pdp11

# RSX-11D Task Builder Reference Manual

Order No. DEC-11-OXDLA-D-D

digital

# RSX-11D Task Builder Reference Manual

Order No. DEC-11-OXDLA-D-D

RSX-11D Version 6B (Version 6.1)

digital equipment corporation · maynard. massachusetts

First Printing, June 1975
Revised, March 1976

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

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

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by DIGITAL.

Copyright © 1975, 1976 by Digital Equipment Corporation

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

The following are trademarks of Digital Equipment Corporation:

| | | |
|---|---|---|
| DIGITAL | DECsystem-10 | MASSBUS |
| DEC | DECtape | OMNIBUS |
| PDP | DIBOL | OS/8 |
| DECUS | EDUSYSTEM | PHA |
| UNIBUS | FLIP CHIP | RSTS |
| COMPUTER LABS | FOCAL | RSX |
| COMTEX | INDAC | TYPESET-8 |
| DDT | LAB-8 | TYPESET-10 |
| DECCOMM | DECsystem-20 | TYPESET-11 |

# RSX-11D TASK BUILDER

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

## FIGURES

# 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-11D 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-11D system described in the RSX-11D Executive Reference Manual and with basic operating procedures described in the RSX-11D User's Guide.

## 0.2 STRUCTURE OF THE DOCUMENT

The manual has seven chapters. The first four chapters describe the basic capabilities of the Task Builder. The last three 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 shareable global areas which can be used for communication between tasks or to reduce the system's memory requirements.

A Glossary of terms is given in Appendix G.

## 0.3 ASSOCIATED DOCUMENTS

Associated RSX-11D documents are described and their readerships are defined in the RSX-11D Documentation Directory, Order Number DEC-11-OXUGA-C-D.

# CHAPTER 1

# INTRODUCTION

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

The fundamental executable unit in the RSX-11D 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 an 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. The task is then installed.
6. 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 6.

If a single routine is to be executed, the user provides the object module filename to be used as Task Builder input and a task image filename to be assigned to the Task Builder output.

In typical applications, a collection of routines is run rather than a single program. In this case the user names each of the object module files. Then the Task Builder links the object modules, resolves any references to any shareable global areas and produces a single task image which is ready for installation and execution.

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 which gives information about how the task is mapped into memory. The user can examine the memory allocation file to determine what support routines and storage reservations are included in the task image. A cross reference file of global symbols used can be appended to the memory allocation file.

Further, and also upon request, the Task Builder produces a symbol table file suitable for input to the Task Builder during the build of another task. For example, such a procedure is used in binding tasks to shareable global areas.

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 (i.e., a multi-segment task), overlay segments are loaded using either the autoload or manual method.

The autoload method makes the loading of overlays transparent to the user and special calls are not 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 shareable global areas and to create new shareable global areas for future reference.

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, and finally a shareable global area is added.

The memory allocation (map) 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 writes a FORTRAN program which is entered through a text editor as file PROG. Then, the following commands are typed in response to the Monitor Console Routine´s request for input:

```
MCR>FOR CALC=PROG
MCR>TKB IMG=CALC
MCR>INS IMG
MCR>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.

The example just given includes the command

```
MCR>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 an equal sign and the input file specifications. 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, mem-allocation, symbol-definition=input, ...
```

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

| Command | Output Files |
|---|---|
| MCR>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. |
| MCR>TKB IMG1=IN1 | The task image file is IMG1.TSK. |
| MCR>TKB ,MP1=IN1 | The memory allocation file is MP1.MAP. |
| MCR>TKB ,,SF1=IN1 | The symbol definition file is SF1.STB. |
| MCR>TKB IMG1,,SF1=IN1 | The task image file is IMG1.TSK and the symbol definition file is SF1.STB. |
| MCR>TKB =IN1 | This is a diagnostic run with no output files. However, any errors encountered will produce relevant error message. |

### 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

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

produces the same result as the single line command:

```
MCR>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.

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

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

The syntax and interpretation of each RSX-11D Task Builder option are given in Chapter 3.

The general form of an option is a keyword followed by an equal sign '=' and 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 to direct the Task Builder to stop accepting input, build the task, and request information for the next task build.

Consider the Sequence:

```
MCR>TKB
TKB>IMG1=IN1
TKB>IN2,IN3
TKB>/
ENTER OPTIONS:
```

```
TKB>PRI=1ØØ
TKB>COMMON=JRNAL:RØ
TKB>/
TKB>IMG2=SUB1
TKB>//
MCR>
```

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.

## 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. Then, the contents of the indirect command file are invoked by typing '@' followed by the file specification.

Suppose the text file AFIL is prepared, as follows:

```
IMG1,MP1=IN1
IN2,IN3
/
PRI=1ØØ
COMMON=JRNAL:RO
//
```

Later, the user can type:

```
MCR>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 which enables Task Builder to accept commands from the indirect file AFIL is equivalent to the keyboard sequence:

```
MCR>TKB
TKB>IMG1,MP1=IN1
TKB>IN2,IN3
TKB>/
ENTER OPTIONS:
TKB>PRI=1ØØ
TKB>COMMON=JRNAL:PRO
TKB>//
```

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 three levels of indirection in file references. The indirect file referenced in a terminal sequence can contain a reference to another indirect file, which in turn references the third 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 a task building file, the user modifies AFIL to include an indirect file reference to BFIL.CMD as a separate line in the option sequence.

Then the contents of AFIL.CMD 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

```
MCR>TKB @AFIL
```

then is:

