATS

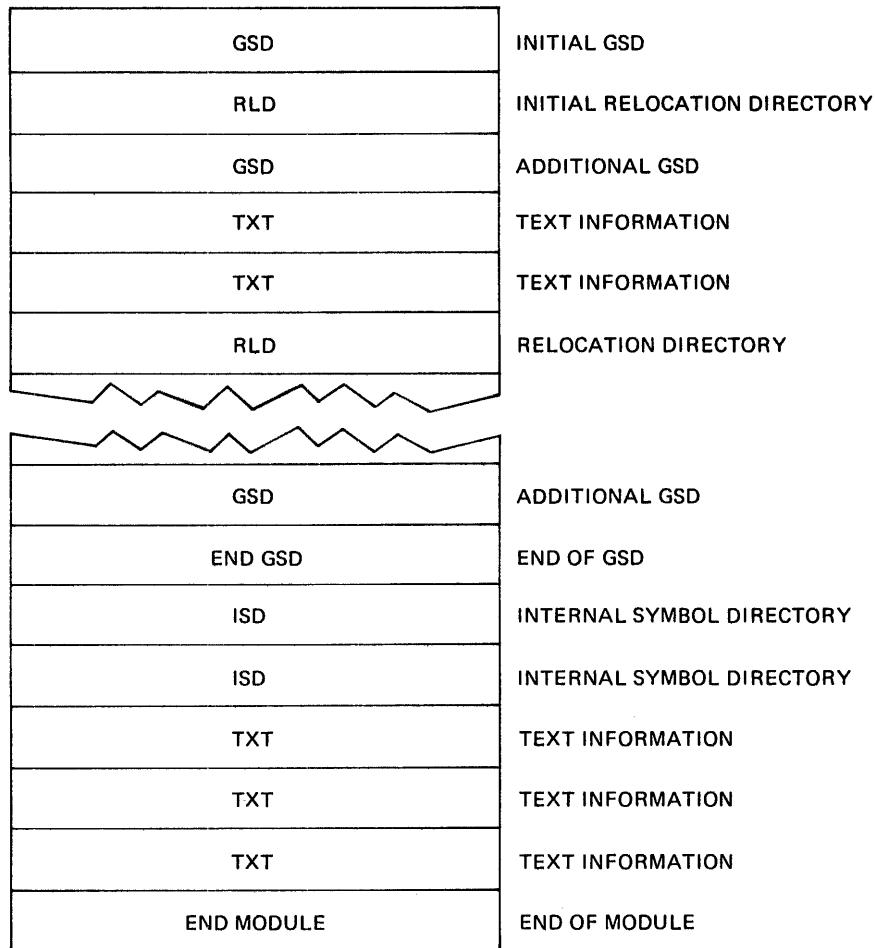


Figure C-1 General Object Module Format

C.1 GLOBAL SYMBOL DIRECTORY

Global symbol directory (GSD) records contain all the information necessary to assign addresses to global symbols and to allocate the memory required by a task.

GSD records are the only records processed in the first pass. You can save a significant amount of time if you put all GSD records at the beginning of a module, because less of the file must be read on the first pass.

GSD records contain seven types of entries:

Type	Entry
0	Module Name
1	Control Section Name
2	Internal Symbol Name

TASK BUILDER DATA FORMATS

Type	Entry
3	Transfer Address
4	Global Symbol Name
5	Program Section Name
6	Program Version Identification

There are four words in the GSD record for each entry type. The first two words contain six Radix-50 characters. The third word contains a flag byte and the entry type identification. The fourth word contains additional information about the entry. See Figure C-2 below.

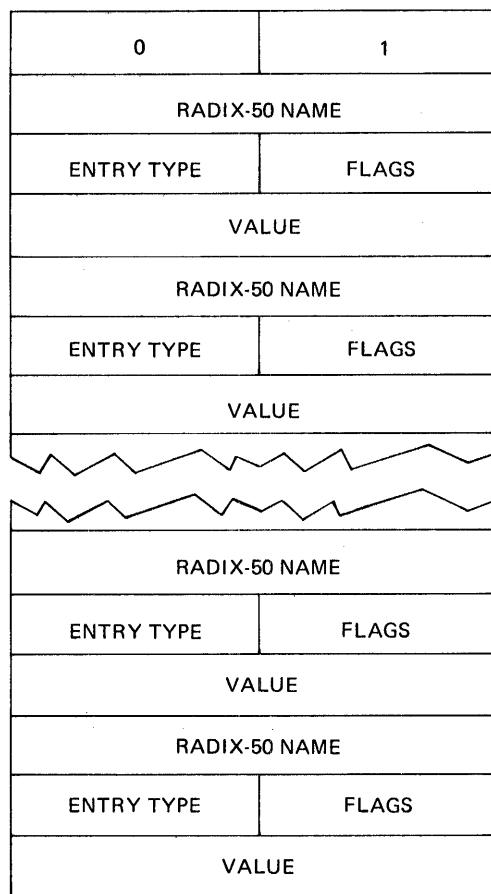


Figure C-2 GSD Record and Entry Format

C.1.1 Module Name

The module name entry, as illustrated in Figure C-3, declares the name of the object module. The name need not be unique with respect to other object modules because modules are identified by file, not module name. Only one module name entry can occur in any given object module.

TASK BUILDER DATA FORMATS

MODULE NAME	
0	0
0	

Figure C-3 Module Name Entry Format

C.1.2 Control Section Name

Control sections, which include ASECTS, blank CSECTS, and named CSECTS, are supplanted by PSECTS. For compatibility with other systems, Task Builder processes ASECTS and both forms of CSECTS. Section C.1.6 details the entry generated for a PSECT statement. In terms of the PSECT directive, ASECT and CSECT statements can be defined as follows:

- For a blank CSECT, a PSECT definition is:

.PSECT ,LCL,REL,CON,RW,I,LOW

- For a named CSECT, the PSECT definition is:

.PSECT name, GBL,REL,OVR,RW,I,LOW

- For an ASECT, the PSECT definition is:

.PSECT . ABS.,GBL,ABS,I,OVR,RW,LOW

ASECTS and CSECTS are processed by the Task Builder as PSECTS with the fixed attributes defined above. The entry generated for a control section is shown in Figure C-4.

CONTROL SECTION	
NAME	
1	(Ignored)
MAXIMUM LENGTH	

Figure C-4 Control Section Name Entry Format

C.1.3 Internal Symbol Name

The internal symbol name entry declares the name of an internal symbol (with respect to the module). The Task Builder does not support internal symbol tables, so the detailed format of this entry is not defined (Figure C-5). Any internal symbol entry encountered while the Task Builder reads the GSD is ignored.

TASK BUILDER DATA FORMATS

SYMBOL NAME	
2	0
UNDEFINED	

Figure C-5 Internal Symbol Name Entry Format

C.1.4 Transfer Address

The transfer address entry, as illustrated in Figure C-6, declares the transfer address of a module relative to a PSECT. The first two words of the entry define the name of the PSECT, and the fourth word, the relative offset from the beginning of that PSECT. If no transfer address is declared in a module, a transfer address entry either must not be included in the GSD, or a transfer address 000001 relative to the default absolute PSECT (. ABS.) must be specified.

PSECT NAME	
3	0
OFFSET	

Figure C-6 Transfer Address Entry Format

NOTE

If the PSECT is absolute and OFFSET is not 000001, then OFFSET is the actual transfer address.

C.1.5 Global Symbol Name

The global symbol name entry, as illustrated in Figure C-7, declares either a global reference or a definition. All definition entries must appear after the declaration of the PSECT they are defined in, and before the declaration of another PSECT. Global references can appear anywhere within the GSD.

The first two words of the entry define the name of the global symbol. The flag byte declares the attributes of the symbol, and the fourth word, the value of the symbol relative to the PSECT it is defined in.

The flag byte of the symbol declaration entry has the following bit assignments.

Bit 0 - Weak Qualifier

0 = Symbol is a strong definition or reference, and is resolved in the normal manner.

TASK BUILDER DATA FORMATS

1 = Symbol is a weak definition or reference. A weak reference (Bit 3=0) is ignored. A weak definition (Bit 3=1) is ignored unless a previous reference has been made.

Bit 1 - Not used.

Bit 2 - Definition Type

0 = Normal Definition or reference.

1 = Library definition. If the symbol is defined in a resident library STB file, the base address of the library is added to the value, and the symbol is converted to absolute (bit 5 is reset); otherwise the bit is ignored.

Bit 3 - Reference or Definition

0 = Global symbol reference.

1 = Global symbol definition.

Bit 4 - Not used.

Bit 5 - Relocation

0 = Absolute symbol value.

1 = Relative symbol value.

Bit 6 - 7 - Not used.

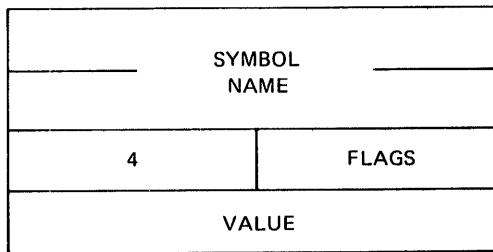


Figure C-7 Global Symbol Name Entry Format

C.1.6 PSECT Name

The PSECT name entry, as illustrated in Figure C-8, declares the name of a PSECT and its maximum length in the module. It also declares the attributes of the PSECT via the flag byte.

GSD records must be constructed such that once a PSECT name has been declared, all global symbol definitions pertaining to it must appear before another PSECT name is declared. Global symbols are declared in symbol declaration entries. Thus, the normal format is a series of PSECT names each followed by optional symbol declarations.

TASK BUILDER DATA FORMATS

The flag byte of the PSECT entry has the following bit assignments:

Bit 0 - Memory Speed

- 0 = PSECT is to occupy low speed (core) memory.
- 1 = PSECT is to occupy high speed (i.e., MOS/Bipolar) memory.

Bit 1 - Library PSECT

- 0 = Normal PSECT.
- 1 = Relocatable PSECT that references a resident library or common block.

Bit 2 - Allocation

- 0 = PSECT references are to be concatenated with other references to the same PSECT to form the total memory allocated to the PSECT.
- 1 = PSECT references are to be overlaid. The total memory allocated to the PSECT is the largest request made by individual references to the same PSECT.

Bit 3 - Reserved for the Task Builder

Bit 4 - Access

- 0 = PSECT has read/write access.
- 1 = PSECT has read-only access.

Bit 5 - Relocation

- 0 = PSECT is absolute and requires no relocation.
- 1 = PSECT is relocatable and references to the control PSECT must have a relocation bias added before they become absolute.

Bit 6 - Scope

- 0 = The scope of the PSECT is local. References to the same PSECT will be collected only within the segment in which the PSECT is defined.
- 1 = The scope of the PSECT is global. References to the PSECT are collected across segment boundaries. The segment in which a global PSECT is allocated storage is determined either by the first module that defines the PSECT on a path, or by direct placement of a PSECT in a segment by the .PSECT directive.

TASK BUILDER DATA FORMATS

Bit 7 - Type

0 = The PSECT contains instruction (I) references.

1 = The PSECT contains data (D) references.

NOTE

Compare these bit assignments with the PSECT attributes in Table 4-1.

