RSX-11M
Task Builder Reference Manual

DEC-11-OMTBA-B-D

RSX-11M Version 2

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		

LIMITED RIGHTS LEGEND

CONTENTS

FIGURES

FIGURES (Cont.)

TABLES

PREFACE

0.1 MANUAL OBJECTIVES AND READER ASSUMPTIONS

This manual is a tutorial, intended to introduce the user to the basic
concepts and capabilities of the RSX-11M Task Builder.

Examples are used to introduce and describe features of the Task
Builder. These examples proceed from the simplest case to the most
complex. The reader may wish to try out some of the sequences to test
his understanding of the document.

The user should be familiar with the basic concepts of the RSX-11M
system described in the Introduction to RSX-11M and with basic
operating procedures described in the RSX-11M Operator Procedures
Manual (see Section 0.3 below).

0.2 STRUCTURE OF THE DOCUMENT

The manual has nine chapters. The first four chapters describe the
basic capabilities of the Task Builder. The last four chapters
describe the advanced capabilities. The Appendices list error
messages and give detailed descriptions of the structures used by the
Task Builder.

Chapter 1 outlines the capabilities of the Task Builder.

Chapter 2 describes the command sequences used to interact with the
Task Builder.

Chapter 3 lists the switches and options.

Chapter 4 describes memory allocation for the task and for the system
and gives examples of the memory allocation file.

Chapter 5 describes the overlay capability and the language used to
define an overlay structure.

Chapter 6 gives the two methods that can be used for loading overlay
segments.

Chapter 7 introduces shared regions, which can be used for
communication between tasks or to reduce the system's memory
requirements.

Chapter 8 describes the considerations for building a task on one
system to run on a system with a different hardware configuration.

Chapter 9 describes two types of memory dumps - Post-Mortem and
Snapshot.

A Glossary of terms is given in Appendix G.

0.3 ASSOCIATED DOCUMENTS

Other manuals closely allied to the purposes of this document are described briefly in the RSX-11M/RSX-11S Documentation Directory, Order No. DEC-11-OMUGA-B-D. The Documentation Directory defines the intended readership of each manual in the RSX-11M/RSX-11S set and provides a brief synopsis of each manual's contents.

CHAPTER 1

INTRODUCTION

This manual introduces the user to the Task Builder and defines the role of the Task Builder in the RSX-11M System.

The fundamental executable unit in the RSX-11M System is the task. A routine becomes an executable task image, as follows:

1. The routine is written in a supported source language.

2. The routine is entered as a text file, through the editor.

3. The routine is translated to an object module, using the appropriate language translator.

4. The object module is converted to a task image using the Task Builder.

5. Finally, the task is run.

If errors are found in the routine as a result of executing the task, the user edits the text file created in step 2 to correct the errors, and then repeats steps 3 through 5.

If a single routine is to be executed, the use of the Task Builder is appropriately simple. The user gives as input only the name of the file that contains the object module produced from the translation of the program and gives as output a name for the task image.

In typical applications, generally a collection of routines is run rather than a single program. In this case the user names each of the object module files and the Task Builder links the object modules, resolves any references to the system library, and produces a single task image, ready to be installed and executed.

The Task Builder makes a set of assumptions (defaults) about the task image based on typical usage and storage requirements. These assumptions can be changed by including switches and options in the task-building terminal sequence, thus directing the Task Builder to build a task which more closely represents the input/output and storage requirements of the task.

The Task Builder also produces, upon request, a memory allocation file that contains information describing the allocation of storage, the modules that comprise the task image, and the value of all global symbols. The user may optionally request that a list of global symbols, accompanied by the name of each referencing modules, be appended to the file (global cross-reference).

If a reduction in the amount of memory required by the task is necessary, the overlay capability can be used to divide the task into overlay segments. Overlaying a task allows it to operate in a smaller memory area and thus makes more space available to other tasks in the system.

If the task is configured as an overlay structure, (that is, a multi-segment task), the user becomes responsible for loading segments into memory as they are needed. There are two methods provided for loading overlay segments: autoload and manual.

The autoload method makes the loading of overlays transparent to the user. No special calls are required to load the overlay segments of the task. Loading of the overlay segments is accomplished automatically by the Overlay Runtime System according to the structure defined by the user at the time the task was built.

The manual load method requires that specific calls to the Overlay Runtime System be included in the coding of the task, and gives the user full control over the loading process.

If the task communicates with another task, or makes use of resident subroutines to save memory, the Task Builder allows the user to link to existing shared regions and to create new shared regions for future reference.

To move a task from one system to another with different memory management status, a special switch (/MM) is included in the Task Builder. The use of this switch allows tasks to be built on .one system and to run on another.

The user can become familiar with the capabilities of the Task Builder by degrees. Chapter 2 of this manual gives the basic information about Task Builder commands. This information is sufficient to handle many applications. The remaining chapters deal with special features and capabilities for handling advanced applications and tailoring the task image to suit the application. The appendices give detailed information about the structure of the input and output files processed by the Task Builder.

This manual describes the handling of an example application, CALC. In the first treatment of CALC, the user builds a task using all the default assumptions. Successive treatments illustrate the main points of each chapter in a realistic manner. Switches and options are added as they are required, an overlay structure is defined when the task increases in size, the loading of overlays is optimized, a shared region is added and finally the task is moved from a development system to a system which does not have memory management.

The memory allocation files for the various stages of task development are included. The effect of a change can be observed by examining the map for the previous example and the map for the example in which the change is made.

CHAPTER 2

COMMANDS

2.1 GENERAL COMMAND DISCUSSION

This chapter describes command sequences that can be used to build
tasks. Each command sequence is presented, by example, from the
simplest case to the most complex. All commands are then summarized
by a set of syntactic rules.

The first of a set of examples, designed to illustrate some of the
important features of the command language, concludes this chapter.
This example illustrates a simple task building sequence for a typical
application.

The convention of underlining system-generated text to distinguish it
from user type-in is used in this manual. For example:

         TKB>IMG1=IN1

The underline in the dialogue indicates that the system printed 'TKB>'
and the user typed 'IMG1=IN1'.

Consider again the creation and execution of a task. Suppose a user
has written a FORTRAN program. He enters the program through a text
editor as the file PROG.FTN. Then he types the following commands in
response to the Monitor Console Routine's request for input:

              >FOR CALC=PROG
              >TKB IMG=CALC
              >INS IMG
              >RUN IMG

The first command (FOR) causes the FORTRAN compiler to translate the
source language of the file PROG.FTN into a relocatable object module
in the file CALC.OBJ. The second command (TKB) causes the Task
Builder to process the file CALC.OBJ to produce the task image file
IMG.TSK. The third command (INS) causes Install to add the task to
the directory of executable tasks. Finally, the fourth command (RUN)
causes the task to execute.

CHAPTER 2. COMMANDS

The example just given includes the command

>TKB IMG=CALC

This command illustrates the simplest use of the Task Builder. It gives the name of a single file as output and the name of a single file as input. This chapter describes, first by example and then by syntactic definition, the complete facility for the specification of input and output files to the Task Builder.

2.1.1 Task Command Line

The task-command-line contains the output file specifications, followed by the input file specifications, separated by an equal sign. There can be up to three output files and any number of input files.

The output files must be given in a specific order: the first file named is the task image file, the second is the memory allocation file, and the third is the symbol definition file. The memory allocation file contains information about the size and location of components within the task. The symbol definition file contains the global symbol definitions in the task and their virtual or relocatable addresses in a format suitable for re-processing by the Task Builder. The Task Builder combines the input files to create a single executable task image.

Any of the output file specifications can be omitted. When all three output files are given, the task-command line has the form:

task-image, map, symbol-definition = input, ...

Consider the following commands and the ways in which the output filenames are interpreted.

Command	Output Files
>TKB IMG1,MP1,SF1=IN1	The task image file is IMG1.TSK, the memory allocation file is MP1.MAP, and the symbol definition file is SF1.STB.
>TKB IMG1=IN1	The task image file is IMG1.TSK.
>TKB ,MP1=IN1	The memory allocation file is MP1.MAP.
>TKB ,,SF1=IN1	The symbol definition file is SF1.STB.
>TKB IMG1,,SF1=IN1	The task image file is IMG1.TSK and the symbol definition file is SF1.STB.
>TKB =IN1	This is a diagnostic run with no output files.

CHAPTER 2. COMMANDS

2.1.2 Multiple Line Input

Although there can be a maximum of three output files, there can be
any number of input files. When several input files are used, a more
flexible format is sometimes necessary, one that consists of several
lines.   This multi-line format is also necessary for the inclusion of
options, as discussed in the next section.

If the user types 'TKB' alone, the Monitor Console Routine (MCR)
invokes the Task Builder.   The Task Builder then prompts for input
until it receives a line consisting of only the terminating sequence
"//".

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

This 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 subsequent lines.

The terminating symbol '//' directs the Task Builder to stop accepting
input, build the task, and return to the Monitor Console Routine
level.

2.1.3 Options

Options are used to specify the characteristics of the task being
built.   If the user types a single slash '/', the Task Builder
requests option information by displaying 'ENTER OPTIONS:'   and
prompting for input.

        >TKB
        TKB>IMG1,MP1=IN1
        TKB>IN2,IN3
        TKB>/
        ENTER OPTIONS:
        TKB>PRI=100
        TKB>COMMON=JRNAL:RO
        TKB>//

In this sequence the user entered the options PRI=100 and
COMMON=JRNAL:RO and then typed a double slash to end option input.
It also returned to MCR!!

CHAPTER 2. COMMANDS

The RSX-11M Task Builder provides 20 options. The syntax and
interpretation of each option are given in Chapter 3.

The general form of an option is a keyword followed by an equal sign
'=' followed by an argument list. The arguments in the list are
separated from one another by colons. In the example given, the first
option consists of the keyword 'PRI' and a single argument '100'
indicating that the task is to be assigned the priority 100. The
second option consists of the keyword 'COMMON' and an argument list
'JRNAL:RO', indicating that the task accesses a common region named
JRNAL and the access is read-only.

More than one option can be given on a line. The symbol exclamation
point '!' is used to separate options on a single line. For example:

    TKB>PRI=100 ! COMMON=JRNAL:RO

is equivalent to the two lines

    TKB>PRI=100
    TKB>COMMON=JRNAL:RO

Some options have argument lists that can be repeated. The symbol
comma ',' is used to separate the argument lists. For example:

    TKB>COMMON=JRNAL:RO,RFIL:RW

In this command, the first argument list indicates that the task has
requested read-only access to the shared region JRNAL. The second
argument list indicates that the task has requested read-write access
to the shared region RFIL.

The following three sequences are equivalent:

    TKB>COMMON=JRNAL:RO,RFIL:RW

    TKB>COMMON=JRNAL:RO ! COMMON=RFIL:RW

    TKB>COMMON=JRNAL:RO
    TKB>COMMON=RFIL:RW

2.1.4 Multiple Task Specification

If more than one task is to be built, the terminating symbol, '/'
(slash), can be used following option input to direct the Task Builder
to stop accepting input, build the task, and request information for
the next task build.

CHAPTER 2. COMMANDS

Consider the Sequence:

```
>TKB

TKB>IMG1=IN1
TKB>IN2,IN3
TKB>/
ENTER OPTIONS:
TKB>PRI=100
TKB>COMMON=JRNAL:RO
TKB>/
TKB>IMG2=SUB1
TKB>//
```

The Task Builder accepts the output and input file specifications and the option input, then stops accepting input when it encounters the '/' during option input. The Task Builder builds IMG1.TSK and returns to accept more input for building IMG2.TSK.

2.1.5 Indirect Command File Facility

The sequence of commands to the Task Builder can be entered directly or entered as a text file and later invoked through the indirect command file facility.

To use the indirect command file facility, the user first prepares a file that contains the user command input for the desired interaction with the Task Builder. He then invokes its contents by typing '@' followed by the file specification.

Suppose the text file AFIL is prepared, as follows:

```
IMG1,MP1=IN1
IN2,IN3
/
PRI=100
COMMON=JRNAL:RO
//
```

Later, the user can type:

```
>TKB   @AFIL
```

When the Task Builder encounters the symbol '@', it directs its search for commands to the file specified following the '@' symbol. When the Task Builder is accepting input from an indirect file, it does not display prompting messages on the terminal. The 1-line command to take commands from the indirect file AFIL is equivalent to the keyboard sequence:

```
>TKB
TKB>IMG1,MP1=IN1
TKB>IN2,IN3
TKB>/
ENTER OPTIONS:
TKB>PRI=100
TKB>COMMON=JRNAL:RO
TKB>//
```

CHAPTER 2. COMMANDS

When the Task Builder encounters a double-slash in the indirect file, it terminates indirect file processing, builds the task, and exits to the monitor upon completion.

However, if the Task Builder encounters an end-of-file in the indirect file before a double slash, it returns its search for commands to the terminal and prompts for input.

The Task Builder permits two levels of indirection in file references. The indirect file referenced in a terminal sequence can contain a reference to another indirect file.

Suppose the file BFIL.CMD contains all the standard options that are used by a particular group at an installation. That is every programmer in the group uses the options in BFIL.CMD. To include these standard options in his task building file, the user modifies AFIL to include an indirect file reference to BFIL.CMD as a separate line in the option sequence.

The contents of AFIL.CMD then are:

        IMG1,MP1=IN1
        IN2,IN3
        /
        PRI=100
        COMMON=JRNAL:RO
        @BFIL
        //

Suppose the contents of BFIL.CMD are:

        STACK=100
        UNITS=5 ! ASG=DT1:5

The terminal equivalent of the command

        >TKB @AFIL

then is:

        >TKB
        TKB>IMG1,MP1=IN1
        TKB>IN2,IN3
        TKB>/
        ENTER OPTIONS:
        TKB>PRI=100
        TKB>COMMON=JRNAL:RO
        TKB>STACK=100
        TKB>UNITS=5 ! ASG=DT1:5
        TKB>//

The indirect file reference must appear as a separate line. For example, if AFIL.CMD were modified by adding the '@BFIL' reference on the same line as the 'COMMON=JRNAL:RO' option, the substitution would not take place and an error would be reported.

CHAPTER 2. COMMANDS

2.1.6 Comments

Comment lines can be included at any point in the sequence. A comment line begins with a semicolon ';' and is terminated by a carriage return. All text on such a line is a comment. Comments can be included in option lines. In this case, the text between the semicolon and the carriage return is a comment.

Consider the annotation of the file just described; the user adds comments to provide more information about the purpose and the status of the task he is working on. Specifically, he adds some identifying lines, notes the function of his input files and shared region, and concludes with a comment on the current status of the task. The contents of the file are as follows:

```
;
; TASK 33A
;
; DATA FROM GROUP E-46 WEEKLY
;
IMG1,MP1=
;
;                      PROCESSING ROUTINES
;
                       IN1
;
;                      STATISTICAL TABLES
;
                       IN2
;
;                      ADDITIONAL CONTROLS
;
                       IN3
/
PRI=100
;
COMMON=JRNAL:RO ; RATE TABLES
;
; TASK STILL IN DEVELOPMENT
;
//
```

2.1.7 File Specification

Thus far the interaction with the Task Builder has been illustrated in terms of filenames. The Task Builder adheres to the standard RSX-11M conventions for file-specification. For any file, the user can specify the device, the user identification code, the filename, the type, the version number, and any number of switches.

Thus, the file specification has the form:

```
device:[group,member]filename.type;version/sw...
```

CHAPTER 2. COMMANDS

Consider, once again, the commands:

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

When the files are specified by name only, the default assumptions for
device, group, member, type, version and switch settings are applied.
For example, if the default user identification code is [200,200], the
task image file specification of the example is assumed to be:

```
SY0:[200,200]IMG1.TSK;1
```

That is, the task image file is produced on the system device (SY0)
under user identification code [200,200]. The default type for a task
image file is TSK and since the name IMG1.TSK is new, the version
number is 1. The default settings for all the task image switches
also apply. Switch defaults are described in full in Chapter 3.

Consider the following commands:

```
>TKB
TKB>[20,23]IMG1/CP/DA,LP:/CR=IN1
TKB>IN2;3,IN3
TKB>//
```

This sequence of commands produces the task image file IMG1.TSK;1
(actually, it produces IMG1.TSK with a version 1 higher than the
current version of IMG1.TSK) under user identification code [20,23]
on the system device. The task image is checkpointable and contains
the standard debugging aid. The memory allocation map, including a
global cross-reference listing, is produced on the line printer. The
task is built from the latest versions of IN1.OBJ and IN3.OBJ and a
specific version, number 3, of IN2.OBJ. The input files are all found
on the system device.

For some files, a device specification is sufficient. In the above
example, the memory allocation file is fully specified by the device
LP. The memory allocation file is produced on the line printer, but
is not retained as a file.

In this example, switches CP, CR, and DA are used. There are 20 Task
Builder switch settings. The code, syntax and meaning for each switch
are given in Chapter 3.

2.2 EXAMPLE: VERSION 1 OF CALC

An example task, CALC, is developed in this manual from the simple
case given here through successive refinements and increasing
complexity. The successive versions of CALC are designed to summarize
the major points of each chapter and to illustrate possible uses for
the facilities described.

CHAPTER 2.   COMMANDS

As the first step in the development of the task CALC, three separate
FORTRAN routines are entered by means of a text editor, translated by
the FORTRAN compiler, and built into a task by the Task Builder.

The routines are:

      RDIN   which reads and analyzes input data and selects a data
                    processing routine on the basis of the analysis.

      PROC1   which processes the input according to a specified set
                    of rules;   and

      RPRT   which outputs the results as a series of reports.

The three routines communicate with each other through a common block
named 'DTA'.

CHAPTER 2. COMMANDS

2.2.1 Entering the Source Language

The source for the FORTRAN programs of the example CALC is entered and
filed by means of the text editor EDI. The user invokes EDI and types
in the source for the FORTRAN programs. The relevant parts of the
programs are shown below:

```
>EDI
EDI>RDIN.FTN
[CREATING NEW FILE]
INPUT
C       READ AND ANALYZE INPUT DATA,
C
C           SELECT A PROCESSING ROUTINE
C
C           ESTABLISH COMMON DATA BASE
C
        COMMON /DTA/ A(200), I
C           READ IN RAW DATA
        READ (6,1) A
1       FORMAT (200F6.2)
        . . .
C           CALL DATA PROCESSING ROUTINE
        CALL PROC1
C           GENERATE REPORT
        CALL RPRT
        . . .
        END
*CL
EDI>PROC1.FTN
[CREATING NEW FILE]
INPUT
C       FIRST DATA PROCESSING ROUTINE
C       COMMUNICATION REGION
        COMMON /DTA A(200),I
        ...
        RETURN
        END

*CL
EDI>RPRT.FTN
[CREATING NEW FILE]
INPUT
C       INTERIM REPORT PROGRAM
C       COMMUNICATION REGION
        COMMON /DTA/ A(200),I
        ...
        RETURN
        END
*EX
[EXIT]
```

CHAPTER 2. COMMANDS

2.2.2 Compiling the FORTRAN Programs

The FORTRAN programs are compiled by the following sequence:

```
>FOR
FOR>RDIN,LRDIN=RDIN
FOR>PROC1,LPROC1=PROC1
FOR>RPRT,LRPRT=RPRT
```

The first command invokes the FORTRAN compiler. The second command directs the compiler to take source input from RDIN.FTN, place the relocatable object code in RDIN.OBJ and write the listing in LRDIN.LST. The remaining commands perform similar actions for the source files PROC1 and RPRT.

2.2.3 Building the Task

The task for the three programs is built in the following way:

```
>TKB CALC;1,LP:=RDIN,PROC1,RPRT
```

The task building command specifies the name of the task image file (CALC.TSK;1), the device for the memory allocation file (LP) and the names of the input files (RDIN.OBJ, PROC1.OBJ and RPRT.OBJ). The task makes use of all the default assumptions for switches and options.

2.3 SUMMARY OF SYNTAX RULES

Syntactic rules for the interaction between the user and the Task Builder are given here. These rules do not present any new information; rather, they define, in a more formal and concise way, the syntax of the commands already described in this chapter.

In the syntax rules, the symbol '...' indicates repetition. For example,

        input-spec, ...

means one or more input-spec items separated by commas; that is, one of the following forms:

        input-spec

        input-spec, input-spec

        input-spec, input-spec, input-spec

        ... etc.

CHAPTER 2. COMMANDS

As another example,

        arg: ...

means one or more arg items separated by colons.

As a final example,

        TKB>input-line
        ...

means one or more of the indicated 'TKB input-line' items.

2.3.1 Syntax Rules

The syntax rules are as follows:

1. A task-building-command can have one of several forms.   The
   first form is a single line:

   >TKB task-command-line

   The second form has additional lines for input file names:

   >TKB
   TKB>task-command-line
   TKB>input-line
   ...
   TKB>terminating-symbol

   The third form allows the specification of options:

   >TKB
   TKB>task-command-line
   TKB>/
   ENTER OPTIONS:
   TKB>option-line
   ...
   TKB>terminating-symbol

   The fourth form has both input lines and option lines:

   >TKB
   TKB>task-command-line
   TKB>input-line
   ...
   TKB>/
   ENTER OPTIONS:
   TKB>option-line
   ...
   TKB>terminating-symbol

The terminating symbol can be:

   /   if more than one task is to be built, or
   //   if control is to return to the Monitor
       Console Routine.

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 defined in Rule 7.

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 of the 20 options is given in Chapter 3.

7. A file-specification conforms to standard RSX-11M conventions. It has the form

    device:[group,member]filename.type;version/sw...

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

    device   is the name of the physical device on which the
                volume containing the desired file is mounted.
                The name consists of two ASCII characters
                followed by an optional 1- or 2-digit octal unit
                number; for example, 'LP' or 'DT1'.

    group   is the group number and is in the range 1
                through 377 (octal).

    member   is the member number in the range 1 through 377
                (octal).

The combination of the group number and the member number is
called the user identification code (UIC).

    filename   is the name of the desired file. The file name
                can be from 1 to 9 alphanumeric characters, for
                example, CALC.

    type   is the 3-character type identification. Files
                with the same name but a different function are
                distinguished from one another by the file type;
                for example, CALC.TSK and CALC.OBJ.

    version   is the octal version number of the file.
                Various versions of the same file are
                distinguished from each other by this number;
                for example, CALC;1 and CALC;2.

    sw   is a switch specification. More than one switch
                can be used, each separated from the previous
                one by a '/'. The switch is a 2-character
                alphabetic name which identifies the switch
                option. The permissable switch options and
                their syntax are given in Chapter 3.

The device, the user identification code, the type, the version, and the switch specifications are all optional.

The following table of default assumptions applies to missing components of a file-specification:

    item   default

    device SY0, the system device *

    group   the system group number currently in effect *

    member the system member number currently in effect *

    type   task image   TSK
         memory allocation MAP
         symbol definition STB
         object module   OBJ
         object module library OLB
         overlay description ODL
         indirect command CMD

    version for an input file, the highest-numbered existing version.

         for an output file, one greater than the highest-numbered existing version.

    switch (the default for each switch is given in Chapter 3.)

*Except when appearing with an overlay description, an explicit device or UIC designation becomes the default for subsequent files separated by commas.

For example:

    DT1:IMG1,MP1=IN1,DF:IN2,IN3

      File   Device

     IMG1.TSK   DT1
     MP1.MAP   DT1
     IN1.OBJ   SY0
     IN2.OBJ   DF0
     IN3.OBJ   DF0

CHAPTER 3

SWITCHES AND OPTIONS

This chapter describes the ways in which additional directions can be
given to the Task Builder for the construction of a task image. Much
of the information in this chapter is quite specialized and refers to
topics that are described later in the manual. A quick reading of
this chapter will show the user the range of ways he can adjust the
task image he builds. Later, the chapter can be used as a reference
for practical applications with specific requirements.

3.1 SWITCHES

The syntax for a file specification, as given in Chapter 2, is:

        dev:[group,member]filename.type;version/sw-1/sw-2.../sw-n

The file specification concludes with zero or more switches, sw-1,
sw-2, ..., sw-n, and these are described in what follows:

When a switch is not given by the user, the Task Builder establishes a
setting for the switch, called a default assumption.

A switch is designated by a 2-character switch code.   The allowable
code values are defined by the processor which interprets the code.
The code is an indication that the switch applies or does not apply.
For example, if the switch code is CP (checkpointable), then the
switch settings recognized are:

        /CP   The task is checkpointable.
        /-CP   The task is not checkpointable.
        /NOCP   The task is not checkpointable.

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.

Table 3-1
Task Builder Switches

CODE	MEANING	APPLIES TO FILE*	DEFAULT
AC	Task is an ancillary control processor.	T	-AC
CC	Input File Consists of concatenated object modules.	I	CC
CP	Task is checkpointable.	T	-CP
CR	Append a global cross-reference listing to the memory allocation file.	M	-CR
DA	Task contains a debugging aid.	T,I	-DA
EA	Task uses extended arithmetic element.	T	-EA
FP	Task uses the floating point processor.	T	-FP
HD	Task image includes a header.	T,S	HD
LB	Input file is a library file.	I	-LB
MM	System has memory management.	T	MM or-MM**
MP	Input file contains an overlay description.	I	-MP
PI	Task is position independent.	T,S	-PI
PM	Post mortem dump is requested.	T	-PM
PR	Task has privleged access rights.	T	-PR
SH	Short memory allocation file is requested.	M	-SH
SP	Spool map output.	M	SP
SQ	Task p-sections are allocated sequentially.	T	-SQ
SS	Selective Search for global symbols.	I	-SS
TR	Task is to be traced.	T	-TR
XT:n	Task Builder exits after n diagnostics.	T	-XT

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

**   The default for the memory management switch is MM if the host
       system has memory management hardware and -MM if the host system
       does not have memory management hardware.

CHAPTER 3. SWITCHES AND OPTIONS

3.1.1 Task Builder Switches

The switches recognized by the Task Builder are described in this section. For each switch, the following information is given:

o   the switch mnemonic,

o   the file(s) to which the switch can be applied.

o   a description of the effect of the switch on the Task Builder, and

o   the default assumption made if the switch is not present.

The switches are given in alphabetical order.

3.1.1.1 AC (Ancillary Control Processor)

file: task image

meaning: The task is an ancillary control processor. An ancillary control processor is a privileged task that extends certain Executive functions. For example, the system task 'F11ACP' is an ancillary control processor that receives and processes file related input and output requests.

effect: The task is privileged. The Task Builder sets the AC attribute flag and the privileged attribute flag in the task label block flag word.

default: -AC

3.1.1.2 CC (Concatenated Object Modules)

file: input

meaning: The file contains more than one object module.

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

default: CC

CHAPTER 3.   SWITCHES AND OPTIONS

3.1.1.3   <u>CP (Checkpointable)</u>

file: task image

meaning: The task is checkpointable.

effect: The Task Builder allocates in the task image a checkpoint
           area equal to the size of the partition for which the task
           is built. If the task is checkpointed, the entire parti-
           tion is recorded in this area. The checkpoint area is
           described in connection with the task image in Chapter 4.

default: -CP

3.1.1.4   <u>CR (Cross Reference)</u>

file: task image or memory allocation

meaning: A global cross-reference is to be appended to the memory
           allocation file.

effect: The cross-reference task is invoked to append a list of
           global symbols, and their referencing modules, to the
           memory allocation file. The format and content of the
           cross-reference is explained in Section 4.4.2.

default: -CR

3.1.1.5   <u>DA (Debugging Aid)</u>

file: task image or input

meaning: The task includes a debugging aid.

effect: The Task Builder performs the special processing described
           in Appendix F. 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

3.1.1.6   <u>EA (Extended Arithmetic Element)</u>

file: task image

meaning: The task uses the KE-11A Extended Arithmetic Element.

effect: The Task Builder allocates three words in the task header
           for the extended arithmetic element save area.

default: -EA

CHAPTER 3. SWITCHES AND OPTIONS

3.1.1.7 FP (Floating Point)

file: task image

meaning: The task uses the Floating Point Processor.

effect: The Task Builder allocates 25 words in the task header for the floating point save area.

default: -FP

3.1.1.8 HD (Header)

file: task image or symbol definition

meaning: A header is to be included in the task image. The negation of this switch to produce a shared region is described in Chapter 7.

effect: The Task Builder constructs a header in the task image. The content of the header is described in Appendix C.

default: HD

3.1.1.9 LB (Library File)

This switch has two forms:

1. Without arguments: LB

2. With arguments: LB:mod-1:mod-2...:mod-8

The interpretation of the switch depends upon the form.

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 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 the 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 and extracts from the library for inclusion in the task image any modules that contain definitions for such references.

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

CHAPTER 3. SWITCHES AND OPTIONS

<div align="center">NOTE</div>

1. If the user wants the Task Builder to
   search a library file both to resolve
   global references and to select named
   modules for inclusion in the task image,
   he must name the library file twice:
   once, with the LB switch and no
   arguments to direct the Task Builder to
   search the file for undefined global
   references, and a second time with the
   desired modules to direct the Task
   Builder to include those modules in the
   task image being built.

2. The SS option may be used with the LB
   option (with or without arguments)
   to perform a selective search for
   global definitions.

default: -LB

3.1.1.10 MM (Memory Management)

file: task image

meaning: The system on which the task will run has memory management
          hardware. Mapped and unmapped systems are described in
          Chapter 4. The use of this switch to build a task to run on
          another system with different mapping status is illustrated
          in Chapter 8.

effect: The Task Builder allocates memory for a mapped or unmapped
          system independent of the mapping status of the system on
          which the task is being built. If -MM is specified, the
          Task Builder assumes an unmapped system and no memory
          managmenet.

default: MM or -MM. The Task Builder allocates memory according to
          the mapping status of the system on which the task is being
          built.

3.1.1.11 MP (Overlay Description)

file: input

meaning: The input file describes an overlay structure for the task.
          Overlay descriptions are discussed in Chapter 5.

effect: The Task Builder receives all the input file specifications
          from this file and allocates memory as directed by the
          overlay description.