```
MCR>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>//
MCR>
```

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.

## 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. Specifically, some identifying lines are added along with notes on the function of the input files and shareable global area. Then, a comment on the current status of the task is added at the end of the file. The content of the file is 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:R0 ; 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-11D conventions for file specifier. For any file, the user can specify the device, the user file directory (ufd), the filename, the type, the version number, and any number of switches.

Thus, the file specifier has the form:

```
device:[ufd]filename.type;version/sw...
```

```
              RETURN
              END

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

       *EX

       [EDI -- EXIT]
```

### 2.2.2 Compiling the FORTRAN Programs

The FORTRAN programs are compiled by the following sequence:

```
MCR>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 image for the three programs is built in the following way:

```
MCR>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.
```

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:

   ```
   MCR>TKB task-command-line
   ```

   The second form has additional lines for input file names:

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

   The third form allows the specification of options:

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

The fourth form has both input lines and option lines:

```
MCR>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 or Batch Processor.
```

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 specifier as defined in Rule 7.

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

   ```
   task-file, mem-allocation-file, symbol-file

   task-file, mem-allocation-file,

   task-file
   ```

   where task-file is the file specifier for the task image file; mem-allocation-file, is the file specifier for the memory allocation file; and symbol-file is the file specifier for the symbol definition file. Any of the specifiers 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 specifiers.

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

   ```
   option ! ...
   ```

```
    @indirect-file
```

where indirect-file is a file specifier.

6. An option has the form:

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

   where the argument-list is

   ```
   arg: ...
   ```

   The syntax for each of the options is given in Chapter 3.

7. A file specifier conforms to standard RSX-11D conventions. It has the form

   ```
   device:[ufd]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´. |
   | ufd | is the user file directory number consisting of two octal numbers each of which is in the range of 1 through 377 (octal). These numbers must be enclosed in brackets and separated by a comma, and must be in the following format: [group, member] For example, member 220 of group 200 would use the following entry: [200,220] |
   | 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 in the range 1 through 77777 (octal). 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 permissible switch options and their syntax are given in Chapter 3.

The device, the user file directory 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 | SYØ, the system device * |
| group | the group number currently in effect * |
| member | the member number currently in effect * |
| type | task image TSK<br>memory allocation MAP<br>symbol definition STB<br>object module OBJ<br>object module library OLB<br>overlay description ODL<br>indirect command CMD |
| version | for an input file, the highest-numbered existing version.<br>for an output file, one greater than the highest-numbered existing version. |
| switch | (the default for each switch is given in Chapter 3.) |

*If an explicit device or [ufd] is given, it becomes the default for subsequent files separated by commas on the same side of the equal (=) sign.

For example:

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

    File          Device

 IMG1.TSK          DT1
 MP1.MAP           DT1
 IN1.OBJ           SYØ
 IN2.OBJ           DFØ
 IN3.OBJ           DFØ
```

# 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 illustrates to the user various methods of modifying the task image file. Later, the chapter can be used as a reference for practical applications with specific requirements.

## 3.1 SWITCHES

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

```
dev:[ufd]filename.type;version/sw-1/sw-2.../sw-n
```

The file specifier concludes with zero or more switches, sw-1, sw-2, ..., sw-n, and the switches are described in the following text.

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.

Switches are primarily used to designate the task attributes which are recorded in the task image file during task build and in the System Task Directory (STD) entry on Install. They are also used to instruct

Table 3-1
Task Builder Switches

| Code | Meaning | Applies to File* | Default Assumption |
|---|---|---|---|
| AB | Task can be aborted. | T | AB |
| CC | Input file consists of concatenated object modules. | I | CC |
| CP | Task is checkpointable | T | CP |
| CR | Cross reference to be appended to memory allocation file | M | -CR |
| DA | Task contains a debugging aid. | T,I | -DA |
| DS | Task can be disabled. | T | DS |
| FP | Task uses the PDP-11/45 floating point processor. | T | FP |
| FX | Task can be fixed in memory | T | -FX |
| HD | Task image includes a header. | T,S | HD |
| LB | Input file is a library file. | I | -LB |
| MP | Input file contains an overlay description. | I | -MP |
| MU | Task is a multi-user task | T | -MU |
| PI | Task code is position independent. | T,S | -PI |
| PR | Task has privileged access rights. | T | -PR |
| SH | Short memory allocation file is required | M | -SH |
| SP | Memory allocation file is spooled | M | SP |
| SQ | Task p-sections are allocated sequentially. | T | -SQ |
| SS | Selective Symbol Search | I | -SS |
| TA | Task is accountable | T | -TA |
| 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 |

TKB to interpret the input file in a special way (e.g., /DA is used when the task contains a debugging aid) and to control the listing of the memory allocation file (e.g., /SH is used to request the short memory allocation file).

### 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:

- the switch mnemonic,
- the file(s) to which the switch can be applied.
- a description of the effect of the switch on the Task Builder, and
- the default assumption made if the switch is not present.

The switches are given in alphabetical order.