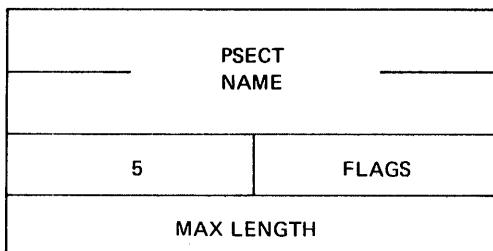


Figure C-8 PSECT Name Entry Format

NOTE

The length of all absolute PSECTs is zero.

C.1.7 Program Version Identification

The program version identification entry, as illustrated in Figure C-9, declares the version of the module. The Task Builder saves the version identification of the first module that defines a nonblank version. This identification is then included on the memory allocation map and is written in the label block of the task image file.

The first two words of the entry contain the version identification. The flag byte and fourth words are not used and contain no meaningful information.

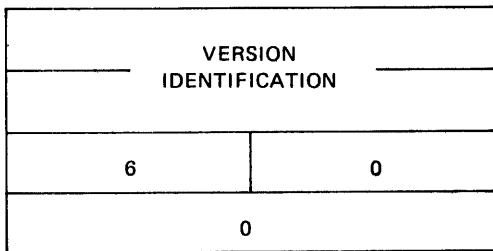


Figure C-9 Program Version Identification Entry Format

TASK BUILDER DATA FORMATS

C.2 END OF GLOBAL SYMBOL DIRECTORY

The end-of-global-symbol-directory record, as illustrated in Figure C-10, declares that no other GSD records are contained farther on in the module. Exactly one end-of-GSD record must appear in an object module. Its length is one word.

0	2

Figure C-10 End-of-GSD Record Format

C.3 TEXT INFORMATION

The text information record, as illustrated in Figure C-11, contains a byte string of information that is to be written directly into the task image file. The record consists of a load address followed by the byte string.

Text records can contain words and/or bytes of information whose final contents have not been determined yet. This information will be bound by a relocation directory record that immediately follows the text record (see Section C.4). If the text record does not need modification, then no relocation directory record is needed. Thus, multiple text records can appear in sequence before a relocation directory record.

The load address of the text record is specified as an offset from the current PSECT base. At least one relocation directory record must precede the first text record. This directory must declare the current PSECT.

LOAD ADDRESS	
TEXT	TEXT
TEXT	TEXT
TEXT	TEXT
~~~~~	
TEXT	TEXT

Figure C-11 Text Information Record Format

## TASK BUILDER DATA FORMATS

The Task Builder writes a text record directly into the task image file and computes the value of the load address minus four. This value is stored in anticipation of a subsequent relocation directory that modifies words and/or bytes that are contained in the text record. When added to a relocation directory displacement byte, this value yields the address of the word and/or byte to be modified in the task image.

### C.4 RELOCATION DIRECTORY

Relocation directory records (see Figure C-12) contain the information necessary to relocate and link the preceding text information record. Every module must have at least one relocation directory record that precedes the first text information record. The first record does not modify a preceding text record but rather defines the current PSECT and location. Relocation directory records contain 15 types of entries. These entries are classified as relocation or location modification entries. The following types are defined:

Type	Definition
1	Internal Relocation
2	Global Relocation
3	Internal Displaced Relocation
4	Global Displaced Relocation
5	Global Additive Relocation
6	Global Additive Displaced Relocation
7	Location Counter Definition
10	Location Counter Modification
11	Program Limits
12	PSECT Relocation
13	Not used
14	PSECT Displaced Relocation
15	PSECT Additive Relocation
16	PSECT Additive Displaced Relocation
17	Complex Relocation
20	Additive Relocation

Each type of entry is represented by a command byte (specifies type of entry and word/byte modification), followed by a displacement byte, and then by the information required for the particular type of entry. The displacement byte, when added to the value calculated from the load address of the preceding text information record (see Section C.3), yields the virtual address in the image that is to be modified. The command byte of each entry has the following bit assignments.

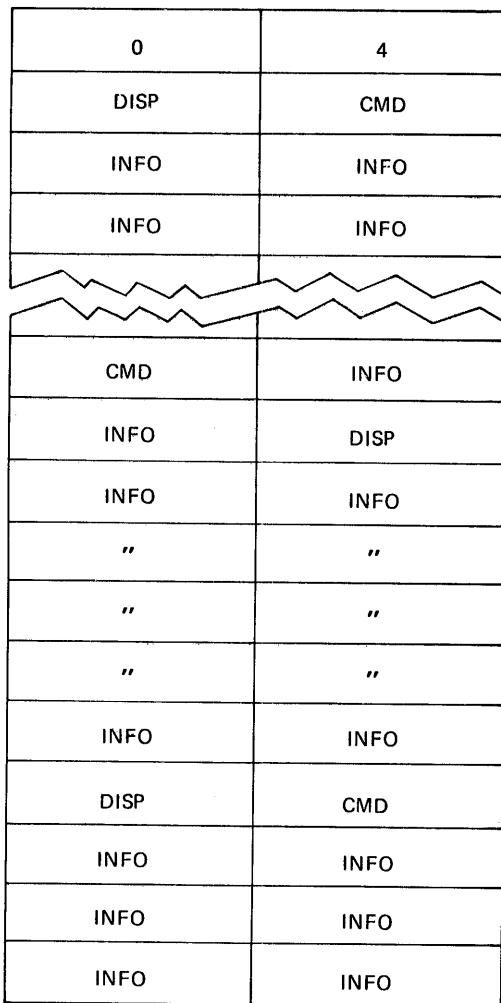
Bits 0 - 6 Specify the type of entry. Potentially, 128 command types can be specified although only 15 (decimal) are implemented.

Bit 7 - Modification

0 = The command modifies an entire word.

1 = The command modifies only one byte. The Task Builder checks for truncation errors in byte modification commands. If truncation is detected, that is, if the modification value has a magnitude greater than 255, an error occurs.

## TASK BUILDER DATA FORMATS



**Figure C-12 Relocation Directory Record Format**

### C.4.1 Internal Relocation

The internal relocation entry illustrated in Figure C-13 relocates a direct pointer to an address within a module. The current PSECT base address is added to a specified constant, and the result is written into the task image file at the calculated address. (That is, a displacement byte is added to the value calculated from the load address of the preceding text block.)

**Example:**

```
A:      MOV      #A,R0
        or
        .WORD    A
```

## TASK BUILDER DATA FORMATS

DISP	B	1
CONSTANT		

Figure C-13 Internal Relocation Entry Format

### C.4.2 Global Relocation

The global relocation entry in Figure C-14 relocates a direct pointer to a global symbol. The definition of the global symbol is obtained and the result is written into the task image file at the calculated address.

Example:

```
MOV      #GLOBAL,R0  
or  
.WORD   GLOBAL
```

DISP	B	2
SYMBOL NAME		

Figure C-14 Global Relocation Entry Format

### C.4.3 Internal Displaced Relocation

The internal displaced relocation entry in Figure C-15 relocates a relative reference to an absolute address from within a relocatable control section. The address plus 2 that the relocated value is to be written into is subtracted from the specified constant. The result is then written into the task image file at the calculated address.

Example:

```
CLR      177550  
or  
MOV      177550,R0
```

DISP	B	3
CONSTANT		

Figure C-15 Internal Displaced Relocation Entry Format

## TASK BUILDER DATA FORMATS

### C.4.4 Global Displaced Relocation

The global displaced relocation entry in Figure C-16 relocates a relative reference to a global symbol. The definition of the global symbol is obtained, and the address plus 2 that the relocated value is to be written into is subtracted from the definition value. The result is then written into the task image file at the calculated address.

Example:

CLR GLOBAL

or

MOV GLOBAL,R0

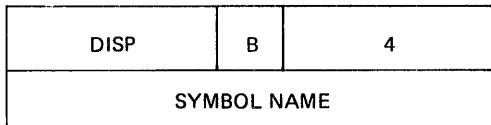


Figure C-16 Global Displaced Relocation Entry Format

### C.4.5 Global Additive Relocation

The global additive relocation entry in Figure C-17 relocates a direct pointer to a global symbol with an additive constant. The definition of the global symbol is obtained, the specified constant is added, and the resultant value is then written into the task image file at the calculated address.

Example:

MOV #GLOBAL+2,R0

or

.WORD GLOBAL-4

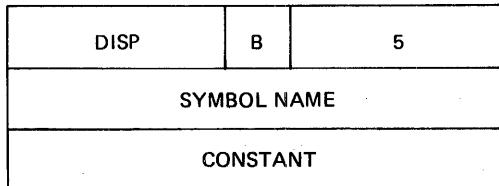


Figure C-17 Global Additive Relocation Entry Format

### C.4.6 Global Additive Displaced Relocation

The global additive displaced relocation entry in Figure C-18 relocates a relative reference to a global symbol with an additive constant. The definition of the global symbol is obtained, and the

## TASK BUILDER DATA FORMATS

specified constant is added to the definition value. The address plus 2 that the relocated value is to be written into is subtracted from the resultant additive value. The result is then written into the task image file at the calculated address.

Example:

CLR GLOBAL+2

or

MOV GLOBAL-5,R0

DISP	B	6
SYMBOL NAME		
CONSTANT		

Figure C-18 Global Additive Displaced Relocation Entry Format

### C.4.7 Location Counter Definition

The location counter definition in Figure C-19 declares a current PSECT and location counter value. The control base is stored as the current control section, and the current control section base is added to the specified constant and stored as the current location counter value.

0	B	7
PSECT NAME		
CONSTANT		

Figure C-19 Location Counter Definition

### C.4.8 Location Counter Modification

The location counter modification entry in Figure C-20 modifies the current location counter. The current PSECT base is added to the specified constant and the result is stored as the current location counter.

Example:

.=.+N

or

.BLKB N

## TASK BUILDER DATA FORMATS

0	B	10
CONSTANT		

Figure C-20 Location Counter Modification

### C.4.9 Program Limits

The program limits entry in Figure C-21 is generated by the .LIMIT assembler directive. The first address above the header (normally the beginning of the stack) and highest address allocated to the task, are obtained and written into the task image file at the calculated address, and at the calculated address plus 2 respectively.

Example:

```
.LIMIT
```

DISP	B	11
------	---	----

Figure C-21 Program Limits Entry Format

### C.4.10 PSECT Relocation

The PSECT relocation entry in Figure C-22 relocates a direct pointer to the beginning address of another PSECT (other than the PSECT in which the reference is made) within a module. The current base address of the specified PSECT is obtained and written into the task image file at the calculated address.

Example:

```
.PSECT A
B:
.
.
.
.PSECT C
MOV #B,R0
or
.WORD B
```

## TASK BUILDER DATA FORMATS

DISP	B	12
PSECT NAME		

Figure C-22 PSECT Relocation Entry Format

### C.4.11 PSECT Displaced Relocation

The PSECT displaced relocation entry in Figure C-23 relocates a relative reference to the beginning address of another PSECT within a module. The current base address of the specified PSECT is obtained and the address plus 2 that the relocated value is to be written into is subtracted from the base value. The result is then written into the task image file at the calculated address.

Example:

```
.PSECT A
B:
.
.
.
.PSECT C
MOV B, R0
```

DISP	B	14
PSECT NAME		

Figure C-23 PSECT Displaced Relocation Entry Format

### C.4.12 PSECT Additive Relocation

The PSECT additive relocation entry in Figure C-24 relocates a direct pointer to an address in another PSECT within a module. The current base address of the specified PSECT is obtained and added to the specified constant. The result is written into the task image file at the calculated address.

Example:

```
.PSECT A
B:
.
.
.
C:
.
.
.
.PSECT D
```

## TASK BUILDER DATA FORMATS

```
MOV      #B+10,R0  
MOV      #C,R0
```

or

```
.WORD    B+10  
.WORD    C
```

DISP	B	15
PSECT NAME		
CONSTANT		

Figure C-24 PSECT Additive Relocation Entry Format

### C.4.13 PSECT Additive Displaced Relocation

The PSECT additive displaced relocation entry in Figure C-25 relocates a relative reference to an address in another PSECT within a module. The current base address of the specified PSECT is obtained and added to the specified constant. The address plus 2 that the relocated value is to be written into is subtracted from the resultant additive value. The result is then written into the task image file at the calculated address.

Example:

```
.PSECT  A  
B:  
.  
.  
.  
C:  
.PSECT  D  
MOV      B+10,R0  
MOV      C,R0
```

DISP	B	16
PSECT NAME		
CONSTANT		

Figure C-25 PSECT Additive Displaced Relocation Entry Format

## TASK BUILDER DATA FORMATS

### C.4.14 Complex Relocation

The complex relocation entry in Figure C-26 resolves a complex relocation expression. In such an expression any of the MACRO-11 binary or unary operations are permitted. Any type of argument is permitted, regardless of whether the argument is unresolved global, relocatable to any PSECT base, absolute, or a complex relocatable subexpression.

The RLD command word is followed by a string of numerically-specified operation codes and arguments. Each operation code occupies one byte. The entire RLD command must fit in a single record. The following operation codes are defined.

- 0 - No operation
- 1 - Addition (+)
- 2 - Subtraction (-)
- 3 - Multiplication (*)
- 4 - Division (/)
- 5 - Logical AND (&)
- 6 - Logical inclusive OR (!)
- 10 - Negation (-)
- 11 - Complement (^C)
- 12 - Store result (command termination)
- 13 - Store result with displaced relocation (command termination)
- 16 - Fetch global symbol. It is followed by four bytes containing the symbol name in Radix-50 representation.
- 17 - Fetch relocatable value. It is followed by one byte containing the sector number, and two bytes containing the offset within the sector.
- 20 - Fetch constant. It is followed by two bytes containing the constant.
- 21 - Fetch resident library base address. If the file is a resident library STB file, the library base address is obtained; otherwise, the base address of the Task Image is fetched.

The STORE commands indicate that the value is to be written into the task image file at the calculated address.

All operands are evaluated as 16-bit signed quantities using two's complement arithmetic. The results are equivalent to expressions that are evaluated internally by the assembler. The following rules should be noted.

1. An attempt to divide by zero yields a zero result. The Task Builder issues a nonfatal diagnostic.

## TASK BUILDER DATA FORMATS

2. All results are truncated from the left in order to fit into 16 bits. No diagnostic is issued if the number was too large. If the result modifies a byte, the Task Builder checks for truncation errors as described in Section C.4.
3. All operations are performed on relocated (additive) or absolute 16-bit quantities. PC displacement is applied to the result only.

For example:

```
.PSECT ALPHA
A:
.
.
.
.PSECT BETA
B:
.
.
.
MOV      #A+B-<G1/G2&^C<177120!G3>>,R1
```

DISP	B	17
COMPLEX STRING		
12		

Figure C-26 Complex Relocation Entry Format

### C.4.15 Additive Relocation

The shared run-time system additive relocation entry in Figure C-27 relocates a direct pointer to an address within a SRTS.

If the current file is a symbol table file (STB), the base address of the SRTS is obtained and added to the specified constant. The result is written into the task image file at the calculated address. If the file is not associated with a SRTS, the task base address is used.

DISP	B	20
CONSTANT		

Figure C-27 Additive Relocation Entry Format

## TASK BUILDER DATA FORMATS

### C.5 INTERNAL SYMBOL DIRECTORY

Internal symbol directory records, as in Figure C-28, declare definitions of symbols that are local to a module. This feature is not supported by the Task Builder and therefore a detailed record format is not specified. If the Task Builder encounters this type of record, it ignores it.

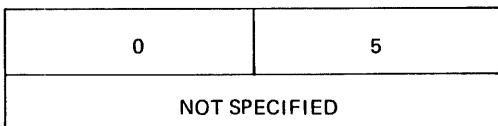


Figure C-28 Internal Symbol Directory Record Format

### C.6 END OF MODULE

The end-of-module record in Figure G-29 declares the end of an object module. Exactly one end-of-module record must appear in each object module. It is one word in length.

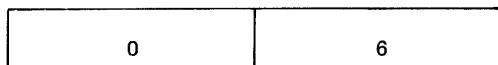


Figure C-29 End-of-Module Record Format

APPENDIX D  
TASK IMAGE FILE STRUCTURE

The task image as it is recorded on the disk appears in Figure D-1.

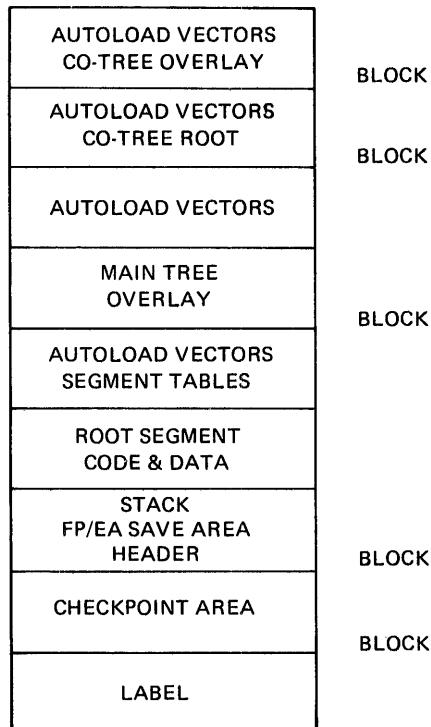


Figure D-1 Task Image on Disk

D.1 LABEL BLOCK GROUP

The label block group shown in Figure D-2 precedes the task on the disk, and contains data that need not be resident during task execution. This group is composed of two elements:

- task and resident library data (Label Block 0)
- table of LUN assignments (Label Block 1)

The task and resident library data elements are described in the paragraphs following Figure D-3.

The table of LUN assignments contains the name and logical unit number of each device assigned.

## TASK IMAGE FILE STRUCTURE

Label	Offset	
L\$BTSK	0	Task
	2	Name
L\$BPAR	4	Task
	6	Partition
L\$BSA	10	Base Address of Task
L\$BHGV	12	Highest Window 0 Virtual Address
L\$BMXV	14	Highest Virtual Address in Task
L\$BLDZ	16	Load Size in 64-Byte Blocks
L\$BMXZ	20	Max Size in 64-Byte Blocks
L\$BOFF	22	Task Offset Into Partition
L\$BWND	24	Number of Task Windows (Less Libraries)
L\$BSEG	26	Size of Overlay Segment Descriptors
L\$BFLG	30	Task Flag Word
L\$BDAT	32	Task Creation Date - Year
	34	- Month
	36	- Day
L\$BLIB	40	Library/Common
	42	Name
	44	Base Address of Library
	46	Highest Address in First Library Window
	50	Highest Address in Library
	52	Library Load Size (64-Byte Blocks)
	54	Library Max Size (64-Byte Blocks)
	56	Library Offset into Region
	60	Number of Library Window Blocks
	62	Size of Library Segment Descriptors
	64	Library Flags Word
	66	Library Creation Date - Year
	70	- Month
	72	- Day
	⋮	
	344	0
L\$BPRI	346	Task Priority
L\$BXFR	350	Task Transfer Address
L\$BEXT	352	Task Extension 64-Byte Blocks
L\$BSGL	354	Block Number of Segment Load List
L\$BHRB	356	Block Number of Header
L\$BBLK	360	Number of Blocks in Label
L\$BLUN	362	Number of Logical Units
	⋮	
	364	0

Table of LUN assignments:

<b>LUN BLOCK 1</b>	Device Name	<b>LUN 1</b>  :  Device Name Unit Number : Device Name Unit Number
	Unit Number	
	:	
	Device Name	
	Unit Number	
	:	
	Device Name	
	Unit Number	
	:	
	Device Name	
	Unit Number	
	:	
	Device Name	
	Unit Number	

Segment load list:

Length of Root Segment
Length of First Overlay Segment
Length of Second Overlay Segment
:
0

**Figure D-2 Label Block Group**

## TASK IMAGE FILE STRUCTURE

Task and resident library data are described below.

L\$BTSK	Task name consisting of two words in Radix-50 format. This parameter is set by the TASK keyword.
L\$BPAR	Partition name consisting of two words in Radix-50 format. This parameter is set by the PAR keyword.
L\$BSA	Starting address of task. Marks the lowest task virtual address. This parameter is set by the PAR keyword.
L\$BHGV	Highest virtual address mapped by address window 0.
L\$BMXV	Highest task virtual address. This value is set to L\$BHGV.
L\$BLDZ	Task load size* in units of 64-byte blocks. This value represents the size of the root segment.
L\$BMXZ	Task maximum size in units of 64-byte blocks. This value represents the size of the root segment plus any additional physical memory needed to contain task overlays.
L\$BOFF	Task offset into partition in units of 64-byte blocks.
L\$BWND	Number of task windows (excluding SRTS).
L\$BSEG	Size of overlay segment descriptors (in bytes).
L\$BFLG	Task flags word. Contains flags meaningful to RSX-11M only.
L\$BDAT	Three words containing the task creation date as 2-digit integer values as follows:  Year (since 1900) Month of year Day of month
L\$BLIB	Resident library entries
L\$BPRI	Task priority set by the PRI keyword (ignored by RSTS/E).
L\$BXFR	Task transfer address. (Not used by RSTS/E).
L\$BEXT	Task extension size in units of 64-byte blocks. This parameter is set by the EXTTSK keyword.
L\$BSGL	Relative block number of segment load list. Set to zero if no list is allocated.
L\$BHRB	Relative block number of header.
L\$BBLK	Number of blocks in label block group.
L\$BLUN	Number of logical units.

The contents of the SRTS/common name block are described below. This block is constructed by referencing the disk image of the SRTS/common

## TASK IMAGE FILE STRUCTURE

block. The format is identical to words 3 through 16 of the label block.

R\$LNAM            Library/common name consisting of two words in Radix-50 format.

R\$LSA            Base virtual address of library or common.

R\$LHGV            Highest address mapped by first library window.

R\$LMXV            Highest virtual address in library or common.

R\$LLDZ            Library/common block load size in 64-byte blocks.

R\$LMXZ            Library maximum size in units of 64-byte blocks.

R\$LOFF            (Not used by RSTS/E.)

R\$LWND            Number of window blocks required by library.

R\$LSEG            Size of library overlay segment descriptors in bytes.

R\$LFLG            Library flags word. The following flags are defined:

Bit

15    LD\$ACC Access intent (1=RW, 0=RO)

3      LD\$REL PIC (Position-Independent Code) flag (1=PIC)

R\$LDAT            Three words containing the library/common block creation date in the following format:

WORD 0: Year since 1900

WORD 1: Month of year

WORD 2: Day of month

### D.2 HEADER

The task header starts on a block boundary and is immediately followed by the task image. The task is read into memory starting at the base of the root segment. Because the root segment is a set of contiguous disk blocks, it is loaded with a single disk access.

The header is divided into two parts: a fixed part as shown in Figure D-3, and a variable part as shown in Figure D-4.

The variable part of the header contains window blocks that describe the following:

- the task's virtual-to-physical mapping
- logical unit data
- task context

The task header is used by RSTS/E mainly for setting the initial conditions for the task. Only locations 46 through 56 have identical meanings as in RSX-11M.

## TASK IMAGE FILE STRUCTURE

H.CSP	0	Current Stack Pointer (R6)
H.HDLN	2	Header length
H.EFLM	4	Event flag mask
	6	Event flag address
H.CUIC	10	Current UIC
H.DUIC	12	Default UIC
H.IPS	14	Initial PS
H.IPC	16	Initial PC (R7)
H.ISP	20	Initial Stack Pointer (R6)
H.ODVA	22	ODT SST vector address
H.ODVL	24	ODT SST vector length
H.TKVA	26	Task SST vector address
H.TKVL	30	Task SST vector length
H.PFVA	32	Power fail AST control block
H.FPVA	34	Floating Point AST control block
H.RCVA	36	Receive AST control block
H.EFSV	40	Address of event flag context
H.FPSA	42	Address of floating point context
H.WND	44	Pointer to number of window blocks
H.DSW	46	Directive Status Word
H.FCS	50	Address of FCS impure storage
H.FORT	52	Address of language impure storage
H.OVLY	54	Address of overlay impure storage
H.VEXT	56	Address of impure vectors
H.SPRI	60	Swapping priority
H.NML	61	Mailbox LUN
H.RRVA	62	Receive by reference AST control block
	64	Reserved
	66	Reserved
	70	Reserved
H.GARD	72	Header guard word pointer
H.NLUN	74	Number of LUNs

**Figure D-3 Task Header Fixed Part**

## TASK IMAGE FILE STRUCTURE

H.LUN	LUN Table (2 words/LUN)	
	•	
	•	
	•	
	Number of Window Blocks	Offsets
	Partition Control Block Address	W.BPCB
	Low Virtual Address Limit	W.BLVR
	High Virtual Address Limit	W.BHVR
	Address of Attachment Descriptor	W.BATT
	Window Size (in 32-word blocks)	W.BSIZ
	Offset into Partition (in 32-word blocks)	W.BOFF
	First PDR Address	W.BFPD
	Number of PDRs to Map	W.BNPD
	Contents of Last PDR	W.BLPD
	•	
	•	
	•	
	Current PS	
	Current PC	INITIAL VALUES
	Current R5	relative block number of header
	Current R4	indent word #2
	Current R3	indent word #1
	Current R2	task name word #2
	Current R1	task name word #1
	Current R0	program transfer address
	Header Guard Word	

**Figure D-4 Task Header Variable Part**

### NOTE

To save the identification, the initial value set by Task Builder should be moved to local storage. When the program is fixed in memory and being restarted without reloading, the

## TASK IMAGE FILE STRUCTURE

reserved program words must be tested for their initial values to determine whether the contents of R3 and R4 should be saved.

The contents of R0, R1, and R2 are only set when a debugging aid is present in the task image.

### D.2.1 Low Core Context

The low core context for a task consists of the Directive Status Word and the Impure Area vectors. The Task Builder recognizes the following global names:

```
.FSRPT File Control Services work area and buffer pool  
vector  
  
$OTSV Language OTS work area vector  
  
N.OVPT Overlay Runtime System work area vector  
  
$VEXT Vector extension area pointer
```

The only proper reference to these pointers is by symbolic name.

The Impure Area Pointers contain the addresses of storage used by the reentrant library routines described above.

The address contained in the vector extension pointer locates an area of memory that can contain additional impure area pointers.

The format of the vector extension area is shown in Figure D-5. Each location within this region contains the address of an impure storage area that is referenced by subroutines that must be reentrant. Addresses below \$VEXTA, referenced by negative offsets, are reserved for DIGITAL applications. Addresses above this symbol, referenced by positive offsets, are allocated for user applications.

```
.PSECTS $$VEX0 and $$VEX1 have the attributes D, GBL, RW, REL, and OVR.
```

The .PSECT attribute OVR, facilitates the definition of the offset to the vector, and the initialization of the vector location at link time as shown by the following example:

```
.GLOBL    $VEXTA      ; MAKE SURE VECTOR AREA IS LINKED  
  
.PSECT    $$VEX1,D,GBL,RO,REL,OVR  
  
BEG=.          ; POINT TO BASE OF POINTER TABLE  
  
.BLKW     N          ; OFFSET TO CORRECT LOCATION  
                  ; IN VECTOR AREA  
  
LABEL:      .WORD      IMPURE    ; SET IMPURE AREA ADDRESS  
                  ; DEFINE OFFSET  
  
OFFSET==LABEL-BEG
```

## TASK IMAGE FILE STRUCTURE

```
.PSECT  
IMPURE:  
;  
.  
.  
.
```

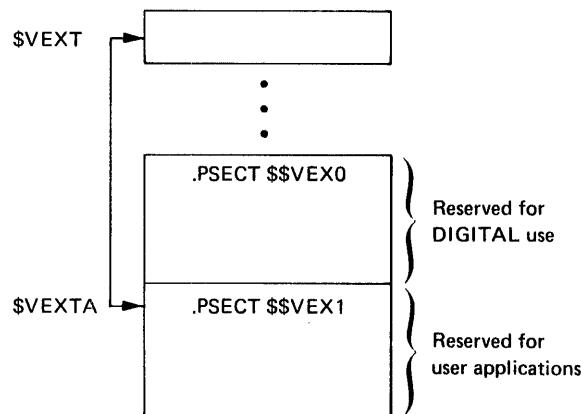


Figure D-5 Vector Extension Area Format

### D.3 OVERLAY DATA STRUCTURE

Figure D-6 illustrates the structure and principal components of the task-resident overlay data base.

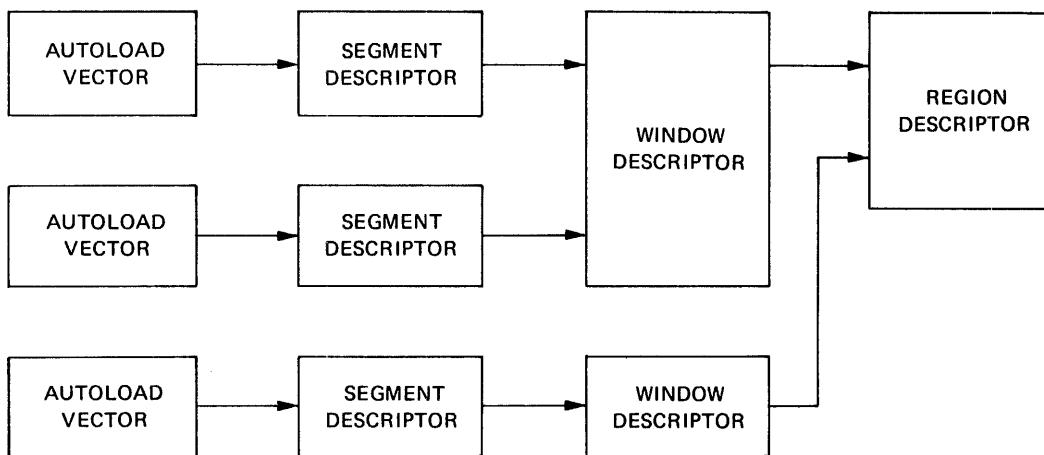


Figure D-6 Task-Resident Overlay Data Base

Autoload vectors are generated whenever a reference is made to an autoloadable entry point in a segment located farther away from the root than the referencing segment.

One segment descriptor is generated for each overlay segment in the task or shared region. The segment descriptor contains information on

## TASK IMAGE FILE STRUCTURE

the size, virtual address, and location of the segment within the task image file. In addition, it contains a set of link words that point to other segments. The overlay structure determines the link word contents.

The following sections describe the composition of each element.

### D.3.1 Autoload Vectors

The autoload vector table consists of one entry per autoload entry point in the form shown in Figure D-7.

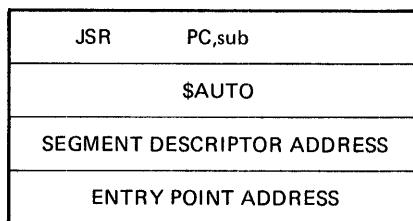


Figure D-7 Autoload Vector Entry

The autoload vector contains a JSR to the autoload processor, \$AUTO, followed by a pointer to the descriptor for the segment to be loaded, and the real address of the entry point.

### D.3.2 Segment Descriptor

The segment descriptor is composed of a 6-word fixed length portion. Segment descriptor contents are shown in Figure D-8.

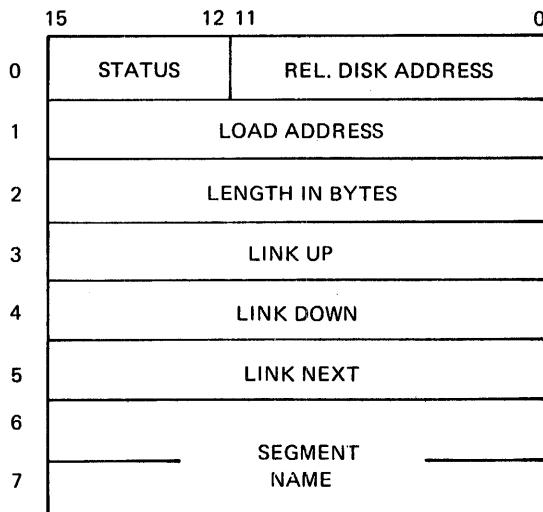


Figure D-8 Segment Descriptor

## TASK IMAGE FILE STRUCTURE

Word 0 contains the relative disk address in bits 0-11, and the segment status in bits 12-15. Each segment in the task image file begins on a disk block boundary. The relative disk address is the block number of the segment relative to the start of the root segment.

The segment flags are defined as follows:

Bit 15	Always set to 1
Bit 14	0 = Segment loaded and mapped 1 = Segment is either not loaded or not mapped
Bit 13	0 = Segment has disk allocation 1 = Segment does not have disk allocation
Bit 12	0 = Segment not loaded from disk 1 = Segment loaded from disk

Word 1 contains the load address of the segment. This address is the first virtual address of the area where the segment will be loaded.

Word 2 specifies the length of the segment in bytes.

The next three words point to the following segment descriptor:

Link Up	points to the next segment away from the root. Link Up equals 0 if you are already at the leaf.
Link Down	points to the next segment toward the root. Link Down equals 0 if you are already at the root.
Link Next	points to the adjoining segment. Link Next equals the address of the current segment if there are no others on the same level with the same Link Down. Link Next links all segments on the same level that have the same Link Down in a circular fashion. Thus, in Figure D-9, Link Next in A3 points to A1, but Link Next in A1 points to A1 itself and Link Next in A0 points to A0 itself.

When a segment is loaded, the overlay run-time system follows the links to determine which segments are being overlaid, and should therefore be marked out of memory.

Using the tree in Figure D-9 as an example:

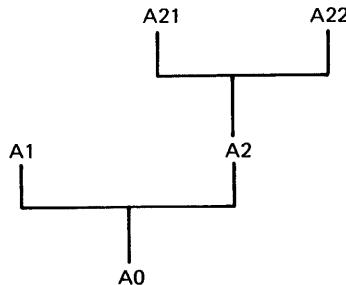


Figure D-9 Sample Tree

## TASK IMAGE FILE STRUCTURE

The segment descriptors are linked as in Figure D-10:

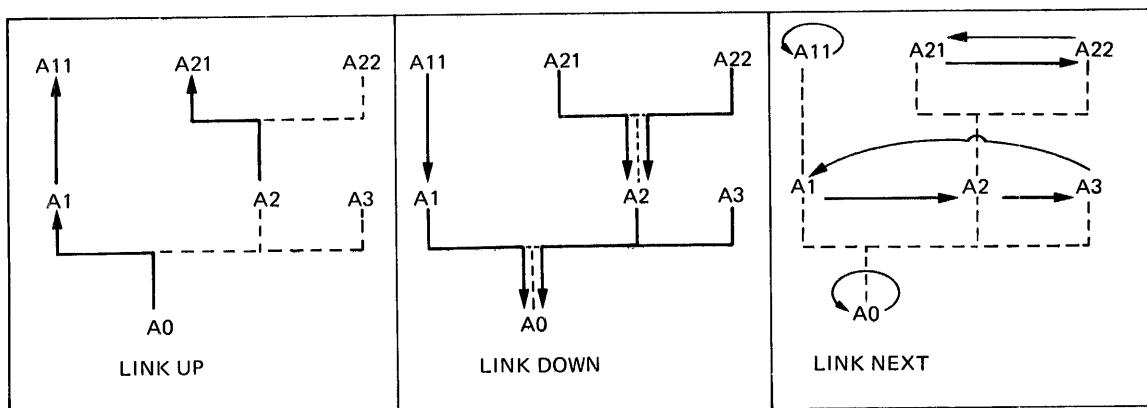


Figure D-10 Segment Linkage Directives

If there is a co-tree, the link next for the root segment descriptor points to the co-tree root segment descriptor.

Words 6 and 7 contain the segment name in Radix-50 format.

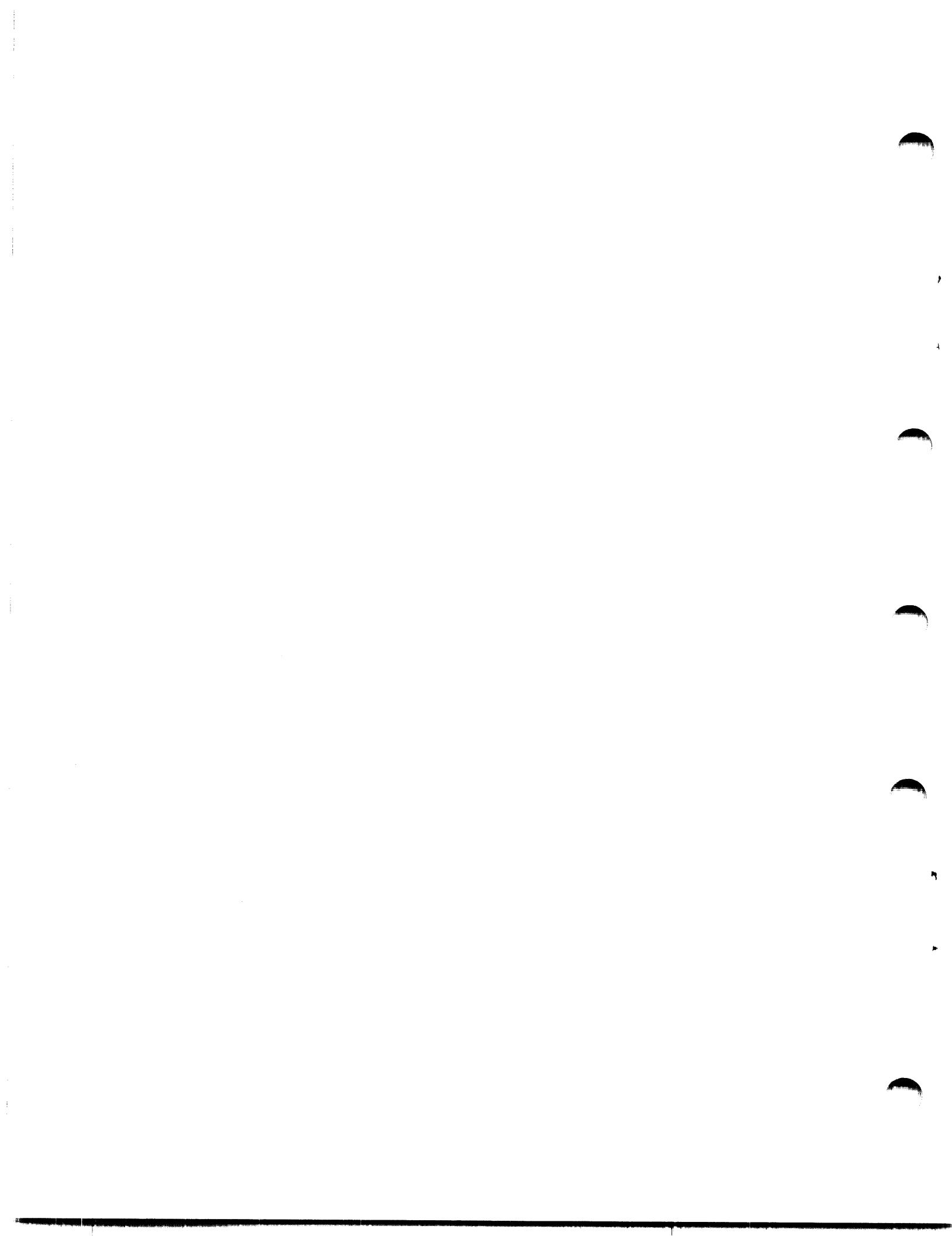
Word 8 points to the window descriptor used to map the segment (0 = none).

### D.4 ROOT SEGMENT

The root segment is written as a contiguous group of blocks. The root segment is the first segment loaded and remains in memory for the entire life of the program execution.

### D.5 OVERLAY SEGMENTS

Each overlay segment begins on a block boundary. The relative block number for the segment is placed in the segment table. Note that a given overlay segment occupies as many contiguous disk blocks as it needs to supply its space request. The maximum size for any segment, including the root, is 28K words.



APPENDIX E  
RESERVED SYMBOLS

All symbols and PSECT¹ names containing a . or \$ are reserved for DIGITAL-supplied software. Several global symbols and PSECT* names are reserved for use by the Task Builder. Special handling occurs when a definition of one of these names is encountered in a task image.

The definition of a reserved global symbol in the root segment causes a word in the task image to be modified with a value calculated by the Task Builder. The relocated value of the symbol is taken as the modification address.

Table E-1 shows global symbols reserved by the Task Builder:

Table E-1  
Task Builder Reserved Global Symbols

Global Symbol	Modification Value
.FSRPT	Address of File Storage Region work area (.FSRCB)
.MOLUN	Error message output device
.NLUNS	The number of logical units used by the task, not including the Message Output and Overlay units
.NOVLY	The overlay logical unit number
N.OVPT	The address of the Overlay Run-time System work area (.NOVLY)
.NSTBL	The address of the segment description tables. Note that this location is modified only when the number of segments is greater than one.
.ODTLL	Logical unit number for the ODT terminal device TI:

(continued on next page)

¹ PSECTS are created by .ASECT, .CSECT, or .PSECT directives. The .PSECT directive eliminates the need for either the .ASECT or .CSECT directive, which are retained only for compatibility with other systems. In this document all sections are referred to as PSECTS unless the specific characteristics of .ASECT or .CSECT apply.

## RESERVED SYMBOLS

Table E-1 (Cont.)  
Task Builder Reserved Global Symbols

Global Symbol	Modification Value
.ODTL2	Logical unit number for the ODT line printer device CL:
\$OTSVA	Address of Object Time System work area (\$OTSVA)