NOTE

When an overlay description file is
specified as the input file for a task,
it must be the only input file
specified. The Task Builder does not
accept any other input files.

default: -MP

3.1.1.12 PI (Position Independent)

file: task image or symbol definition

meaning: The task contains only position independent code or data.
           Position independent shared regions are described in Chapter
           7.

effect: The Task Builder sets the Position Independent Code (PIC)
           attribute flag in the task label block flag word.

default: -PI

3.1.1.13   PM (Post Mortem Dump)

file: task image

meaning: In the event of an abnormal task termination, the system
           will automatically list the contents of task memory.
           (See Chapter 9 for information on memory dumps.)

effect: The Task Builder sets the Post-Mortem Dump flag in the
           Task label flag word.

default: -PM

3.1.1.14 PR (Privileged)

file: task image

meaning: The task is privileged with respect to memory access rights.
           The task can access the I/O page, and the Executive in
           addition to its own partition. Privileged tasks are
           described in Chapter 4.

effect: The Task Builder sets the Privileged Attribute flag in the
           task label block flag word.

default: -PR

3.1.1.15   SH (Short Map)

file: memory allocation

meaning: The short version of the memory allocation file is produced.
           Chapter 4 describes the memory allocation file and gives a
           short and a long version of a memory allocation file.

effect: The Task Builder does not produce the 'File Contents'
           section of the memory allocation file.

default: -SH

3.1.1.16   SP (Spool Map Output)

file: memory allocation

meaning: Invoke the print symbiont to list the map file on LPØ.
           The file is deleted after printing.

effect: The Task Builder creates a map file on SYØ and then
           uses the print symbiont to list the file. The file is
           deleted after printing.

default: SP

3.1.1.17   SQ (Sequential)

file: task image

meaning: The task image is constructed from the specified program
           sections in the order in which they are input. Chapter 4
           describes the allocation of the task image and gives an
           example which shows the allocation performed under the
           default assumption and the allocation performed when the   SQ
           switch is specified.

effect: The Task Builder does not re-order the program sections
           alphabetically.

default: -SQ

3.1.1.18   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 concatenated
           object file, the switch is applied to each module in the
           file.

default: -SS

CHAPTER 3. SWITCHES AND OPTIONS

3.1.1.19 TR (Traceable)

file: task image

meaning: The task is traceable.

effect: The Task Builder sets the T bit in the initial PS word of
          the task. When the task is executed, a trace trap occurs on
          the completion of each instruction.

default: -TR

3.1.1.20 XT:n (Exit on Diagnostic)

file: task image

meaning: More than n error diagnostics are 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, using the convention:

                 n.   means a decimal number (the decimal point must be
                        included).
                 #n or n means an octal number.

          If n is not specified, it is assumed to be 1.

default: -XT

3.1.2 Examples

The following terminal sequences illustrate the use of switches in
file specifications and the resulting interpretation.

Terminal Sequence	Interpretation
>TKB IMG1/CP/DA=IN1/-CC	The task IMG1.TSK is checkpointable and includes the debugging aid SY:[1,1]ODT.OBJ. The input file IN1 contains only one object module.
>TKB TKB>IMG2/PR,MP1/SH= TKB>IN2,RSX11M.STB TKB>//	The task IMG2.TSK is a privileged task. The short map MP1.MAP is requested. The inputs for the task are the file IN2.OBJ and the symbol definition file RSX11M.STB which links the task to the subroutines and data base of the Executive.
>TKB TKB>IMG3=IN3 TKB>LB1/LB:SUB1:SUB2 TKB>LB1/LB,DBG1/DA TKB>//	The task IMG3.TSK contains the input file IN3.OBJ, the modules SUB1 and SUB2 from the library file LB1, and the debugging aid DBG1.OBJ. The library file LB1.OLB is specified a second time without arguments so that the Task Builder will search the file for undefined global references.

>TKB IMG4/XT:5=TREE/MP The Task IMG4.TSK is built from the overlay description contained in the file TREE.ODL. If more than five diagnostics occur, the Task Builder aborts the run.

3.1.3 Override Conditions

In some cases, it is not reasonable to apply two particular switches to a file. When such a conflict occurs, the Task Builder selects the overriding switch according to the following table:

switch	switch	overriding switch
AC	PR	AC
EA	FP	FP
CC	LB	LB

For example, in the terminal sequence:

    MCR>TKB IMG5=IN6,IN5/LB/CC

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

3.2 OPTIONS

Twenty options are available to the user of the RSX-11M Task Builder. These options give the Task Builder information about the characteristics of the task.

Some of these options are of interest to all users of the system, some of interest only to the FORTRAN programmer, and some of interest only to the MACRO-11 programmer. The interest range is given with the description of the option. For more detailed descriptions of the FORTRAN-related options, read the appropriate FORTRAN IV or FORTRAN IV-PLUS manual.

Options can be divided into seven categories. The identifying mnemonics and a brief description for each category are listed below:

1. contr — Control options are used to affect Task Builder execution. ABORT is the only member of this category. The user can direct the Task Builder to abort the task build by the use of the option ABORT.

2. ident — Identification options are used to identify task characteristics. The task name, priority, user identification code, and partition can be specified by the use of options in this category.

3. alloc — Allocation options are used to modify the task's memory allocation. The size of stack, program-sections in the task, and FORTRAN work areas and buffers can be adjusted by the use of options in this category.

4. share — Storage sharing options are used to indicate the task's intention to access a shared region.

5. device — Device specifying options are used to specify the number of units required by the task and the assignment of physical devices to logical unit numbers.

6. alter — Content altering options are used to define a global symbol and value or to introduce patches in the task image

7. synch — Synchronous trap options are used to define synchronous trap vectors.

Table 3-2 lists all the options alphabetically. A brief description of each option is given. The interest range of the option is indicated by the following codes:

F   option is of interest to FORTRAN programmers only.
M   option is of interest to MACRO-11 programmers only.
FM option is of interest to both.

The mnemonic for the category to which the option belongs is also indicated in the table.

The options are then described in more detail by category.

CHAPTER 3. SWITCHES AND OPTIONS

Table 3-2
Task Builder Options

Option	Meaning	Interest	Category
ABORT	Direct TKB to terminate build.	FM	contr
ABSPAT	Declare absolute patch values.	M	alter
ACTFIL	Declare number of files open simultaneously.	F	alloc
ASG	Declare device assignment to logical units.	FM	device
COMMON	Declare task's intention to access a memory resident shared region.	FM	share
EXTSCT	Declare extension of a program section.	FM	alloc
EXTTSK	Extend the amount of memory owned by a task.	FM	alloc
FMTBUF	Declare extension of buffer used for processing format strings at run-time.	F	alloc
GBLDEF	Declare a global symbol definition.	M	alter
GBLPAT	Declare a series of patch values relative to a global symbol.	M	alter
LIBR	Declare task's intention to access a memory resident shared region.	FM	share
MAXBUF	Declare an extension to the FORTRAN record buffer.	F	alloc
ODTV	Declare the address and size of the debugging aid SST vector.	M	synch
PAR	Declare partition name and dimensions.	FM	ident
PRI	Declare priority.	FM	ident
STACK	Declare the size of the stack.	FM	alloc
TASK	Declare the name of the task.	FM	ident
TSKV	Declare the address of the task SST vector.	M	synch
UIC	Declare the user identification code under which the task runs.	FM	ident
UNITS	Declare the maximum number of units.	FM	device

CHAPTER 3. SWITCHES AND OPTIONS

3.2.1 Control Option

There is only one control option. This option is of interest to all users of the system.

3.2.1.1 ABORT (Abort the Task Build) - The ABORT option directs the Task Builder to abort the task build.

This option is used when it is discovered that an earlier error in the terminal sequence will cause the Task Builder to produce an unusable task image.

The task Builder, on recognizing the keyword ABORT, stops accepting input and restarts for another task build.

An example of the use of the ABORT option is given in section 3.3.

syntax: ABORT = n

       where n   is an integer value. The integer is required to satisfy the general form of an option; however, the value is ignored in this case.

default: none

NOTE

The use of CTRL/Z causes the Task Builder to stop accepting input and build the task.

The ABORT option is the only proper way to restart the Task Builder if an error is discovered and the Task Builder output is not desired.

3.2.2 Identification Options

Four options are available for providing identifying information for the task. These options are of interest to all users of the system.

The identification options specify the name of the task, the user identification code, the priority, and the partition. The user identification code can be specified when the task is run. If such a specification is not made at run time, the user identification code established when the task was built is used.

CHAPTER 3. SWITCHES AND OPTIONS

3.2.2.1 TASK (Task Name) - The TASK option specifies the name of the task.

syntax: TASK = task-name

where: task-name is a 1- to 6-character radix-50 name identifying
          the task.

default: The first six characters of the name of the task image file
          are used to identify the task when the task is installed.

3.2.2.2 UIC (User Identification Code) - The UIC option declares the
User Identification Code (UIC) for the task if no UIC is specified
when execution is requested.

syntax: UIC = [group,member]

where: group   is an octal number in the range 1 - 377 which
                     specifies the group.

          member   is an octal number in the range 1 - 377 which
                     specifies the member.

default: The UIC under which the Task Builder is running
                     (normally the terminal UIC).

3.2.2.3 PRI (Priority) - The PRI declares the priority at which the
task executes. If no priority is specified when the task is
installed, this priority is used.

syntax: PRI = priority-number

where: priority-number is a decimal integer in the range 1 - 250

default: (established by Install)   See   the RSX-11M   Operator's
          Procedures Manual.

3.2.2.4 PAR (Partition) - The PAR option identifies the partition for
which the task is built.

In a mapped system a task can be installed   in   any system or user
partition large enough to contain it.

In an unmapped system, the task is bound to physical memory and must
be installed in the partition for which it was built or in a partition
starting at the same memory address as that partition.

syntax: PAR = pname [:base:length]

where   pname   is the name of the partition

          base   is the octal byte address defining the start of
                     the partition.

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

           In a mapped system a length of zero implies a
           system-controlled partition. A non-zero   length
           implies a user-controlled partition.

default: PAR = GEN

If the base and length are not specified, the Task Builder tries to
obtain   that   information   from the system on which the task is being
built. If the partition named is resident in that system, the base
and length can be obtained.

The Task Builder binds the task to the addresses defined   by
the partition base and verifies that the task does not exceed the
length specification if the partition is user-controlled.

To determine the validity of the task the Task Builder must   consider
two   types of task images, runnable and non-runnable, and two types of
systems, mapped and unmapped. A runnable task image must have a
header and can be installed and run. A non-runnable image must not
have a header and can not be executed directly.   The Task Builder,
therefore, enforces the address limits according to the type of image
and type of system, as follows:

	Runnable tasks		Non-runnable images	
	mapped	unmapped	mapped	unmapped
base	0	on 32word boundary	on 4k boundary	on 32word boundary
length	multiple of 32words	multiple of 32words	multiple of 32words	multiple of 32words
high address bound	(32K-32) words	28K Words	(32K-32) words	(32K-32) Words

3.2.3 Allocation Options

There are six options that direct the Task Builder to change the
length of an allocation. The first three options are of interest only
to the FORTRAN programmer. The remaining options are of interest to
all.

3.2.3.1 ACTFIL (Number of Active Files) - The ACTFIL option declares
the   number   of files that the task can have open simultaneously. For
each active file, an allocation of approximately 512 bytes is made.

If the number of active files used by a task is less than the default assumption of four, the ACTFIL option can be used to save space. If the number of active files is more than the default assumption, the ACTFIL option must be used to direct the Task Builder to make the additional allocation so that the task can run.

The FORTRAN Object Time System (OTS) and File Control Services (FCS) must be included in the task image for the extension to take place. The p-section that is extended has the reserved name '$$FSR1'.

syntax: ACTFIL = file-max

where: file-max is a decimal integer indicating the maximum number of files which can be open at the same time.

default: ACTFIL = 4

3.2.3.2 MAXBUF (Maximum Record Buffer Size) - The MAXBUF option declares the maximum record buffer size required for any file used by the task.

This option must be used to extend the buffer whenever a file is to be processed in which the maximum record size exceeds the default buffer length.

The FORTRAN Object Time System must be included in the task image for the extension to take place. The program section that is extended has the reserved name '$$IOB1'.

syntax: MAXBUF = max-record

where: max-record   is a decimal integer, larger than the default, which specifies the maximum record size in bytes.

default: MAXBUF = 132

3.2.3.3 FMTBUF (Format Buffer Size) - The FMTBUF option declares the length of internal working storage allocated for the compilation of format specifications at run-time. The length of this area must equal or exceed the number of bytes in the longest format string to be processed.

Run-time compilation occurs whenever an array is referenced as the source of formatting information within a FORTRAN I/O Statement. The program section to be extended has the reserved name '$$OBF1'.

syntax: FMTBUF = max-format

where: max-format   is a decimal integer larger than the default, which specifies the number of characters in the longest format specification.

default: FMTBUF = 132

3.2.3.4 <u>EXTSCT (Program Section Extension)</u> - The EXTSCT option declares an extension in size for a p-section. P-sections and their attributes are described in Chapter 4.

If the p-section has the attribute CON (concatenated), the section is extended by the specified number of bytes. If the p-section has the attribute OCR (overlay), the section is extended only if the length of the extension is greater than the length of the p-section.

For example, suppose that p-section BUFF is 200 bytes long and the option below is given:

        EXTSCT = BUFF:250

The extension specified for the p-section depends on the CON/OVR attribute; specifically:

        for CON the extension is 250 bytes.

        for OVR the extension is 50 bytes.

The extension occurs when the p-section name is encounted in an input object file or in the overlay description file.

Syntax: EXTSCT = p-sect-name:extension

where: p-sect-name   is a 1- to 6-character radix-5C name
                          specifying the p-section to be extended.

           extension   is an octal integer that specifies the number
                          of bytes by which to extend the p-section.

default: none

3.2.3.5 <u>EXTTSK (Extend Task Memory)</u> - The EXTTSK option declares the amount of additional memory to be allocated to the task when installed in a system-controlled partition.

The amount of memory available to the task will be the sum of the task size plus the increment specified in the EXTTSK keyword (rounded up to the nearest 32-word boundary). If the task is built for a user-controlled partition, the allocation of task memory reverts to the partition size. The maximum size of the task may not exceed 32K - 32 words.

In an unmapped system the EXTTSK keyword is ignored.

syntax: EXTTSK = length

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

default: EXTTSK = 0

3.2.3.6 <u>STACK (Stack Size)</u> - 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 interrupt service linkages. The stack is referenced by hardware register R6 (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

3.2.3.7 <u>Examples of Allocation Options</u> - Suppose the FORTRAN routines contained in file GRP1 use eight files simultaneously and the maximum record length in one of these files is 160 characters.

The terminal sequence used to build the task that would permit these programs to run is:

        >TKB
        TKB>IMG1,MP1=GRP1
        TKB>/
        ENTER OPTIONS:
        TKB>ACTFIL = 8
        TKB>MAXBUF = 160
        TKB>//

3.2.4 <u>Storage Sharing Options</u>

Two options indicate the task's intention to access a shared region. These options are of interest to all users of the system.

By convention, the COMMON option indicates the use of a shared region that contains only data and the LIBR option indicates the use of a shared region that contains only code. The two options have the same effect, however, and can be used interchangeably.

3.2.4.1 <u>COMMON (Resident Common Block)</u> - The COMMON option declares a resident common block for use by the task.

syntax: COMMON = common-name:access-code[:apr]

where: common-name   is the 1- to 6-character radix-50 name of the
                          common block.

           access-code   is the code RW (read-write) or the code RO
                          (read-only) indicating the type of access the
                          task requires.

                    apr   is an integer in the range   0-7 which
                                        specifies the first Addressing Page Register
                                        to be reserved for the common block.

default: none

The apr is optional and accepted only for a mapped system.

3.2.4.2 LIBR (Resident Library) - The LIBR option declares a resident
library for use by the task.

syntax: LIBR = library-name:access-code[:apr]

where: library-name   is the 1- to 6-character radix-50   name
                         specifying the library.

          access-code   is the code RW (read-write) or the code RO
                         (read-only) indicating the type of access the
                         task requires.

          apr   is an integer in the range 0 - 7 which
                         specifies the first Addressing Page Register
                         to be reserved for the library.

default: none

The apr is optional and is accepted only for a mapped system.

3.2.4.3 Example of Storage Sharing Options - Suppose   the   task
composed of the MACRO-11 programs TST1 and TST2 accesses a shared
region DTST that contains data and a shared region STST that contains
code.

The terminal sequence used to build the task is:

```
>TKB
TKB>CHK,LP:=TST1,TST2
TKB>/
ENTER OPTIONS:
TKB>COMMON = DTST:RW
TKB>LIBR = STST:RO
TKB>//
```

CHAPTER 3. SWITCHES AND OPTIONS

3.2.5 Device Specifying Options

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

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

Since the options are processed as they are encountered, to increase the number of units and assign devices to these units, the user should enter the UNITS option first and then the ASG option. 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.

syntax: UNITS = max-units

where: max-units   is a decimal integer in the range 0 - 250
                            specifying the maximum number of logical
                            units.

default: UNITS = 6

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 a 1- or 2-digit decimal unit
                            number.

            unit-num-1   are decimal integers indicating the
            unit-num-2   logical unit numbers.
            ...
            unit=num-8

default: ASG = SY0:1:2:3:4, TI0:5, CL0:6

3.2.5.3 Example of Device Specifying Options - Suppose the FORTRAN programs specified in the file GRP1 require nine logical units. The device assignments for units 1-6 agree with the default assumptions and logical units 7,8 and 9 are assigned to DECtape 1 (DT1). The terminal sequence of the example of 3.2.3.6 is changed to include device assignment options, as follows.

CHAPTER 3. SWITCHES AND OPTIONS

```
>TKB
TKB>IMG1,MP1=GRP1
TKB>/
ENTER OPTIONS:
TKB>ACTFIL = 8 !  MAXBUF = 160
TKB>UNITS=9 !  ASG = DT1:7:8:9
TKB>//
```

3.2.6 Storage Altering Options

These options alter the task image and are of interest only to the MACRO-11 programmer. The GBLDEF option declares a global symbol and value. The options ABSPAT and GBLPAT introduce patches into the task image.

3.2.6.1 GBLDEF (Global Symbol Definition) - The GBLDEF option declares the definition of a global symbol.

The symbol definition is considered absolute.

syntax: GBLDEF = symbol-name:symbol-value

where: symbol-name   is a 1- to 6-character radix-50 name 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 ABSPAT (Absolute Patch) - The ABSPAT option declares a series of patches starting at the specified base address. Up to 8 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 radix-50 name of the segment.

           address   is the octal address of the first patch. The address may be on a byte boundary; however, two bytes are always modified for each patch.

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

           val-2   is an octal number in the range 0-177777 to be assigned to address+2

           ...   ...

           val-8   is an octal number in the range 0-177777 to be assigned to address+20.

CHAPTER 3. SWITCHES AND OPTIONS

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

3.2.6.3 GBLPAT (Global Relative Patch) - The GBLPAT option declares a
series of patch values starting at an offset relative to a global
symbol.   Up to 8 patch values can be given.

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

where:
seg-name	is as defined for ABSPAT
sym-name	is a 1- to 6-character radix-50 name specifying the global symbol.
offset	is an octal number specifying the offset from the global symbol.
val-1 val-2 ... val-8	are as defined for ABSPAT

default: none

NOTE

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

3.2.6.4 Example of Storage Altering Options - Suppose that in the
example composed of the MACRO-11 programs TST1 and TST2, GAMMA is a
referenced symbol whose value is to be specified when the task is
built.   The user defines the symbol GAMMA to have the value 25. He
introduces 10 patch values relative to the global symbol DELTA.

The terminal sequence of Example 3.2.4.3 is modified to include the
options GBLPAT and GBLDEF as follows:

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

3.2.7 Synchronous Trap Options

There are two options which declare that the specified vector address
is to be preloaded into the task header thus enabling the task to
receive control on the occurrence of synchronous traps. These options
are of interest only to the MACRO-11 programmer.

3.2.7.1 ODTV (ODT SST Vector) - The ODTV option declares a global
symbol to be the address of the ODT Synchronous System Trap vector.
The defined global symbol must exist in the part of the task that is
always in memory.

syntax: ODTV = symbol-name:vector-length

where: symbol-name   is a 1- to 6-character radix-50 name of a
                         global symbol.

          vector-length is a decimal integer in the range 1 - 32
                         specifying the length of the SST vector in
                         words.

default: none

3.2.7.2 TSKV (Task SST Vector) - The TSKV option declares a global
symbol to be the address of the task SST vector. The defined symbol
must exist in the part of the task that is always in memory.

syntax: TSKV = symbol-name:vector-length

where: symbol-name
                          } are as defined for ODTV
          vector-length

default: none

3.3 EXAMPLE: CALC;2

Suppose that in the first execution of the task CALC several logical
errors are found. The user corrects the program and is now ready to
make the changes in the program and some adjustments in the task image
file based on the information he obtained about the size of the task
in the first task build.

In this example, he modifies the text file for the program, recompiles
the program, and rebuilds the task so that only one active file buffer
is reserved and the task is built for a larger partition.

CHAPTER 3. SWITCHES AND OPTIONS

3.3.1 Correcting the Errors in Program Logic

The FORTRAN source language for the program 'RDIN' is corrected to be:

```
C       READ AND ANALYZE INPUT DATA
C          SELECT A PROCESSING ROUTINE
C
C          ESTABLISH COMMON DATA BASE
C
        COMMON /DTA/ A(200), I
C          READ IN RAW DATA
        READ (6,1) A
1       FORMAT (200 F6.2)
        ...
        CALL PROC1
        ...
        CALL RD1
        ...
        CALL RPRT
        END
        SUBROUTINE RD1
        ...
        RETURN
        END
```

Next, the program 'RDIN' is recompiled:

```
>FOR RDIN,LRDIN=RDIN
```

Observe that the corrections to 'RDIN' included the addition of a subroutine 'RD1'. The object file produced by the FORTRAN compiler as a result of the above terminal sequence now contains two object modules.

3.3.2 Building the Task

Suppose that the user knows from the logic of the program that only one file is open at a time. The Task Builder assumes that four files are open simultaneously, so some space can be saved in the task by use of the ACTFIL option. In addition, the task is moved from the default partition 'GEN' which on the host system is 8192 words to a larger partition 'PAR14K'. Since 'PAR14K' is resident in the host system, the base and length are known to the Task Builder.

CHAPTER 3.   SWITCHES AND OPTIONS

He builds the task with the following terminal sequence:

```
>TKB
TKB>CALC;2,=RDIN,RPRT,PROC1
TKB>/
ENTER OPTIONS:
TKB>PAR=PAR14K
TKB>ABORT=1
TKB -- *FATAL* - ABORTED VIA REQUEST
ABORT=1
TKB>CALC;2,LP:/SH=RDIN,PROC1,RPRT
TKB>/
ENTER OPTIONS:
TKB>PAR=PAR14K
TKB>ACTFIL=1
TKB>//
```

The user introduced the ABORT option to end the task build when he
realized that he had omitted the memory allocation file.

The effect of these options on the memory allocation is seen in the
next chapter. After the description of the task and memory allocation
files, the memory allocation files for the first two examples are
given.

CHAPTER 4

MEMORY ALLOCATION

This chapter describes the allocation of task and system memory. The two types of systems supported by RSX-11M, mapped and unmapped, are described and the memory access rights of tasks within those systems are defined. The memory allocation file is described in detail and examples of memory allocation files in mapped and unmapped systems are illustrated. The memory allocation file for the example CALC;1 of Chapter 2 and CALC;2 of Chapter 3 are included and discussed. The effect of the options used in CALC;2 can be observed by comparing the two memory allocation files.

4.1 TASK MEMORY

Task memory in RSX-11M consists of a header, stack, and a set of named areas called program section (p-sections). Each p-section has associated with it attributes from which the Task Builder can determine its base and length.

Task Memory can be represented by the following diagram:

```
                        +-----------------+  -- end of partition
                        | unused portion  |
                        +-----------------+
                        |    p-section    |
                        +-----------------+
                        |    p-section    |
                        +-----------------+
                        |                 |
                        |      ...        |
                        |                 |
                        +-----------------+
                        |    p-section    |
                        +-----------------+
                        |     stack       |
                        +-----------------+
                        |     header      |
                        +-----------------+  -- partition base
```
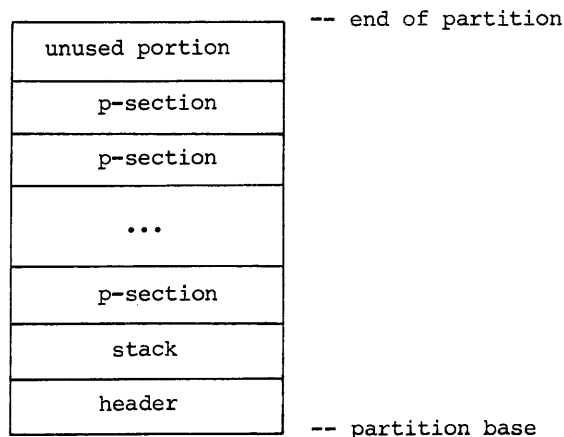
Task Memory

The header contains task parameters and data required by the Executive and provides a storage area for recording the tasks context. The contents of the header are described in detail in Appendix C.

The stack is an area that can be used for temporary storage and subroutine linkages and is referenced by general register R6, the stack pointer. The size of the stack can be changed by the use of the STACK option, as described in Chapter 3.

4.1.1 P-Sections

A program section, or p-section, is the basic unit of memory for the task. A source language program is translated into an object module consisting of p-sections. For example, the object module produced by compiling a typical FORTRAN program consists of a p-section containing the code generated by the compiler, a p-section for each common block defined in the FORTRAN program, and a set of p-sections required by the FORTRAN Object Time System.

A name and a set of attributes are associated with each p-section. The p-section attributes are given in Table 4-1.

CHAPTER 4. MEMORY ALLOCATION

Table 4-1
P-Section Attributes

ATTRIBUTE	VALUE	MEANING
access-code	RW	(read/write). Data can be read from and written into the p-section.
	RO	(read only). Data can be read from, but cannot be written into the p-section.
type-code	D**	(data). The p-section contains data.
	I**	(instruction). The p-section contains instructions or data and instructions.
scope-code	GBL	(global). The p-section name is considered across segment boundaries. The Task Builder allocates storage for the p-section from references outside the defining segment.
	LCL	(local). The p-section name is considered only within the defining segment. The Task Builder allocates storage for the p-section from references within the defining segment only.
alloc-code	CON	(concatenate). P-sections with the same name are concatenated. The total allocation is the sum of the individual allocations.
	OVR	(overlay). P-sections with the same name overlay each other. The total allocation is the length of the longest individual allocation.
reloc-code	REL	(relocatable). Storage in the p-section is allocated relative to the virtual base address of the partition.
	ABS	(absolute). Storage in the p-section is always allocated relative to zero.
memory-code	HIGH	(high). The p-section is to be loaded into high speed memory.
	LOW	(low). The p-section is to be loaded into core.

** Not to be confused with the I and D space hardware on the PDP 11/45 and 11/70.

CHAPTER 4. MEMORY ALLOCATION

The scope-code and type-code are only meaningful when an overlay
structure is defined for the task. The scope-code is described in
connection with the resolution of p-section in Chapter 5. The
type-code is described in connection with the generation of autoload
vectors in Chapter 6. The memory-code is not used by the Task
Builder.

The access-code and alloc-code are used by the Task Builder to
determine the placement and the size of the p-section in task memory.

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

The alloc-code is used to determine the starting address and length of
p-sections with the same name. If the alloc-code indicates that
p-sections with the same name are to be overlaid, the Task Builder
places each reference at the same position in task memory and
determines the total allocation from the length of the longest
reference. If the alloc-code indicates that p-sections with the same
name are to be concatenated, the Task Builder places each reference
one after another in task memory and determines the total allocation
from the sum of the lengths of each reference.

The initial reference to any p-section is always aligned on a word
boundary. When a p-section has the concatenate attribute, all
references to that p-section are placed adjacent to one another in task
memory. If any of these references ends on a byte boundary, the next
reference to that p-section is not word-aligned.

4.1.2 Allocation of P-sections

Suppose the user enters the following command:

>TKB IMG1,MP1=IN1,IN2,IN3,LBR1/LB

The user is directing 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, and to search the library file LBR1.OLB
for any undefined global references. Suppose the input files are
composed of p-sections with the following access-codes, alloc-codes,
and sizes:

File-name	P-section name	Access Code	Alloc 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

CHAPTER 4. MEMORY ALLOCATION

First, the Task Builder collects all p-sections with the same name to
determine the allocation for each uniquely named p-section.

In this example, there are two occurrences of the p-section named B
with attributes RW and CON. The total allocation for B is the sum of
the lengths of each reference;   that is, 100 + 120 = 220.   The
allocation for each uniquely named p-section then is:

P-section Name	Total Allocation
B	22Ø
A	3ØØ
C	22Ø

The Task Builder then re-organizes the p-sections alphabetically and
places them in memory according to their access-code, as follows:

C (220)	⎤ ⎦ read only	
B (220)	⎤	task memory
A (300)	⎟ read/write	
stack	⎦	
header		
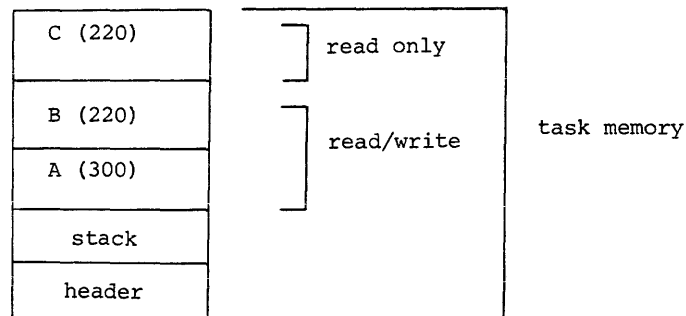
4.1.2.1 Sequential Allocation of P-sections - The SQ (sequential)
switch affects only the placement of p-sections in task memory.
P-sections with the same name and attributes are collected as
described;   then uniquely named p-sections are placed in memory in the
order of input sequence according to the access-code.

Suppose the user adds the SQ switch to the previous example:

>TKB IMG1/SQ,MP1=IN1,IN2,IN3,LBR1/LB

The Task Builder collects the p-sections and places them in memory in
the input sequence, as follows:
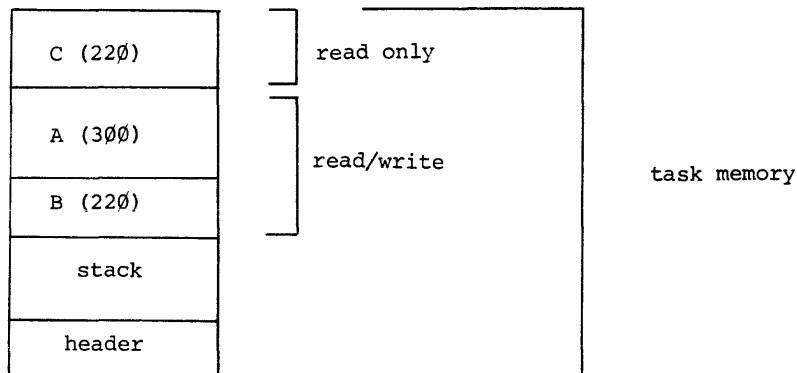
4.1.3 The Resolution of Global Symbols

When creating the task image file, the Task Builder resolves global references. Suppose the global symbols are defined and referenced in the p-sections in the following way:

File Name	P-section Name	Global Defn.	Global Name
IN1	B	B1	A1
		B2	L1
	A		C1
			XXX
	C		
IN2	A	A1	B2
	B	B1	
IN3	C		B1

In processing the first file, IN1, the Task Builder finds definitions for B1 and B2 and references to A1,L1,C1, and XXX. Since no definition exists for these references, the Task Builder defers the resolution of these global symbols. In processing the next file, IN2, the Task Builder finds a definition for A1, which resolves the previous reference, and a reference to B2, which can be immediately resolved.

CHAPTER 4. MEMORY ALLOCATION

When all the input object files have been processed, the Task Builder has three unresolved global references, namely: Cl, Ll, and XXX. A search of the library file LBR1 resolves Ll and the Task Builder includes the defining module in the task image. A search of the System Library resolves XXX. The global symbol Cl remains unresolved and is, therefore, listed as an undefined global symbol.