#### 3.1.1.1 AB (Abortable)

file: task image

meaning: The task can be aborted when it is running

effect: The Task Builder clears the not-abortable flag in the task label block flags word.

default: AB

#### 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

#### 3.1.1.3 CP (Checkpointable)

file: task image

meaning: The task is checkpointable.

effect: The Task Builder clears the not-checkpointable flag in the task label block flags word.

default: CP

### 3.1.1.4 CR (Cross Reference)

file: memory allocation

meaning: Produce a global cross reference.

default: -CR

### 3.1.1.5 DA (Debugging Aid)

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. When the /DA switch is applied to an input file, the Task Builder sets up the specified file as a user-written debugging aid.

default: -DA

### 3.1.1.6 DS (Disableable)

file: task image

meaning: The task can be disabled.

effect: The Task Builder clears the not - disableable flag in the task label flags word.

default: DS

### 3.1.1.7 FP (Floating Point)

file: task image

meaning: The task uses the PDP11/45 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 FX (Fixable)

file: task image

meaning: The task can be fixed in memory.

effect: The Task Builder clears the non-fixable flag in the task label block flags word. Note that a fixed task will not be checkpointed even when built as checkpointable.

default: -FX

### 3.1.1.9 HD (Header)

file: task image or symbol definition

meaning: A header is to be included in the task image. A description for using the negation form of this switch (-HD or NOHD) to produce a shareable global area is contained in Chapter 7.

effect: The Task Builder constructs a header in the task image. The contents of the header are described in Appendix C.2.

default: HD

### 3.1.1.10 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 (created by the Librarian) which are to be searched for the resolution of undefined global references.

2. If the switch is applied with arguments, the input file is assumed to be a library file of relocatable object modules from which the modules named in the argument list are to be taken for inclusion in the task image. The module names are those defined at assembly time by the .TITLE directive (or if no .TITLE directive, the filename (first 6 characters) when inserted by the Librarian).

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.

NOTE

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.

default: -LB

### 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 (except those defined in the .ODL file).

default: -MP

### 3.1.1.12 MU (Multi-user)

meaning: Multiple versions of the task can run simultaneously. Note that only read-write parts of the task will be duplicated in memory.

effect: The multi-user flag is set in the task label block flags word and any read-only section of the root segment is aligned on a disk block boundary.

default: -MU

### 3.1.1.13 PI (Position Independent)

file: task image or symbol definition

meaning: The task contains only position independent code or data. Position independent shareable global areas 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

### 3.1.1.14 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 SCOM data area (including node pool) in addition to its own task space. 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 Memory Allocation File)

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 the Memory Allocation File)

file: memory allocation

meaning: If spooling is enabled on the system the memory allocation file will be spooled to the output device and deleted.

effect: The memory allocation file is spooled and deleted.

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 stated in the TKB command lines. 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. This switch must not be used for modules that rely upon alphabetical p-section allocation. In RSX-11D such modules include FORTRAN I/O handling and FCS modules from SYSLIB.

default: -SQ

#### 3.1.1.18 SS (Selective Search)

file: The input file is to be used only to define the required symbols.

effect: The Task Builder includes only the required symbol definitions from the specified file as distinct from all global symbols of that file. This switch is useful when the input file is an .STB type, because it reduces the size of symbol table searches.

default: -SS

#### 3.1.1.19 TA (Task Accounting)

file: task image

meaning: Accounting data for the task is accumulated if this switch is specified and the accounting function of the operating system is running.

effect: The Task Builder allocates an extra 16Ø (decimal) words in the task header for accumulation of accounting data when the task is running.

default: -TA

#### 3.1.1.20 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 |

### 3.1.1.21 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:<br><br>n. means a decimal number<br>#n or n means an octal number.<br><br>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 |
|---|---|
| `MCR>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. |
| `MCR>TKB`<br>`TKB>IMG2/PR,MP1/SH=`<br>`TKB>IN2,[1,1]EXEC.STB`<br>`TKB>//` | The task IMG2.TSK is a privileged task. The short memory allocation file MP1.MAP is requested. The inputs for the task are the file IN2.OBJ and the symbol definition file SY:[1,1] EXEC.STB which links the task to the subroutines and data base of the Executive. |
| `MCR>TKB`<br>`TKB>IMG3=IN3`<br>`TKB>LB1/LB:SUB1:SUB2`<br>`TKB>LB1/LB,DBG1/DA`<br>`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. |

| | |
|---|---|
| `MCR>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 |
|---|---|---|
| Any task image switch except PI and SQ | -HD | -HD |
| CC | LB | LB |
| CP | FX | -FX |

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

Options are available to the user of the RSX-11D Task Builder to supply information about the characteristics of the task to the Task Builder.

Some of these options are of interest to all users of the system, some are of interest only to the FORTRAN programmer, and some are of interest only to the MACRO-11 programmer. The interest range is given with the description of the option.

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

1. contr - Control options 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 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 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 indicate the task´s intention to access a shareable global area. |
| 5. | device | - Device specifying options specify the number of units required by the task and the assignment of physical devices to logical unit numbers. |
| 6. | alter | - Content altering options define a global symbol and value or to introduce patches in the task image. |
| 7. | synch | - Synchronous trap options define synchronous trap vectors. |

Table 3-2 lists all the options alphabetically. A brief description of each option is given and 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.

NOTE

Many of the TKB options can be overridden at install time (See RSX-11D User´s Guide).

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 |
| BASE | Define lowest virtual address. | FM | alloc |
| COMMON | Declare task´s intention to access a shareable global area. | FM | share |
| EXTSCT | Declare extension of a program section. | FM | alloc |
| EXTTSK | Extend task memory allocation at install time. | 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 shareable global area. | 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 |
| POOL | Declare pool usage limit | FM | alloc |
| PRI | Declare priority. | FM | ident |
| STACK | Declare the size of the stack. | FM | alloc |
| TASK | Declare the name of the task. | FM | ident |
| TOP | Define highest virtual address. | FM | alloc |

Table 3-2 (Cont.)
Task Builder Options

| Option | Meaning | Interest | Category |
|---|---|---|---|
| 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 |

## 3.2.1 Control Option

Only one control option is available. This option is of interest to all users of the system.

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

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

The Task Builder, on recognizing the keyword ABORT, stops accepting input, closes all opened files, deletes all output files 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, build the task, and then terminate.