.TRLUN	The trace subroutine output logical unit number
\$VEXT	Address of vector extension area (\$VEXTA)

The PSECT names in Table E-2 are reserved by the Task Builder. In some cases, the definition of a reserved PSECT causes the PSECT to be extended if the appropriate option input is specified (see Section 3.2.3.2).

Table E-2  
PSECT Names Reserved by the Task Builder

PSECT Name	Description
\$\$ALVC	Contains segment autoload vectors
\$\$DEVT	The extension length (in bytes) is calculated from the formula  $\text{EXT} = \langle S.FDB+52 \rangle * \text{UNITS}$  Where the definition of S.FDB is obtained from the root segment symbol table and UNITS is the number of logical units used by the task, excluding the Message Output, Overlay, and ODT units.
\$\$RGDS	Contains region descriptors for resident libraries referenced by the task
\$\$RTS	Contains return instruction
\$\$SGD0	PSECT adjoining task segment descriptors
\$\$SGD1	Contains task segment descriptors
\$\$SGD2	PSECT following task segment descriptors
\$\$WNDS	Contains task window descriptors

## APPENDIX F

### IMPROVING TASK BUILDER PERFORMANCE

This appendix contains procedures and suggestions to help you maximize Task Builder performance. Procedures are given for:

- Evaluating and improving Task Builder throughput
- Modifying command switch defaults to provide a more efficient user interface

#### F.1 EVALUATING AND IMPROVING TASK BUILDER PERFORMANCE

Task Builder throughput is determined by these factors:

1. The amount of memory available for table storage
2. The amount of disk latency due to input file processing

The discussion in the following paragraphs outlines methods for improving throughput in each case. The methods approach their goals through judicious use of system resources and Task Builder features.

##### F.1.1 The Task Builder Work File

The largest factor affecting Task Builder performance is the amount of memory available for table storage. To reduce memory requirements, the Task Builder uses a work file to store symbol definitions and other tables. If the total size of these tables is within the limits of available memory, the work file is kept in core and not shunted to a disk. If the tables exceed the amount of memory available, some information must be moved to the disk, which degrades performance.

Work file performance can be gauged by consulting the statistics portion of the Task Builder map. The following parameters are displayed:

Number of work file references:

Total number of times that work file data was referenced.

Work file reads:

Number of work file references that resulted in disk accesses to read work file data.

## IMPROVING TASK BUILDER PERFORMANCE

### NOTE

If work file reads and writes equal zero and the number of work file references is greater than zero, you can be sure that the work file remained in memory.

#### Work file writes:

Number of work file references that resulted in disk accesses to write work file data.

#### Size of Core Pool:

Amount of in-core table storage in words. This value is also expressed in units of 256-word pages (information is read from and written to disk in blocks of 256 words).

#### Size of Work File:

Amount of work file storage in words. If this value is less than the core pool size, the number of work file reads and writes is zero. That is, no work file pages are removed to the disk. This value is also expressed in pages (256-word blocks).

#### Elapsed Time:

Amount of time required to build the task image, and produce the map. This value excludes ODL processing, option processing, and the time required to produce the global cross-reference.

The overhead for accessing the work file can be reduced in one or more of the following ways:

- by increasing the amount of memory available for table storage
- by placing the work file on the fastest random access device
- by decreasing system overhead required to access the file
- by reducing the number of work file references

The Task Builder automatically increases its size up to the maximum job size, which may be as large as 28K words. See the RSTS/E System Manager's Guide for information on how to change the maximum job size.

The size of the work file can be reduced by:

- Linking your task to a core-resident run-time system containing commonly used routines (e.g., BASIC-PLUS-2 Object Time System) whenever possible
- Including common modules, such as components of an object time system, in the root segment of an overlaid task
- Using an object library or file of concatenated object modules if many modules are to be linked

In the last two cases, system overhead is also significantly reduced because fewer files must be opened to process the same number of modules.

## IMPROVING TASK BUILDER PERFORMANCE

The number of work file references can be reduced by eliminating unneeded output files and cross-reference processing, or by obtaining the short map. In addition, selected files such as the default system object module library, can usually be excluded from the map. In this case, a full map can be obtained at less frequent intervals and retained.

Try the following procedures to improve work file performance.

- Increase maximum task size by raising the system limit for dynamic task extension.
- Decrease work file size by using resident run-time systems, concatenated object files, and object libraries.
- Decrease work file size by moving common modules into the root segment of an overlaid task.
- Decrease the number of work file references by eliminating the map and global cross-reference, obtaining the short map, or excluding files from the map.

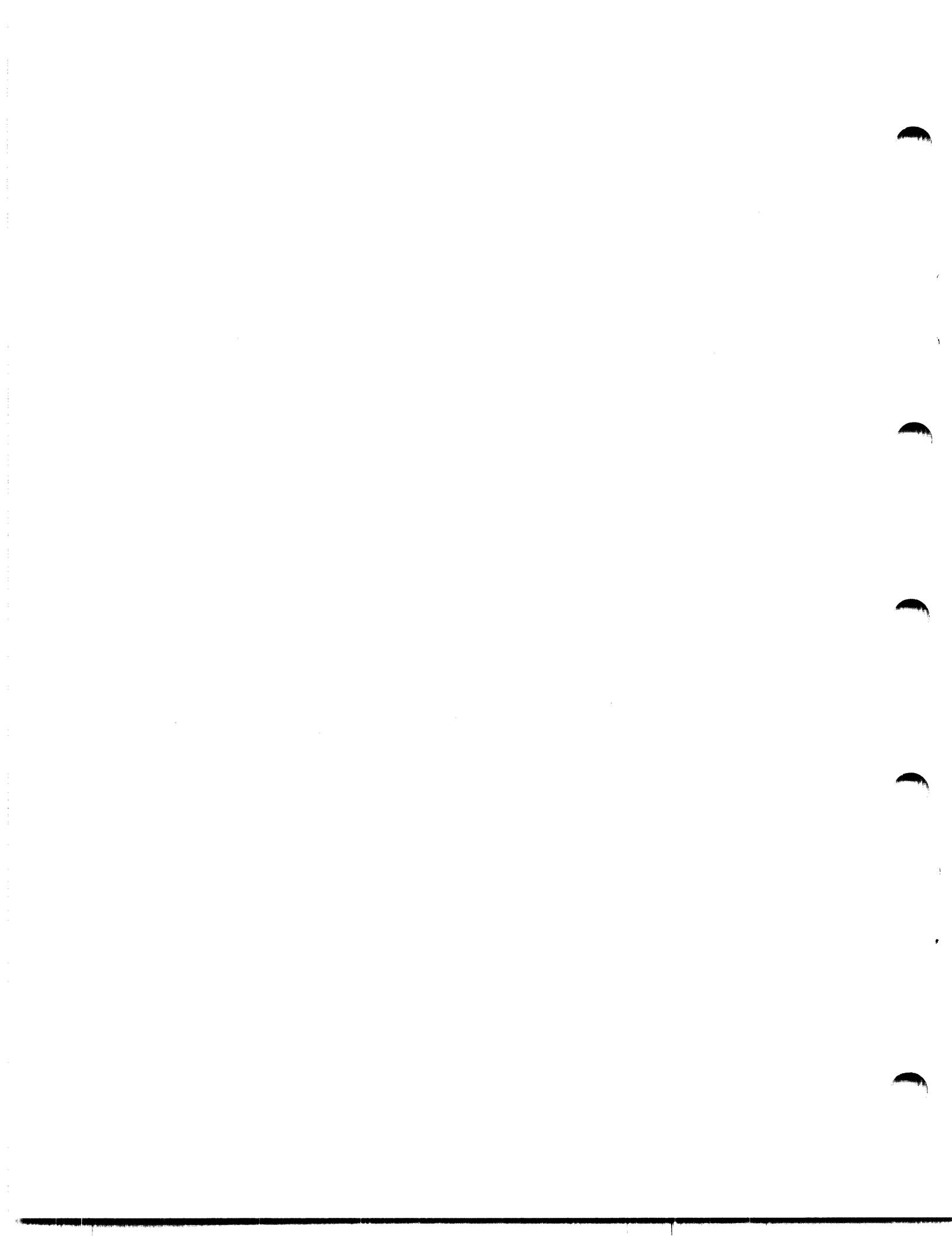
### F.1.2 Input File Processing

The suggestions for minimizing the size of the work file and number of work file accesses also drastically reduce the amount of input file processing.

A given module can be read up to three times when building the task:

- to build the symbol table
- to produce the task image
- to produce the long map

Files that are excluded from the long map are read only twice. The third pass is completely eliminated for all modules when a short map is requested. So, if you do not need the long map, use the /SH switch (described in Section 3.1.8) to eliminate the third pass.



## APPENDIX G

### INCLUDING A DEBUGGING AID

You can include a program that controls the execution of a task by naming the appropriate object module as an input file and applying the /DA switch.

When such a program is read, the Task Builder causes control to be passed to the program when the task starts.

Such control programs might trace a task, printing out relevant debugging information, or monitor the task's performance for analysis.

The switch has the following effects:

1. The transfer address in the debugging aid overrides the task transfer address.
2. On initial task load, the following registers have the indicated value:

R0 - Transfer address of task  
R1 - Task name in Radix-50 format (word #1)  
R2 - Task name (word #2)



## APPENDIX H

### GLOSSARY

AUTOLOAD	The method of loading overlay segments, in which the Overlay Run-Time System automatically loads overlay segments when they are needed and handles any unsuccessful load requests.
CO-TREE	An overlay tree whose segments, including the root segment, are made resident in memory through calls to the Overlay Run-Time System.
FRAGMENTED MEMORY	The state existing when portions of memory are non-contiguous.
GLOBAL COMMON BLOCK	An area of memory reserved for a resident library or common block.
GLOBAL SYMBOL	A symbol whose definition is known outside the defining module.