The relocatable global symbol Bl is defined twice and is listed as a multiply-defined global symbol on the terminal. The first definition of a multiply defined symbol is used by the Task Builder. An absolute global symbol can be defined more than once without being listed as multiply defined as long as each occurrence of the symbol has the same value.

4.2 SYSTEM MEMORY

In RSX-11M, system memory consists of the resident Executive and a set of named areas. These named areas are partitions, sub-partitions, and common blocks; associated with each of them are parameters of base and length.

System memory can be represented by the following diagram:
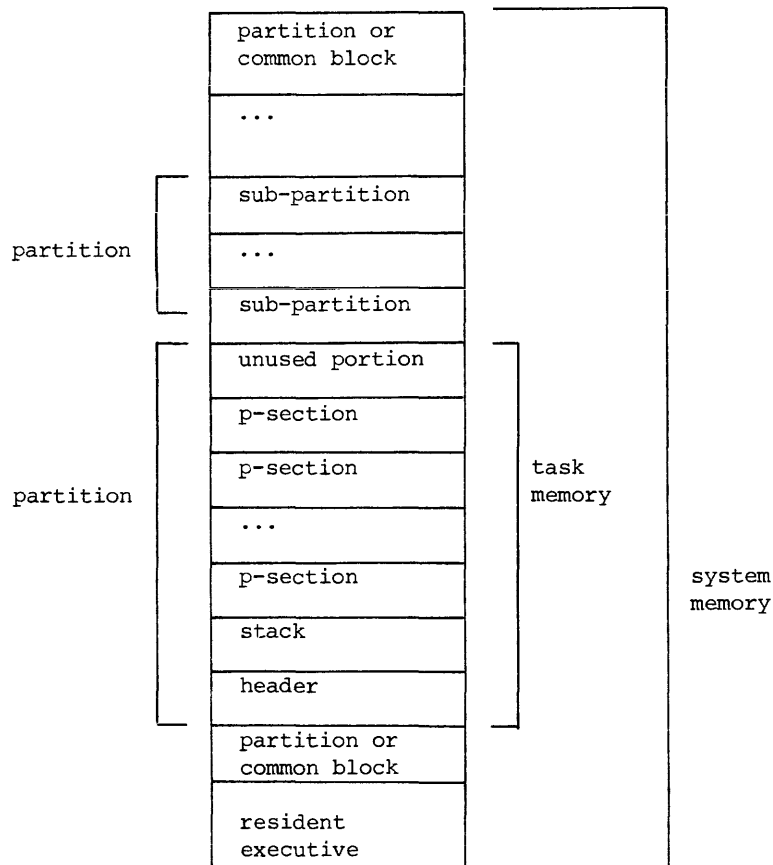
CHAPTER 4. MEMORY ALLOCATION

4.2.1 Mapped and Unmapped Systems

RSX-11M supports two types of systems, mapped and unmapped. A system with memory management hardware is called a mapped system. Mapped systems differ from unmapped systems in three respects:

1. Binding

In an unmapped system, the task is relocated to the base specified by the partition at the time the task is built, and therefore, the task can not be installed in a partition with a different base address.

In a mapped system, the task is bound to virtual zero and relocated by the mapping hardware, and therefore, the task can be installed in any partition large enough to contain it.

2. Protection

In an unmapped system, the task can access all physical memory.

In a mapped system, the task can only access memory specifically owned by the task.

3. Size

In an unmapped system, the largest task size is 28K minus the size of the Executive.

In a mapped system, the largest task size is 32K - 32.

The structure of task memory is identical in both systems. No object code alterations are required to run a task in either a mapped or unmapped system.

4.2.2 Privileged Tasks

A privileged task has special memory access rights. A non-privileged task can access only its own partition and any referenced shared regions, but a privileged task can, in addition, access the Executive and the I/O page.

In an unmapped system, a task cannot be prevented from accessing the entire memory, but the users of the system are expected to observe the access rules and preserve the distinction between privileged and non-privileged tasks.

In a mapped system, however, the task can only access the memory specifically owned by the task, so the distinction between privileged tasks and non-privileged tasks is a real one.

The memory allocation for a privileged task in a mapped system can be represented by the following diagram:

CHAPTER 4. MEMORY ALLOCATION


```
            +-------------------+
            |                   |
            |    I/O page       |
            |                   |      -- virtual 16ØØØØ
            +-------------------+
            |                   |
            |                   |      -- end of partition
            +-------------------+
            |  unused portion   |
            |  available to     |
            |  task             |
            +-------------------+
            |  task code and    |
            |  data             |
            +-------------------+
            |  stack            |
            +-------------------+
            |  header           |
            |                   |      -- partition base
            +-------------------+         (virtual 100000)
            |                   |
            +-------------------+
            |  executive        |
            |  R/W mapping      |
            +-------------------+
            |  low core         |
            |  context          |
            +-------------------+      -- virtual Ø
```
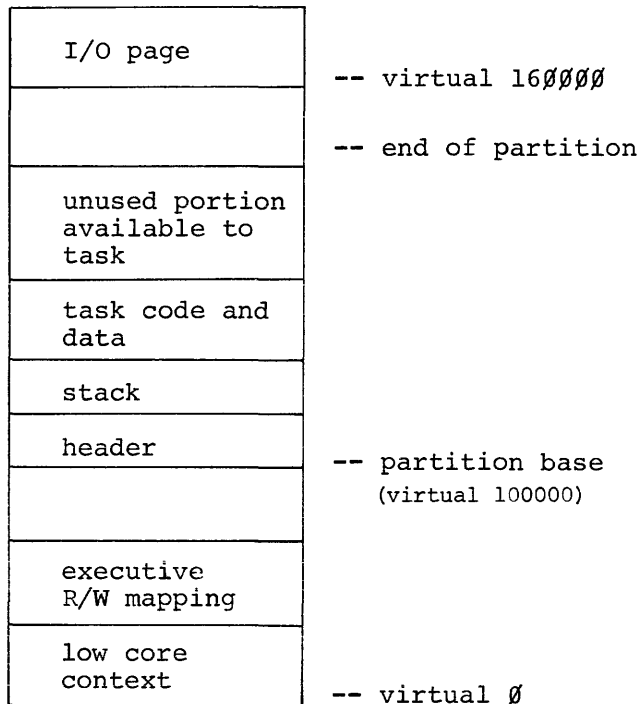

The Executive and system tables occupy virtual locations within
address limits 0 - 77777. The task can occupy virtual locations
100000 - 160000. A privileged task can not reside in a partition
whose length exceeds 12K (checked by the Task Builder and by Install).

4.3 TASK IMAGE FILE

In addition to the task memory, or core image, the task image file
contains a label block group and possibly a checkpoint area. The
label block group contains data that is used by the Install processor
to create an entry in the system task directory for the task. The
label is described in detail in Appendix C.

The checkpoint area is allocated if the user specifies that his task
is checkpointable in building the task:

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

The switch CP is appended to the task image file indicating that the
task is checkpointable.

4.3.1 Checkpoint Area

If the task is checkpointable, the Task Builder must reserve space in the task image file large enough to save all of the memory owned by the task. If this area is too small, the task must be installed with the checkpointable attribute disabled.

When the task is to reside in a system-controlled partition, the size of this area is the task size plus any increment specified through the EXTTSK keyword (see paragraph 3.2.3.5). If the task is to reside in a user-controlled partition, the size of this area is equal to the partition size.

When building a task for an unmapped system, the Task Builder always allocates the checkpoint area based on partition size (all partitions must be user-controlled).

When building a task for a mapped system, the Task Builder assumes that the partition is system-controlled if:

1. A length of zero is specified in the PAR keyword (see paragraph 3.2.2.4),

2. A resident, system-controlled partition is specified in the PAR keyword,

3. The resident, default partition, GEN, is system-controlled,

4. The EXTTSK keyword is used after any partition specifications (see paragraph 3.2.3.5).

The following examples illustrate option input for the above conditions.

Example 1: Explicitly specifying a system-controlled partition.

    TKB>PAR=SYSCNT::

The two colon delimiters define a base and length of 0.

Example 2: Specifying a resident partition.

    TKB>PAR=GEN

In this case the assumption made by the Task Builder depends upon whether or not the resident partition, GEN, is system-controlled.

Example 3: Default partition is to be used.

    ENTER OPTIONS

    TKB>/

The default partition, GEN, is assumed. The allocation of checkpoint space in the task file depends upon whether this partition is user-controlled.

Example 4: Use of the EXTTSK keyword.

    TKB_EXTTSK=4000

    TKB>/

    The EXTTSK keyword is used to increase the amount of memory by
    4000 words. The partition in which the task will reside is
    assumed to be system-controlled.

4.4 MEMORY ALLOCATION FILE

The memory allocation file lists information about the allocation of
task memory and the resolution of global symbols. Optionally, a global
cross reference list may be appended to the file by means of the /CR
switch.

4.4.1 Memory Allocation File Format

In the discussion of task memory allocation, the following example was
used:

        >TKB IMG1,MP1=IN1,IN2,IN3

The requested memory allocation file, MP1, is shown in Figure 4-1 for
a mapped system and in Figure 4-2 for an unmapped system. In the
mapped system, the task is bound to virtual address zero and can be
relocated by the mapping hardware into various partitions. In the
unmapped system, the task is bound to physical address 50100, the base
address of the default partition 'GEN'.

The memory allocation file header contains information that identifies
Task Builder, the task, and the task-build time.

The segment description gives memory limits, identification, and
attributes.   The task IMG1.TSK has a read/write memory allocation of
1744 bytes (that is, the header, the stack, and p-sections A and B)
and a read-only memory allocation of 220 bytes (p-section C).

The PROGRAM SECTION ALLOCATION SYNOPSIS shows the placement and size
of all p-sections.

The file contents section lists the input files, the p-sections that
make up the file, and the global symbols that are defined in the
p-sections. Undefined global symbols are listed following   the
absolute p-section and summarized at the end of the listing.

CHAPTER 4. MEMORY ALLOCATION

```
FILE IMG1.TSK;1 MEMORY ALLOCATION MAP
THIS ALLOCATION WAS DONE ON 25-SEP-74
AT 14:15 BY TASK BUILDER VERSION M08


*** ROOT SEGMENT: IN1


R/W MEM  LIMITS: 000000 001743 001744
R-O MEM  LIMITS: 001744 002163 000220
STACK    LIMITS: 000204 001203 001000
DISK BLK LIMITS: 000002 000004 000003
IDENTIFICATION : 00
TASK ATTRIBUTES: NC


PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 001204 001204 000000
<A     >: 001204 001503 000300
<B     >: 001504 001723 000220
<C     >: 001744 002163 000220
<$$$   >: 001724 001743 000020
<. ABS.>: 000000 000000 000000


*** FILE: IN1.OBJ;1  TITLE: .MAIN.  IDENT:

<. ABS.>: 000000 000000 000000

>>>>>>>>>>>> UNDEFINED REFERENCE: C1


<B     >: 001504 001603 000100

   B1     001506-R  B2     001506-R

<A     >: 001204 001503 000300


<C     >: 001744 002113 000150
```

Figure 4-1
Memory Allocation File for IMG1.TSK on a Mapped System

```
*** FILE: IN2.OBJ;1  TITLE: .MAIN.  IDENT:

<A      >: 001204 001453 000250

    A1      001210-R

<B      >: 001604 001723 000120

    B1      001506-R


*** FILE: IN3.OBJ;3  TITLE: .MAIN.  IDENT:

<C      >: 002114 002163 000050


*** FILE: LBR1.OLB;1  TITLE: L1      IDENT: 00

<. BLK.>: 001204 001204 000000

    L1      001204-R


*** FILE: SYSLIB.OLB;1  TITLE: XXX      IDENT: 00

<$$$    >: 001724 001743 000020

    XXX     001724-R



***********

UNDEFINED REFERENCES:

    C1
```

Figure 4-1 (Cont.)
Memory Allocation File for IMG1.TSK on a Mapped System

```
          FILE IMG2.TSK;1 MEMORY ALLOCATION MAP
          THIS ALLOCATION WAS DONE ON 25-SEP-74
          AT 14:15 BY TASK BUILDER VERSION M08


          *** ROOT SEGMENT: IN1


          R/W MEM  LIMITS: 050100 052043 001744
          R-O MEM  LIMITS: 052044 052263 000220
          STACK    LIMITS: 050304 051303 001000
          DISK BLK LIMITS: 000002 000004 000003
          IDENTIFICATION : 00
          TASK ATTRIBUTES: NC


          PROGRAM SECTION ALLOCATION SYNOPSIS:

          <. BLK.>: 051304 051304 000000
          <A    >: 051304 051603 000300
          <B    >: 051604 052023 000220
          <C    >: 052044 052263 000220
          <$$$  >: 052024 052043 000020
          <. ABS.>: 000000 000000 000000


          *** FILE: IN1.OBJ;1  TITLE: .MAIN.  IDENT:

          <. ABS.>: 000000 000000 000000

          >>>>>>>>>>>> UNDEFINED REFERENCE: C1


          <B    >: 051604 051703 000100

             B1      051606-R  B2      051606-R

          <A    >: 051304 051603 000300


          <C    >: 052044 052213 000150
```

Figure 4-2
Memory Allocation File for IMG1.TSK on an Unmapped System

```
*** FILE: IN2.OBJ;1  TITLE: .MAIN.  IDENT:
<A     >: 051304 051553 000250
    A1     051310-R
<B     >: 051704 052023 000120
    B1     051606-R


*** FILE: IN3.OBJ;3  TITLE: .MAIN.  IDENT:
<C     >: 052214 052263 000050


*** FILE: LBR1.OLB;1  TITLE: L1     IDENT: 00
<. BLK.>: 051304 051304 000000
    L1     051304-R

*** FILE: SYSLIB.OLB;1  TITLE: XXX     IDENT: 00
<$$$   >: 052024 052043 000020
    XXX    052024-R



************
UNDEFINED REFERENCES:
    C1
```

Figure 4-2 (Cont.)
Memory Allocation File for IMG1.TSK on an Unmapped System

4.4.2 Global Cross-Reference Format

In addition to the memory allocation information described, the user
may also request that a listing of all global symbols, accompanied
by the name of each referencing module, be appended to the memory
allocation file described above. A listing of this type, termed a
global cross-reference, is frequently useful when debugging and
maintaining a task that consists of many modules.

Cross-reference processing is performed by a separate task that is
invoked by including the CR switch in the memory allocation file
specification as shown in the following example.

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

A portion of the cross-reference output is shown in Figure 4-3.

The page header contains the name of the memory allocation file, along
with the originating task (TKB), and the time and date the file was
created. Cross-reference information consisting of the global symbol
name, value, relocatability (-R), and the name of each referencing
module is shown in the body of the listing.

For this unoverlayed task, the name of the module containing the
symbol definition is preceded by a #. A complete summary of the
cross-reference format is included in section 4.4.3. The reader
should consult Appendix H for detailed information on cross-reference
processing and data formats.

4.4.3 Structure of the Memory Allocation File

The structure of the memory allocation file can be described as
follows:

1. The memory allocation file consists of the following sequence
    of items:

        heading
        segment description
        program section allocation synopsis
        file contents description
        undefined references summary

    If the /CR switch was used to request a global cross-reference,
    the following items are added to the memory allocation file:

        cross-reference page header
        cross-reference list entries

    Each of the above named items is defined in 2 through 9.

2. The heading gives the time and date of the task-build in the
    following form:

        FILE task-image-file-name MEMORY ALLOCATION MAP
        THIS ALLOCATION WAS DONE ON date
        AT time BY TASK BUILDER VERSION version-no.

```
SYMBOL      VALUE      REFERENCES...

$OTSV       000052     $CLOSE      $EOL        $ERRPT      $FIO        $INITI
                       $ISNLS      $OTI        $RETS       $STPPA      $VTRAN
$OTSVA      016206-R  # $OTV        .MAIN.
$PUTRE      011662-R   $IFW       # $PUTRE
$RLCB       027034-R  # RQLCB       RSTFDB
$RQCB       027136-R   OPFNB      # RQLCB
$R50        011714-R   $ERRPT     # $R50
$SAVRG      027274-R   RQLCB      # SAVRG
$SBR        001516-R  # $FADD
$SEQC       016206-R  # $OTV
$SST        016734-R  # $OTV
$SST0       004310-R  # $ERRPT      $OTV
$SST1       004316-R  # $ERRPT
$SST2       004330-R  # $ERRPT      $OTV
$SST3       004336-R  # $ERRPT      $OTV
$SST4       004344-R  # $ERRPT      $OTV
$SST5       004352-R  # $ERRPT      $OTV
$SST6       004436-R  # $ERRPT      $OTV
$SST7       004362-R  # $ERRPT      $OTV
$SVTK$      006002-R  # $ERRPT
.ASLUN      030436-R  # ASSLUN
.CLOSE      024702-R  # CLOSE       $CLOSE
.FATAL      031436-R  # COMMON      WAIT1
.FINIT      025140-R  # FINIT       $OTI
.FSRCB      024264-R  # FCSFSR      $OTV
.FSRPT      000050     ASSLUN      CREATE      FCSFSR      FINIT       OPFNB
                       PARD1       RSTFDB      WAITI       XQIO1
.GTDID      025244-R  # GETDID      $OPEN
.MBFCT      024364-R  # FCSFSR
.MOLUN      016214-R  # $OTV
.NLUNS      016212-R  # $OTV
.OPFNB      025262-R  # OPFNB       $OPEN
.PPASC      032604-R   PARD1      # PPNASC
.PPR50      032132-R   DIDFND     # PPNR50
.PUTSQ      027320-R  # PUTSQ       $PUTRE
.SAVR1      030402-R   ASSLUN      CKALOC      CLOSE       FINIT       GETDID
                       OPFNB       PUTSQ      # SAVR1      $ERRPT      $OPEN
..ALC1      034012-R  # CKALOC
..ALOC      033742-R  # CKALOC      WTWAIT
..ALUN      030442-R  # ASSLUN      GETDI       OPFNB
..BDRC      030632-R  # BDBREC      OPFNB       RDWAIT      WTWAIT
..BKRG      030714-R  # BKRG        CLOSE       RDWAIT      WTWAIT
..CREA      030734-R  # CREATE      OPFNB
..DEL1      031122-R  # DEL         OPFNB
..DID       032216-R   DIDFND     # DIFND
..DIDF      032076-R  # DIDFND      PARDID
..EFCK      031162-R  # EOFCHK      PUTSQ
..EFC1      031170-R  # EOFCHK
..ENTR      031336-R   DEL        # DIRECT      OPFNB
..EXTD      034042-R  # CKALOC
..EXT1      034116-R  # CKALOC
..FCSX      031424-R   CLOSE      # COMMON      CREATE      OPFNB       PUTSQ
                       WAITI       WATSET      WTWAIT
..FIND      031352-R   DIFND      # DIRECT      OPFNB
..FINI      025150-R  # FINIT       OPFNB
..GTDI      031440-R  # GETDI       GETDID
```

Figure 4-3
Cross Reference Listing for MP1.MAP

3. The segment description consists of the following sequence of items:

        ***SEGMENT segment-name

        R/W MEM   LIMITS: start-addr   end-addr   length
        R/O MEM   LIMITS: start-addr   end-addr   length
        STACK   LIMITS: start-addr   end-addr   length
        DISK BLK   LIMITS: start-blk   end-blk   blk-length
        IDENTIFICATION: name
        ODT XFR ADDRESS: address
        PRG XFR ADDRESS: address
        TASK ATTRIBUTES: attr-1 ...   attr-n

    Any line in the sequence is omitted if it does not apply to a given task image.

    The constructs in this sequence are defined in paragraph 7.

4. The program section allocation synopsis has the form:

     p-sect-name-1 start-addr end-addr length
     ...

    If the SQ switch is applied, the p-sect-names are listed in input order; otherwise p-sect-names are listed in alphabetical order. Since p-sections are allocated according to their access-code, the alphabetical listing is not necessarily sequential.

5. The file contents description contains an entry for each input file in the form:

        ***FILE filename   TITLE title-name   IDENT ident-name

    <. ABS.>   start-addr end-addr length
                        g-name-1 value g-name-2 value...
    >>>>>   UNDEFINED REFERENCE g-name-n
    <p-sect-name-1>   start-addr end-addr length...
                        g-name-1 value-R g-name-2 value-R...
    . BLK.   start-addr end-addr length
                        g-name-1 value g-name-2 value

    The absolute global symbols are listed in the p-section named . ABS, which is collated first. The blank p-section . BLK. is collated last in the listing.

6. The undefined references summary has the form:

        ****
        UNDEFINED   REFERENCES

        g-name-1
        ...

7. The Cross-Reference Page header gives the name of the memory
    allocation file, the originating task (TKB), the date and time
    the memory allocation file was created and the cross reference
    page number, in the following format:

    GLOBAL CROSS REFERENCE

    FILE: map-file-name **CREATED BY TKB ON date AT time**

    SYMBOL   **VALUE**   **REFERENCES ...**

8. The cross-reference list contains an alphabetic listing of
    each global symbol along with its value and the name of each
    referencing module. When a symbol is defined in several
    segments within an overlay structure, the last defined value
    is printed. Similarly, if a module is loaded in several
    segments within the structure, the module name will be dis-
    played more than once within each entry.

    Entries have the form:

    g-name-1   value (-R)   title-name-1.....
                               title-name-n

    g-name-2   value (-R)   title-name-1.....
                               title-name-n

    The suffix '-R' is appended to the value if the symbol is
    relocatable.

    **Prefix symbols accompanying each module name define the type
    of reference as follows:**

    Prefix Symbol   **Reference Type**

       blank   Module contains a reference that is resolved
                      in the same segment or in a segment towards
                      the root.

         ↑   Module contains a reference that is resolved
                      directly in a segment away from the root or
                      in a co-tree.

         @   Module contains a reference that is resolved
                      through an autoload vector.

         #   Module contains a non-autoloadable
                      definition.

         *   Module contains an autoloadable definition.

                               <u>NOTE</u>

    The reader should consult the glossary and
    Chapter 5 for a discussion of unfamiliar
    terms.

CHAPTER 4. MEMORY ALLOCATION

9. The remaining constructs are defined as follows:

segment-name   is the name of the segment.

start-addr   is the first storage address in octal byte format.

end-addr   is the last storage address in octal byte format.

length   is the number (in octal) of bytes occupied.

start-blk   is the relative block number (in octal) for the starting disk location.

end-blk   is the last relative block number for the disk allocation.

blk-length   is the number (in octal) of blocks occupied.

address   is a byte address (in octal).

name   is the name attached to the first non-blank .IDENT entry encountered.

attr   is an attribute code that applies to the task image. The list of codes printed is:

NC	Task is not checkpointable
FP	Task uses the floating point processor
DA	Task includes the standard debugging aid SY0:[1,1]ODT.OBJ
PI	Task contains only position independent code and data
PM	Post mortem dump will be produced in the event of an abnormal task termination
PR	Task is privileged
TR	Task initial PS word has T-bit enabled
EA	Task uses KE-11A extended arithmetic element
AC	Task is an ancillary control processor
NH	Task does not contain a header

p-sect-name	is the name of a p-section.
file-name	is the name of an input object file.
title-name	is the name of the first non-blank .TITLE encountered.
ident-name	is the name of the first non-blank .IDENT encountered.
g-name	is the name of a global symbol.

4.5 EXAMPLE: CALC;1 MAP

The first run of CALC, discussed in Chapter 2, produces the memory allocation file shown in Figure 4-4. This memory allocation file contains all the parts described in this chapter. For inclusion in the manual, the map was truncated after the second entry in the file contents description. The truncated entries are described in general terms in the section on the file contents.

4.5.1 Heading

The heading contains the date and time the example was run.

4.5.2 Segment Description

The task code and data for CALC;1 occupies 37024 octal bytes of read-write memory. After examining the map, the user decided to build the next version of CALC for a partition larger than the default partition GEN, which on the system he is using consists of 40000 bytes.

There is no entry for read-only memory because this task does not have any read-only p-sections.

The stack occupies 1000 bytes because the user did not change the default stack size.

The identification $FORT is assigned by the FORTRAN compiler to all main programs.

The program transfer address is the virtual address 1210 (that is, the starting address of the program.)

The task has the attribute NC (not checkpointable).

4.5.3 Program Section Allocation Synopsis

The blank program section '. BLK.' contains the object code produced from the translation of the modules for CALC;1. The code begins at virtual address 1210, ends at virtual address 26127, and occupies 24720 bytes.

The program section 'DTA ' is the memory allocation reserved for the common block DTA.

The remaining program sections are storage regions required by the FORTRAN object time system (OTS) and File Control Services (FCS), which were called in by the FORTRAN compiler to perform services for the FORTRAN program.

4.5.4 File Contents Description

The file contents description lists for each file the program sections that the file contributed to the segment. In CALC;1 there are three input files, RDIN.OBJ, PROC1.OBJ, and RPRT.OBJ. In addition to these files, the library file SYSLIB.OLB is required to contribute the FORTRAN run-time routines.

The input file RDIN.OBJ contains three p-sections; namely, '.$$$$.', '. BLK.', and 'DTA '. The p-section '.$$$$.' is the common block reservation for unnamed or blank common. Since this task does not use blank common, the storage reservation is zero. The p-section '. BLK.' contains the code for RDIN.OBJ, starts at virtual address 1210, and occupies 110 bytes. 'DTA ' is the p-section containing the common block DTA. This section starts at virtual address 26130, and occupies 1442 bytes.

The input file, PROC1.OBJ, also contains three p-sections; namely, '.$$$$.', '. BLK.', and 'DTA '. The p-section '. BLK.' contains the code for PROC1 and the definition for global symbol 'PROC1', the name of the subroutine.

The map reproduced below does not contain the modules contributed by the library file SYSLIB.OLB.

CHAPTER 4. MEMORY ALLOCATION

FILE CALC.TSK;1 MEMORY ALLOCATION MAP
THIS ALLOCATION WAS DONE ON 25-JUL-74
AT 14:53 BY TASK BUILDER VERSION M06

*** ROOT SEGMENT: RDIN

R/W MEM LIMITS: 000000 037023 037024
STACK   LIMITS: 000210 001207 001000
DISK BLK LIMITS: 000002 000041 000040
IDENTIFICATION : $FORT
PRG XFR ADDRESS: 001210
TASK ATTRIBUTES: NC

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 001210 026127 024720
<DTA   >: 026130 027571 001442
<$$AOTS>: 027572 030347 000556
<$$DEVT>: 030350 031557 001210
<$$FSR1>: 031560 035657 004100
<$$FSR2>: 035660 035761 000102
<$$IOB1>: 035762 036165 000204
<$$IOB2>: 036166 036166 000000
<$$OBF1>: 036166 036275 000110
<$$OBF2>: 036276 036276 000000
<$$RESL>: 036276 037023 000526
<. ABS.>: 000000 000000 000000
<.$$$.>: 037024 037024 000000

*** FILE: RDIN.OBJ:1 TITLE: .MAIN. IDENT: $FORT

<.$$$.>: 037024 037024 000000
<DTA   >: 026130 027571 001442
<. BLK.>: 001210 001317 000110

*** FILF: PROC1.OBJ:1 TITLE: PROC1 IDENT: $FCRTS

<. BLK.>: 001320 001320 000000

    PROC1 001320-R

<.$$$.>: 037024 037024 000000

<DTA   >: 026130 027571 001442

<. BLK.>: 001320 003003 001464

Figure 4-4
Memory Allocation File for CALC;1
(Mapped System)

4.6 EXAMPLE: CALC;2 MAP

In the example CALC;2 in Chapter 3, the user added some code to RDIN, and entered two options during option input:

   o ACTFIL=1   to eliminate the three active file buffers not needed by CALC.

   o PAR=PAR14K to direct the Task Builder to use a larger partition for CALC since the user intends to expand the task.

The memory allocation file shown in Figure 4-5 reflects these changes:

```
FILE CALC.TASK;2 MEMORY ALLOCATION MAP
THIS ALLOCATION WAS DONE ON 25-JUL-74
AT 15:07 BY TASK BUILDER VERSION M06

*** ROOT SEGMENT: RDIN

R/W MEM  LIMITS: 000000 033777 034000
STACK    LIMITS: 000210 001207 001000
DISK BLK LIMITS: 000002 000035 000034
IDENTIFICATION : $FORT
PRG XFR ADDRESS: 001210
TASK ATTRIBUTES: NC

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 001210 026163 024754
<DTA   >: 026164 027625 001442
<$$AOTS>: 027626 030403 000556
<$$DEVT>: 030404 031613 001210
<$$FSRT>: 031614 032633 001020
<$$FSR2>: 032634 032735 000102
<$$IOB1>: 032736 033141 000204
<$$IOB2>: 033142 033142 000000
<$$OBF1>: 033142 033251 000110
<$$OBF2>: 033252 033252 000000
<$$RESL>: 033252 033777 000526
<. ABS.>: 000000 000000 000000
<.$$$.>: 034000 034000 000000
```

Figure 4-5
Memory Allocation File for CALC;2
(Mapped System)

CHAPTER 4. MEMORY ALLOCATION

Because of the additional logic in the program RDIN, the task code allocation increased from 24720 in CALC;1 to 24754 in CALC;2.

Because the ACTFIL keyword was used, the File Storage Region buffer pool, $$FSR1, decreased from 4100 in CALC;1 to 1020 in CALC;2.

	CALC;1	CALC;2	Difference
task code	24720	24754	+ 34
$$FSR1	4100	1020	−3060
			─────
			−3024

The use of the ACTFIL keyword saved 3060 bytes. The net saving of 3024 bytes, when added to the memory requirements for CALC;2, gives the memory requirement for CALC;1

CALC;2	34000
DIFF	3024
	─────
CALC;1	37024

CHAPTER 5

OVERLAY CAPABILITY

This chapter describes the use of the overlay capability to reduce the
memory requirements of a task. The concept of tree structured
overlays is introduced and a language for representing this structure
is defined. Examples are given that illustrate the use of the
language and the allocation of memory for an overlayed task.

5.1 OVERLAY DESCRIPTION

To create an overlay structure, the user divides his task into a
series of segments; specifically:

- a single root segment, which is always in memory, and

- any number of overlay segments, which share memory with one
  another.

A segment consists of a set of modules and p-sections that can be
loaded by a single disk access. Segments that overlay each other must
be logically independent. Two segments are said to be logically
independent if the components of one segment do not reference and are
not referenced by any of the components of the other segment.

When the user defines an overlay structure, he must consider the
general flow of control within his task in addition to the logical
independence of the overlay segments. Dividing a task into overlays
saves space, but introduces the overhead activity of loading these
segments into memory as they are needed. The programmer must make
optimization decisions in constructing the overlay just as he does in
writing the programs.

There are several large classes of tasks that can be handled
effectively by an overlay structure. A task that moves sequentially
through a set of modules is well suited to the use of an overlay
structure.   A task which selects one of a set of modules according to
the value of an item of input data is also well suited to an overlay
structure.