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 specifying task identifying information. 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 installed and also when it is run. If such a specification is not made, the user identification code established when the task was built is used. The task runs under the most recently specified UIC.

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

syntax: TASK = task-name

where: task-name is a 1- to 6-character radix-5Ø name identifying the task.

default: The name of the task image file is used to identify the task when the task is installed. This option is used when it is desirable to store the task image file under one name, but run the task under another name (e.g., MAX.TSK and CORMAC.TSK both use the task name ...MAC).

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 number. |
| default: | | The UIC specified by the MCR HEL[LO] command. |

3.2.2.3 PRI (Priority) - The PRI option declares the priority at which the task executes. If priority is not specified when the task is installed, the priority declared in the PRI option is used.

syntax: PRI = priority-number

where: priority-number is a decimal integer in the range 1 - 25Ø

default: 5Ø(established by Install)

3.2.2.4 PAR (Partition) - The PAR option identifies the partition in which the task is run unless overriden at install or run time.

syntax: PAR = pname

| | | |
|---|---|---|
| where | pname | is the name of the partition |
| default: | none | (established at install time) |

## 3.2.3 Allocation Options

These options 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 users.

### 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. If a double buffered FCS is used, the ACTFIL specification must be doubled also.

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 all files used by the task.

This option must be used to extend the buffer size whenever a file is to be processed in which the maximum record size exceeds the default buffer length as specified for the device by the System Manager during system generation.

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 parsing 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 processing 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 EXTSCT (Program Section Extension) - 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 OVR (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 encountered 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-50 name that specifies 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 EXTTSK (Extend Task Space) - The read/write space of the task may be extended at Install time.

| | |
|---|---|
| syntax: | EXTTSK = task-extension |
| where: | task-extension is the decimal number of words by which Install extends the upper read/write area of the task. The value is rounded up to the next 32-word block boundary. |
| default: | 0 |

This parameter can be overridden by Install option /INC. If the EXTTSK option has been overridden, the task must be removed and reinstalled without the /INC option to revert back to the EXTTSK option.

This option is used in conjunction with the .LIMIT directive to the assembler and the Executive directive Get Task Parameters.

It is useful in saving disk space that would otherwise be allocated for initially empty buffers, for example. Further, the Install /INC option provides the ability to vary the size of such buffers.

3.2.3.6 POOL(Pool Limit) - The POOL option declares the maximum number of eight word pool nodes that the task may use simultaneously.

| | | |
|---|---|---|
| syntax: | POOL = pool - limit | |
| where: | pool-limit | is the decimal number of eight-word nodes in the range of 1 through 511. For MU tasks this indicates the pool-limit per version. |
| default: | 40 | |
| Example: | Set the pool-limit to 60<br>POOL=60 | |

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

The stack is an area of memory used for temporary storage, subroutine calls, and 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 that specifies the number of words required for the stack. |
| default: | STACK = 256 | |

3.2.3.8 BASE (Base Address) - The BASE option specifies the base address of the task to be at a particular 4K boundary.

syntax: BASE = bound

where: bound is a decimal number between 0 and 28 which specifies the lowest 4K boundary of the image.

default: 0

Creation of shareable global area images which are not position-independent provides one example of how to use this option. In fact, the BASE (and TOP) options are primarily used to locate shareable global areas, such as libraries and common block, and are not recommended for use in building normal tasks.

Task image addresses are normally allocated upward from zero. This type of library file must appear in the same virtual address range of each task that shares it. To avoid conflicts with task addresses the library may be allocated toward the top of the virtual address range (i.e., 140000 to 177776) by using a base address declaration (also see Section 3.2.3.9)

NOTE

The BASE option will override any previous TOP specification.

3.2.3.9 TOP (TOP ADDRESS) - The TOP option declares the ending address of a task to be within a 4K boundary.

syntax: TOP=bound

where: bound is a decimal number between 0 and 28 which specifies the highest 4K boundary of the image.

default: none

The purpose of this option is the same as the purpose of the BASE option except it allows defintion of the last 4K boundary, rather than the first 4K boundary.

NOTE

The TOP option will override any previous BASE specification.

3.2.3.10 Examples of Allocation Options - If 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 following terminal sequence can be used to build the task:

```
    MCR>TKB
```

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

### 3.2.4 Storage Sharing Options

Two options indicate the task's intention to access a shareable global area. These options are of interest to all users of the system.

By convention, the COMMON option indicates the use of a shareable global area that contains only data and the LIBR option indicates the use of a shareable global area that contains only code. The two options have the same effect, except that when all tasks accessing a common area have exited, the area is written back to its file on disk. For a library file, the memory allocated to it is simply released.

3.2.4.1 COMMON (Sharable Common Block) - The COMMON option declares a 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 either RW (read/write) or RO (read-only) to indicate the type of access required for the task. |
| | apr | is an integer in the range of 1 to 7 which specifies the first Addressing Page Register to be reserved for the common block. The apr is optional. |

default: none

3.2.4.2 LIBR (Sharable Library) - The LIBR option declares a sharable 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 which specifies the library. |
| | access-code | is either RW (read/write) or RO (read-only) to indicate the type of access required for the task. |
| | apr | is an integer in the range of 1 to 7 which |

specifies the first Addressing Page Register to be reserved for the library. The apr is optional.

default: none

3.2.4.3 Example of Storage Sharing Options - If the task composed of the MACRO-11 programs TST1 and TST2 accesses a shareable common area DTST that contains data and a shareable library area STST that contains code, the following terminal sequence can be used to build the task:

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

## 3.2.5 Device Specifying Options

The two options in this category are of interest to all system users. 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 of 0 to 255 which specifies 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<br>unit-num-2<br>...<br>unit=num-8 | are decimal integers indicating the logical unit numbers. |

default: ASG = SYØ:1:2:3:4, TIØ:5, CLØ: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 command sequence of the example shown in Section 3.2.3.7 is changed to include device assignment options, as follows:

```
MCR>TKB
TKB>IMG1,MP1=GRP1
TKB>/
ENTER OPTIONS:
TKB>ACTFIL = 8 !  MAXBUF = 16Ø
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-5Ø name of the defined symbol. |
| | symbol-value | is an octal number in the range of Ø to 177777 which is 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-5Ø 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 of Ø to 177777 to be assigned to address. |
| | val-2 | is an octal number in the range of Ø to 177777 to be assigned to address+2 |
| | ... | ... |
| | val-8 | is an octal number in the range of Ø to 177777 to be assigned to address+16(octal).. |

NOTE

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: | sym-name | is a 1 to 6-character radix-5Ø name which specifies the global symbol. |
| | offset | is an octal number used to specify the offset from the global symbol. |
| | seg-name<br>val-1<br>val-2<br>...<br>val-8 | are identical to that 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 and introduces 10 patch values relative to the global symbol DELTA.

The terminal sequence of example shown in Section 3.2.4.3 is modified to include the options GBLPAT and GBLDEF as follows:

```
MCR>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

Two options are available to 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 of 1 to 32 which specifies 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 reserve.

### 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:

```
MCR>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

The user knows from the program logic that only one file is open at a time, but the Task Builder assumes that four files are open simultaneously. Therefore, the user can utilize the ACTFIL option to reduce the space required for the task.

The task is built with the following terminal sequence:

```
MCR>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

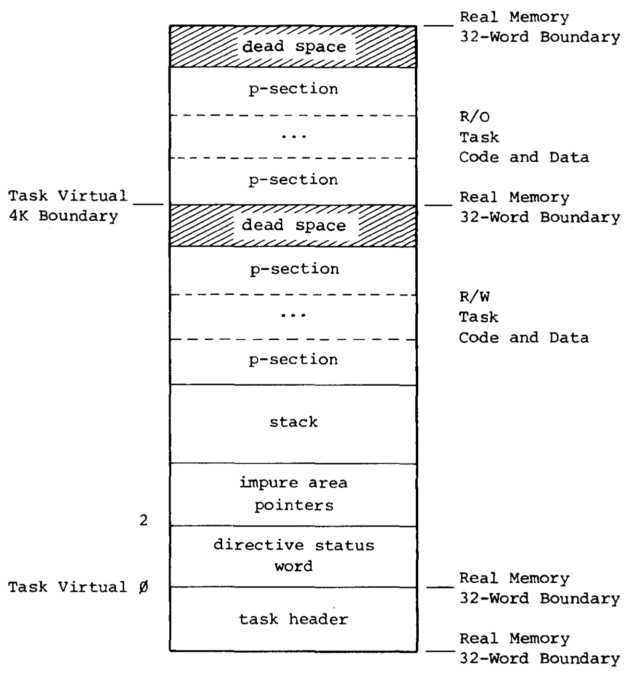
# MEMORY ALLOCATION

This chapter describes the allocation of task and system memory. The memory allocation file and the cross reference file are described in detail and examples of memory allocation files 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-11D 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 for a single segment task can be represented by the following diagram:



Task Memory for a Single Segment Task

### 4.1.1 Task Header

The task header contains task parameters and data required by the executive for controlling execution of a task. It also provides an area for saving the task's contents when a switch is made to another task and for accumulating accounting information about the task. It is resident at all times when the task is resident, but is not a part of the task's virtual address space. A detailed description of the header can be found in Appendix C.2.

### 4.1.2 Directive Status Word

Virtual location zero of every RSX-11D task is a reserved word for the executive to report the status of all executive directives issued by the task.

### 4.1.3 Impure Area Pointers

The words following the directive status word are used as pointers to the following areas of the task.

| Address | Use |
|---|---|
| 2 | Address of FCS data storage area |
| 4 | Address of FORTRAN-OTS work area |
| 6 | Address of overlay run time system work area |

Like the directive status word these parameters occupy the low address end of the task stack.

### 4.1.4 Stack

A default stack of 256(decimal) words is allocated for each task. The STACK=option may be used to override this allocation. A STACK=0 specification is useful in building shareable global areas which do not require a stack. (libraries use the task stack space because they are within the address space of the binding task).

### 4.1.5 R/W Task Code(and Data)

The R/W p-sections of a task are concatenated to the end of the stack. The memory allocation is rounded up to a 32(decimal) word boundary by the addition of dead space.

### 4.1.6 R/O Task Code (and Data)

If there are p-sections in the task which have the read-only attribute (read/write is the default), the Task Builder concatenates these and allocates them so they occupy an integral number of 32(decimal) word blocks. The task will be mapped so that the R-O p-sections will begin at the next 4K virtual address. Further, if the task is multi-user, these R-O areas will begin on the next available disk block to facilitate separate loading of R/W and R-O portions of the task at run time. This enables multiple copies of such tasks to share one copy of the read-only code (and/or data). The executive treats these shared read-only areas of a task much as it treats shareable global areas.

Note that ODT cannot be used to modify read-only parts of a task. This also means that breakpoints cannot be set in such code. Any attempt to use ODT in the read-only areas of a task will cause the task to terminate with a segment fault.

### 4.1.7 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.

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.

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 the each reference.

When a p-section has the concatenate attribute, all references to that p-section are placed one after 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.