MAIN TREE	An overlay tree whose root segment is loaded by the Monitor when the task is made active.
MEMORY ALLOCATION FILE	The output file created by the Task Builder that describes the allocation of task memory.
MEMORY RESIDENT OVERLAY	An overlay segment that shares virtual addresses with other segments, but which resides in its own portion of memory. The segment is read from disk at the same time as all other segments in the resident library; thereafter, mapping directives are issued in place of disk load requests.
OVERLAY DESCRIPTION LANGUAGE (ODL)	A language that describes the overlay structure of a task.
OVERLAY RUN-TIME SYSTEM	A set of subroutines linked as part of an overlaid task that are called to load segments into memory.
OVERLAY SEGMENT	A segment that shares physical memory and/or virtual address space with other segments and is loaded when needed.
OVERLAY TREE	A tree structure consisting of a root segment and optionally one or more overlay segments.

## GLOSSARY

PARTITION	The area of memory where a task or resident library is located within the job's virtual address space.
PATH	A route that is traced from the root of the overlay tree to any one leaf in that tree.
PATH-LOADING	The technique used by the autoload method to load all segments on the path between a calling segment and a called segment.
POSITION INDEPENDENT CODE (PIC)	Code that can be loaded and run anywhere in memory without modification.
PSECT (P-SECTION)	A section of memory that is a unit of the total allocation. A source program is translated into object modules that consist of PSECTS with attributes describing access, allocation, relocatability, etc.
RESIDENT LIBRARY	A collection of reentrant, shareable routines or data that is accessible to user tasks.
ROOT SEGMENT	The segment of an overlay tree that, once loaded, remains in memory during the execution of the task.
RUNNABLE TASK	A task that can be executed.
SEGMENT	A group of modules and/or PSECTS that occupy memory simultaneously and that can be loaded by a single disk access.
SYMBOL DEFINITION FILE	The output file created by the Task Builder that contains the global symbol definitions and values in a format suitable for reprocessing by the Task Builder. Symbol definition files are used to link tasks to shared run-time systems.
TASK IMAGE FILE	The output file created by the Task Builder that contains the executable portion of the task.
VIRTUAL ADDRESS SPACE	The set of addresses ranging from 0 to 177777 octal that are contained in a 16-bit word and referenced directly by a user program.
VIRTUAL ADDRESS WINDOW	The amount of virtual address space that is allocated to a p-section or common block.

## INDEX

ABORT option, 3-9  
  use of, 3-16  
ABSPAT option, 3-14  
  used for patching, 5-14  
Access code, 4-3, 4-4  
Additive Relocation entry, C-19  
  Global, C-13  
  PSECT, C-16  
Address register, 1-2  
Allocation code, 4-3, 4-4  
Allocation option, 3-8, 3-10  
ALTER statement,  
  COBOL, 5-14  
ASECT, C-4  
  see also PSECT  
ASG option, 3-12  
  see also Option  
Asterisk as autoload  
  indicator, 6-1  
At character,  
  commercial, 2-6  
Attribute,  
  attached to segment, 5-2  
  task, section contents, 4-8  
\$AUTO,  
  autoload routine, 6-4  
Autoload indicator,  
  application, 6-2  
  asterisk as, 6-1  
  function, 6-3  
  in overlay tree, 5-13  
Autoload mechanism, 6-1  
Autoload routine (\$AUTO), 6-4  
Autoload vectors,  
  excess, 6-4  
  table, D-9

BASIC-PLUS-2,  
  compilation sequence, 2-13  
  example ODL file, 5-20  
Bit assignments, flag byte,  
  C-5  
Blank CSECT, C-4  
  see also PSECT  
Block,  
  common name, D-3  
  shared run-time system  
    (SRTS), D-3  
  task label, D-4  
Block diagrams, 5-4  
Branch in overlay tree, 5-4

Calls, overlay, 6-4  
\$CBLMRG creates overlay  
  description, 2-14  
/CC switch, 3-2  
  see also Switch  
Character, commercial at,  
  2-6, 5-15  
Characters, unique kernel, 5-19  
Co-tree,  
  defining a, 5-16  
  in multiple tree structure,  
    5-15  
  linkage, 5-12  
  root in .ROOT directive,  
    5-16  
  root segment descriptor,  
    D-11  
Co-tree definition,  
  comma operator in, 5-16  
COBOL,  
  ALTER statement, 5-14  
  compilation sequence, 2-13  
  example ODL file, 5-21  
  /KER:xx switch, 2-13  
  ODL file, 5-20  
  PSECT names, 2-13  
Code,  
  access, 4-3, 4-4  
  allocation, 4-3, 4-4  
  relocation, 4-3  
  scope, 4-3, 4-4  
  type, 4-4  
Comma operator, 5-9, 5-10  
  in co-tree definition, 5-16  
Command file, indirect,  
  contents, 2-7  
  primary, 2-7  
  secondary, 2-7  
Command line,  
  examples of, 2-3  
  format of, 2-2  
  input file in, 2-2  
  output file in, 2-2  
  task image file in, 2-2  
Command mode, 2-5  
Comments, 2-9  
Commercial at character, 5-15  
COMMON statement creates  
  PSECT, 4-3  
Common storage forced to  
  root, 5-20  
Compilation sequence,  
  BASIC-PLUS-2, 2-13  
  COBOL, 2-13

## INDEX (Cont.)

Complex Relocation entry, C-18  
Content-altering option, 3-8, 3-13  
Control option, 3-8, 3-9  
Control Section Name entry, C-4  
Conversion table, octal-decimal, B-11 to B-14  
Core pool size, F-2  
Cross-reference processing, F-3  
CSECT, blank, C-4  
    see also PSECT  
CSECT, named, C-4  
    see also PSECT

/DA switch, 3-3  
    see also Switch  
Data structure, overlay, D-8  
Debugging aid, including a, G-1  
Decimal to octal conversion, B-1 to B-10  
Defaults, filename extensions, 2-2  
    table of filename extensions, 2-2  
    Task Builder, 1-1  
Device-specifying option, 3-8, 3-12  
Diagnostic error messages, A-1 to A-7  
Diagrams, block, 5-4  
Directive, co-tree root in .ROOT, 5-16  
.END, in overlay description, 5-10  
.FCTR, 5-9, 5-11  
.NAME, 5-11, 5-12  
.PSECT, 5-14  
Directive Status Word, D-7  
Displaced Relocation entry, Global, C-13  
    Internal, C-12  
    PSECT, C-16  
/DL switch, 3-3  
    see also Switch

Elapsed time message, F-2  
.END directive in overlay description, 5-10

End of Module Record, C-20  
Error messages, diagnostic, A-1 to A-7  
    fatal, A-1 to A-7  
Extensions, filename  
    see Defaults  
EXTSCT option, 3-10  
    see also Option  
EXTTSK option, 3-10  
    see also Option  
    used to extend memory, 4-1

Fatal error messages, A-1 to A-7  
.FCTR directive, 5-9, 5-11  
File, BASIC-PLUS-2 ODL, 5-20  
COBOL ODL, 5-20, 5-21  
complexity, COBOL ODL, 5-20  
contents, indirect command, 2-7  
    memory allocation, 2-3, 4-7  
    symbol definition, 2-3  
    Task Builder map, 1-2  
    task image, 4-7  
indirect reference, 2-9  
indirect command, using an, 2-6  
input, in task command line, 2-2  
    processing, F-3  
    specifications, 2-5  
label block group in task image, 4-7  
output, omitting, 2-3  
    specifications, 2-5  
    unnecessary, F-3  
overhead for accessing work, F-2  
overlay description from indirect, 5-15  
primary indirect command, 2-7  
secondary indirect command, 2-7  
Task Builder work, F-1  
task image, 2-3, C-10  
work, F-1, F-2  
File extensions as default entries, 2-2  
File extensions, default, table, 2-2  
Fixed part, task header, D-4

## INDEX (Cont.)

Flag byte,  
    bit assignments, C-5  
Flow of control and path, 5-5  
Flow of control in segment,  
    5-1  
/FP switch, 3-3  
    see also Switch  
/FU switch, 3-4  
    see also Switch

GBLDEF option, 3-13  
    see also Option  
GBLPAT option, 3-15  
    see also Option  
    used for patching, 5-14  
GBLREF option, 3-14  
    see also Option  
Global Additive Relocation  
    entry, C-13  
Global Displaced Relocation  
    entry, C-13  
Global PSECT resolution, 5-8  
Global Relocation entry, C-12  
Global symbol,  
    ambiguously defined, 5-7  
    definition, 1-1  
    in a tree, 5-7  