CHAPTER 5. OVERLAY CAPABILITY

5.1.1 Overlay Structure

Consider a task, TK1, which consists of four input files. Each input file consists of a single module of the same name as the file. The task is built by the command:

>TKB TK1=CNTRL,A,B,C

Suppose the user knows that the modules A, B, and C are logically independent. In this example:

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.

The user can define an overlay structure in which A, B, and C are overlay segments that occupy the same storage. Suppose further that the flow of control for the task is as follows:

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 and A returns to CNTRL.

The loading of overlays occurs only four times during the execution of the task. Therefore, the user can reduce the memory requirements of the task without unduly increasing the overhead activity.

Consider the effect of introducing an overlay structure on the allocation of memory for the task. Suppose the lengths of the modules are as follows:

CNTRL	10000 bytes
A	6000 bytes
B	5000 bytes
C	1200 bytes

The memory allocation produced as a result of building the task as a single segment on a system with memory mapping hardware is as follows:

C	- 24200
B	- 23000
	- 15000
A	- 10000
CNTRL	- 0

The memory allocation for a single-segment task requires 24200 bytes.

CHAPTER 5. OVERLAY CAPABILITY

The memory allocation produced as a result of using the overlay capability and building a multi-segment task is as follows:

The multi-segment task requires 16000 bytes. In addition to the module storage, additional storage is required for overhead connected with handling the overlay structure. This overhead is described later and illustrated in the example CALC.

Observe that the amount of storage required for the task is determined by the length of the root segment and the length of the longest overlay segment. Overlay segments A and B in this representation are much longer than overlay segment C. If the user can divide A and B into sets of logically independent modules, he can further reduce the storage requirements of his task. Suppose he divides A into a control program A0 and two overlays A1 and A2. A2 is then further divided into the main part A2 and two overlays A21 and A22. Similarly, he divides the B overlay into a control module B0 and two overlays B1 and B2.

The memory allocation for the task produced by the additional overlays defined for A and B is given by the diagram:

As a single-segment task, TK1 required 24200 bytes of storage. The first overlay structure reduced the requirement by 6200 bytes. The second overlay structure further reduced the storage requirement by 2200 bytes.

Observe that a vertical line can be drawn through the memory diagram to indicate a state of memory. In the diagram given here, the leftmost such line gives memory when CNTRL, A0, and A1 are loaded: the next such line gives memory when CNTRL, A0, A2, and A21 are loaded: and so on.

Observe also that a horizontal line can be drawn through the memory diagram to indicate segments that share the same storage. In the given diagram, the uppermost such line gives A1, A21, A22, B1, B2 and C, all of which can use the same memory; the next such line gives A1, A2, B1, B2, and C; and so on.

CHAPTER 5. OVERLAY CAPABILITY

5.1.2 Overlay Tree

The Task Builder provides a language for representing an overlay structure consisting of one or more trees.

A single overlay tree is described first and then the procedure for describing multiple overlay trees is given.

The memory allocation for the previous example can be represented by the single overlay tree shown below:

```
                A21      A22
                 |_____|
                     |
        A1          A2                 B1     B2
         |_____|                  |_____|
              |                            |
             A0                           B0              C
              |_____|_____|
                                 |
                               CNTRL
```
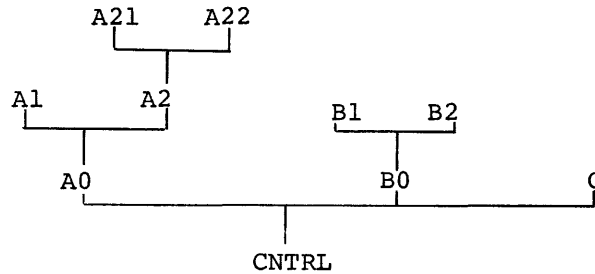
The tree has a root, CNTRL, and three main branches, A0, B0, and C. The tree has six leaves, A1, A21, A22, B1, B2, and C.

The tree has as many paths as it has leaves. The path down is defined from the leaf to the root, for example:

        A21-A2-A0-CNTRL

The path up is defined from the root to the leaf, for example:

        CNTRL-B0-B1.

Understanding the tree and its paths is important to the understanding of the overlay loading mechanism and the resolution of global symbols.

5.1.2.1 Loading Mechanism - Modules can call other modules that exist on the same path. The module CNTRL is common to every path of the tree and, therefore, can call and be called by every module in the tree. The module A2 can call the modules A21, A22, A0, and CNTRL; but A2 can not call A1, B1, B2, B0 or C.

When a module calls a module in another overlay segment, the overlay segment must be in memory or must be brought into memory. The methods for loading overlays are described in the next chapter.

5.1.2.2 Resolution of Global Symbols in a Multi-segment Task - The Task Builder performs the same activities in resolving global symbols for a multi-segment task as it does for a single segment task. The rules defined in Chapter 4 for the resolution of global symbols in a single segment task still apply, but the scope of the global symbols is altered by the overlay structure.

CHAPTER 5. OVERLAY CAPABILITY

In a single segment task, any global definition can be referenced by any module. In a multi-segment task, a module can only reference a global symbol that is defined on a path that passes through the segment to which the module belongs.

In a single segment task, if two global symbols with the same name are defined, the symbols are multiply defined and an error message is produced. In a multi-segment task two global symbols can be defined with the same name as long as the definitions are on separate paths. A reference is said to be ambiguous if there are multiple definitions on common paths to which the reference could be resolved.

Consider the task TK1 and the global symbols Q, R, S, and T.

```
           A21       A22
          T(def)    R(ref)
            |       Q(ref)
            └─────────┘
                 |
   A1        A2                B1                    B2
  Q(ref)    R(def)            Q(ref)
  R(ref)
    └─────────┘                └──────────────────────┘
         |                                |
        A0                               B0                        C
      Q(def)                           Q(def)                      |
      S(def)                           S(def)                      |
      T(def)                             |                         |
        └────────────────────────────────────────────────────────┘
                              |
                           CNTRL
                           S(ref)
```
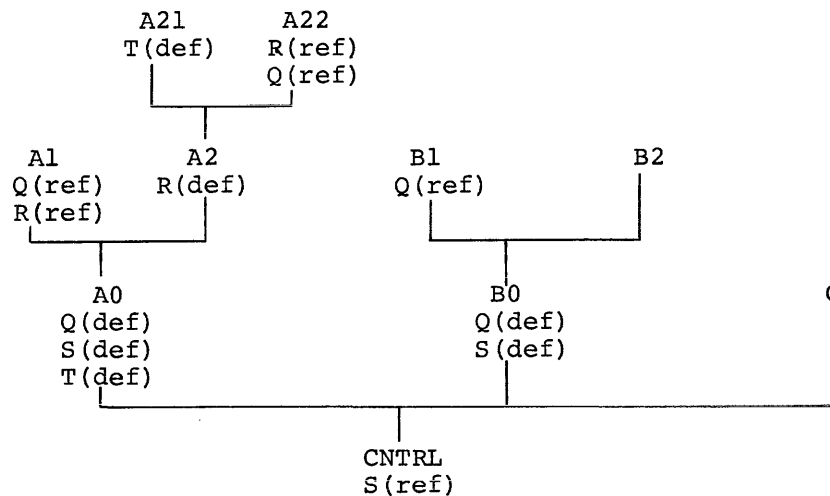
The following remarks apply to the use of each of the symbols shown in the diagram:

Q   The global symbol Q is defined in the segment A0 and in the segment B0. The reference to Q in segment A22 and the reference to Q in segment A1 are resolved to the definition in A0. The reference to Q in B1 is resolved to refer to the definition of B0. The two definitions of Q are distinct in all respects and occupy different memory allocations.

R   The global symbol R is defined in the segment A2. The reference to R in A22 is resolved to 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 a path through A1.

S   The global symbol S is defined in A0 and B0. References to S
    from A1, A21 or A22 are resolved to the definition in A0 and
    references to S in B1 and B2 are resolved to 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. S is ambiguously defined.

T   The global symbol T is defined in A21 and A0. Since there is
    a single path through the two definitions (CNTRL-A0-A2-A21),
    the global symbol T is multiply defined.

5.1.2.3 Resolution of P-sections in a Multi-segment Task - A
p-section has an attribute that indicates whether the p-section is
local (LCL) to the segment in which it is defined or of global (GBL)
extent.

Local p-sections with the same name can appear in any number of
segments. Storage is allocated for each local p-section in the
segment in which it is declared. Global p-sections of the same name,
however, must be resolved by the Task Builder.

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

FORTRAN common blocks are translated into global p-sections with the
overlay attribute. Suppose that in the task TK1 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 which contains A21.

However, if the programs A0 and B0 use a common block COMAB, the Task
Builder allocates the storage for COMAB in both the segment which
contains A0 and the segment which contains B0. A0 and B0 can not
communicate through COMAB. When the overlay segment containing B0 is
loaded, any data stored in COMAB by A0 is lost.

The tree for the task TK1 including the allocation of the common
blocks COMA and COMAB is:
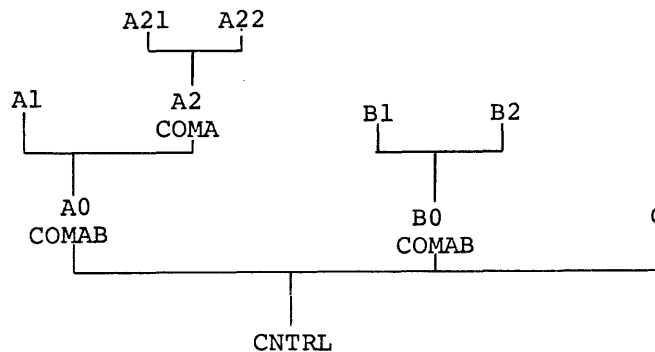
CHAPTER 5. OVERLAY CAPABILITY

The allocation of p-sections can be specified by the user. If A0 and B0 need to share the contents of COMAB, the user can force the allocation of this p-section into the root segment by the use of the .PSECT directive, described in Section 5.1.3.4.

5.1.3 Overlay Description Language (ODL)

The Task Builder provides a language that allows the user to describe the overlay structure. The overlay description language (ODL) contains five directives by which the user can describe the overlay structure of his task.

An overlay description consists of a series of ODL directives. There must be one .ROOT directive and one .END directive. The .ROOT directive tells the Task Builder where to start building the tree and the .END directive tells the Task Builder where the input ends.

5.1.3.1 .ROOT and .END Directives - The arguments of the ROOT directive make use of two operators to express concatenation and overlaying. A pair of parentheses delimits a group of segments that start at the same location in memory. The maximum number of nested parentheses cannot exceed 32.

- The operator dash '-' indicates the concatenation of storage. For example, 'X-Y' means that the memory allocation must contain X and Y simultaneously. So X and Y are allocated in sequence.

- The operator comma ',' appearing within parentheses indicates the overlaying of storage. For example, 'Y,Z' means that memory can contain either Y or Z. Therefore Y and Z are share storage.

  This operator is also used to define multiple tree structures, as described in 5.1.4.

Consider the overlay description lanquage directives:

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

These directives describe the following tree and its corresponding memory diagram:
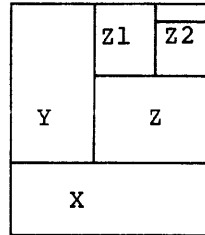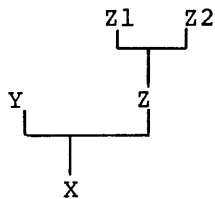
To create the overlay description for the task TK1 described earlier in this chapter, the user creates a file TFIL that contains the directives:

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

To build the task with that overlay structure, the user types:

```
>TKB TK1=TFIL/MP
```

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

5.1.3.2 .FCTR Directive - The tree that represents the overlay structure can be complicated. The overlay description language includes another directive, .FCTR, which allows the user to build large trees and represent them systematically.

The .FCTR directive allows the user to extend the tree description beyond a single line. Since there can be only one .ROOT directive, the .FCTR directive must be used if the tree definition exceeds one line. The .FCTR directive, however, can also be used to introduce clarity in the representation of the tree.

CHAPTER 5. OVERLAY CAPABILITY

The maximum number of nested .FCTR levels is 32.

To simplify the tree given in the file TFIL the .FCTR directive is introduced into the overlay description language as follows:

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

The label 'BFCTR', is used in the .ROOT directive to designate the argument of the .FCTR directive, '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. The user can modify TFIL as follow:

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

The decision to use the .FCTR directive is based on considerations of space and style.

5.1.3.3 .NAME Directive - The .NAME directive allows a segment name to be defined and included at any appropriate point in the tree. The defined name must be unique with respect to filenames, p-section names, .FCTR labels and other segment names that are used in the overlay description.

The .NAME directive is used to uniquely identify a segment that is to be loaded into memory by means of the Manual Load Method described in Chapter 6.

Suppose that, in the definition of the tree for TK1, the user wants to give a name to every main branch of the tree. He defines three names and includes these new names in the overlay description for the tree. TFIL is modified as follows:

```
        .NAME BRNCH1
        .NAME BRNCH2
        .NAME BRNCH3
        .ROOT CNTRL-(BRNCH1-AFCTR,BRNCH2-BFCTR,BRNCH3-C)
AFCTR:  .FCTR AO-(A1,A2-(A21,A22))
BFCTR:  .FCTR BO-(B1,B2)
        .END
```

5.1.3.4 .PSECT Directive - The .PSECT directive allows the placement
of a global p-section to be specified directly. The name of the
p-section and its attributes are given in the .PSECT directive. Then,
the name can be used explicitly in the definition of the tree to
indicate the segment in which the p-section is to be allocated.

Suppose the user encountered a problem in communication resulting from
the overlay description for TKl.   The user was careful about the
logical independence of the modules in the overlay segment, but he
failed to take into account the logical independence requirement of
multiple executions of the same overlay segment.

The flow of the task TKl, as described earlier in this chapter, can be
summarized in the following way.   CNTRL calls each of the overlay
segments and the overlay segment returns to CNTRL in the following
order: A,B,C,A. The module A is executed twice. The overlay segment
containing A must be reloaded for the second execution of A.

The module A uses the common block DATA3. The Task Builder allocates
DATA3 in the overlay segment containing A. The first execution of A
stores some results in DATA3. The second execution of A requires
these values. In the present overlay description, 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.

The use of a .PSECT directive forces the allocation of DATA3 into the
root segment to permit the two executions of A to communicate. TFIL
is modified as follows:

```
          .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
```

The attributes RW,GBL,REL and OVR are described in Chapter 4.

5.1.4 Multiple Tree Structures

The Task Builder allows the specification of more than one tree within
the overlay structure. A structure containing multiple trees has the
following properties:

1. Storage is not shared among trees.   The total storage
    required is the sum of the longest path on each tree.

2. Each path in a tree is common to all paths on every other
    tree.

These properties allow modules, that would otherwise have to reside in
the root segment, to be contained in an overlay tree.

CHAPTER 5. OVERLAY CAPABILITY

Such overlay trees within the structure consist of a main tree and one or more co-trees. The root segment of the main tree is loaded by the monitor when the task is made active while segments within each co-tree are loaded through calls to the overlay runtime system.

Except for the above distinction, all overlay trees have identical characteristics. That is, each tree must have a root segment and possibly one or more overlay segments.

The following paragraphs describe the procedure for specifying multiple trees in the overlay description language and illustrate the use of co-trees to reduce the memory required by a task.

5.1.4.1 Defining a Multiple Tree Structure - Multiple tree structures are specified within the overlay description language by extending the function of the comma ',' operator. As previously discussed, this operator, when included within parentheses, defines a pair of segments that share storage. The inclusion of the comma operator outside all parentheses delimits overlay trees. The first overlay tree thus defined is the main tree. Subsequent trees are co-trees.

Consider the following:

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

Two overlay trees are specified. A main tree containing the root segment X0 and three overlay segments and a co-tree consisting of root segment Y0 and two overlay segments. The Executive loads segment X0 into memory when the task is activated. The task then loads the remaining segments through calls to the overlay runtime system.

A co-tree must have a root segment to establish linkages to the overlay segments within the co-tree. Logically, these root segments need not contain code or data. (Such modules can be resident in the main root). A segment of this type termed a 'null segment', may be created by means of the .NAME directive. The previous example is modified as shown below to include a null segment.

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

The null segment 'YNUL' is created, using the .NAME directive, and replaces the co-tree root that formerly contained Y0.OBJ. Y0 now resides in the main root.

CHAPTER 5. OVERLAY CAPABILITY

5.1.4.2 <u>Multiple Tree Example</u> - The following example illustrates the use of multiple trees to reduce the size of the task.

Suppose that in the task TK1, the root segment CNTRL consists of a small dispatching routine and two long modules, CNTRLX and CNTRLY. CNTRLX and CNTRLY are logically independent of each other, are approximately equal in length, and must access modules on all the paths of the main tree.

The user can define a co-tree for CNTRLX and CNTRLY and effect a saving in the storage required by the task. He modifies the overlay description in TFIL as follows:

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

The co-tree is defined at the 'zeroth' parenthesis level in the .ROOT directive. A co-tree must have a root segment, to establish linkages to the overlay segments within the co-tree. When no code or data logically belong in the root, the .NAME directive can be used to create a null root segment.

The tree for the task TK1 now is:
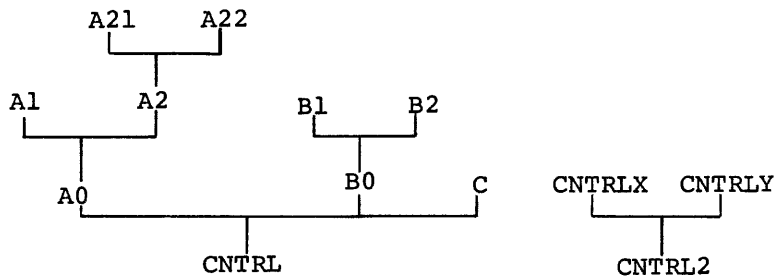
The corresponding memory diagram is:

5-12
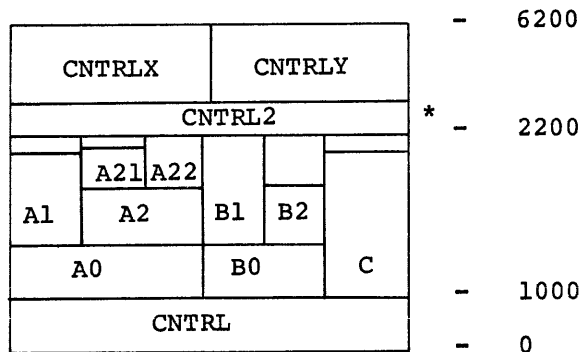
CHAPTER 5.   OVERLAY CAPABILITY

The specification of the co-tree decreases the storage allocation by 4000 bytes. CNTRLX and CNTRLY can still access modules on all the paths of the main tree. The only requirement imposed by the introduction of the co-tree is the logical independence of CNTRLX and CNTRLY.

Any number of co-trees can be defined. Additional co-trees can access all the modules in the main tree and in the other co-trees.

5.1.5 Overlay Core Image

The core image for a task with an overlay structure can be represented by the following diagram:

co-tree overlay segment
co-tree root segment
...
co-tree overlay segments
co-tree root segment
main tree overlay segments
main tree root segment
stack
header
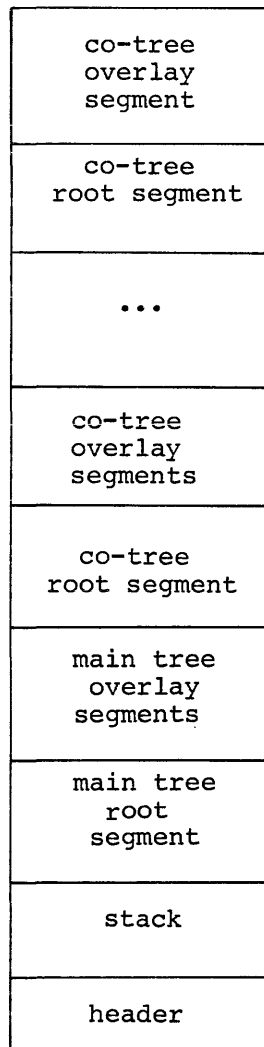
CHAPTER 5. OVERLAY CAPABILITY

The header and stack are described in Chapter 4.

The root segment of the main tree contains all the modules that are
resident in memory throughout the entire execution of the task, along
with the segment tables, and if the autoload loading method is used,
the autoload vectors.

```
┌──────────────────┐  ┐
│    autoload      │  │
│    vectors       │  │
├──────────────────┤  │
│                  │  │
│    segment       │  │  main tree
│    tables        │  │  root segment
│                  │  │
├──────────────────┤  │
│                  │  │
│    code and      │  │
│    data          │  │
│                  │  │
│                  │  │
└──────────────────┘  ┘
```
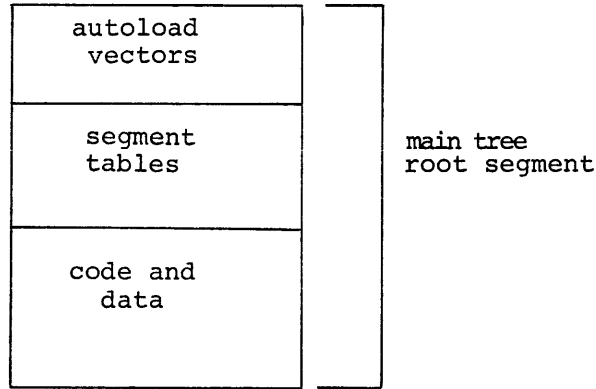
The segment table contains a segment descriptor for every segment in
the task. The descriptor contains information about the load address,
the length of the segment, and the tree linkages. The segment table
is described in detail in Appendix C.

Autoload vectors appear in every segment that calls modules in another
segment that is further from the root of the tree. Autoload vectors
are described in connection with loading mechanisms in Chapter 6 and
the detailed composition of the autoload vector is given in Appendix
C.

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.

```
┌──────────────────┐  ┐               ┐
│ autoload vectors │  │               │
├──────────────────┤  │ overlay       │
│  code and data   │  │ segment       │
│                  │  │               │
├──────────────────┤  ┘               │  overlay
│        •         │                  │
│        •         │                  │
│        •         │                  │
├──────────────────┤  ┐               │
│ autoload vectors │  │ overlay       │
├──────────────────┤  │ segment       │
│  code and data   │  │               │
│                  │  │               │
└──────────────────┘  ┘               ┘
```
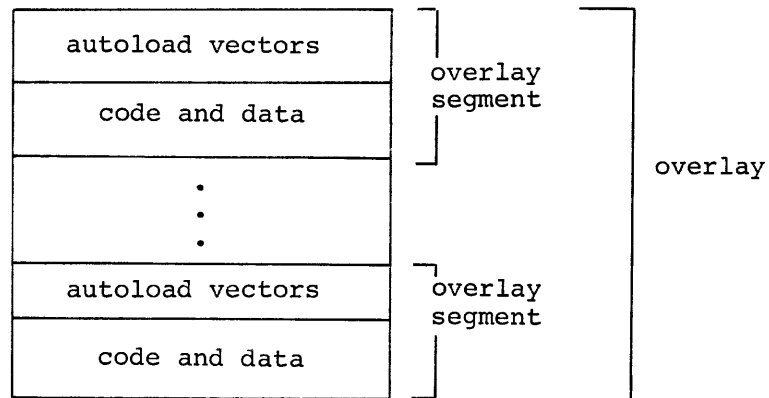
The co-tree overlay region consists of memory allocated for the
overlay segments of the co-trees.

The co-tree root segment contains the modules that, once loaded, must
remain resident in memory.

CHAPTER 5. OVERLAY CAPABILITY

5.2 EXAMPLE: CALC;3

The version of CALC introduced earlier is now ready for the addition
of two more data processing routines, PROC2 and PROC3. These new
algorithms are logically independent of each other and of PROC1.   The
third algorithm, PROC3, contains two independent routines SUB1 and
SUB2.

The user defines an overlay structure for CALC as follows:

```
                                        SUB1   SUB2
                                         |_____|
                                            |
        PROC1           PROC2            PROC3
         |_____|
                        |
                      RDIN
                      RPRT
```
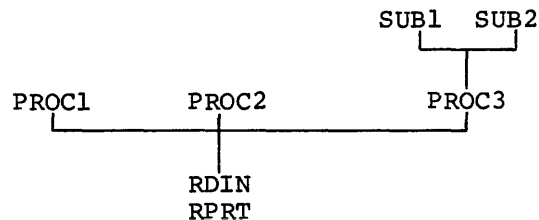
5.2.1 Defining the ODL File

The user constructs a file, CALTR, of ODL directives to represent the
tree for CALC, as follows:

```
        >EDI
        EDI>CALTR.ODL
        [CREATING NEW FILE]
        INPUT
                .ROOT  RDIN-RPRT-*(PROC1,PROC2,P3FCTR)
        P3FCTR:  .FCTR PROC2-(SUB1,SUB2)
                .END
        *EX
```

                                  NOTE

                The '*' in the ODL description is the
                autoload indicator and is described in
                Chapter 6.

CHAPTER 5. OVERLAY CAPABILITY

5.2.2 Building the Task

The user builds the task with the same options as in the example of Chapter 3. He replaces the names of the input files by a single filename that designates the file containing the overlay description:

```
>TKB
TKB>CALC;3,LP:/SH=CALTR/MP
ENTER OPTIONS:
TKB>PAR=PAR14K
TKB>ACTFIL=1
TKB>//
```

5.2.3 Memory Allocation File for CALC;3

The short memory allocation file for this multi-segment task consists of one page per segment. For convenience the pages are compressed in this manual. See Figure 5-1.

The memory diagram for CALC;3 is:

			- 36400
			-- 35724
		SUB1 | SUB2	
			-- 35310
PROC1	PROC2	PROC3	
segment tables and autoload vectors			- 33254
			-- 33012
FORTRAN buffers			
			-- 26220
DTA			
			-- 24556
RPRT RDIN			
			-- 1214
stack			
			-- 214
header			-- 0

If the user had not used an overlay structure for the task, the memory requirement of the task would have been:

```
ROOT      33012
PROC1      3124
PROC2      2304
PROC3      2034
SUB1        414
SUB2        404
          -----
          43516
```

CHAPTER 5. OVERLAY CAPABILITY

FILE CALC.TSK;3 MEMORY ALLOCATION MAP
THIS ALLOCATION WAS DONE ON 25-JUL-74
AT 15:36 BY TASK BUILDER VERSION M06

*** ROOT SEGMENT:RDIN

R/W MEM LIMITS: 000000 033253 033254
STACK   LIMITS: 000214 001213 001000
DISK BLD LIMITS: 000002 000035 000034
IDENTIFICATION : $FORT
PRG XFR ADDRESS: 001214
TASK ATTRIBUTES: NC

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 001214 024555 023342
<DTA   >: 024556 026217 001442
<$$ALER>: 026220 026243 000024
<$$AOTS>: 026244 027021 000556
<$$DEVT>: 027022 030231 001210
<$$FSR1>: 030232 031251 001020
<$$FSR2>: 031252 031353 000102
<$$IOB1>: 031354 031557 000204
<$$IOB2>: 031560 031560 000000
<$$OBF1>: 031560 031667 000110
<$$OBF2>: 031670 031670 000000
<$$OVDT>: 000000 000000 000000
<$$RESL>: 031670 033011 001122
<$$SGDF>: 000000 000000 000000
<. ABS.>: 000000 000000 000000
<.$$$.>: 033012 033012 000000

*** SEGMENT: PROC1

R/W MEM LIMITS: 033254 036377 003124
DISK BLK LIMITS: 000036 000041 000004

Figure 5-1
Memory Allocation File for CALC;3
(Mapped System)

CHAPTER 5. OVERLAY CAPABILITY

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 033254 034737 001464
<ADTA >: 034740 036377 001440

*** SEGMENT: PROC2

R/W MEM LIMITS: 033254 035557 002304
DISK BLK LIMITS: 000042 000044 000003

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 033254 034117 000644
<ADTA >: 034120 035557 001440

*** SEGMENT: PROC3

R/W MEM LIMITS: 033254 035307 002034
DISK BLK LIMITS: 000045 000047 000003

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 033254 033627 000354
<ADTA >: 033630 035267 001400

***SEGMENT: SUB1

R/W MEM LIMITS: 035310 035723 000414
DISK BLK LIMITS: 000050 000050 000001

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 035310 035723 000414

*** SEGMENT: SUB2

R/W MEM LIMITS: 035310 035713 000404
DISK BLK LIMITS: 000051 000051 000001

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 035310 035713 000404

Figure 5-1 (Cont.)
Memory Allocation File for CALC;3
(Mapped System)

5.3 EXAMPLE CALC;4

After examining the memory allocation file for CALC;3, the user observes that the Task Builder has allocated ADTA in the overlay segments PROC1, PROC2, and PROC3, since all of these segments are equidistant from the root.

The user knows, however, that these segments need to communicate with each other through ADTA. In the existing allocation, any values placed in ADTA by PROC1 are lost when PROC2 is loaded. Similarly, any values stored in ADTA by PROC2 are lost when PROC3 is loaded.

The user adds a .PSECT directive to the overlay description to force ADTA into the root segment so that PROC1, PROC2, and PROC3 can communicate with each other. He modifies CALTR as follows:

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

He builds the task as in CALC;3 and the resulting memory allocation file can be represented by the following diagram:

				-- 36400
				-- 35724
		SUB1	SUB2	-- 35310
PROC1	PROC2	PROC3		
				-- 34714
segment table and autoload vectors				-- 34452
FORTRAN buffers				-- 27660
DTA				-- 26216
ADTA				-- 24556
RPRT RDIN				
				-- 1214
stack				-- 214
header				-- 0

CHAPTER 5. OVERLAY CAPABILITY

FILE CALC.TSK;4 MEMORY ALLOCATION MAP
THIS ALLOCATION WAS DONE ON 25-JUL-74
AT 15:44 BY TASK BUILDER VERSION M06

***ROOT SEGMENT: RDIN

R/W MEM LIMITS: 000000 034713 034714
STACK   LIMITS: 000214 001213 001000
DISK BLK LIMITS: 000002 000036 000035
IDENTIFICATION : $FORT
PRG XFR ADDRESS: 001214
TASK ATTRIBUTES: NC

PROGRAM SECTION ALLOCATION SYNOPSIS

<. BLK.>: 001214 024555 023342
<ADTA >: 024556 026215 001440
<DTA   >: 026216 027657 001442
<$$ALER>: 027660 027703 000024
<$$AOTS>: 027704 030461 000556
<$$DEVT>: 030462 031671 001210
<$$FSR1>: 031672 032711 001020
<$$FSR2>: 032712 033013 000102
<$$IOB1>: 033014 033217 000204
<$$IOB2>: 033220 033220 000000
<$$OBF1>: 033220 033327 000110
<$$OBF2>: 033330 033330 000000
<$$OVDT>: 000000 000000 000000
<$$RESL>: 033330 034451 001122
<$$SGOF>: 000000 000000 000000
<. ABS.>: 000000 000000 000000
<.$$$.>: 034452 034452 000000

Figure 5-2
Memory Allocation File for CALC;4
(Mapped System)

CHAPTER 5. OVERLAY CAPABILITY

*** SEGMENT: PROC1

R/W MEM LIMITS: 034714 036377 001464
DISK BLK LIMITS: 000037 000040 000002

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 034714 036377 001464

*** SEGMENT: PROC2

R/W MEM LIMITS: 034714 035557 000644
DISK BLK LIMITS: 000041 000041 000001

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 034714 035557 000644

*** SEGMENT: PROC3

R/W MEM LIMITS: 034714 035307 000374
DISK BLK LIMITS: 000042 000042 000001

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 034713 035267 000354

*** SEGMENT: SUB1

R/W MEM LIMITS: 035310 035723 000414
DISK BLK LIMITS: 000043 000043 000001

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 035310 035723 000414

*** SEGMENT: SUB2

R/W MEM LIMITS: 035310 035713 000404
DISK BLK LIMITS: 000044 000044 000001

Figure 5-2 (cont.)
Memory Allocation File for CALC;4
(Mapped System)

CHAPTER 5. 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 p-sections 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.