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. |
| 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. |

* Indicates the default attribute

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

*** Not implemented

## 4.1.8 Allocation of P-sections

Suppose the user enters the following command:

```
MCR>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:

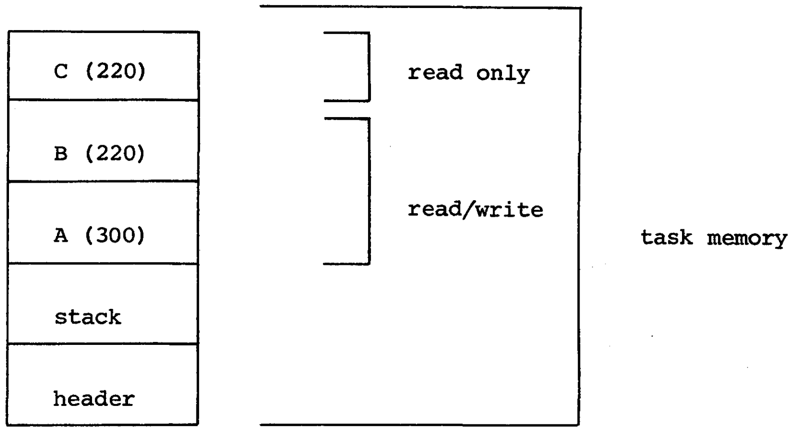
| File-name | P-section name | Access Code | Alloc Code | Size (octal) |
|---|---|---|---|---|
| IN1 | B | RW | CON | 100 |
| | A | RW | OVR | 300 |
| | C | RO | CON | 150 |
| IN2 | A | RW | OVR | 250 |
| | B | RW | CON | 120 |
| IN3 | C | RO | CON | 50 |

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, 1ØØ + 12Ø = 22Ø. 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:

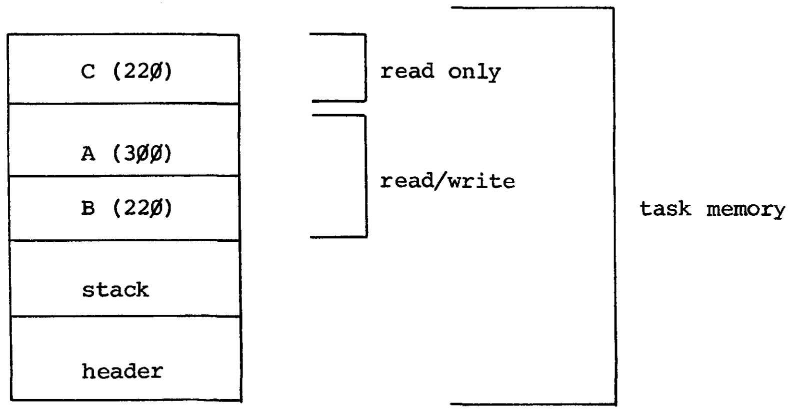


4.1.8.1 <u>Sequential Allocation of P-sections</u> - The SQ (sequential) switch affects only the placement of p-sections in task memory. P-sections with the same name and attributes are collected as described; then uniquely named p-sections are placed in memory in the order of input sequence according to the access-code.

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

```
MCR>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:



### 4.1.9 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 Ref. |
|---|---|---|---|
| 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.

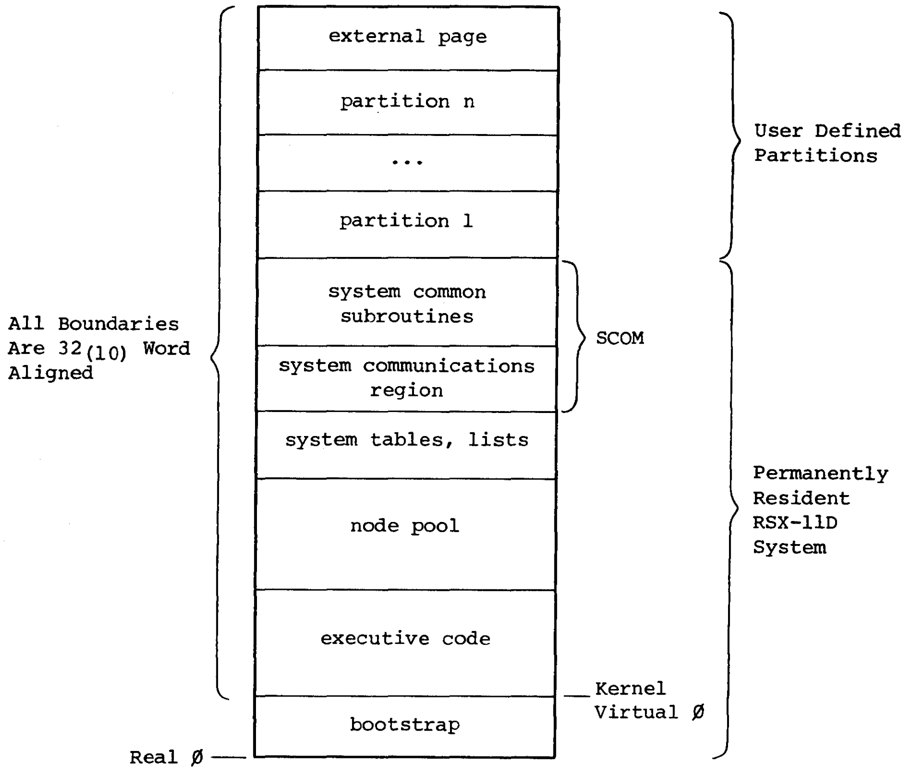
When all the input object files have been processed, the Task Builder has three unresolved global references, namely: C1, L1, and XXX. A search of the library file LBR1 resolves L1 and the Task Builder includes the defining module in the task image. A search of the System Library resolves XXX. The global symbol C1 remains unresolved and is, therefore, listed as an undefined global symbol.