    in multi-segment task, 5-6  
    in single-segment task, 5-6  
    multiply defined, 5-7  
    rules, 5-6  
    undefined, 5-7  
Global Symbol Directory  
    entries, C-2, C-3  
Global Symbol Name entry, C-5  
Global symbol reporting,  
    undefined, 5-8  
Global symbol resolution, 4-6,  
    4-7  
    in a task, 5-6  
    procedure, 5-6  
Group, label block,  
    contents, D-1  
    in task image file, 4-7

Header contents, 4-2  
    page, 4-8  
Header, task, D-4  
    fixed part, D-4  
    in task image memory, 4-2  
    variable part, D-4  
HISEG option, 3-11  
    see also Option  
Hyphen operator, 5-9, 5-10

Identification option, 3-8,  
    3-10  
Image file, task, C-10  
    in task command line, 2-2  
Impure Area Vectors, D-7  
Independence, logical, of  
    segment, 5-1  
Indicator, autoload,  
    application, 6-2  
    asterisk as, 6-1  
    function, 6-3  
    in overlay tree, 5-13  
Indirect command file,  
    contents, 2-7  
    primary, 2-7  
    secondary, 2-7  
    using an, 2-6  
Indirect file,  
    overlay description from,  
        5-15  
    reference, 2-9  
Input,  
    multi-line, 2-4  
Input file,  
    in task command line, 2-2  
    processing, F-3  
    specifications, 2-5  
Internal Displaced Relocation  
    entry, C-12  
Internal Relocation entry,  
    C-11  
Internal Symbol Directory  
    Record, C-20  
Internal Symbol Name entry,  
    C-4

/KER:xx,  
    see Kernel switch  
Kernel characters and object  
    module, 2-13  
Kernel characters, unique,  
    5-19  
Kernel switch, 2-13  
    use of, 5-19

Label block, D-4  
Label block group,  
    contents, D-1  
    in task image file, 4-7  
/LB switch, 3-4  
    see also Switch  
Leaf in overlay tree, 5-4  
Library,  
    default object module, 5-8

## INDEX (Cont.)

Library module, forced to root, 5-18  
Link down, D-11  
Link next, D-11  
Link up, D-11  
Linkage,  
    co-tree, 5-16  
    subroutine, 4-2  
Load address, C-10  
Loading, overlay, and tree properties, 5-5  
Local PSECT resolution, 5-8  
Location Counter Definition entry, C-14  
Location Counter Modification entry, C-14  
Logical independence of segment, 5-1  
Logical unit number, 3-12  
Low core context, D-7

/MA switch, 3-5  
    see also Switch  
Main tree in multiple tree structure, 5-15  
Map,  
    debugging information in, 4-10  
Map file contents,  
    Task Builder, 1-2  
MAP statement creates PSECT, 4-3  
Mechanism,  
    autoload, 6-1  
Memory,  
    EXTTSK option used to extend, 4-1  
    header in task image, 4-2  
    PSECT in task image, 4-2  
    stack in task image, 4-2  
    task image, 4-2  
Memory allocation file, 2-3, 4-7  
Memory allocation synopsis, 4-8  
Memory parts,  
    physical, 4-1  
Memory structure,  
    task, 4-1  
Messages, error, A-1 to A-7  
Mode,  
    command, 2-5  
    option, 2-5  
Modification entry,  
    Location Counter, C-14  
Module, object,  
    kernel characters and, 2-13  
    relationship between, 5-5

Module Name entry, C-3  
/MP switch, 3-5  
    see also Switch  
Multi-line input, 2-4  
Multi-segment task,  
    global symbol in, 5-6  
Multiple tree structure, 5-15  
    co-tree in, 5-15  
    main tree in, 5-15  
    parentheses in, 5-15  
Multiply defined global symbol, 5-7  
  
.NAME directive, 5-11, 5-12  
Named CSECT, C-4  
    see also PSECT  
Negating a switch, 3-1  
Null segment, 5-16

Object module, 1-1  
    generation, 1-1  
    kernel characters and, 2-13  
    library, default, 5-8  
Octal calculations, 5-3, B-1  
    to B-10  
Octal-decimal conversion table, B-11 to B-14  
ODL,  
    see Overlay Description Language  
ODL file,  
    BASIC-PLUS-2 example, 5-20  
    COBOL example, 5-20, 5-21  
    COBOL-generated, 5-20  
Operator,  
    comma, 5-9, 5-10  
    hyphen, 5-9, 5-10  
    parentheses, 5-10  
Operator, comma, in co-tree definition, 5-16  
Option,  
    ABORT, 3-9, 3-16  
    ABSPAT, 3-14  
    ASG, 3-12  
    EXTSCT, 3-10  
    EXTTSK, 3-10  
    GBLDEF, 3-13  
    GBLPAT, 3-15  
    HISEG, 3-11  
    STACK, 3-11  
    table, 3-9  
    TASK, 3-10  
    UNITS, 3-12  
Option mode, 2-5  
Option used for patching,  
    ABSPAT, 5-14  
    GBLPAT, 5-14

## INDEX (Cont.)

Option used to extend memory,  
EXTTSK, 4-1  
Options,  
    allocation, 3-8, 3-10  
    content-altering, 3-8, 3-10  
    control, 3-8, 3-9  
    device-specifying, 3-8, 3-12  
    extending memory, 4-1  
    format, 2-5  
    identification, 3-8, 3-10  
    patching, 5-14  
    storage altering, 3-13  
    storage sharing, 3-8, 3-11  
Output file,  
    in task command line, 2-2  
    omitting, 2-3  
    specifications, 2-5  
    unnecessary, F-3  
Overhead for accessing work  
    file, F-2  
Overlay calls, task TK1, 6-4  
Overlay data structure, D-8  
Overlay description, 5-1  
    \$CBLMRG creates, 2-14  
    comma within parentheses  
        in, 5-15  
    contents, 4-8  
    .END directive in, 5-10  
    from indirect file, 5-15  
Overlay Description Language,  
    5-1  
    summary, 5-42 to 5-44  
Overlay loading and tree  
    properties, 5-5  
Overlay segment descriptor,  
    D-11  
Overlay structure,  
    task suited to, 5-1  
Overlay structure and overlay  
    tree, 5-4  
Overlay tree, 5-4  
    autoload indicator in, 5-13  
    branch in, 5-4  
    leaf in, 5-4  
    overlay structure and, 5-4  
    USER, 5-19

Parentheses,  
    comma within, function, 5-15  
    in multiple tree structure,  
        5-15  
    operator, 5-10  
    treatment, 5-42  
Patching,  
    ABSPAT option used for, 5-14  
    GBLPAT option used for, 5-14  
Path down, 5-5

Path up, 5-5  
Path-loading, 6-3  
/PM switch, 3-6  
    see also Switch  
Primary indirect command file,  
    2-7  
Processing,  
    cross-reference, F-3  
    input file, F-3  
Program Limits entry, C-15  
Prompt, task builder, 2-1  
PSECT,  
    ASECT definition as, C-4  
    attributes, C-8  
        function of, 4-2  
        table, 4-3  
    blank CSECT definition as,  
        C-4  
    COMMON statement creates,  
        4-3  
    composition, 4-2  
    creation, 4-3  
    definition, 4-2  
        in task image memory, 4-2  
        length, maximum, C-6  
    MAP statement creates, 4-3  
    name, C-8  
    named CSECT definition as,  
        C-4  
    names and COBOL, 2-13  
    placement and /SQ switch,  
        4-6  
    referencing, 4-3  
    resolution,  
        global, 5-8  
        local, 5-8  
    segment number creates, 4-3  
PSECT Additive Relocation  
    entry, C-16  
.PSECT directive, 5-14  
PSECT Displaced Relocation  
    entry, C-16  
PSECT Name entry, C-6  
PSECT Relocation entry, C-15

Reads,  
    work file, F-1, F-2  
Rebuilding a task, 3-1, 3-15,  
    3-16  
Record,  
    End of Module, C-20  
    Internal Symbol Directory,  
        C-20  
    Relocation Directory, C-10  
    text, C-10  
Text Information, C-9

INDEX (Cont.)

References,  
    work file, F-1  
Relocation code, 4-3  
Relocation Directory entries,  
    C-10  
Relocation Directory Record,  
    C-10  
Relocation entry,  
    Additive, C-19  
    Complex, C-18  
    Global, C-12  
    Global Additive, C-13  
    Global Displaced, C-13  
    Internal, C-11  
    Internal Displaced, C-12  
    PSECT, C-15  
    PSECT Additive, C-16  
    PSECT Displaced, C-16  
Reserved symbols, E-1, E-2  
Resolution,  
    global PSECT, 5-8  
    global symbol, 4-6, 4-7  
        in a task, 5-6  
        procedure, 5-6  
    local PSECT, 5-8  
RMS-11 task,  
    /SQ switch and, 3-6  
Root,  
    common storage forced to,  
        5-20  
    library module forced to,  
        5-18  
.ROOT directive,  
    co-tree root in, 5-16  
Root segment contents, 5-18  
Root segment descriptor, D-11  