2.   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, since it
          terminates input.

3.   The tree structure is defined by the operators '-' (hyphen)
     and ',' (comma) and by the use of parentheses.

     The operator '-' (hyphen) indicates that its arguments are to
     be concatenated and thus co-exist in memory. The operator ','
     (comma) within parentheses indicates that its arguments are
     to be overlaid and thus share memory. The operator ',' not
     enclosed in parentheses delimits overlay trees. The
     parentheses group segments that begin at the same point in
     memory.

     For example,

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

     defines 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 parenthesis pair
     indicates that the overlay segments C and D start at the same
     location in memory.

4.   The simplest overlay description consists of two directives,
     as follows:

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

Any number of the optional directives (.FCTR, .PSECT, and .NAME) can be included.

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 2.3.1

    ● Factor labels

    ● Segment names

    ● P-section names

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 effectively 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:

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

    The second representation makes it clear that the tree has three main branches.

8. The .PSECT directive provides a means for directly specifying the segment in which a p-section is placed.

    The .PSECT directive gives the name of the p-section and its attributes. For example:

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

    ALPHA is the p-section name and the remaining arguments are attributes. P-section attributes are described in Chapter 4.

The p-section name must appear first on the .PSECT directive, but the attributes can appear in any order or can be omitted. If an attribute is omitted, a default assumption is made. For p-section attributes the default assumptions are:

RW,I,LCL,REL,CON

In the above example, therefore, it is only necessary to specify the attributes that do not correspond to the default assumption:

.PSECT ALPHA,GBL

9. The .NAME directive provides a means for defining a segment name for use in the overlay description. This directive is useful for creating a null segment or naming a segment that is to be loaded manually. If the .NAME directive is not used, the name of the first file, or p-section in the segment is used to identify the segment.

The .NAME directive defines a name, as follows:

.NAME NEWNM

The defined name must be unique with respect to the names of p-sections, segments, files, and factor labels.

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)

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.

CHAPTER 6

LOADING MECHANISMS

When the user divides his task into overlay segments, he becomes
responsible for loading these overlay segments into memory as they are
needed. The degree of involvement on the part of the user can range
from minimum, in which he specifies that the loading of all segments
be handled automatically, to maximum, in which he explicitly controls
the asynchronous loading of each segment and handles any errors that
occur as a result of the load request.

This chapter describes the loading mechanisms available to the user.

There are two methods for loading overlays:

      Autoload   in which the Overlay Runtime System is
                        automatically invoked to load those segments that
                        are marked by the user.

      Manual Load   in which the user includes explicit calls to the
                        Overlay Runtime System in his programs.

In the autoload method, loading and error recovery are handled by the
Overlay Runtime System. In the manual load method, the user handles
loading and error recovery explicitly. The user has more control and
can specify whether loading is to be done synchronously or
asynchronously.

The user must decide which method he is going to use, because both
methods can not be used in a single task. Both methods offer
advantages. The autoload method allows the user to divide his task
into segments without explicit calls to load overlays. The manual
load method saves space and gives the user full control over the
loading process.

The user is responsible for loading the overlay segments of the main
tree, and if co-trees are used, the root segment as well as the
overlay segments of the co-tree. Once loaded, the root segment of the
co-tree remains in memory.

CHAPTER 6. LOADING MECHANISMS

6.1 AUTOLOAD

If the user decides to use the autoload method, he places the autoload
indicator '*' in the ODL description of the task at the points where
loading must take place. The execution of a transfer of control
instruction to an autoloadable segment up-tree automatically initiates
the autoload process.

6.1.1 Autoload Indicator

The autoload indicator, '*', marks the construct to which it is
applied as autoloadable. If the autoload indicator is applied to a
parenthesized construct then every name within the parentheses is
marked autoloadable. Applying the autoload indicator at the outermost
parentheses level of the ODL tree description marks every module in
the overlay segments autoloadable.

Consider the example TK1 of Chapter 5, and suppose further that
segment C consists of a set of modules C1, C2, C3, C4 and C5. The
tree diagram for TK1 then is:

```
                    A21    A22
                     └──────┤
                            │
   A1          A2          B1        B2          C5
   └───────────┤           └─────────┤           C4
               │                     │           C3
              A0                    B0           C2
               └─────────────────────┤           C1
                                     │           └──
                            CNTRL
```
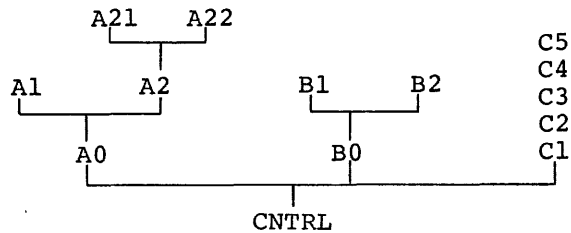
If the user introduces the autoload indicator at the outermost
parentheses level, he is assured that, regardless of the flow of
control within the task, a module is always properly loaded when it is
called. The ODL description for the task with this provision then is:

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

To be assured that all modules of a co-tree are properly loaded, the
user must mark the root segment as well as the outermost parentheses
level of the co-tree, as follows:

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

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

CHAPTER 6. LOADING MECHANISMS

The autoload indicator can be applied to the following constructs:

- Filenames - to make all the components of the file autoloadable.

- Parenthesized ODL tree descriptions - to make all the names within the parentheses autoloadable.

- P-section names - to make the p-section autoloadable. The p-section must have the I (instruction) attribute.

- Defined names introduced by the .NAME directive - to make all components of the segment to which the name applies autoloadable.

- Factor label names - to make the first irreducible component of the factor autoloadable. If the entire factor is enclosed in parentheses, then the entire factor is made autoloadable.

Suppose the user introduces two .PSECT directives and a .NAME directive into the ODL description for TK1 and then applies autoload indicators in the following way:

```
              .ROOT CNTRL-(*AFCTR,*BFCTR,*CFCTR)
AFCTR:        .FCTR A0-*ASUB1-ASUB2-*(A1,A2-(A21,A22))
BFCTR:        .FCTR (B0-(B1,B2))
CFCTR:        .FCTR CNAM-C1-C2-C3-C4-C5
              .NAME CNAM
              .PSECT ASUB1,I,GBL,OVR
              .PSECT ASUB2,I,GBL,OVR
              .END
```

The interpretation for each autoload indicator in the overlay description is as follows:

*AFCTR   The autoload indicator is applied to a factor label name, so the first irreducible component of that factor, A0, is made autoloadable.

*BFCTR   The autoload indicator is applied to a factor label name, so the first irreducible component of that factor, (B0-(B1,B2)), is made autoloadable.

*CFCTR   Again, the autoload indicator is applied to a factor label name, so the first irreducible component, CNAM, of the factor is made autoloadable. CNAM, however, is a defined name introduced by a .NAME directive, so all the components of the segment to which the name applies are made autoloadable; that is, C1, C2, C3, C4, and C5.

*ASUB1   The autoload indicator is applied to a p-section name, so the p-section ASUB1 is made autoloadable.

*(Al,A2-(A21,A22)) The autoload indicator is applied to a parenthesized ODL description so every name within the parentheses is made autoloadable; that is, Al, A2, A21, and A22.

The net effect of the above ODL description is to make every name except ASUB2 autoloadable.

6.1.2 Path-Loading

Autoload uses the technique of path-loading. That is, a call from a segment to a segment up-tree (farther away from the root) requires that all the segments on the path from the calling segment to the called segment to be resident in memory. Path loading is confined to the tree in which the called segment resides. A call from a segment in another tree results in the loading of all segments on the path in the second tree from the root to the called module.

Consider again the example TK1 and the tree diagram:

```
        A21      A22
         |_____|
             |
 A1        A2        B1        B2         C5
  |_____|         |_____|         C4
       |                   |              C3
      A0                  B0              C2
       |_____|_____ C1
                 |
               CNTRL
```
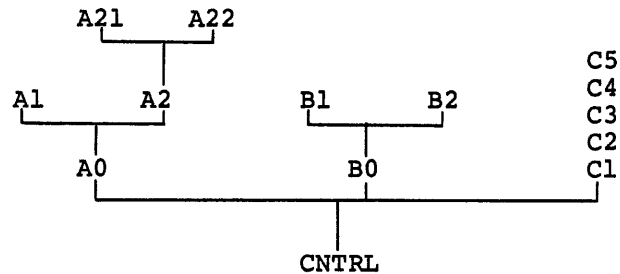
If CNTRL calls A2, then all the modules between the calling module CNTRL and the called module A2 are loaded. In this case modules A0 and A2 are loaded.

The Overlay Runtime System keeps track of the segments in memory and only issues load requests for those segments not in memory. If, in the above example, CNTRL called A1 and then called A2, A0 and A1 are loaded first and then A2 is loaded. A0 is not loaded when A2 is loaded because it is already in memory.

A reference from a segment to a segment down-tree (closer to the root) is resolved directly. For example, if A2 calls A0, then the reference is resolved directly because A0 is known to be in memory as a result of the path-loading that took place in the call to A2.

CHAPTER 6. LOADING MECHANISMS

6.1.3 Autoload Vectors

When the Task Builder sees a reference from a segment to an autoloadable segment up-tree, it generates an autoload vector for the referenced global symbol. The definition of the symbol is changed to an autoload vector table entry. The autoload vector has the following format:

JSR   PC
$AUTO
SEGMENT DESCRIPTOR ADDR.
ENTRY POINT ADDRESS

Observe that a Transfer of Control instruction to the referenced global symbol executes the call to the autoload routine, $AUTO contained in the autoload vector.

An exception is made in the case of a p-section with the D (data) attribute. References from a segment to a global symbol up-tree in a p-section with the D attribute are resolved directly.

Since the Task Builder can obtain no information about the flow of control within the task, it often generates more autoload vectors than are necessary. The user, however, can apply his knowledge of the flow of control of his task and his knowledge of path-loading to determine the placement of autoload indicators. By placing the autoload indicators only at the points where loading is actually required, the user can minimize the number of autoload vectors generated for the task.

CHAPTER 6. LOADING MECHANISMS

Suppose that in TK1 all the calls to overlays originate in the root segment. That is, no module in an overlay segment calls outside its overlay segment. Suppose further 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
```

If the autoload indicator is placed at the outermost parentheses level, thirteen autoload vectors are generated for this task.

The user observes that since A2 and A0 are loaded by path loading to A21, the autoload vectors for A2 and A0 are unnecessary. He observes, further, that the call to C1 loads the segment which contains C2, C3, C4 and C5; therefore autoload vectors for C2 through C5 are unnecessary.

The user eliminates the unnecessary autoload vectors by placing the autoload indicator only at the points where loading is required, as follows:

```
               .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, namely those for A1, A21, A22, B0, B1, B2, and C1.

6.2 MANUAL LOAD

If the user decides to use the manual load method of loading segments, he must include explicit calls to the $LOAD routine in his programs. These load requests give the name of the segment to be loaded and optionally give information necessary to perform asynchronous load requests and to handle unsuccessful load requests.

The $LOAD routine does not path-load. A call to $LOAD always results in the segment named in the load request being loaded and only that segment being loaded.

CHAPTER 6. LOADING MECHANISMS

The MACRO-11 programmer calls the $LOAD routine directly. The FORTRAN programmer is provided with the subroutine 'MNLOAD'.

6.2.1 Manual Load Calling Sequence

The MACRO-11 programmer calls $LOAD, as follows:

```
        MOV     #PBLK,R0
        CALL    $LOAD
```

where PBLK labels a parameter block with the following format:

```
PBLK:       .BYTE   length,event-flag
            .RAD50  /seg-name/
            .WORD   I/O-status
            .WORD   AST-trp
```

The user must specify the following parameters:

length   the length of the parameter block (3 - 5 words)

event-flag   the event flag number, used for asynchronous loading. If the event-flag number is zero, synchronous loading is performed.

seg-name   the name of the segment to be loaded, a 1- to 6-character radix-50 name, occupying two words.

The following parameters are optional:

I/O-status   the address of the I/O status doubleword. Standard QIO status codes apply.

AST-trp   the address of an asynchronous trap service routine to which control is transferred at the completion of the load request.

The condition code C is set or cleared on return, as follows:

If the condition code C = 0, the load request was successfully executed.

If condition code C = 1, the load request was unsuccessful.

For a synchronous load request, the return of the condition code 0 means that the desired segment has been loaded and is ready to be executed. For an asynchronous load request, the return of the code 0 means that the load request has been successfully queued to the device driver, but the segment is not necessarily in memory. The user must ensure that loading has been completed by waiting for the specified event flag before calling any routines or accessing any data in the segment.

6.2.2 FORTRAN Subroutine for Manual Load Request

To use manual load in a FORTRAN program, the program makes explicit reference to the $LOAD routine by means of the 'MNLOAD' subroutine. The subroutine call has the following form:

        CALL MNLOAD (seg-name,event-flag,I/O-status,ast-trp,ld-ind)

where:

seg-name   is a 2 word real variable containing the segment name in radix-50 format.

event-flag   is an optional integer event flag number, to be used for an asynchronous load request. If the event flag number is zero, the load request is considered synchronous.

I/O-status   is an optional 2-word integer array to contain the I/O status doubleword, as described for the QIO directive in the RSX-11M Executive Reference Manual.

ast-trp   is an optional asynchronous trap subroutine to be entered at the completion of a request. MNLOAD requires that all pending traps specify the same subroutine.

ld-ind   is an optional integer variable to contain the results of the subroutine call. One of the following values is returned:

                +1 request was successfully executed.

                -1 request had bad parameters or was not executed successfully.

Optional arguments can be omitted. The following calls are all legal:

Call	Effect
CALL MNLOAD (SEGA1)	Load segment named in SEGA1 synchronously
CALL MNLOAD (SEGA1,0,,,LDIND)	Load segment named in SEGA1 synchronously and return success indicator to LDIND.
CALL MNLOAD (SEGA1,1,IOSTAT,ASTSUB,LDIND)	
	Load segment named in SEGA1 asynchronously, transferring control to ASTSUB upon completion of the load request, storing the I/O-status doubleword in IOSTAT and the success indicator in LDIND

Consider the program CNTRL, discussed in connection with the autoload method, and suppose that between the calls to the overlay segments there is sufficient processing to make asynchronous loading effective. The user removes the autoload indicators from his ODL description and recompiles his FORTRAN programs with explicit calls to the MNLOAD subroutine, as follows:

```
PROGRAM CNTRL
EXTERNAL ASTSUB
DATA SEGA1 /6RA1    /
DATA SEGA21 /6RA21  /
...
CALL MNLOAD (SEGA1,1,IOSTAT,ASTSUB,LDIND)
...
CALL A1
...
CALL MNLOAD (SEGA21,1,IOSTAT,ASTSUB,LDIND)
...
CALL A21
...
...

END
SUBROUTINE ASTSUB
DIMENSION IOSTAT(2)

...
END
```

When the AST trap routine is given as shown in the preceding example, the IO status doubleword is automatically supplied to the dummy variable IOSTAT.

CHAPTER 6. LOADING MECHANISMS

6.3 ERROR HANDLING

If the manual load method is selected, the user must provide error handling routines which diagnose load errors and provide appropriate recovery.

If the autoload method is selected, a simple recovery procedure is provided, which checks the Directive Status Word (DSW) for the presence of an error indication. If the DSW indicates that no system dynamic storage is available, the routine issues a 'wait for significant event' directive and tries again; if the problem is not dynamic storage, the recovery procedure generates a breakpoint synchronous trap. If the using routine is set to service the trap and return without altering the state of the program, the request can be retried.

A more comprehensive user-written error recovery subroutine can be substituted for the system-provided routine if the following conventions are observed:

1. The error recovery routine must have the entry point name $ALERR.

2. The contents of all registers must be saved and restored.

On entry to $ALERR, R2 contains the address of the segment descriptor that could not be loaded. Before recovery action can be taken, the routine must determine the cause of the error by examining the following words in the sequence indicated:

1. $DSW -   The Directive Status Word may contain an error status code, indicating that the I/O request to load the overlay segment was rejected by the Executive.

2. N.OVPT - The contents of this location, offset by N.IOST, point to a 2-word I/O Status block containing the results of the load overlay request returned by the device driver. The status code occupies the low-order byte of word 0.

6.4 EXAMPLE: CALC;5

Suppose the task CALC is now complete and checked out and the user wants to adjust the autoload vectors to minimize the amount of storage required.

From his knowledge of the flow of control of the task he can determine that PROC3 is always in memory as a result of path-loading when it is called and therefore, the autoload vector for PROC3 can be eliminated.

He modifies the ODL description in CALTR, as follows:

CHAPTER 6. LOADING MECHANISMS

```
                .ROOT RDIN-RPRT-ADTA-(*PROC1,*PROC2,P3FCTR)
P3FCTR:         .FCTR PROC3-*(SUB1,SUB2)
                .END
```

He builds the task and the resulting memory allocation file in Figure
6-1 shows that the repositioning of the autoload indicator saved 10
bytes.

```
        FILE CALC.TSK;5 MEMORY ALLOCATION MAP
        THIS ALLOCATION WAS DONE ON 25-JUL-74
        AT 15:50 BY TASK BUILDER VERSION M06



        *** RDOT SEGMENT: RDIN


        R/W MEM  LIMITS: 000000 034703 034704
        STACK    LIMITS: 000214 001213 001000
        DISK BLK LIMITS: 000002 000036 000035
        IDENTIFICATION : $FORT
        PRG XFR ADDRESS: 001214
        TASK ATTRIBUTES: NC


        PROGRAM SECTION ALLOCATION SYNOPSIS:

        <. BLK.>: 001214 024555 023342
        <ADTA  >: 024556 026215 001440
        <DTA   >: 026216 027657 001442
        <$$ALER>: 027660 027703 000024
        <$$AOTS>: 027704 030461 000556
        <$$DEVT>: 030462 031671 001210
        <$$FSR1>: 031672 032711 001020
        <$$FSR2>: 032712 033013 000102
        <$$IOB1>: 033014 033217 000204
        <$$IOB2>: 033220 033220 000000
        <$$OBF1>: 033220 033327 000110
        <$$OBF2>: 033330 033330 000000
        <$$OVDT>: 000000 000000 000000
        <$$RESL>: 033330 034451 001122
        <$$BGDF>: 000000 000000 000000
        <. ABS.>: 000000 000000 000000
        <.$$$.>: 034452 034452 000000
```

Figure 6-1
Root Segment of Memory Allocation
File for CALC;5 (Mapped System)

CHAPTER 6. LOADING MECHANISMS

6.5 GLOBAL CROSS-REFERENCE OF AN OVERLAID TASK

Chapter 4 introduced the global cross-reference feature. This section
illustrates a global cross-reference that has been created for an
overlaid task. The task consists of a root segment containing the
module ROOT.OBJ, and two overlay segments comprised of modules
OVR1 and OVR2. The overlay description of the file is as follows:

        .ROOT ROOT-(OVR1,*OVR2)

Only segment OVR2 is autoloadable.

The resulting cross-reference listing is shown in Figure 6-2. By
consulting the cross-reference listing, the following observations
can be made.

The global symbol OVR1 is defined in the module OVR1 and a single
non-autoloadable, up-tree reference is made to this symbol by the
module ROOT (as indicated by the up-arrow, i.e., circumflx, character).

The asterisk preceding the module OVR2 indicates that the global
symbol OVR2 is an autoload symbol and is referenced from the module
ROOT through an autoload vector as shown by the @ character.

Down-tree references to the global symbol ROOT are made from modules
OVR1 and OVR2. These references are resolved directly.

GLOBAL CROSS REFERENCE

FILE: OVRTST   CREATED BY   TKB   ON 21-MAR-75 AT 14:04   PAGE 1

SYMBOL   VALUE   REFERENCES...

N.ALER   000010   AUTO   # OVRES
N.IOST   000004   OVCTL   # OVRES
N.OVLY   000000   OVCTL   # OVRES
N.OVPT   000054   AUTO   OVCTL
N.STBL   000002   # OVRES
OVR1   001764-R   # OVR1   ⌐ ʳROOT
OVR2   001764-R   * OVR2   @ ROOT
O$VEF   000037   # OVCTL
ROOT   001210-R   OVR1   OVR2   # ROOT
$ALERR   001226-R   # ALERR   OVDAT
$AUTO   001252-R   # AUTO
$DSW   000046   ALERR
$MARKS   001406-R   AUTO   # OVCTL
$RDSEG   001512-R   AUTO   # OVCTL
$SAVRG   001662-R   AUTO   # SAVRG
.NALER   001224-R   # OVDAT
.NIOST   001220-R   # OVDAT
.NOVLY   001214-R   # OVDAT
.NOVPT   000042   # OVDAT
.NSTBL   001216-R   # OVDAT

Figure 6-2
Sample Overlaid Cross-reference Listing

CHAPTER 7

SHARED REGIONS

This chapter describes the use of shared regions. A shared region is
a block of data or code that can be shared by any number of tasks.

Shared regions are useful because they make more efficient use of
memory:

1. Shared regions provide a way in which two or more tasks can
    communicate.

2. Shared regions provide a way in which a single copy of a data
    base or commonly used subroutines can be shared by several
    tasks.

Consider the first case, in which two tasks, Task A and Task B, need
to communicate a large amount of data.   A convenient method of
transporting this data is the use of a shared region.   Tasks can
communicate independent of their time of execution. This case is
illustrated by the following diagram:

shared region S	shared region S
Task A	
	Task B
Resident Executive	Resident Executive

System Memory (Time t)	(System Memory) Time t+n

Task A and Task B communicate through the shared region. Any number of tasks can link to a shared region.

Consider the second case, in which tasks make use of common routines. The common subroutines are not included in each task image; instead, they are included in a shared region so that a single copy is accessible to all tasks. This case is shown in the following diagram:

Routine R
Task A
Routine R
Task B
Resident Executive

System Memory

Routine R
Task A
Task B
Resident Executive

System Memory

CHAPTER 7. SHARED REGIONS

A task can link to as many as three shared regions. A privileged task in a mapped system, however, can link to a maximum of two shared regions.

A shared region has associated with it a task image file and a symbol definition file. When a task links to a shared region the Task Builder uses the symbol definition file of the shared region to establish the linkages between the task and the shared region.

7.1 USING AN EXISTING SHARED REGION

The user can link to any of the system shared regions by using the COMMON or LIBR keyword option and specifying the name of the shared region and the type of access he is requesting.

Suppose JRNAL is a system shared region and the user wants his task IMG1 to link to that region and examine some relevant data. He specifies the name in the COMMON keyword with read-only access as follows:

```
>TKB
TKB>IMG1,LP:=IN1,IN2,IN3
TKB>/
ENTER OPTIONS:
TKB>COMMON=JRNAL:RO
TKB>//
```

A task can link to any shared region on the disk. However, before the task can be activated, any shared region it uses must be resident in memory.

7.2 CREATING A SHARED REGION

To create a shared region, the task image and symbol definition files must be built under UIC [1,1] on the system device.

In Chapter 4, runnable tasks were described. A shared region differs from a runnable task in that it does not have a header or a stack. The user must therefore specify that no header and stack are to be produced for the task image file in creating a shared region.

In summary, to create a shared region the following steps are taken:

- The task image file and symbol definition file are built under UIC [1,1] on the system device.

- The task image file or symbol definition file has the switch /-HD, indicating that no header is required.

- The option STACK=0 is entered during option input to eliminate the stack.

Suppose the user wants to create a resident library, ZETA, from the files Z1, Z2, and Z3. He builds the shared region, as follows:

```
>TKB
TKB>[1,1]ZETA/-HD,LP:,SY:[1,1]ZETA=Z1,Z2,Z3
TKB>/
ENTER OPTIONS:
TKB>STACK=0
TKB>//
```

A task can now link to the shared region. However, before the task can be installed and activated, the shared region must be made resident in memory. Space is allocated for the library and the library is loaded into memory by the following commands (see the RSX-11M Operator's Procedures Manual for a description of the commands).

```
>! ALLOCATE SPACE FOR RESIDENT LIBRARY
>SET /MAIN=ZETA:1400!100!COM
>INS [1,1]ZETA/PAR=ZETA
```

7.3 POSITION INDEPENDENT AND ABSOLUTE SHARED REGIONS

A shared region can be either position independent or absolute. Position independent shared regions can be placed anywhere in the task's virtual address space when the system on which the task runs has memory management hardware. Absolute regions must be fixed in the virtual address space.

The user must ensure that the region is position independent, if he applies the PI switch. The PI switch directs the Task Builder to treat the region as position independent, but the Task Builder can not determine whether or not the region is position independent. If the PI switch is applied to a region which is not truly position independent, the execution of a task linked to that region is unpredictable.

Data is always position independent. Code can be position independent, but the code produced as a result of compiling a FORTRAN program is not position independent. Furthermore, FORTRAN programs can not be used as shared libraries because these programs do not satisfy the re-entrancy requirements necessary for shared regions.

FORTRAN common blocks can be included in shared regions. However, the only way FORTRAN programs can communicate through the use of common blocks is by the common block name; to retain this name, the shared region must be declared position independent. If the region is not declared position independent, the name is not retained and no FORTRAN program can link to the common block.

Chapter 8 illustrates the use of a FORTRAN common block as a shared region on an unmapped system.

Absolute shared regions are used for code which satisfies the re-entrancy requirements for a shared region but is not position independent.

7.4 EXAMPLE: CALC;6 BUILDING AND USING A SHARED REGION

Suppose the task CALC has been completely debugged and the user wants
to replace the dummy reporting routine RPRT by a generalized reporting
program that operates as a separate task. This generalized reporting
program GPRT was developed by another programmer in parallel with the
development of CALC. Now both routines are ready and the user wants
to create a shared region so that the two tasks can communicate.

In addition to creating the shared region, the user must modify his
FORTRAN routine to replace the call to the dummy reporting routine by
a call to REQUEST for the task GPRT and he must remove the dummy
routine from his ODL description for the task.

7.4.1 Building the Shared Region

The common block into which CALC places its results and from which
GPRT takes its input is named DTA. The user wants to make DTA into a
shared region so that the two tasks can communicate.

The user first creates a separate input file for DTA:

```
>EDI
EDI>DTA.FTN
[CREATING NEW FILE]
INPUT
C
C     GLOBAL COMMON AREA FOR 'CALC' AND
C     REPORTING TASK 'GPRT'
      BLOCK DATA
      COMMON /DTA/ A(200),I
      END
  *EX
```

He then compiles DTA:

```
>FOR DTA,LP:=DTA
```

He then builds the task image and symbol definition file for the
shared region DTA:

```
>TKB
TKB>[1,1]DTA/PI,LP:/SH,SY0:[1,1]DTA/-HD=DTA
TKB>/
ENTER OPTIONS:
TKB>STACK=0
TKB>//
```

He marks the task image file for DTA as position independent in order
to retain the name of the referenced common block, DTA.

As required, he creates the task image and symbol definition file on the system device under the User Identification Code [1,1,], applies the switch -HD to the symbol definition file to specify that the task has no header, and enters the option STACK=0 to eliminate the stack. It was necessary to specify the system device SY0 for the symbol definition file; if the user does not specify a device, the last named device applies. In this case, failure to specify the system device would have resulted in the application of the device specification LP to the symbol definition file.

The shared region DTA now exists on the disk as an eligible candidate for inclusion in an active system. The user can now modify his task to link to that shared region. However, before the task can be executed, the shared region must be made resident in memory.

7.4.2 Modifying the Task to Use the Shared Region

The user now modifies the task CALC. He edits the file containing the program RDIN to include the name of the reporting task in radix-50 format:

        DATA RPTSK/6RGPRT /

And he replaces the call to the dummy reporting routine RPRT by the call:

        CALL REQUES (RPTSK)

The relevant part of the program RDIN is shown below:

        C   READ AND ANALYZE INPUT DATA
        C   ESTABLISH COMMON DATA BASE
             COMMON /DTA/ A(200), I
        C   SET UP NAME OF REPORTING TASK IN RADIX 50
             DATA RPTSK /6RGPRT /
        C   READ IN RAW DATA
             ...
             CALL REQUES (RPTSK)
             ...
             END

The user now modifies the ODL description of the task CALC to remove the file RPRT.OBJ. He changes the .ROOT directive from:

        .ROOT RDIN-RPRT-ADTA-(*PROC1,*PROC2,P3FCTR)

to:

        .ROOT RDIN-ADTA-(*PROC1,*PROC2,P3FCTR)

CHAPTER 7. SHARED REGIONS

He then builds an indirect command file to include the COMMON keyword:

```
>EDI
EDI>CALCBLD.CMD
[CREATING NEW FILE]
INPUT
CALC,LP:/SH=CALTR/MP
PAR=PAR14K
ACTFIL=1
COMMON=DTA:RW
//
*EX
```

And then he builds the task with the single command referencing the indirect file:

```
>TKB @CALCBLD
```

The communication between the two tasks, CALC and GPRT, is now established. When the shared region DTA is made resident, the two tasks can run.

7.4.3 The Memory Allocation Files

Figure 7-1 gives the memory allocation file for the shared region DTA. The attribute list indicates that the task image was built with no header (NH) and is position independent (PI).

Figure 7-2 gives the memory allocation file for the task CALC after the shared region DTA was created and the dummy reporting routine removed from the task. The read-write memory limits for the root segment code have increased due to the call to REQUES. The read-write memory limits for the entire task have decreased because the common block DTA is now a shared region allocated at 160000 and no longer part of the task.