The relocatable global symbol B1 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. The results of these resolutions can be obtained in Figure 4-1.

## 4.2 SYSTEM MEMORY

In RSX-11D, system memory consists of the resident executive and a set of named areas which are defined at system generation time. These named areas are partitions, each of which has parameters of base and length.

A typical system memory layout can be represented by the following diagram:

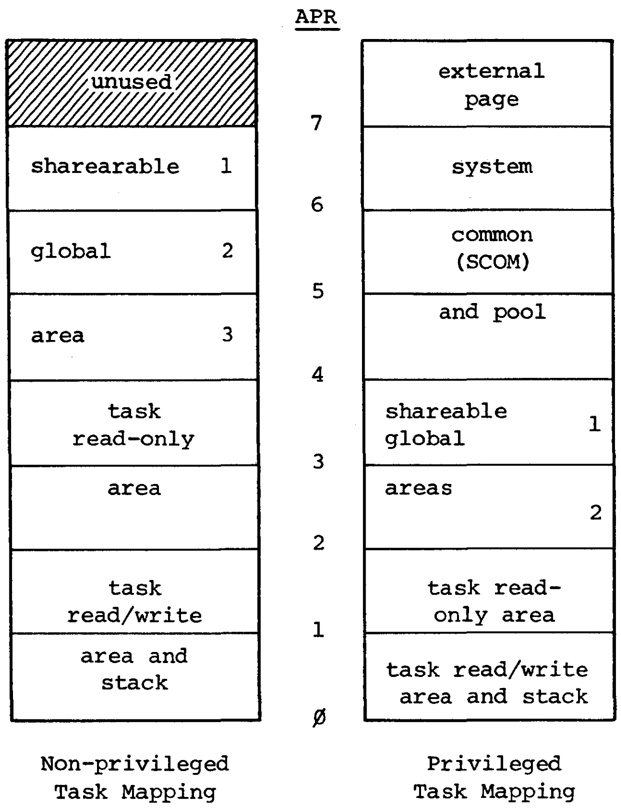


Typical System Memory Layout

### 4.2.1 Privileged Tasks

A privileged task has special memory access rights. A non-privileged task can access only its own partition and any referenced shareable global areas, but a privileged task can, in addition, access SCOM and the external page.

The following diagram illustrates typical privileged and non-privileged tasks. Note that APR boundaries are aligned at 4K virtual addresses and 32(decimal) word real addresses when in memory.



## 4.3 TASK IMAGE FILE

In addition to the task memory, or core image, the task image file contains a label block group. 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, as is the task image file structure.

## 4.4 MEMORY ALLOCATION FILE

The memory allocation file lists information about the allocation of task memory and the resolution of global symbols. Figure 4-1 is a listing of the memory allocation file produced by example IMG1 of Section 4.1.

```
FILE IMG1.TSK;1 MEMORY ALLOCATION MAP
THIS ALLOCATION WAS DONE ON 13-FEB-75
AT 15:40 BY TASK BUILDER VERSION R09


*** ROOT SEGMENT: IN1


R/W MEM   LIMITS: 000000 001577 001600
R-O MEM   LIMITS: 020000 020277 000300
STACK     LIMITS: 000000 000777 001000
DISK BLK  LIMITS: 000003 000005 000003
IDENTIFICATION  :
PRG XFR ADDRESS: 001300
TASK ATTRIBUTES: FP.NF


PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 001000 001000 000000
<A     >: 001000 001277 000300
<B     >: 001300 001517 000220
<C     >: 020000 020217 000000
<. ABS.>: 000000 000000 000000


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

<. ABS.>: 000000 000000 000000

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


<A     >: 001000 001277 000300


<B     >: 001300 001377 000100

    B1     001300-R  B2    001302-R

<C     >: 020000 020147 000150
```

Figure 4-1
Memory Allocation File for IMG1.TSK

```
                 MEMORY ALLOCATION

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

<A      >: 001000 001247 000250

    A1      001000-R

<B      >: 001400 001517 000120

    B1      001300-R


*** FILE: IN3.OBJ;2  TITLE: IN3   IDENT:

<C      >: 020150 020217 000050


*** FILE: SYSLIB.OLB;1  TITLE: MODN   IDENT:

<. ABS.>: 000000 000000 000000

    XXX     000005


*** FILE: LBR1.OLB;1  TITLE: LIB1   IDENT:

<. ABS.>: 000000 000000 000000

    L1      000022


************

UNDEFINED REFERENCES:

    C1
```

Figure 4-1 (Cont.)
Memory Allocation File for IMG1.TSK

### 4.4.1 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
   ```

   These items are defined in 2 through 6.

2. The heading gives the name of the task, date and time of the task-build, and the version number of the Task Builder 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.
   ```

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-l ...   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 Rule 7.