    co-tree, D-11  
Rules,  
    global symbol, 5-6  
Task Builder syntax, 2-14

Scope code, 4-3, 4-4  
Secondary indirect command  
    file, 2-7  
Section Name, Control, entry,  
    C-4  
Segment,  
    attribute attached to, 5-12  
    flow of control in, 5-1  
    logical independence of, 5-1  
    null, 5-16  
    root, 5-18  
Segment description, 4-8  
Segment descriptor, D-9 to D-11  
    co-tree root, D-11  
    overlay, D-11  
    root, D-11

Segment number creates PSECT,  
    4-3  
Segment status flags, D-10  
Segment table, 5-18  
Settings, switch,  
    recognized, 3-1  
/SH switch, 3-6  
    see also Switch  
Single-segment task,  
    global symbol in, 5-6  
Size,  
    core pool, F-2  
    decreasing stack, 3-11  
    task, 5-19  
    work file, F-2  
Slash,  
    as switch identifier, 2-17,  
        3-1  
    as terminator,  
        double, 2-5, 2-6  
        return, double, 2-7  
        single, 2-5, 2-6  
Source lines,  
    example program, 2-10  
Specifications,  
    input file, 2-5  
    output file, 2-5  
/SQ switch, 3-6  
    see also Switch  
and RMS-11 task, 3-6  
    PSECT placement and, 4-6  
SRTS/common name block, D-3  
/SS switch, 3-6  
    see also Switch  
Stack, 4-2  
    in task image memory, 4-2  
    size, decreasing, 3-11  
STACK option discussion, 3-11  
Storage,  
    common, forced to root,  
        5-20  
    temporary, 4-2  
Storage altering option, 3-13  
Storage sharing option, 3-8,  
    3-11  
Structure,  
    co-tree in multiple tree,  
        5-15  
    main tree in multiple tree,  
        5-15  
    multiple tree, 5-15  
    overlay data, D-8  
    parentheses in multiple tree,  
        5-15  
    task memory, 4-1  
    task suited to overlay, 5-1  
    task TK1 modified, 5-13  
Structure, overlay, and overlay  
    tree, 5-4

## INDEX (Cont.)

Subroutine linkage, 4-2  
Switch,  
  /CC, 3-2  
  conflicting, 3-7  
  /DA, 3-3  
  /DL, 3-3  
  /FP, 3-3  
  /FU, 3-4  
  /KER:xx, 2-13  
    use of, 5-19  
  /LB, 3-4  
  /MA, 3-5  
  /MP, 3-5  
  negating a, 3-1  
  /PM, 3-6  
  /SH, 3-6  
  /SQ, 3-6  
    and RMS-11 task, 3-6  
    PSECT placement and, 4-6  
  /SS, 3-6  
  table, 3-2  
  /WI, 3-7  
  /XT, 3-7  
Switch identifier,  
  slash as, 2-17, 3-1  
Switch settings,  
  recognized, 3-1  
Switch specification, 2-17  
Symbol, global, 1-1  
  ambiguously defined, 5-7  
  in a tree, 5-7  
  in multi-segment task, 5-6  
  in single-segment task, 5-6  
  multiply defined, 5-7  
  undefined, 5-7  
Symbol definition file, 2-3  
Symbol Directory entries,  
  Global, C-2, C-3  
Symbol Directory Record,  
  Internal, C-20  
Symbol Name entry,  
  Global, C-5  
  Internal, C-4  
Symbols, reserved, E-1, E-2  
Synopsis, memory allocation, 4-8  
Syntax rules,  
  Task Builder, 2-14

Task,  
  aborting a, 3-15  
  global symbol in multi-  
    segment, 5-6  
  global symbol in single-  
    segment, 5-6  
  global symbol resolution in  
    a, 5-6  
  memory structure, 4-1

Task (Cont.),  
  rebuilding a, 3-15, 3-16  
  size, 5-19  
  /SQ switch and RMS-11, 3-6  
  suited to overlay structures,  
    5-1

Task attribute section, 4-8  
Task Builder, 1-1, 2-4  
  defaults, basis for, 1-1  
  functions, 1-1  
  map file contents, 1-2  
  options, table, 3-9  
  performance, F-1  
  prompt, 2-1  
  switches, table, 3-2  
  syntax rules, 2-14  
  throughput, F-1  
  work file, F-1

Task command line, 2-2, 2-3  
  input file in, 2-2  
  output file in, 2-2  
  task image file in, 2-2

Task header, D-4  
  fixed part, D-4  
  variable part, D-4

Task image file, 2-3, C-10  
  contents, 4-7  
  in task command line, 2-2  
  label block group in, 4-7

Task image memory,  
  contents, 4-2  
  header in, 4-2  
  PSECT in, 4-2  
  stack in, 4-2

Task label block, D-4

Task memory structure, 4-1

TASK option, 3-10  
  see also Option

Task size, 5-19

Task suited to overlay  
  structure, 5-1

Task TK1,  
  overlay calls, 6-4

Task TK1 discussion, 5-2

Task TK1 modified structure, 5-13

Tasks,  
  building multiple, 2-6

Terminator,  
  double slash as, 2-5, 2-6, 2-7  
  single slash as, 2-5, 2-6

Text Information Record, C-9

Text record, C-10

Throughput,  
  Task Builder, F-1

Time, elapsed,  
  message, F-2

TK1, task, 5-2  
  modified structure, 5-13  
  overlay calls, 6-4

INDEX (Cont.)

Transfer Address entry, C-5  
Tree, main, in multiple  
    tree structure, 5-15  
Tree, overlay, 5-4  
    autoload indicator in, 5-13  
    branch in, 5-4  
    global symbol in, 5-7  
    leaf in, 5-4  
    overlay structure and, 5-4  
USER, 5-19  
Tree properties,  
    overlay loading and, 5-5  
Tree search, 5-8  
Tree structure,  
    co-tree in multiple, 5-15  
    main tree in multiple, 5-15  
    parentheses in multiple, 5-15  
Type code, 4-4  
  
Undefined global symbol, 5-7  
    reporting, 5-8  
UNITS option, 3-12  
    see also Option  
USER overlay tree, 5-19

Variable part,  
    task header, D-4  
Vector table,  
    autoload, D-9  
Vectors,  
    excess autoload, 6-4  
    Impure Area, D-7  
  
/WI switch, 3-7  
    see also Switch  
Work file,  
    overhead for accessing, F-2  
    reads, F-1, F-2  
    references, F-1  
    size, F-2  
Task Builder, F-1  
writes, F-2  
  
/XT switch, 3-7  
    see also Switch

READER'S COMMENTS

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

Did you find errors in this manual? If so, specify by page.

---

---

---

---

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

---

---

---

---

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

---

---

---

---

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

- Assembly language programmer
- Higher-level language programmer
- Occasional programmer (experienced)
- User with little programming experience
- Student programmer
- Non-programmer interested in computer concepts and capabilities

Name _____ Date _____

Organization _____

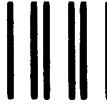
Street _____

City _____ State _____ Zip Code _____  
or  
Country _____

Please cut along this line.

— — — Do Not Tear - Fold Here and Tape — — —

digital



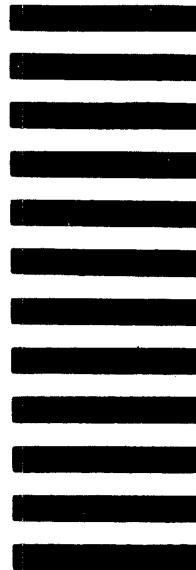
No Postage  
Necessary  
if Mailed in the  
United States

**BUSINESS REPLY MAIL**

FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

ATTN: Commercial Engineering Publications MK1-2/2H3  
DIGITAL EQUIPMENT CORPORATION  
CONTINENTAL BOULEVARD  
MERRIMACK N.H. 03054



— — — Do Not Tear - Fold Here and Tape — — —

Cut Along Dotted Line

AD-5072A-T1

READER'S COMMENTS

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

Did you find errors in this manual? If so, specify by page.

---

---

---

---

---

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

---

---

---

---

---

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

---

---

---

---

---

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

- Assembly language programmer
- Higher-level language programmer
- Occasional programmer (experienced)
- User with little programming experience
- Student programmer
- Non-programmer interested in computer concepts and capabilities

Name _____ Date _____

Organization _____

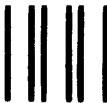
Street _____

City _____ State _____ Zip Code _____  
or  
Country _____

Please cut along this line.

-----Do Not Tear - Fold Here and Tape-----

**digital**



No Postage  
Necessary  
if Mailed in the  
United States

**BUSINESS REPLY MAIL**

FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

ATTN: Commercial Engineering Publications MK1-2/2H3  
DIGITAL EQUIPMENT CORPORATION  
CONTINENTAL BOULEVARD  
MERRIMACK N.H. 03054

-----Do Not Tear - Fold Here and Tape-----

Cut Along Dotted Line