FILE DTA.TSK:2 MEMORY ALLOCATION MAP
THIS ALLOCATION WAS DONE ON 25-JUL-74
AT 16:25 BY TASK BUILDER VERSION M06

*** ROOT SEGMENT: DTA

R/W MEM LIMITS: 000000 001443 001444
DIS BLK LIMITS: 000002 000003 000002
IDENTIFICATION:
TASK ATTRIBUTES:NC,NH,PI

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 000000 000000 000000
<DTA   >: 000000 001441 001442
<. ABS.>: 000000 000000 000000
<.$$$$.>: 001442 001442 000000

*** FILE: DTA.OBJ:2   TITLE: .DATA.   IDENT:

<.$$$$.>: 001442 001442 000000

<DTA   >: 000000 001441 001442

Figure 7-1
Memory Allocation File for the Shared Region DTA
(Mapped System)

```
FILE CALC.TSK:6 MEMORY ALLOCATION MAP
THIS ALLOCATION WAS DONE ON 25-JUL-74
AT 16:20 BY TASK BUILDER VERSION M06

*** ROOT SEGMENT: RDIN

R/W MEM  LIMITS: 000000 033337 033340
STACK    LIMITS: 000214 001213 001000
DISK BLK LIMITS: 000002 000035 000034
IDENTIFICATION : $FORT
PRG XFR ADDRESS: 001214
TASK ATTRIBUTES: NC

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 001214 024651 023436
<ADTA  >: 024652 026311 001440
<DTA   >: 160000 161441 001442
<$$ALER>: 026312 026335 000024
<$$AOTS>: 026336 027113 000556
<$$DEVT>: 027114 030323 001210
<$$FSR1>: 030324 031343 001020
<$$FSR2>: 031344 031445 000102
<$$IOR1>: 031446 031651 000204
<$$IOB2>: 031652 031652 000000
<$$OBF1>: 031652 031761 000110
<$$OBF2>: 031762 031762 000000
<$$OVOT>: 000000 000000 000000
<$$RESL>: 031762 033103 001122
<$$SGDF>: 000000 000000 000000
<. ABS.>: 000000 000000 000000
<.$$$.>: 161442 161422 000000

*** FILE: DTA.STB;2  TITLE: DTA     IDENT:

<DTA   >; 160000 161441 001442

<. ABS.>: 000000 000000 000000

N.OVPT 000054   $DSW    000046   $OTSV   000052   .FSPRT 000050

<.$$$.>: 161442 161442 000000

*** FILE: RDIN.OBJ;5  TITLE: .MAIN.  IDENT: $FORT

<.$$$.>: 161442 161442 000000

<.$$$.>: 161442 161442 000000
```

Figure 7-2
Memory Allocation File for CALC;6
(Mapped System)

CHAPTER 8

HOST AND TARGET SYSTEMS

This chapter describes the construction of a task destined to   run   on
another system.

8.1   BUILDING THE TASK FOR THE TARGET SYSTEM

The user can transfer a task from the host system (the system on which
the task is built) to the target system (the system on which the task
will run) by following a few simple steps:

1. He builds the task image specifying a partition that has the
    base address and size of the partition in which the task will
    run on the target system.

2. He ensures that any shared regions accessed by the task are
    present in both systems under UIC[1,1].

3. If the target system and the host system do not have the same
    mapping status, he sets the Memory Management switch (MM) to
    reflect the mapping status of the target system.

8.1.1   Example

Suppose that in a given installation, there is one large computer
system with mapping hardware and several smaller systems without
mapping hardware. The programmers in this installation create and
debug their tasks on the large host system and when the tasks are
ready to go into production, they transfer them to the smaller
systems. If the programmer is developing the task, TK1, in the
default partition on the host system, his task building sequence is:

    >TKB TK1,LP:=SQ1,SQ2

When he is ready to move his task to a target system, he builds the
task again, indicating the mapping status of the target system and
naming the partition in which the task is to reside on the target
system:

```
>TKB
TKB>TK1/-MM,LP:=SQ1,SQ2
TKB>/
ENTER OPTIONS:
TKB>PAR=PART1:100000:40000
TKB>//
```

The resulting task image is ready to run on the unmapped target system.

8.2   EXAMPLE: CALC;7

Suppose the user has now completed checking out the interface between his task CALC and the generalized reporting routine and he is now ready to move the task to another system. The system on which he has been working has mapping hardware, but the system on which CALC is going to run does not have mapping hardware.

The user knows the configuration of the target system. He knows that there is a partition called PAR8K with base at 40000 in which the task CALC is going to run.

To move the tasks CALC and GPRT, he must also move the shared region DTA. Therefore, he must rebuild the shared region task image for the partition in which it will reside on the target system.

8.2.1   Rebuilding the Shared Region

He builds the task image for the shared region again, this time for a partition in the target system:

```
>TKB
TKB>[1,1]DTA/PI/-MM,LP:/SH,SY0:[1,1]DTA/-HD=DTA
ENTER OPTIONS:
TKB>STACK=0
TKB>PAR=PARS1:156000:2000
TKB>//
```

CHAPTER 8. HOST AND TARGET SYSTEMS

8.2.2 Rebuilding the Task for the Target System

He modifies the indirect command file CALCBLD, so that it includes the
memory mapping switch and the target partitions. He also adds
comments to identify the task building sequence:

```
;       PROCESS ANALYSIS FOR SYSTEM M23
;       VERSION 1 DATE:  AUG 26, 1974
CALC/-MM,LP:/SH=CALTR/MP
PAR=PAR8K:40000:40000
COMMON=DTA:RW        ;COMMUNICATION WITH GPRT
//
```

He then builds his task with the single command;

```
>TKB @CALCBLD
```

His task is now ready to be installed and run on the target system.
Before the task can be installed, the shared region must be made
memory resident on the target system.

8.2.3 The Memory Allocation Files

Figure 8-1 gives the memory allocation file of the shared region DTA
for an unmapped system. The shared region is bound to the partition
base specified by the PAR keyword in the task build. Note that the
shared region is declared position independent in the unmapped system
even though it is bound to the partition base 156000.   The position
independent declaration is necessary to preserve the relocatable
p-section DTA so that other FORTRAN tasks can link to the region.

Figure 8-2 gives the memory allocation file of the task CALC for an
unmapped system.   The task is bound to the specified partition base
40000 and linked to the shared region DTA bound at 156000.

```
FILE DTA.TSK;1 MEMORY ALLOCATION MAP
THIS ALLOCATION WAS DONE ON 20-AUG-74
AT 07:09 BY TASK BUILDER VERSION M06

*** ROOT SEGMENT: DTA

R/W MEM LIMITS: 156000 157443 001444
DISK BLK LIMITS: 000002 000003 000002
IDENTIFICATION:
TASK ATTRIBUTES: NC,NH,PI

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 156000 156000 000000
<DTA   >: 156000 157441 001442
<. ABS.>: 000000 000000 000000
<.$$$$.>: 157442 157442 000000

*** FILE: DTA.OBJ;1 TITLE: .DATA. IDENT:

<.$$$$.>: 157442 157442 000000

<DTA   >: 156000 157441 001442
```

Figure 8-1
The Memory Allocation File for the Shared Region
(Unmapped System)

CHAPTER 8. HOST AND TARGET SYSTEMS

FILE CALC.TSK;>: MEMORY ALLOCATION MAP
THIS ALLOCATION WAS DONE ON 20-AUG-74
AT 08:25 BY TASK BUILDER VERSION M06

*** ROOT SEGMENT: RDIN

R.W MEM LIMITS: 040000 073337 033340
STACK   LIMITS: 040214 041213 001000
DISK BLK LIMITS: 000002 000035 000034
IDENTIFICATION : $FORT
PRG XFR ADDRESS: 041214
TASK ATTRIBUTES: NC

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 041214 064661 023446
<ADTA >: 064662 066321 001440
<DTA   >: 156000 157441 001442
<$$ALER>: 066322 066345 000024
<$$AOTS>: 066346 067123 000556
<$$DEVT>: 067124 070333 001210
<$$FSR1>: 070334 071353 001020
<$$FSR2>: 071354 071445 000102
<$$IOB1>: 071456 071661 000204
<$$IOB2>: 071662 071662 000000
<$$OBF1>: 071662 071771 000110
<$$OBF2>: 071772 071772 000000
<$$OVDT>: 000000 000000 000000
<$$RESL>: 071772 073113 001122
<$$SGDF>: 000000 000000 000000
<. ABS.>: 000000 000000 000000
<.$$$$.> 157442 157442 000000

*** SEGMENT: PROC1

R/W MEM LIMITS: 073340 075023 001464
DISK BLK LIMITS: 000036 000037 000002

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 073340 075023 001464

Figure 8-2
The Memory Allocation File for
CALC;7 (Unmapped System)

*** SEGMENT, PROC2

R/W MEM LIMITS: 073340 074203 000644
DISK BLK LIMITS: 000040 000040 000001

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 073340 074203 000644

*** SEGMENT: PROC3

R/W MEM LIMITS: 073340 073673 000334
DISK BLK LIMITS: 000041 000041 000001

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 073340 073673 000334

*** SEGMENT: SUB1

R/W MEM LIMITS: 073674 074307 000414
DISK BLK LIMITS: 000042 000042 000001

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 073674 074307 000414

*** SEGMENT: SUB2

R/W MEM LIMITS: 073674 074307 000414
DISK BLK LIMITS: 000043 000043 000001

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 073674 074307 000414

Figure 8-2 (Cont.)
The Memory Allocation File for CALC;7
(Unmapped System)

CHAPTER 9

MEMORY DUMPS

9.1 POST-MORTEM DUMPS

The task PMD... generates a Post-mortem Dump of a task that is ab-
normally terminated. A task may be made eligible for a Post-Mortem
dump in one of two ways:

    1. At task build time by specifying the /PM switch on the
        task file. /-PM will disable dumps and is the default.

    2. At task installation time the /PMD switch may be used
        to override the task built option. /PMD=YES will enable
        dumping; /PMD=NO will disable dumping.

The Post-mortem Dump task PMD... should be installed in a 4K parti-
tion in which all other tasks are checkpointable. This allows the
dump to be generated in a timely manner and prevents the system from
being locked up while the dump is being generated. The dump task
is capable of dumping from memory or from the checkpoint image of
the user's task. The Post-mortem Dump task is sensitive to the
location of the aborted task. Thus, if the aborted task is check-
pointed during the dump, the dump task will switch to reading the
checkpoint image. If the aborted task returns to memory after
being checkpointed, the Post-mortem Dump will switch to using the
core image again.

Dumps are always generated on the system disk under UIC [1,4]. Thus,
to avoid errors from PMD... the user must allocate a UFD for [1,4]
before installing the task. When the dump task finishes generating
the dump, it will attempt to queue it to the print spooler for subse-
quent printing. If no spooler is installed, the dump file will be
left on the disk and may be printed at a later time using PIP.

                              NOTE

        Dump files tend to be rather large. The dump of an
        8K partition will average about 340 blocks. There-
        fore, if there is little space on the disk, it is
        important to print and delete the dump file without
        delay.

CHAPTER 9. MEMORY DUMPS

9.1.1 <u>Description of a Dump Generated by PMD...</u>

The following description of the contents of Post-Mortem and Snapshot dumps is keyed to Figure 9-1. Snapshot dumps are explained more fully in Section 9.2.

<u>Item</u>	<u>Description</u>
1	Type of dump - Post-Mortem or Snapshot. If it is a Snapshot dump, the dump ID is printed.
2	The name of the task being dumped and the date and time the dump was generated.
3	The program counter at the time of the dump; and if it is a post-mortem dump, the reason the task was aborted.
4	The general registers, stack pointer, and processor status at the time of the dump.
5	The task status flags at the time of the dump. See the description of ATL or TAL in the <u>RSX-11M Operator's Procedures Manual</u> for the meaning of the flags.
6	The task event flag mask words at the time of the dump. If the dump is a Snapshot dump, the EFN specified in the SNAP macro will be ON.
7	The task UIC and the current value of the directive status word.
8	The task's priority, number of outstanding I/O requests, and the terminal from which the task was initiated (TI:).
9	The task load device and the logical block number of the start of the task image on the device.
10	The floating point unit (FPU) registers or the extended arithmetic element (EAE) registers if the task is using one of these pieces of hardware. If the task is not using the FPU or EAE, these registers will not be printed. If the task uses the FPU and doesn't specify /FP on the task image file, or if it uses the EAE unit and has not specified the /EA switch, the registers will not be printed. If the machine being used has both an FPU and an EAE, PMD will assume the user is using the FPU since it is the unit of choice for arithmetic computations.
11	The logical unit assignments at the time of the dump. UNIT is the logical unit number, and DEVICE is the device to which the logical unit is assigned. For Snapshot dumps, file status will display the file name of any open files. Post-mortem dumps will not display this information since all of the files will have been closed as a result of the I/O rundown on the aborted task.

CHAPTER 9. MEMORY DUMPS

Item   Description

12   The overlay segments loaded at the time of the dump, the
          relative block number of the segment, the base address, the
          length of the segment, and, for tasks using manual loads,
          the segment names are displayed. The block number may be
          used to determine which segment is loaded by reference to
          the memory allocation file generated by the Task Builder.
          The starting block number for each segment is the relative
          block number of the segment. By obtaining a match, the
          name of the segment in memory may be determined. Zero
          length segments are usually co-tree roots.

13   The task stack at the time of the dump. The address is
          displayed, along with the contents, in octal, ASCII, and
          RAD50. Each word on the stack is dumped. If the stack
          pointer is above the initial value of the stack (H.ISP),
          only one word will be dumped. The rest will be dumped
          as part of the task image.

14   The task image itself. The partition being dumped and the
          limits of interest are displayed. For post-mortem dumps,
          this is the entire task limit. For Snapshot dumps, this is
          the virtual task limits requested by the user. The dump
          routine rounds the requested low limit down to the nearest
          multiple of 8 bytes and rounds the requested high limit up
          to the nearest multiple of 8 bytes. The dump image displays
          the virtual starting address of a four-word block of memory,
          the data in both octal and RAD50 on the first line, and
          byte octal and ASCII on the second line. A four-word block
          which is repeated in a contiguous region of memory will be
          printed once, and then noted by the message

              *** DUPLICATE THROUGH xxxxxx ***

          where xxxxxx indicates the last word which is duplicated.
          If the task was aborted, the task partition, and the
          libraries and common blocks it is linked to, will be dumped.
          If the dump is a Snapshot dump, up to four contiguous
          blocks of memory will be dumped, if requested.

CHAPTER 9.   MEMORY DUMPS

POST-MORTEM DUMP   ①

TASK: SNPTST   ②   TIME: 27-JUN-75 09:31

PC: 000726   ③   IOT EXECUTION   ③

REGS: R0 - 000353   R1 - 074400   R2 - 000120   R3 - 140130 ⎫
                                                                    ⎬ ④
          R4 - 000000   R5 - 000002   SP - 000312   PS - 170000 ⎭

TASK STATUS: OST MSG AST -CHK HLT   ⑤

EVENT FLAG MASKS: <01-16> 000001 <17-32> 056502 ⎫
                   <33-48> 000000 <49-64> 000000 ⎭ ⑥

CURRENT UIC: [200,200]   DSW: 1. ⑦

PRIORITY: 50.   I/O COUNT: 0.   TI DEVICE - TT10:   ⑧

LOAD DEVICE - SY0:   LBN: 0,002265   ⑨

FLOATING POINT UNIT   ⎫
                                                 ⎪
     STATUS - 044400   ⎪
                                                 ⎪
     R0 - 000000 000000 000000 000000   ⎪
     R1 - 000000 000000 000000 000000   ⎬ ⑩
     R2 - 000000 000000 000000 000000   ⎪
     R3 - 000000 000000 000000 000000   ⎪
     R4 - 000000 000000 000000 000000   ⎪
     R5 - 000000 000000 000000 000000   ⎭

  LOGICAL UNITS   ⎫
                                         ⎪
   UNIT DEVICE   FILE STATUS   ⎬ ⑪
                                         ⎪
     1   SY0:   ⎪
     2   SY0:   ⎪
     3   SY0:   ⎭

OVERLAY SEGMENTS LOADED   ⎫
                                                                        ⎪
STARTING RELATIVE BLOCK: 000002   BASE: 000000   LENGTH: 001464   ⎬ ⑫
STARTING RELATIVE BLOCK: 000004   BASE: 001464   LENGTH: 000264   ⎭

  TASK STACK   ⎫
                                       ⎬ ⑬
    ADDRESS CONTENTS ASCII RAD50   ⎪
     000312 000045   X   7   ⎭

Figure 9-1
Sample Post-Mortem Dump

```
                    TASK IMAGE


          PARTITION: GEN        LIMITS: 000000 - 001777

000000   000312    000170    056424    000000   ! EB  C  N5.     !
         312 000   170 000   024 135   000 000                   !J        !        !
000010   001777    003200    000000    000000   ! YW AAX         !
         377 003   200 006   000 000   000 000                   !         !
000020   000000    000000    000000    000000   !               !
         000 000   000 000   000 000   000 000                   !         !

***  DUPLICATE THROUGH   000036  ***

000040   000000    000000    000000    000001   !             A! !
         000 000   000 000   000 000   001 000                   !         !
000050   000000    000000    000726    000000   !         K0    !
         000 000   000 000   326 001   000 000                   !      V  !
000060   000001    056502    000000    000000   !  A N64         !
         001 000   102 135   000 000   000 000                   !  B]     !
000070   100200    100200    170017    000360   !TVP TVP 8PO  F !
         200 200   200 200   017 360   360 000                   !         !
000100   000312    000000    000000    000000   ! EB            !
         312 000   000 000   000 000   000 000                   !J        !
000110   000000    000000    000000    000000   !               !
         000 000   000 000   000 000   000 000                   !         !
000120   000000    140170    000000    063536   !     012    PUV!
         000 000   170 300   000 000   136 147                   !    ·  A !
000130   000003    052352    000000    052352   !  C MWR    MWR!
         003 000   352 124   000 000   352 124                   !    T   T!
000140   000000    052352    000000    170000   !    MWR    8P !
         000 000   352 124   000 000   000 360                   !    T    !
000150   000726    000000    000000    140000   ! K0    B    01 !
         326 001   002 000   000 000   130 300                   !V        X·!
000160   000120    074400    000353    000000   ! B   SNP  E5   !
         120 000   000 171   353 000   000 000                   !P        !
000170   044400    000000    000000    000000   !KSH            !
         000 111   000 000   000 000   000 000                   ! I       !
000200   000000    000000    000000    000000   !               !
         000 000   000 000   000 000   000 000                   !         !

***  DUPLICATE THROUGH   000246  ***

000250   000000    000732    000000    001464   !     K4     TT!
         000 000   332 001   000 000   064 003                   ! Z     4 !
000260   000264    000000    000000    000001   ! DT         A!
         264 000   000 000   000 000   001 000                   !4        !


001720   006303    006101    020116    103402   !BAX ABO EFO UXB!
         300 014   101 014   116 040   002 207                   !·  A  N  !
001730   161601    005200    005366    000002   !6PA ASH A0F  B!
         201 343   200 012   366 012   002 000                   !         !
001740   003367    022626    000207    000000   !ADW F V  CO   !
         367 006   226 045   207 000   000 000                   !    X    !
001750   000000    000000    000000    000000   !               !
         000 000   000 000   000 000   000 000                   !         !

***  DUPLICATE THROUGH   001776  ***
```

⑭

Figure 9-1 (Cont.)
Sample Post-Mortem Dump

9-5

9.2 SNAPSHOT DUMP

The task PMD... is also capable of producing edited dumps for running tasks. These dumps are called Snapshot Dumps, and they are useful as debugging aids. A Snapshot Dump may be requested any number of times during the execution of a task. The information generated is under the control of the programmer.

Snapshot Dumps are generated by the following macros:

SNPDF$ Defines offsets in the Snapshot Dump Control Block and control bits which control the format of the dump.

SNPBK$ Allocates the Snapshot Dump Control Block (see Figure 9-2).

SNAP$ Causes a Snapshot Dump to be generated.

SNPBK$ and SNAP$ issue calls to SNPDF$, so, in most cases, the programmer does not have to explicitly issue the SNPDF$ macro call.

9.2.1 Format of the SNPBK$ Macro

The format of the SNPBK$ macro call is as follows:

SNPBK$ dev,unit,ctl,efn,id,L1,H1,L2,H2,L3,H3,L4,H4

where:

dev   Is the 2 character ASCII name of the device to which the dump is directed. If it is a directory device, the UFD [1,4] must be on the volume. The dump will be written to the disk and then spooled to the line printer. If there is no print spooler, the file will be left on the disk. If the device is not a directory device, the dump will go directly to the device.

unit   Is the unit number of the device to which the dump is directed.

ctl   Are the flags which control the format of the dump and the data to be printed. The flags are:

SC.HDR   Print the dump header. (Items 1-10 in Figure 9-1.)
SC.LUN   Print information on all assigned LUNs. (Item 11)
SC.OVL   Print information about all loaded overlay segments. (Item 12)
SC.STK   Print the user stack. (Item 13)
SC.WRD   Print the requested memory in octal words and RAD50. (Item 14)
SC.BYT   Print the requested memory in octal bytes and ASCII. (Item 14)

efn   Is the event flag to be used to synchronize the user
                program and the task PMD... .

id   Is a number which identifies the Snapshot Dump. Since
                dumps can be requested at different times and under
                different conditions, this ID is used to identify the
                place or reason for the dump.

L1,L2   Are the starting addresses of the memory blocks to
L3,L4   be dumped.

H1,H2,   Are the ending addresses of the memory blocks to be
H3,H4   dumped.

<div align="center">NOTE</div>

<div align="center">
If no memory is to be dumped, each limit
(L1,L2,L3,L4,H1,H2,H3,H4) should be zero.
</div>

Only one snap block is allowed. It generates the global label
..SPBK.

<div align="center">NOTE</div>

<div align="center">
Since SNPBK$ is used to allocate storage
for the snap block, all arguments except
dev must be valid arguments for .WORD or
.BYTE directives.
</div>

9.2.2 Format of the SNAP$ Macro

The format of the SNAP$ macro is as follows:

        SNAP$ ctl,efn,id,L1,H1,L2,H2,L3,H3,L4,H4

where:

ctl   Are the flags which control the format of the dump
                and the data to be printed. The flags are:

                SC.HDR   Print the dump header.
                SC.LUN   Print information on all assigned LUNs.
                SC.STK   Print the user stack.
                SC.OVL   Print information about all loaded
                            overlay segments.
                SC.WRD   Print the requested memory in octal words
                            and RAD50.
                SC.BYT   Print the requested memory in octal bytes
                            and ASCII.

efn   Is the event flag to be used to synchronize the user
                program and the task PMD... . A wait for single
                event flag directive is always generated to perform
                synchronization.

id   Is a number which identifies the snapshot dump.
              Since dumps can be requested at different times
              and under different conditions, this ID is used
              to identify the place or reason for the dump.

L1,L2,   Are the starting addresses of memory blocks to
L3,L4   be dumped.

H1,H2,   Are the ending addresses of memory blocks to be
H3,H4   dumped.

<u>NOTE</u>

1. If no memory is to be dumped, each limit
    (L1,L2,L3,L4,H1,H2,H3,H4) should be zero.

2. The control flags may be set in any combina-
    tion. They are NOT mutually exclusive. Thus,
    any number of options may be obtained; e.g.,
    SC.HDR!SC.LUN!SC.WRD will print the header,
    LUNs, and the requested memory in word octal
    and RAD50 mode.

3. Arguments should only be specified to override
    the information already in the snap control
    block.

4. Since SNAP$ generates instructions to move
    data into the snap block, its arguments must
    be valid source operands for move commands.

Symbol		Offset	Description
SB.CTL		0	Control Flags
SB.DEV		2	Device Mnemonic
SB.UNT		4	Unit Number
SB.EFN		6	Event Flag
SB.ID		10	Snap Identification
SB.LM1	(L1)	12	Memory Block 1 Limits
	(H1)	14	
	(L2)	16	Memory Block 2 Limits
	(H2)	20	
	(L3)	22	Memory Block 3 Limits
	(H3)	24	
	(L4)	26	Memory Block 4 Limits
	(H4)	30	
SB.PMD		32	"PMD..." in RAD50
		34	

Figure 9-2
Format of Snapshot Dump Control Block

9.2.3 Example of a Snapshot Dump

The sample program shown in Figure 9-3 will cause two Snapshot dumps to be printed directly on LP0:. The first dump will use the parameters defined in the Snap Control Block. The header will be generated and the data in relative locations BLK to BLK+220 will be displayed in word octal and RAD50. The identification on the dump will be 1.

The second dump causes the data in the locations BLK to BLK+220 to be displayed in byte octal and ASCII. A header is also generated. The dump identification is 64 (100 octal). Figures 9-4 and 9-5 show the dumps generated by the sample program.

SNPTST - TEST SNAP DUMP AND PMD MACRO MX9C   27-JUN-75 09:24   PAGE 1

```
    1                                          .TITLE   SNPTST - TEST SNAP DUMP AND PMD
    2                                          .IDENT   /01/
    3                                          .MCALL   SNPBKS,SNAPS,CALL
    4 000000                           BLK:    SNPBKS   LP,0,SC.HDR!SC.OVL!SC.WRD,1,1,BLK,BLK+220
    5 000036      123    116    120    BUF:    .ASCIZ   /SNPTST/
      000041      124    123    124
      000044      000
    6                                          .EVEN
    7 000046                           START:  SNAPS                       ;
    8 000216   012700   000036'                MOV      #BUF,R0            ; SET BUFFER ADDRESS
    9 000222                                   CALL     $C AT5            ; CONVERT TO RAD50
   10 000226                                   SNAPS    #SC.HDR!SC.OVL!SC.BYT,,#100
   11 000412   000004                          IOT                        ; ABORT
   12          000046'                         .END     START
```

SNPTST - TEST SNAP DUMP AND PMD MACRO MX9C   27-JUN-75 09:24   PAGE 1-1
SYMBOL TABLE

```
BLK       000000R        SB.EFN= 000006        SC.BYT= 000040        SC.WRD= 000020        ..SPBK  000000RG
BUF       000036R        SB.ID = 000010        SC.HDR= 000001        START   000046R       ...SNP= 000032
IE.ACT= ****** GX        SB.LM1= 000012        SC.LUN= 000002        $CAT5 = ****** GX
SB.CTL= 000000          SB.PMD= 000032        SC.OVL= 000004        $DSW  = ****** GX
SB.DEV= 000002          SB.UNT= 000004        SC.STK= 000010        $SST2 = 000027
```

```
. ABS.     000000        000
           000414        001
ERRORS DETECTED:  0
```

FREE CORE: 6416. WORDS
SNPTST,SNPTST=SNPTST

Figure 9-3
Sample Program That Calls for Snapshot Dumps

```
                        SNAPSHOT DUMP  ID: 64


TASK: SNPTST                                    TIME: 27-JUN-75 09:31

PC: 000724

REGS:      R0 - 000353   R1 - 074400   R2 - 000120   R3 - 140130

           R4 - 000000   R5 - 000002   SP - 000312   PS - 170000

TASK STATUS:   MSG -CHK WFR

EVENT FLAG MASKS:  <01-16> 000001  <17-32> 056502
                   <33-48> 000000  <49-64> 000000

CURRENT UIC: [200,200]   DSW: 1.

PRIORITY: 50.          I/O COUNT: 0.     TI DEVICE - TT10:

LOAD DEVICE - SY0:     LBN: 0,002265


FLOATING POINT UNIT

       STATUS - 044400

       R0 - 000000   000000   000000   000000
       R1 - 000000   000000   000000   000000
       R2 - 000000   000000   000000   000000
       R3 - 000000   000000   000000   000000
       R4 - 000000   000000   000000   000000
       R5 - 000000   000000   000000   000000

OVERLAY SEGMENTS LOADED

STARTING RELATIVE BLOCK: 000002   BASE: 000000   LENGTH: 001464
STARTING RELATIVE BLOCK: 000004   BASE: 001464   LENGTH: 000264


                    TASK IMAGE



           PARTITION: GEN        LIMITS: 000312 - 000532

    000310  001 000  045 000  114 120  000 000   !   %  LP   !
    000320  001 000  100 000  312 000  132 001   !   •  J  Z !
    000330  000 000  000 000  000 000  000 000   !             !
    000340  000 000  000 000  014 146  174 263   !          3!
                           .
                           .
                           .
    000500  006 206  337 045  371 377  046 000   !   ↓%    & !
    000510  002 003  261 000  005 001  346 035   !   ↑       !
    000520  176 377  346 025  051 002  377 210   !      )   !
    000530  300 025  350 000  367 011  312 001   !•      J !
```

Figure 9-4
Sample Snapshot Dump (Words Octal and RAD50)

SNAPSHOT DUMP ID: 1

TASK: SNPTST   TIME: 27-JUN-75 09:31

PC: 000530

REGS: R0 - 000000   R1 - 000000   R2 - 000000   R3 - 140130

           R4 - 000000   R5 - 000002   SP - 000312   PS - 170000

TASK STATUS: MSG -CHK WFR

EVENT FLAG MASKS: <01-16> 000001 <17-32> 056502
                   <33-48> 000000 <49-64> 000000

CURRENT UIC: [200,200]   DSW: 1.

PRIORITY: 50.   I/O COUNT: 0.   TI DEVICE - TT10:

LOAD DEVICE - SY0:   LBN: 0,002265

FLOATING POINT UNIT

     STATUS - 044400

     R0 - 000000   000000   000000   000000
     R1 - 000000   000000   000000   000000
     R2 - 000000   000000   000000   000000
     R3 - 000000   000000   000000   000000
     R4 - 000000   000000   000000   000000
     R5 - 000000   000000   000000   000000

OVERLAY SEGMENTS LOADED

STARTING RELATIVE BLOCK: 000002   BASE: 000000   LENGTH: 001464

                           TASK IMAGE

                 PARTITION: GEN   LIMITS: 000312 - 000532

     000310 000001 000025 050114 000000 | A   U L36 |
     000320 000001 000001 000312 000532 | A   A E.B HZ|
     000330 000000 000000 000000 000000 | |
     000340 000000 000000 063014 131574 | PMD ...|
                                   .
                                   .
                                   .
     000500 103006 022737 177771 000046 |UQ0 FBO 8I   8|
     000510 001402 000261 000405 016746 | SJ DQ FU D1N|
     000520 177576 012746 001051 104377 | 5F CTF M3 U61|
     000530 012700 000350 004767 000712 |CSH E2 AW KR|

                          Figure 9-5
              Sample Snapshot Dump (Bytes Octal and ASCII)

APPENDIX A

ERROR MESSAGES

A.1 TASK BUILDER ERROR MESSAGES

The Task Builder produces diagnostic and fatal error messages.   Error
messages are printed in the following forms:

        TKB -- *DIAG*-error-message

or

        TKB -- *FATAL*-error-message

Some errors are dependent upon correction from the terminal.   If the
user is entering text at the terminal, a diagnostic error message can
be printed, the error corrected, and the task building sequence
continued.   If the same error is detected by the Task Builder in an
indirect file, the Task Builder cannot request correction and thus the
error is termed fatal and the task build is aborted.

Some diagnostic error messages are simply informative and advise the
user of an unusual condition. If the user considers the condition
normal to his task, he can install and run the task image.

This appendix tabulates the error messages produced by the Task
Builder.   Most of the error messages are self-explanatory. The Task
Builder prints the text shown in this manual in upper case letters.
In some cases, the Task Builder prints the line in which the error
occurred, so that the user can examine the line which caused the
problem and correct it.

   0.   ILLEGAL GET COMMAND LINE ERROR

        System error.   (no recovery.)

   1.   COMMAND SYNTAX ERROR
        invalid-line

        The invalid-line printed has incorrect syntax.

   2.   REQUIRED INPUT FILE MISSING

        At least one input file is required for a task build.

24.   ALLOCATION FAILURE ON FILE file-name

The Task Builder could not acquire sufficient contiguous disk space to store the task image file. (If possible, delete unnecessary files on disk to make more room available.)

25.   I/O ERROR ON OUTPUT FILE file-name

This error may occur on any of the three output files.

26.   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.

27.   TRUNCATION ERROR IN MODULE module-name

An attempt has been made to load a global value greater than +127 or less than -128 into a byte. The low-order eight bits are loaded.

28.   number UNDEFINED SYMBOLS SEGMENT seg-name

The Memory Allocation File lists each undefined symbol by segment.

29.   INVALID KEYWORD IDENTIFIER
      invalid-line

The invalid-line printed contains an unrecognizable keyword.

30.   OPTION SYNTAX ERROR
      invalid-line

The invalid-line printed contains unrecognizable syntax.

31.   TOO MANY PARAMETERS
      invalid-line

The invalid-line printed contains a keyword with more parameters than required.

32.   ILLEGAL MULTIPLE PARAMETER SETS
      invalid-line

The invalid-line printed contains multiple parameters for a keyword which only allows a single parameter.

33.   INSUFFICIENT PARAMETERS
      invalid-line

The invalid-line contains a keyword with an insufficient number of parameters to complete the keyword meaning.

APPENDIX A. ERROR MESSAGES

34. TASK HAS ILLEGAL MEMORY LIMITS

    An attempt has been made to build a task whose size exceeds
    the partition boundary.

35. OVERLAY DIRECTIVE HAS NO OPERANDS
    invalid-line

    All overlay directives except .END require operands.

36. ILLEGAL OVERLAY DIRECTIVE
    invalid-line

    The invalid-line printed contains an unrecognizable overlay
    directive.

37. OVERLAY DIRECTIVE SYNTAX ERROR
    invalid-line

    The invalid-line printed contains a syntax error.

38. ROOT SEGMENT MULTIPLY DEFINED
    invalid-line

    The invalid-line printed contains the second .ROOT directive
    encountered. Only one .ROOT directive is allowed.

39. LABEL OR NAME IS MULTIPLY DEFINED
    invalid-line

    The invalid-line printed contains a name that has already
    appeared on a .FCTR, .NAME, or .PSECT directive.

40. NO ROOT SEGMENT SPECIFIED

    The overlay description did not contain a .ROOT directive.

41. BLANK P-SECTION NAME IS ILLEGAL
    invalid-line

    The invalid-line printed contains a .PSECT directive that
    does not have a p-section name.

42. ILLEGAL P-SECTION ATTRIBUTE
    invalid-line

    The invalid-line printed contains a p-section attribute that
    is not recognized.

43. ILLEGAL OVERLAY DESCRIPTION OPERATOR
    invalid-line

    The invalid-line printed contains an unrecognizable operator
    in an overlay description.

44. TOO MANY NESTED .ROOT/.FCTR DIRECTIVES
     invalid-line

     The invalid-line printed contains a .FCTR directive that exceeds the maximum nesting level (32).

45. TOO MANY PARENTHESES LEVELS
     invalid-line

     The invalid-line printed contains a parenthesis that exceeds the maximum nesting level (32).

46. UNBALANCED PARENTHESES
     invalid-line

     The invalid-line printed contains unbalanced parentheses.

47. not used.

48. 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.

49. ILLEGAL NUMBER OF LOGICAL UNITS
     invalid-line

     The invalid-line printed contains a number of logical unit greater than 250.

50. not used.

51. not used.

52. not used.

53. ILLEGAL DEFAULT PRIORITY SPECIFIED
     invalid-line

     The invalid-line printed contains a priority greater than 250.

54. ILLEGAL ODT OR TASK VECTOR SIZE

     SST vector size specified greater than 32 words.

55. ILLEGAL FILENAME
     invalid-line

     The invalid-line printed contains a wild card (*) in a file specification. The use of wild cards is prohibited.

56. not used.

57. LOOKUP FAILURE ON FILE filename
    invalid-line

    The invalid-line printed contains a filename which cannot be located in the directory.

58. ILLEGAL DIRECTORY
    invalid-line

    The invalid-line printed contains an illegal UIC.

59. INCOMPATIBLE REFERENCE TO A LIBRARY P-SECTION p-sect-name

    A task has attempted to reference more storage in a shared region than exists in the shared region definition.

60. ILLEGAL REFERENCE TO LIBRARY P-SECTION p-sect-name

    A task has attempted to reference a p-sect-name existing in a resident library (shared region) but has not named the library in a COMMON or LIBR keyword.

61. RESIDENT LIBRARY MEMORY ALLOCATION CONFLICT
    keyword-string

    One of the following problems has occurred:

    1. More than three shared regions have been specified.

    2. The same shared region has been specified more than once.

    3. Shared regions whose memory allocations overlap have been specified.

62. LOOKUP FAILURE RESIDENT LIBRARY FILE
    invalid-line

    No symbol table or task image file found for the shared region on SY0 under UIC [1,1].

63. INVALID ACCESS TYPE
    invalid-line

    Requested access to shared region was not RW or RO.

64. ILLEGAL PARTITION/COMMON BLOCK SPECIFIED
    invalid-line

    User defined base or length not on 32 word boundary.

65.   NO MEMORY AVAILABLE FOR LIBRARY library-name

Insufficient virtual memory available to cover total memory
needed by referenced shared regions (mapped system only).

66.   PIC LIBRARIES MAY NOT REFERENCE OTHER LIBRARIES
invalid-line

67.   ILLEGAL APR RESERVATION

APR specified on COMMON or LIBR keyword that is outside the
range 0-7;   or APR specified in an unmapped system.

68.   I/O ERROR LIBRARY IMAGE FILE

An I/O error has occurred during an attempt to open or read
the Task Image File of a shared region.

69.   not used.

70.   not used.

71.   INVALID APR RESERVATION

APR specified on a LIBR or COMMON keyword for an absolute
library.

72.   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 divide was set to zero.
(Probable cause- division by an undefined global symbol.)

73.   WORK FILE I/O ERROR

I/O error during an attempt to reference data stored by the
Task Builder in a work file.

74.   LOOKUP FAILURE ON SYSTEM LIBRARY FILE

The   Task   Builder   cannot   find   the   system   Library
(SY0:[1,1]SYSLIB.OLB) file to resolve undefined symbols.

75.   UNABLE TO OPEN WORK FILE

The work file is located on the same device as the Task
Builder.   (Work file device is not mounted or Task Builder
UIC not present on the device.)

76.   NO VIRTUAL MEMORY STORAGE AVAILABLE

Maximum permissible size of the work file exceeded (no
recovery).

APPENDIX A. ERROR MESSAGES

77. MODULE module-name NOT IN LIBRARY

   The Task Builder could not find the module named on the LB
   switch in the library.

78. INCORRECT LIBRARY MODULE SPECIFICATION
   invalid-line

   The invalid-line contains a module name with a non-Radix-50
   character.

79. LIBRARY FILE filename HAS INCORRECT FORMAT

   A module has been requested from a library file that has an
   empty module name table.

80. RESIDENT LIBRARY IMAGE HAS INCORRECT FORMAT
   invalid-line

   The invalid-line specifies a shared region that has one of
   the following problems:

   1. The library file image has a header.

   2. The library references another library with invalid
      address bounds (i.e., not on 4K boundary in a mapped
      system).

   3. The library has invalid address bounds.

81. PARTITION partition-name HAS ILLEGAL MEMORY LIMITS

   1. The partition-name defined in the host system has
      incompatible memory limits with respect to the target
      system.

   2. The user has attempted to build a privileged task in a
      partition whose length exceeds 12K.

82. INVALID PARTITION/COMMON BLOCK SPECIFIED
   invalid-line

   Partition is invalid for one of the following reasons:

   1. The Task Builder cannot find the partition name in the
      host system in order to get the base and length.

   2. The system is mapped, but the base address of the
      partition is not on a 4K boundary for a non-runnable task
      or is not 0 for a runnable task.

   3. The memory bounds for the partition overlap a shared
      region.

   4. The partition name is identical to the name of a
      previously defined COMMON or LIBR shared region.

     5.   The top address of the partition for a runnable task exceeds 32K-32 words for a mapped system or exceeds 28K-1 for an un-mapped system.

     6.   A system-controlled partition was specified for an un-mapped system.

83.   ABORTED VIA REQUEST
input-line

The input-line contains a request from the user to abort the task build.

A.2   CROSS REFERENCE ERROR MESSAGES

The following error messages are output by the Cross Reference Task. Each message is preceded by one of the following prefixes:

```
CRF -- *DIAG*  - <name of originating task> - MESSAGE
CRF -- *FATAL* - <name of originating task> - MESSAGE
```

1.   CREF INPUT FILE <file name> HAS ILLEGAL FORMAT

A premature end-of-file was encountered on the CREF data file. This problem should be reported to the Digital Software representative.

2.   FAILED TO DELETE FILE <file name>

Cross reference processor did not have delete privileges. Verify that the cross-reference task is installed to run under UIC [10,1].

3.   FILE <file name> NOT FOUND

File <file name> could not be located.

4.   ILLEGAL ERROR/SEVERITY CODE

The occurrence of this error indicates a software malfunction that should be reported to the Digital Software Representative.

5.   INPUT FROM UNKNOWN TASK

Cross reference processing requested by the originating task is not supported by CREF.

6.   I/O ERROR ON FILE <file name>

An error has been encountered while reading or writing the specified file. A possible hardware problem is indicated; or the device may have insufficient space to accommodate the CREF output file.

7.   INVALID OUTPUT FORMAT SPECIFIED

This message indicates an inconsistency in the data file submitted for CREF processing. The occurrence of this error should be reported to your Digital software representative.

APPENDIX A. ERROR MESSAGES

8. NO DYNAMIC STORAGE AVAILABLE

The Cross Reference task requires more working storage than is available within the area of memory owned by the task. If possible, install the task in a larger partition.

9. NO VIRTUAL MEMORY STORAGE AVAILABLE

The Cross-Reference processor work file storage requirements exceed 65,536 words. No recovery is possible from this error.

10. OPEN FAILURE ON FILE <file name>

The Cross-reference processor was unable to open the named file to append the cross-reference listing. Ensure that the cross-reference task is installed to run under UIC [10,1] to allow unrestricted access to all files.

11. SYMBOL TABLE SEARCH STACK OVERFLOW

Software error. Notify the DEC Software Representative.

12. **UNABLE TO OPEN WORKFILE**

Possible causes are:

1. Device not mounted
2. Device write protected

The workfile device is assigned to LUN 7 and is normally the device from which CRF was installed.

13. **WORK FILE I/O ERROR**

The Cross Reference Processor encountered an I/O error while reading or writing data on its workfile.

Possible causes:

1. **Device full**
2. **Hardware error**

APPENDIX B

TASK BUILDER DATA FORMATS

An object module is the fundamental unit of input to the Task Builder.

Object modules are created by any of the standard language processors (i.e., MACRO-11, FORTRAN, etc.) or the Task Builder itself (symbol definition file). The RSX-11M librarian (LBR) provides the ability to combine a number of object modules together into a single library file.

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 one 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 B-1.

An object module must begin with a Declare GSD record and end with an end-of-module record. Additional Declare GSD records may occur anywhere in the file but before an end-of-GSD record. An end-of-GSD record must appear before the end-of-module record. At least one relocation directory record must appear before the first text

information record. Additional relocation directory and text information records may appear anywhere in the file. The internal symbol directory records may appear anywhere in the file between the initial declare GSD and end-of-module records.

Object module records are variable length and are identified by a record type code in the first word of the record. The format of additional information in the record is dependent upon the record type.
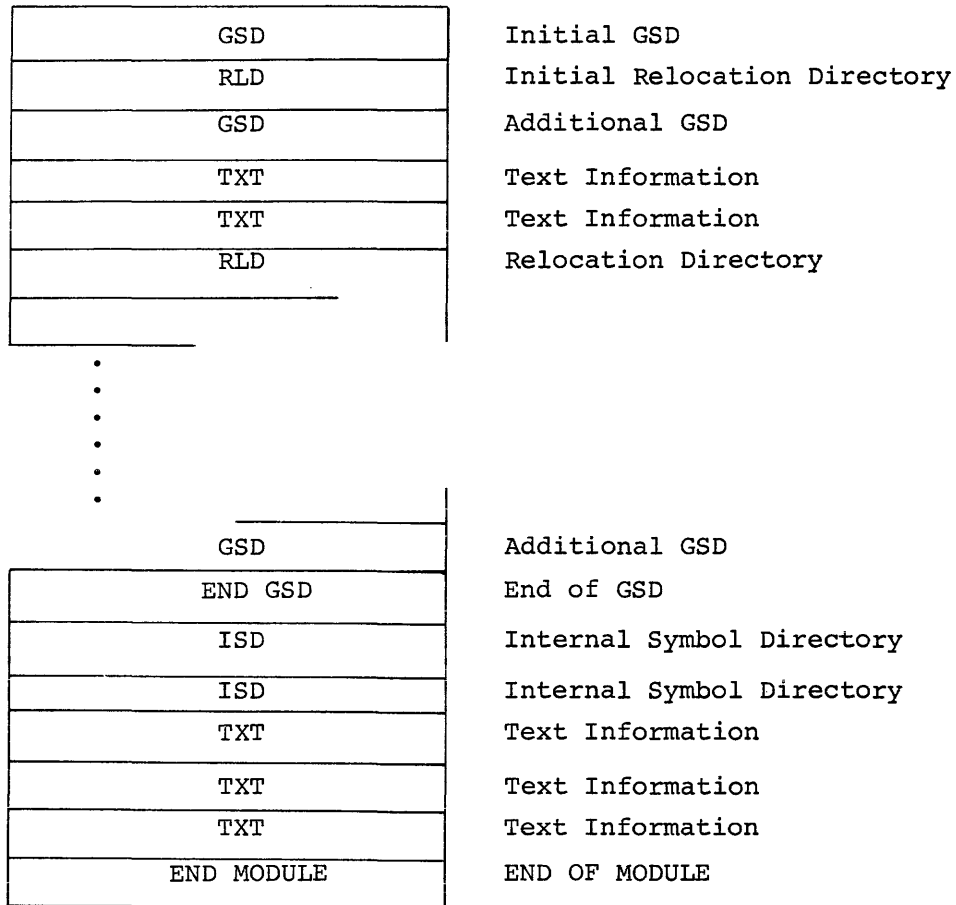
APPENDIX B. TASK BUILDER DATA FORMATS

GSD	Initial GSD
RLD	Initial Relocation Directory
GSD	Additional GSD
TXT	Text Information
TXT	Text Information
RLD	Relocation Directory
⋮	
GSD	Additional GSD
END GSD	End of GSD
ISD	Internal Symbol Directory
ISD	Internal Symbol Directory
TXT	Text Information
TXT	Text Information
TXT	Text Information
END MODULE	END OF MODULE

Figure B-1
General Object Module Format

B.1 GLOBAL SYMBOL DIRECTORY (GSD)

Global symbol directory 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, thus
significant time can be saved if all GSD records are placed at the
beginning of a module (i.e., less of the file must be read in phase
3).

APPENDIX B. TASK BUILDER DATA FORMATS

GSD records contain seven types of entries:

        Type 0 - Module Name

        Type 1 - Control Section Name

        Type 2 - Internal Symbol Name

        Type 3 - Transfer Address

        Type 4 - Global Symbol Name

        Type 5 - Program Section Name

        Type 6 - Program Version Identification

Each entry type is represented by four words in the GSD record.   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 B-2.

APPENDIX B. TASK BUILDER DATA FORMATS

0	RECORD TYPE = 1	
RAD50 NAME		
ENTRY TYPE	FLAGS	
VALUE		
RAD50 NAME		
TYPE	FLAGS	
VALUE		

.
.
.
.
.
.

RAD50 NAME	
TYPE	FLAGS
VALUE	
RAD50 NAME	
TYPE	FLAGS
VALUE	

Figure B-2
GSD Record and Entry Format

B.1.1 Module Name

The module name entry declares the name of the object module. The
name need not be unique with respect to other object modules (i.e.,
modules are identified by file not module name) but only one such
declaration may occur in any given object module. See Figure B-3.

APPENDIX B.   TASK BUILDER DATA FORMATS

MODULE NAME	
0	0
0	

Figure B-3
Module Name Entry Format

B.1.2   Control Section Name

Control sections, which include ASECTs, blank-CSECTS, and named-CSECTs
are supplanted in RSX-11M by PSECTs. For compatibility, the Task
Builder processes ASECTs and both forms of CSECTs. Section B.1.6
details the entry generated for a PSECT statement. In terms of a
PSECT statement we can define ASECT and CSECT statements as follows:

For a blank CSECT, a PSECT is defined with the following attributes:

        .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 B-4.

CONTROL SECTION NAME	
1	IGNORED
MAXIMUM LENGTH	

Figure B-4
Control Section Name Entry Format

APPENDIX B.   TASK BUILDER DATA FORMATS

B.1.3   Internal Symbol Name

The internal symbol name entry declares the name of an internal symbol
(with respect to the module). TKB does not support internal symbol
tables and therefore the detailed format of this entry is not defined
(Figure B-5). If an internal symbol entry is encountered while
reading the GSD, it is merely ignored.

```
+---------------------------------------+
|               SYMBOL                  |
|                NAME                   |
+-------------------+-------------------+
|        2          |         0         |
+-------------------+-------------------+
|              UNDEFINED                |
+---------------------------------------+
```

Figure B-5
Internal Symbol Name Entry Format

B.1.4   Transfer Address

The transfer address entry declares the transfer address of a module
relative to a P-section. The first two words of the entry define the
name of the P-section and the fourth word the relative offset from the
beginning of that P-section. 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 of 000001 relative to the default absolute
P-section (. ABS.) must be specified. See Figure B-6.

```
+---------------------------------------+
|               SYMBOL                  |
|                NAME                   |
+-------------------+-------------------+
|        3          |         0         |
+-------------------+-------------------+
|               OFFSET                  |
+---------------------------------------+
```

Figure B-6
Transfer Address Entry Format

NOTE

If the P-section is absolute,
then OFFSET is the actual
transfer   address   if   not
000001.

APPENDIX B. TASK BUILDER DATA FORMATS

B.1.5 Global Symbol Name

The global symbol name entry (Figure B-7) declares either a global reference or a definition. All definition entries must appear after the declaration of the P-section under which they are defined and before the declaration of another P-section. Global references may 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 P-section under which it is defined.

The flag byte of the symbol declaration entry has the following bit assignments.

Bits 0 - 2 - Not used.

Bit 3 - Definition

    0 = Global symbol references.

    1 = Global symbol definition.

Bit 4 - Not used

Bit 5 - Relocation

    0 = Absolute symbol value.

    1 = Relative symbol value

Bit 6 - 7 - Not used.

SYMBOL NAME	
4	FLAGS
VALUE	
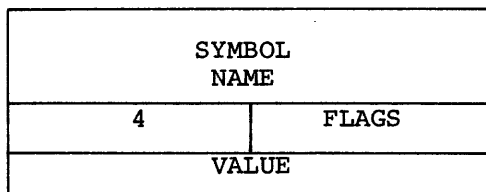
Figure B-7
Global Symbol Name Entry Format

APPENDIX B. TASK BUILDER DATA FORMATS

B.1.6 Program Section Name

The P-section name entry (Figure B-8) declares the name of a P-section and its maximum length in the module. It also declares the attributes of the P-section via the flag byte.

GSD records must be constructed such that once a P-section name has been declared all global symbol definitions that pertain to that P-section must appear before another P-section name is declared. Global symbols are declared via symbol declaration entries. Thus the normal format is a P-section name followed by zero or more symbol declarations, followed by another P-section name followed by zero or more symbol declarations, and so on.

The flag byte of the P-section entry has the following bit assignments:

Bit 0 - Memory Speed

    0 = P-section is to occupy low speed (core) memory.

    1 = P-section is to occupy high speed (i.e., MOS/Bipolar) memory.

Bit 1 - Library P-section

    0 = Normal P-section.

    1 = Relocatable P-section that references a core resident library
        or common block.

Bit 2 - Allocation

    0 = P-section references are to be concatenated with   other
        references to the same P-section to form the total memory
        allocated to the section.

    1 = P-section references are to be overlaid.   The total memory
        allocated to the P-section is the largest request made by
        individual references to the same P-section.

Bit 3 - Not used but reserved.

Bit 4 - Access

    0 = P-section has read/write access.

    1 - P-section has read-only access.

Bit 5 - Relocation

    0 = P-section is absolute and requires no relocation.

1 = P-section is relocatable and references to the control section
    must have a relocation bias added before they become
    absolute.

Bit 6 - Scope

0 = The scope of the P-section is local. References to the same
    P-section will be collected only within the segment in which
    the P-section is defined.

1 = The scope of the P-section is global.   References to the
    P-section are collected across segment boundaries.   The
    segment in which a global P-section is allocated storage is
    determined either by the first module that defines the
    P-section on a path or by direct placement of a P-section in
    a segment via the Overlay Description Language .PSECT
    directive.

Bit 7 - Type

0 = The P-section contains instruction (I) references.

1 = The P-section contains data (D) references.

P-SECTION NAME	
5	FLAGS
MAX LENGTH	

Figure B-8
P-Section Name Entry Format

NOTE

The length of all absolute
sections is zero.

B.1.7 Program Version Identification

The program version identification entry (Figure B-9) declares the
version of the module. TKB 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 contains the version identification.
The flag byte and fourth words are not used and contain no meaningful
information.

SYMBOL NAME	
6	0
0	

Figure B-9
Program Version Identification Entry Format

B.2   END-OF-GLOBAL-SYMBOL-DIRECTORY

The end-of-global-symbol-directory record (Figure B-10) declares that
no other GSD records are contained further on in the file. Exactly
one end-of-GSD-record must appear in every object module and is one
word in length.

0	RECORD TYPE = 2

Figure B-10
End of GSD Record Format

B.3   TEXT INFORMATION

The text information record (Figure B-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 may contain words and/or bytes of information whose final
contents are yet to be determined. This information will be bound by
a relocation directory record that immediately follows the text record
(see B.4 below). If the text record does not need modification, then
no relocation directory record is needed. Thus multiple text records
may appear in sequence before a relocation directory record.

The load address of the text record is specified as an offset from the
current P-section base. At least one relocation directory record must
precede the first text record. This directory must declare the
current P-section.

TKB 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.

0	RECORD TYPE = 3
LOAD ADDRESS	
TEXT	TEXT
"	TEXT
"	"

.
.
.
.
.
.

"	"
"	"
"	"
"	TEXT
TEXT	TEXT

Figure B-11
Text Information Record Format

B.4   RELOCATION DIRECTORY

Relocation directory records (Figure B-12) contain the information necessary to relocate and link a 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 it defines the current P-section and location. Relocation directory records contain 13 types

of entries. These entries are classified as relocation or location modification entries. The following types of entries are defined:

Type 1 - Internal Relocation

Type 2 - Global Relocation

Type 3 - Internal Displaced Relocation

Type 4 - Global Displaced Relocation

Type 5 - Global Additive Relocation

Type 6 - Global Additive Displaced Relocation

Type 7 - Location Counter Definition

Type 10 - Location Counter Modification

Type 11 - Program Limits

Type 12 - P-Section Relocation

Type 13 - Not used

Type 14 - P-Section Displaced Relocation

Type 15 - P-Section Additive Relocation

Type 16 - P-Section Additive Displaced Relocation

Type 17 - Complex Relocation

Each type of entry is represented by a command byte (specifies type of entry and word/byte modification), followed by a displacement byte, followed 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 previous text information record, (see B.3 above) 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
          may 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 (i.e., the modification value has a
       magnitude greater than 255), an error is produced.

APPENDIX B.   TASK BUILDER DATA FORMATS

0	RECORD TYPE = 4
DISP	CMD
INFO	INFO
"	INFO
"	"
"	"
"	"
"	"
"	"
"	"

•
•
•

CMD	"
INFO	DISP
"	INFO
"	"
"	"
"	"
"	"
DISP	CMD
INFO	INFO
INFO	INFO
INFO	INFO

Figure B-12
Relocation Directory Record Format

APPENDIX B. TASK BUILDER DATA FORMATS

B.4.1 Internal Relocation

This type of entry (Figure B-13) relocates a direct pointer to an
address within a module. The current P-section base address is added
to a specified constant and the result is written into the task image
file at the calculated address (i.e., displacement byte added to value
calculated from the load address of the previous text block).

Example:

A: MOV   #A,R0

        or

        .WORD   A

DISP	B	1
CONSTANT		
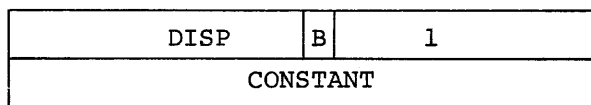
Figure B-13
Internal Relocation Command Format

B.4.2 Global Relocation

This type of entry (Figure B-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		
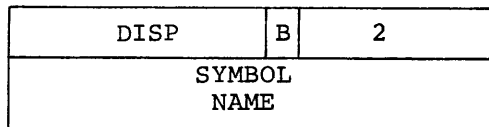
Figure B-14
Global Relocation

APPENDIX B. TASK BUILDER DATA FORMATS

B.4.3 Internal Displaced Relocation

This type of entry (Figure B-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		
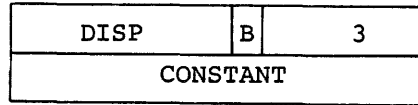
Figure B-15
Internal Displaced Relocation

B.4.4 Global Displaced Relocation

This type of entry (Figure B-16) relocates a relative reference to 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. This value is then written into the task image file at the calculated address.

Example:

     CLR GLOBAL

    or

     MOV GLOBAL,R0

DISP	B	4
SYMBOL NAME		
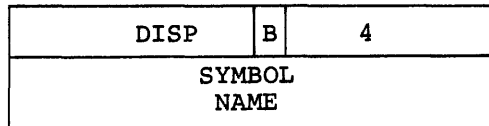
Figure B-16
Global Displaced Relocation

APPENDIX B. TASK BUILDER DATA FORMATS

B.4.5 Global Additive Relocation

This type of entry (Figure B-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

DISP	B	5
SYMBOL NAME		
CONSTANT		

Figure B-17
Global Additive Relocation

B.4.6 Global Additive Displaced Relocation

This type of entry (Figure B-18) relocates a relative reference to a
global symbol with an additive constant. The definition of the global
symbol is obtained and the 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
resultant value 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 B-18
Global Additive Displaced Relocation

APPENDIX B.   TASK BUILDER DATA FORMATS

B.4.7   Location Counter Definition

This type of entry (Figure B-19) declares a current P-section 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
SECTION NAME		
CONSTANT		

Figure B-19
Location Counter Definition

B.4.8   Location Counter Modification

This type of entry (Figure B-20) modifies the current location counter. The current P-section base is added to the specified constant and the result is stored as the current location counter.

Example:

    .=.+N

or

    .BLKB   N

0	B	10
CONSTANT		

Figure B-20
Location Counter Modification

APPENDIX B. TASK BUILDER DATA FORMATS

B.4.9 Program Limits

This type of entry (Figure B-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 tasks 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 B-21
Program Limits

B.4.10 P-Section Relocation

This type of entry (Figure B-22) relocates a direct pointer to the beginning address of another P-section (other than the P-section in which the reference is made) within a module. The current base address of the specified P-section 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

```
┌────────────────┬───┬────────────┐
│      DISP      │ B │     12     │
├────────────────┴───┴────────────┤
│            SECTION               │
│             NAME                 │
└──────────────────────────────────┘
```

Figure B-22
P-Section Relocation

APPENDIX B. TASK BUILDER DATA FORMATS

B.4.11 P-Section Displaced Relocation

This type of entry (Figure B-23) relocates a relative reference to the
beginning address of another P-section within a module. The current
base address of the specified P-section is obtained and the address
plus 2 that the relocated value is to be written into is subtracted
from the base value. This value is then written into the task image
file at the calculated address.

Example:

        .PSECT A
B:

        .
        .
        .
        .
        .PSECT C
        MOV   B,R0

DISP	B	11
SECTION NAME		

Figure B-23
P-Section Displaced Relocation

B.4.12 P-Section Additive Relocation

The type of entry (Figure B-24) relocates a direct pointer to an
address in another P-section within a module.   The current base
address of the specified P-section is obtained and added to the
specified constant. The result is written into the task image file at
the calculated address.

APPENDIX B. TASK BUILDER DATA FORMATS

Example:

```
        .PSECT   A
B:
          •
          •
          •
          •
          •
C:        •
          •
          •
          •
        PSECT    D
        MOV      #B+10,R0
        MOV      #C,R0

        or

        .WORD    B+10
        .WORD    C
```

DISP	B	15
SECTION NAME		
CONSTANT		

Figure B-24
P-Section Additive Relocation

B.4.13 P-Section Additive Displaced Relocation

This type of entry (Figure B-25) relocates a relative reference to an
address in another P-section within a module.   The current base
address of the specified P-section 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. This
value is then written into the task image file at the calculated
address.

APPENDIX B. TASK BUILDER DATA FORMATS

Example:

```
        .PSECT  A
B:
        .
        .
        .
C:      .
        .
        .
        .
        .
        .PSECT  D

        MOV     B+10,R0
        MOV     C,R0
```

DISP		B	16
SECTION NAME			
CONSTANT			

Figure B-25
P-Section Additive Displaced Relocation

B.4.14 Complex Relocation

This type of entry (Figure B-25A) resolves a complex relocation expression. Such an expression is one in which any of the MACRO-11 binary or unary operations are permitted with any type of argument, regardless of whether the argument is unresolved global, relocatable to any P-section base, absolute, or a complex relocatable subexpression.

The RLD command word is followed by a string of numerically-specified operation codes and arguments. All of the operation codes occupy 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 (/)

APPENDIX B.   TASK BUILDER DATA FORMATS

    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.

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 are to be noted.

  1. An attempt to divide by zero yields a zero result. The task Builder issues a nonfatal diagnostic.

  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 3.4.

  3. All operations are performed on relocated (additive) or absolute 16-bit quantities. PC displacement is applied to the result only.

Example:

    .PSECT   ALPHA
A:

      •
      •
      •

    .PSECT   BETA
B:

      •
      •
      •

APPENDIX B.   TASK BUILDER DATA FORMATS

       MOV   #A+B-G1/G2&<↑C<177120!G3>>,R1

DISP	B	17
COMPLEX STRING		
12		

Figure B-25A
Complex Relocation

B.5   INTERNAL SYMBOL DIRECTORY

Internal symbol directory records (Figure B-26) declare definitions of
symbols that are local to a module. This feature is not supported by
TKB and therefore a detailed record format is not specified.   If TKB
encounters this type of record, it will ignore it.

0	RECORD TYPE = 5
NOT SPECIFIED	

Figure B-26
Internal Symbol Directory Record Format

B.6   END OF MODULE

The end-of-module record (Figure B-27) declares the end-of-an object
module.   Exactly one end of module record must appear in each object
module and is one word in length.

0	RECORD TYPE = 6

Figure B-27
End-of-Module Record Format

APPENDIX C

TASK IMAGE FILE STRUCTURE

The task image as it is recorded on the disk appears in Figure C-1.

AUTOLOAD VECTORS CO-TREE OVERLAY	BLOCK
AUTOLOAD VECTORS CO-TREE ROOT	BLOCK
AUTOLOAD VECTORS	
MAIN TREE OVERLAY	BLOCK
AUTOLOAD VECTORS SEGMENT TABLES	
ROOT SEGMENT CODE & DATA	
STACK FP/EA SAVE AREA HEADER	BLOCK
CHECKPOINT AREA	BLOCK
LABEL	
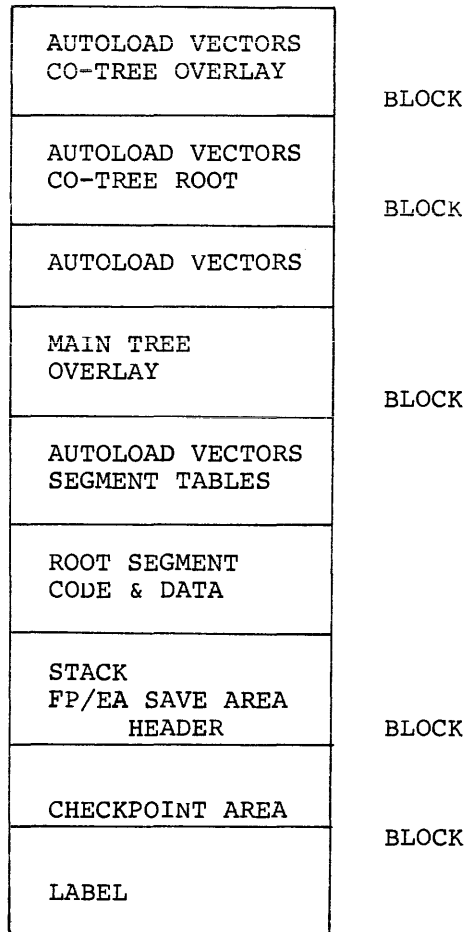
Figure C-1
Task Image on Disk

APPENDIX C. TASK IMAGE FILE STRUCTURE

C.1 LABEL BLOCK GROUP

The label block group, shown in Figure C-2, precedes the task on the disk, and contains data that need not be resident during task execution, and up to two blocks containing device assignment data for LUNs 1-255. The task label blocks (first block in group) are read and verified by Install. The information in these blocks is used to fill in the task header.

APPENDIX C. TASK IMAGE FILE STRUCTURE

```
LABEL
L$BSTK   0   ┌─────────────────────────────────┐
             │              TASK               │
         2   ├─────────────────────────────────┤
             │              NAME               │
L$BPAR   4   ├─────────────────────────────────┤
             │         TASK PARTITION          │
         6   ├─────────────────────────────────┤
             │        in radix 50 format       │
L$BFLG  10   ├─────────────────────────────────┤
             │         TASK FLAG WORD          │
L$BPRI  12   ├─────────────────────────────────┤
             │         TASK PRIORITY           │
L$BLDZ  14   ├─────────────────────────────────┤
             │   LOAD SIZE IN 32-WORD BLOCKS   │
L$BMXZ  16   ├─────────────────────────────────┤
             │   MAX SIZE IN 32-WORD BLOCKS    │
L$BSA   20   ├─────────────────────────────────┤
             │      TASK STARTING ADDRESS      │
L$BHRB  22   ├─────────────────────────────────┤
             │      HEADER RELATIVE BLOCK      │
L$BBLK  24   ├─────────────────────────────────┤
             │    NUMBER OF BLOCKS IN TABLE    │
L$BXFR  26   ├─────────────────────────────────┤
             │      TASK TRANSFER ADDRESS      │
L$BDAT  30   ├─────────────────────────────────┤
             │        YEAR       YEAR          │
        32   ├─────────────────────────────────┤
             │ CREATION           MONTH        │
        34   ├─────────────────────────────────┤
             │ DATE               DAY          │
L$BLIB  36   ├─────────────────────────────────┤  ┐
             │      LIBRARY/COMMON NAME        │  │
        40   ├─────────────────────────────────┤  │
             │       IN RADIX 50 FORMAT        │  │  LIBRARY
        42   ├─────────────────────────────────┤  │  REQUEST
             │  LIBRARY LENGTH  (32w blks)     │  │
        44   ├─────────────────────────────────┤  │  (maximum of
             │ CREATION           YEAR         │  │   three 8
        46   ├─────────────────────────────────┤  │   word entries)
             │ DATE               MONTH        │  │
        50   ├─────────────────────────────────┤  │
             │                    DAY          │  │
        52   ├─────────────────────────────────┤  │
             │ STARTING ADDRESS                │  │
        54   ├─────────────────────────────────┤  │
             │         LIBRARY FLAGS           │  │
             └─────────────────────────────────┘  ┘

                          •
                          •

L$BEXT 116   ┌─────────────────────────────────┐    (In increments of
             │         TASK EXTENSION          │     32 words)
             └─────────────────────────────────┘
```

Figure C-2
Label Block Group

746	
750	FILE
752	ID
754	
756	FILENAME
760	
762	TYPE
764	VERSION
766	
770	DIRECTORY ID
772	
774	DEVICE NAME
776	UNIT

----	.

LUN BLOCK 1	DEVICE NAME	LUN 1
	UNIT NUMBER	
	. . .	
	DEVICE NAME	LUN 127
	UNIT NUMBER	
	.	
LUN BLOCK 2	DEVICE NAME	LUN 128
	UNIT NUMBER	
	. . .	
	DEVICE NAME	LUN 255
	UNIT NUMBER	

Figure C-2 (Cont.)
Label Block Group

APPENDIX C.   TASK IMAGE FILE STRUCTURE

C.1.1   Label Block Details

The information contained in the label block is verified by the install task in creating a system task directory entry for the task, and in linking the task to resident shared regions.

L$BTSK   Task name, consisting of two words in Radix-50 format. The value of this parameter is set by the TASK keyword.

L$BPAR   Partition name, consisting of two words in Radix-50 format. Its value is set by the PAR keyword.

L$BFLG   Task flag word containing bit values that are set or cleared depending on defined task attributes. Attributes are established by appending the appropriate switches to the task image file specification.

	Bit	Attribute if Bit=1
TS$CHK	6	Task is non-checkpointable
TS$PRV	8	Task is privilieged
TS$ACP	13	Task is ancillary control processor
TS$NHD	14	Task image does not have header
TS$PIC	15	Task is position independent

L$BPRI   Default priority, set by the PRI keyword.

L$BLDZ   Load size of the task, expressed in multiples of 32-word blocks.   The value of L$BLDZ is equal to the size of the root segment, in multi-segment tasks.

L$BMXZ   Maximum size of the task, expressed in multiples of 32-word blocks. The header size is included.

                L$BMXZ is used by Install to verify that the task fits into the specified partition.

L$BSA   Starting address of task. Marks the base address of the Task image in the addressing space.

L$BHRB   Relative block of the header with respect to the first block in the task file.

L$BBLK   Number of blocks in the Label Block group.

L$BXFR   Transfer address of the task. Used by BOOT to load and start a bootable core image; for example: the resident executive.

APPENDIX C. TASK IMAGE FILE STRUCTURE

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$BEXT   Task extension size in 32 word blocks. The value of
                  this parameter is set by means of the EXTTSK keyword.

The following paragraphs describe components of the Resident Library
Name Block. An 8-word block is generated for each Resident Library
referenced by the task. Because shared regions need not be resident
in the system, the Task Builder builds the block from the region's
disk image, using information in the header and label blocks.

Library Name   A 2-word Radix-50 name specified in the LIBR or COMMON
                keyword.

Length   Length of the shared region in 32 word blocks. The
                INSTALL routine verifies that this value does not
                exceed the size of the resident common block area.

Creation Date   Obtained from the creation date in the library image
                label block.

Starting Address First address used to map the Library into the task
                addressing space.

                Flag Word Bits 2, 14 and 15 are used as follows:

                      Bit   Value   Meaning

        LD$REL   2   0   Library is absolute
                            1   Library is PIC
        LD$RSV   14   0   Reserved

        LD$ACC   15   0   Access request type is Read only
                            1   Access request type is Read
                                    Write

C.2 HEADER

The task is read into memory starting at the base of the Header.
Since the root segment is a contiguous set of disk blocks it is loaded
with a single disk access. Figure C-3 illustrates the format of the
fixed part, and Figure C-4 the Logical Unit Table. The Floating Point
Save Area is storage for the PDP-11/45 floating point registers when
this option is requested.

The task header starts on a block boundary and is immediately followed
by the task image.

In an unmapped system, the header is fully accessible to the task. In
a mapped system, the Executive copies the header of an active task to
protected memory and restores the header contents when the task is
completed or checkpointed.

APPENDIX C. TASK IMAGE FILE STRUCTURE

H.CSP	0	CURRENT STACK POINTER
H.HDLN	2	LENGTH OF HEADER (bytes)
H.PCBT	4	TASK PCB
	6	
	10	
	12	
H.PCBC	14	COMMON/LIBR PCB#1
	16	
	20	
	22	
	24	COMMON/LIBR PCB#2
	26	
	30	
	32	
	34	COMMON/LIBR PCB#3
	36	
	40	
	42	
	44	END OF PCB DESCRIPTORS
H.DSW	46	DSW CONTEXT SAVE
H.FCS	50	FCS CONTEXT SAVE/PTR
H.FOR	52	FORTRAN OTS CONTEXT SAVE/PTR
H.OVLY	54	OVERLAY RUNTIME SYSTEM PTR
H.RSVD	56	RESERVED

Impure Area Pointers (for H.FCS, H.FOR, H.OVLY)

Figure C-3
Task Header Fixed Part

APPENDIX C.   TASK IMAGE FILE STRUCTURE

Label	Offset	Description
H.EFLM	60	EVENT FLAG MASK WORD (1-16)
	62	" (17-32)
	64	" (33-48)
	66	" (49-64)
H.CUIC	70	CURRENT UIC
H.DUIC	72	DEFAULT UIC
H.IPS	74	INITIAL PS
H.IPC	76	INITIAL PC
H.ISP	100	INITIAL SP
H.ODVA	102	ODT SST VECTOR ADDRESS
H.ODVL	104	ODT SST VECTOR LENGTH
H.TKVA	106	TASK SST VECTOR ADDRESS
H.TKVL	110	TASK SST VECTOR LENGTH
H.PFVA	112	POWER FAIL AST CONTROL BLOCK
H.FPVA	114	FLOATING POINT AST CONT BLK
H.RCVA	116	RECEIVE AST CONTROL BLOCK
	120	RESERVED
H.FPSA	122	FLOATING POINT/EAE SAVE PTR
	124	RESERVED
H.GARD	126	HEADER GUARD WORD POINTER
H.NLUN	130	NUMBERS OF LUNS

Figure C-3 (Cont.)
Task Header Fixed Part

APPENDIX C. TASK IMAGE FILE STRUCTURE

H.LUN

LUN TABLE (2 words/LUN)
CURRENT PS
CURRENT PC
CURRENT R5
CURRENT R4
CURRENT R3
CURRENT R2
CURRENT R1
CURRENT R0
HEADER GUARD WORD

INITIAL VALUES

relative block number of header
ident word #2
ident word #1
task name word #2
task name word #1
program transfer address
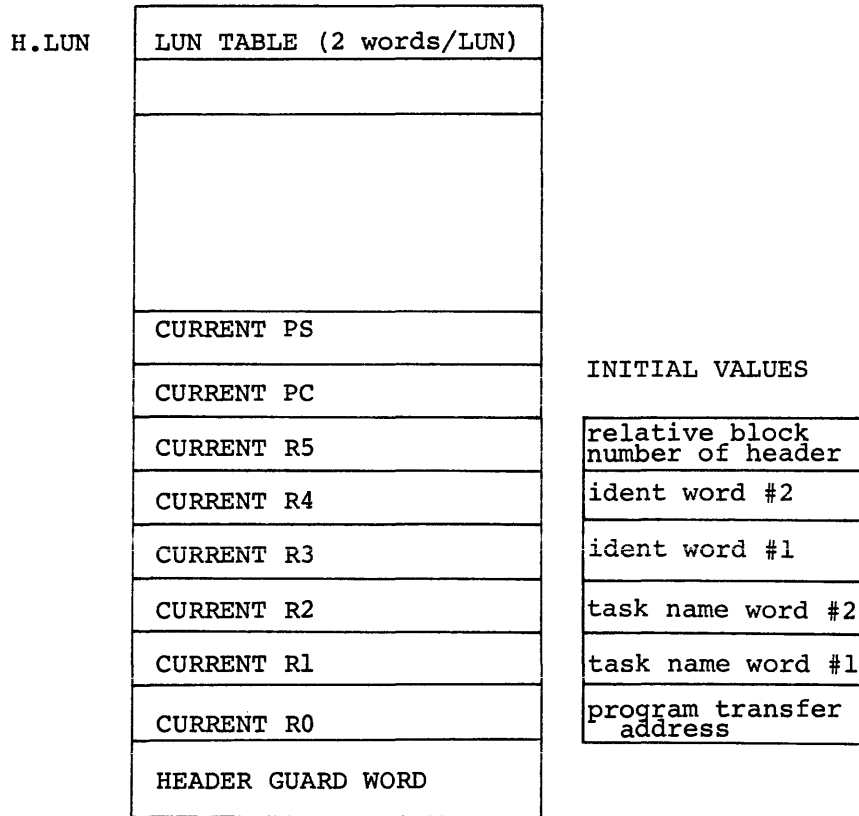
Figure C-4
Task Header Variable Part

APPENDIX C. TASK IMAGE FILE STRUCTURE

C.2.1 Low Core Context

The low core context for a task consists of the Directive Status Word
and the Impure Area Pointers. The Task Builder predefines the
symbolic reference names as follows:

$DSW   Directive Status Word

.FSRPT File Control Services work area and buffer pool
        Pointer

$OTSV   FORTRAN OTS work area Pointer

N.OVPT Overlay Runtime System work area Pointer

The only proper reference to these pointers is by symbolic name. The
pointers are read-only. If they are written into, the result will be
lost on the next context switch.

The Directive Status Word is a one word area used to report the
results of an Executive Directive.

The Impure Area Pointers are necessary to satisfy the re-entrancy
requirements of the associated routines.

C.2.2 Logical Unit Table Entry

Each entry in the Logical Unit Table has the form shown in Figure C-5.

APPENDIX C.   TASK IMAGE FILE STRUCTURE

UCB ADDR
WINDOW BLOCK POINTER
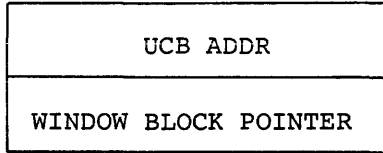
Figure C-5
Logical Unit Table Entry

The first word contains the address of the device unit control block in the Executive system tables that contains device dependent information.

The second word is a pointer to the window block if the device is file-structured.

The UCB address is set at install-time if a corresponding ASG parameter is specified at task-build-time. This word can also be set at run-time with the Assign Lun Directive to the Executive.

The window block pointer is set when a file is opened on the device whose UCB address is specified by word 1. The window block pointer is cleared when the file is closed.

C.3   SEGMENT TABLES

The Segment Table contains a segment descriptor for every segment in the task. The segment descriptor is formatted as shown in Figure C-6. If the autoload method is used, the segment descriptor is six words in length.   If the manual load method is used, the segment descriptor is expanded to be eight words in length to include the segment name.

STATUS	REL. DISK ADDRESS
LOAD ADDRESS	
LENGTH IN BYTES	
LINK UP	
LINK DOWN	
LINK NEXT	
SEGMENT	
NAME	

Figure C-6
Segment Descriptor

C.3.1 Status

The status bit is used in the autoload method to determine if an overlay is in memory, that is:

        bit 12 = 0   segment is in memory.

        bit 12 = 1   segment is not in memory.

C.3.2 Relative Disk Address

Each segment begins on a block boundary and occupies a contiguous disk area to allow an overlay to be loaded by a single device access. The relative disk address is the relative block number of the overlay segment from the start of the task image. The maximum relative block number can not exceed 4096 since twelve bits are allocated for the relative disk address.

C.3.3 Load Address

The load address contains the address into which the loading of the overlay segment starts.

APPENDIX C. TASK IMAGE FILE STRUCTURE

C.3.4 Segment Length

The segment length contains the length of the overlay segment in bytes and is used to construct the disk read.

C.3.5 Link Up

The link up is a pointer to a segment descriptor away from the root.

C.3.6 Link Down

The link down is a pointer to a segment nearer the root.

C.3.7 Link Next

The link next is a pointer to the adjoining segment descriptor. When a segment is loaded, the loading routine follows the link next to determine if a segment in memory is being overlaid and should therefore be marked out-of-memory.

The link next pointers are linked in a circular fashion:

Consider the tree:

```
            A21    A22
             |_____|
                |
   A1          A2
    |_____|
          |
         A0
```
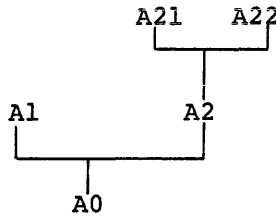

The segment descriptors are linked in the following way:

```
   A21    A22        A21    A22        A21 ⇄ A22

  A1     A2          A1    A2          A1 ⇄ A2

      A0                 A0               A0

   link up           link down        link next
```
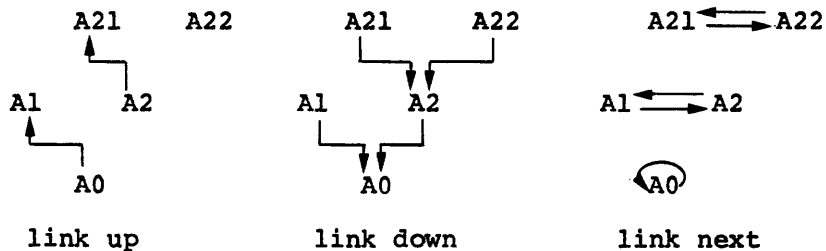
APPENDIX C. TASK IMAGE FILE STRUCTURE

If there is a co-tree, the link next of the segment descriptor for the
root points to the segment descriptor for the root segment of the
co-tree.

C.4 AUTOLOAD VECTORS

Autoload vectors appear in every segment that references autoload
entry points in segments that are farther from the root than the
referencing segment.

The autoload vector table consists of one entry per autoload entry
point in the form shown in Figure C-7.

JSR   PC
$AUTO
SEGMENT DESCRIPTOR ADDR.
ENTRY POINT ADDRESS

Figure C-7
Autoload Vector Entry

C.5 ROOT SEGMENT

The root segment is written as a contiguous number of blocks.

C.6 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 32K-32 words.

APPENDIX D

RESERVED SYMBOLS

Several global symbol and p-section* 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.

The following global symbols are reserved by the Task Builder:

GLOBAL SYMBOL	MODIFICATION VALUE
.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.
.NSTBL	The address of the segment description tables. Note that this location is modified only when the number of segments is greater than one.
.TRLUN	The trace subroutine output logical unit number.
.ODTL1	Logical unit number for the ODT terminal device TI:.
.ODTL2	Logical unit number for the ODT line printer device CL:.

----

* P-sections are created by .ASECT, .CSECT, or .PSECT directives. The
.PSECT directive obviates the need for either the .ASECT or .CSECT
directives, these being retained for compatibility only.   In this
document all sections will be referred to as p-sections unless the
specific characteristics of .ASECTS or .CSECT apply.

APPENDIX D.   RESERVED SYMBOLS

The definition of a reserved p-section causes that p-section to be extended if the appropriate option input is specified (see section 3.2.3.4).

The following p-section names are reserved by the Task Builder:

SECTION NAME	EXTENSION LENGTH
$$DEVT	The extension length (in bytes) is calculated from the formula $$EXT = <S.FDB+52>*UNITS$$ Where the definition of S.FDB is obtained from the root segment symbol table and UNITS is the number of logical units used by the task, excluding the Message Output, Overlay , and ODT units.
$$FSR1	The extension of this section is specified by the ACTFIL option input.
$$IOB1	The extension of this section is specified by the MAXBUF option input.
$$OBF1	FORTRAN OTS uses this area to parse array type format specifications. May be extended by FMTBUF keyword.

APPENDIX E

TAILORING THE TASK BUILDER

There are several ways in which the performance of the Task Builder
can be improved by making use of the resources of a given system.

Like most system programs, the Task Builder is heavily overlaid.
There are two versions of the Task Builder available: BIGTKB, which
occupies more storage and runs faster than TKB and TKB, which has more
overlays, occupies less storage and runs slower.

In order to minimize storage requirements, the Task Builder uses a
work file for storing symbol definitions and other tables. The work
file is organized as a virtual memory file. When tables exceed the
available memory, the information is displaced to the work file and
retrieved when it is required. The work file and the Task Builder
usually exist on the same device, namely: SY0.

The following techniques are available for improving the performance
of the Task Builder based on the resources of the system on which the
Task Builder is to run.

1.   The appropriate version of the Task Builder should be chosen;
     that is, one which conserves space or time depending on the
     system's requirements.

2.   If a device with a faster response time is available, the
     Task Builder should be moved to that device.

3.   If additional memory is available, the Task Builder should be
     installed in a larger partition so that it can make use of
     the extra memory as dynamic storage.

4.   If two moving head disks are available the Task Builder
     should be moved to one disk and the work file to another by
     re-assigning LUN 8. There will be less head movement in this
     case and the disks can, therefore, respond faster.

If the user has the resources to rebuild the Task Builder, he can
alter some parameters at build time which affect the Task Builder's
performance.

APPENDIX E. TAILORING THE TASK BUILDER

1. W$KEXT - defines the number of blocks by which the work file
             is extended when an extension is required. If
             W$KEXT is increased, the access to the work file
             will be faster.

2. N$MPAG - defines a threshold which determines whether a fast
             or slow work file page search is used. The fast
             page search saves about 15% of the execution time,
             but requires 256 words of the Task Builder's dynamic
             storage. This threshold defines the minimum page
             storage capacity of dynamic memory required for the
             fast search method. It is currently set at 20.

APPENDIX F

INCLUDING A DEBUGGING AID

If the user wants to include a program which controls the execution of the task he is building, he can do so by naming the appropriate object module as an input file and applying the /DA switch.

When such a program is input, the Task Builder causes control to be passed to the program when the task execution is initiated.

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 effect:

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 G

RSX-11M TASK BUILDER GLOSSARY

AUTOLOAD -   The method of loading overlay segments, in
                                which the Overlay Runtime 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 Runtime System.

GLOBAL COMMON BLOCK -   An area of memory reserved for a resident
                                library or common block.

GLOBAL CROSS REFERENCE -   A list of global symbols, in alphabetical
                                order, accompanied by the name of each refer-
                                encing module.

GLOBAL SYMBOL -   A symbol whose definition is known outside
                                the defining module.

HOST SYSTEM -   The system on which the task is built.

MAIN PARTITION -   A partition whose memory may be subdivided
                                into fixed-length sub-partitions or dynam-
                                ically allocated to each task by the
                                Executive (system-controlled partitions).

MAIN TREE-   An overlay tree whose root segment is loaded
                                by the Monitor when the task is made active.

MANUAL LOAD -   The method of loading overlay segments in
                                which the user includes explicit calls in his
                                routines to load overlays and handles
                                unsuccessful load requests.

MEMORY ALLOCATION FILE -   The output file created by the Task Builder
                                that describes the allocation of task memory.

OVERLAY DESCRIPTION LANGUAGE - A language that describes the overlay
                                structure of a task.

APPENDIX G.   RSX-11M TASK BUILDER GLOSSARY

OVERLAY RUNTIME 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 storage with other
                           segments and is loaded when it is needed.

OVERLAY TREE -   A tree structure consisting of a root segment
                           and optionally one or more overlay segments.

PARTITION -   An area of memory reserved for the execution
                           of tasks.

PATH -   A route that is traced from one segment in
                           the overlay tree to another segment in that
                           tree.

PATH-DOWN -   A path toward the root of the tree.

PATH-UP -   A path away from the root of the 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.

PRIVILEGED TASK -   A task that has privileged memory access
                           rights. A privileged task can access the
                           Executive and the I/O page in addition to its
                           own partition and referenced shared regions.

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 p-sections with attributes describing
                           access, allocation, relocatability, etc.

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 has a header and stack and that
                           can be installed and executed.

SHARED REGION -   An area of system memory whose contents can
                           be shared by any number of tasks.

SEGMENT -   A group of modules and/or p-sections that
                           occupy memory simultaneously and that can be
                           loaded by a single disk access.

SUB-PARTITION -   A partition that resides within a main
                           partition.

SYMBOL DEFINITION FILE -   The output file created by the Task Builder
                           that contains the global symbol definitions
                           and values in a format suitable for re-
                           processing by the Task Builder. Symbol
                           definition files are used to link tasks to
                           shared regions.

APPENDIX G.   RSX-11M TASK BUILDER GLOSSARY

SYSTEM-CONTROLLED PARTITION - A partition whose   memory   may   be
                             dynamically allocated by the Executive to
                             several concurrently active, resident tasks.

TARGET SYSTEM -   The system on which the task executes.

TASK IMAGE FILE -   The output file created by the Task Builder
                             that contains the executable portion of the
                             task.

USER-CONTROLLED PARTITION - A partition that can accommodate only
                             one active, resident task.

APPENDIX H

CROSS REFERENCE PROCESSING AND DATA FORMATS

H.1 CROSS REFERENCE PROCESSOR

The Cross Reference Processor is an independent task that is invoked
by the Task Builder to convert an input file of the specified format
into a cross reference listing that is appended to a text output file.

Input to the cross reference file consists of the following:

- A text file to which the cross reference listing is to be
  appended. The file must reside on a random access file-
  structured device.

- A cross reference input file having the format shown in
  Figure H-1. The file is assumed to reside on the same
  device and have the same UIC and filename as the text
  file. The input file, however, has an extension of .CRF.

- A "SEND" packet having the format shown in Figure H-2.

Normally the output file will reside on the input device; however, by
means of the appropriate flags within the send packet, the cross ref-
erence task can be instructed to dispose of this file in one of the
following ways:

    The text file may be submitted to the print symbiont
    for output.

    The file may be copied to a record-oriented device,
    then deleted from the system device.

Requesting a global cross-reference listing via the /CR switch causes
the Task Builder to perform the additional processing outlined below.

- If the output device is sequential; or record-oriented;
  i.e., LP:, a temporary memory allocation file is created
  on SY0:. The target device and unit are recorded.

- After the memory allocation file has been written, another
  pass is made over the object files to create the cross
  reference input file.

APPENDIX H. CROSS-REFERENCE PROCESSING AND DATA FORMATS

- The SEND packet is always constructed regardless of the
  target device. The spooling flag is set if the target
  device is a random access device.

- The packet is sent to the cross-reference task.

- The cross reference task is requested to run.

H.2 CREF INPUT FILE FORMAT

The cross-reference input file consists of a single header record
followed by data records as shown in Figure H-1.

The header record consists of the name of the originating task (in
RADIX-50), followed by a numeric value that identifies the task. The
value is used to access internal cross-reference tables that contain
information describing the CREF output format.

The next 5 words contain the creation time and date for the text
file.

Subsequent data records contain the information used by CRF to create
the body of the listing. The first two words comprise the symbol
name (in RADIX-50). The second two words consist of information that
identifies the reference. For a global cross-reference, these words
contain the module name (in RADIX-50). The symbol value is an octal
quantity associated with the symbol name.

The last word in the data record is subdivided into the attributes
flag byte and the format number byte. The contents of the flag byte
cause specific characters to be printed to the left of each reference,
and within the value field. For the Task Builder the flags and re-
lated symbols are:

Flag	Symbol	Definition
200	^	Reference is defined in an overlay segment away from the root.
040	-R	Symbol is relocatable.
020	*	Module contains an autoloadable definition.
010	#	Module contains a non-autoloadable definition.
002	@	Module contains a reference that is resolved through an autoload vector.

The format byte defines the table to which the data record belongs.
This allows several types of cross-reference listings to be generated
by a single originating task. The Task Builder always sets this value
to zero. The format number identifies the cross-reference list to which
the data record belongs. Currently, only one type of cross-reference is
created by the Task Builder and the value is always zero.

HEADER RECORD	NAME OF ORIGINATING TASK (2-WORD RAD50)
	ORIGINATING TASK IDENTIFIER (TKB=0)
	CREATION DATE YEAR
	OF MONTH
	CREF INPUT DAY
	FILE HOUR
	MINUTE
	RESERVED
DATA RECORD	SYMBOL NAME (2 WORD RAD50)
	REFERENCE IDENTIFIER (2 WORDS)
	SYMBOL VALUE
	FORMAT NUMBER ATTRIBUTES
DATA RECORDS TO END-OF-FILE	

Figure H-1
CREF Input File Format

APPENDIX H. CROSS-REFERENCE PROCESSING AND DATA FORMATS

H.3 SEND PACKET FORMAT

The format of the send packet is shown in Figure H-2. The contents
of the packet are as follows.

Words 1 through 10 contain information describing the text output file
name, directory file identification, device name and unit, and a flags
byte. A value of 1 in bit 0 of the flags byte instructs CREF to sub-
mit the text file to the print symbiont after processing.

The remaining three words specify the CREF input data file version
and the device to receive the text output after processing (if a
record-oriented device was specified in the map file portion of the
Task Builder command line).

Using the above information, CREF constructs the data input file name
as follows:

File name:	Same as text file
Version:	As specified
Type:	.CRF
Device:	Same as text file
Directory ID:	Same as text file

TEXT FILE NAME	
3-WORDS	
RAD 50	
FILE TYPE	
FILE VERSION	
DIRECTORY ID	WORD 0
	WORD 1
	WORD 2
DEVICE NAME	
FLAGS	UNIT
CREF FILE VERSION	
TARGET DEVICE	
RESERVED	UNIT

Figure H-2
Send Packet Format

APPENDIX H. CROSS-REFERENCE PROCESSING AND DATA FORMATS

H.4 ADJUSTING THE CROSS-REFERENCE LINE FORMAT

The user may adjust the cross-reference output format for either
132-column or 72-column printers by revising the build file
CRFBLK.CMD under UFD [1,24] (mapped) or [1,20] (unmapped) and re-
building the task as follows.

To obtain the narrow format (default), the user should edit the build
file and insert a semi-colon in front of the command:

        EXTSCT=$$RCB0:204

Conversely, the wide format is obtained by removing the semi-colon
preceding the above command.

INDEX

READER'S COMMENTS

NOTE: This form is for document comments only. Problems
with software should be reported on a Software
Problem Report (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

If you do not require a written reply, please check here. ☐
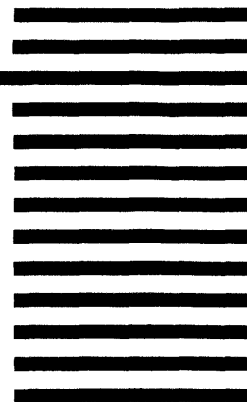
Please cut along this line.

-- Fold Here --

-- Do Not Tear - Fold Here and Staple --

USINESS REPLY MAIL
O POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

ostage will be paid by:

digital

Software Communications
P. O. Box F
Maynard, Massachusetts   01754

digital

DIGITAL EQUIPMENT CORPORATION
MAYNARD, MASSACHUSETTS 01754