4. The program section allocation synopsis has the form:

   ```
   p-sect-name-l  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 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 of (in octal) 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 PDP-11/45 floating point processor |

| | |
|---|---|
| DA | Task includes the standard debugging aid SY0:[1,1]ODT.OBJ |
| PI | Task contains only position independent code and data |
| PR | Task is privileged |
| TR | Task initial PS word has T-bit enabled |
| NH | Task does not contain a header |
| NF | Task is not fixble |
| NA | Task is not abortable |
| ND | Task cannot be disabled |
| MU | Task is multi-user |
| TA | Task is accountable |

| | |
|---|---|
| p-sect-name | is the name of a p-section. |
| file-name | is the name of the file from which the input module was taken. |
| 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. |

### 4.4.2 Structure of the Cross Reference Listing

The cross reference output has two parts: a page header and the actual cross reference listing.

1. The cross reference page header consists of the following items:

   name of the map file
   time the map file was created
   cross reference page number

2. The actual cross reference contains an alphabetic listing of each global symbol, its value, and the name of each module referring to it. The name of each module is prefixed with a character indicating the symbol resolution within the module. The cross reference listing has the following format:

```
symbol name   value (-R)   x module-1   x module-2  ...
                           x module-n
```

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

Module name prefix characters have the following meaning:

| Prefix Character | Reference Type |
|---|---|
| blank | Module contains a reference that is resolved in the same segment or in a segment toward the root, or 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 definition that is not autoloadable. |
| * | Module contains an autoloadable definition. |

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

## 4.5 EXAMPLES - TWO VERSIONS OF CALC

The first run of CALC discussed in Chapter 2 produced the memory allocation file listed in Figure 4-2 and the short form of the memory allocation file obtained for version 2 is shown in Figure 4-3.

### 4.5.1 Segment Description

The memory allocation file heading is self-explanatory. The segment description indicates that the task consists entirely of R/W p-section and that the task is 43240 (octal) bytes in length. The stack size is defaulted to the lower 1000(octal) bytes of the task.

The program transfer address indicates that task execution begins at virtual address 1000. The default attributes assigned to the task are floating point (FP) and not fixable (NF).

The discussion of the label block in Appendix C explains why the task occupies virtual blocks 3 through 46 (octal).

### 4.5.2 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 1000, ends at virtual address 32507, and occupies 31510 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.

### 4.5.3 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 contributed the FORTRAN run-time routines and all FCS routines required.

The input file RDIN.OBJ contains two p-sections; namely, ´.BLK.´, and ´DTA ´. If the program had used a blank or unnamed common, the p-section .$$$$. would have appeared in the p-section allocation synopsis. The p-section ´. BLK.´ contains the code for RDIN.OBJ, starts at virtual address 1000, and occupies 150 bytes. ´DTA ´ is the p-section containing the common block DTA. This section starts at virtual address 32510, and occupies 1442 bytes.

The input file, PROC1.OBJ, also contains two 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 memory allocation file reproduced below does not contain the modules contributed by the library file SYSLIB.OLB.

```
FILE CALC.TSK;1 MEMORY ALLOCATION MAP
THIS ALLOCATION WAS DONE ON 13-FEB-75
AT 14:21 BY TASK BUILDER VERSION R09


*** ROOT SEGMENT: RDIN


R/W MEM  LIMITS: 000000 043237 043240
STACK    LIMITS: 000000 000777 001000
DISK BLK LIMITS: 000003 000046 000044
IDENTIFICATION : 02
PRG XFR ADDRESS: 001000
TASK ATTRIBUTES: FP,NF


PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 001000 032507 031510
<DTA   >: 032510 034151 001442
<$$DEVT>: 034152 036065 001714
<$$FSR1>: 036066 042165 004100
<$$FSR2>: 042166 042267 000102
<$$IOB1>: 042270 042473 000204
<$$IOB2>: 042474 042474 000000
<$$OTST>: 042474 042507 000014
<$$RESL>: 042510 043235 000526
<. ABS.>: 000000 000000 000000


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

<. BLK.>: 001000 001147 000150

    MAIN.  001000-R

<DTA   >: 032510 034151 001442


*** FILE: PROC1.OBJ;2  TITLE: PROC1   IDENT:

<. BLK.>: 001150 001173 000024

    PROC1   001150-R

<DTA   >: 032510 034151 001442


*** FILE: RPRT.OBJ;2  TITLE: RPRT   IDENT:

<. BLK.>: 001174 001217 000024

    RPRT   001174-R

<DTA   >: 032510 034151 001442
```

Figure 4-2
Memory Allocation File for CALC;1

## MEMORY ALLOCATION

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

- ACTFIL=1 to eliminate the three active file buffers not needed by CALC.
- PAR=PAR14K to direct the Task Builder to use a larger partition for CALC since the user intends to expand the task. However, this has no effect on task building other than to set up a partition name for Install to use.

The memory allocation file shown in Figure 4-3 reflects these changes:

```
FILE CALC.TASK;2 MEMORY ALLOCATION MAP
THIS ALLOCATION WAS DONE ON 13-FEB-75
AT 14:29 BY TASK BUILDER VERSION R09

*** ROOT SEGMENT: RDIN

R/W MEM  LIMITS: 000000 040203 040204
STACK    LIMITS: 000000 000777 001000
IDENTIFICATION : 02
PRG XFR ADDRESS: 001000
TASK ATTRIBUTES: FP,NF

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 001000 032533 031534
<DTA   >: 032534 034175 001442
<$$DEVT>: 034176 036111 001714
<$$FSR1>: 036112 037131 001020
<$$FSR2>: 037132 037233 000102
<$$IOB1>: 037234 037437 000204
<$$IOB2>: 037440 037440 000000
<$$OTST>: 037440 037453 000014
<$$RESL>: 037454 040201 000526
<. ABS.>: 000000 000000 000000
```

Figure 4-3
Memory Allocation File for CALC;2

Because of the additional logic in the program RDIN, the task code allocation increased from 31510 in CALC;1 to 31534 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 | 31510 | 31534 | + 24 |
| $$FSR1 | 4100 | 1020 | -3060 |
| | | | ----- |
| | | | -3034 |

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     40204

DIFF        3034
           -----
CALC;1     43240
```

# 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.

### 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:

```
MCR>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:

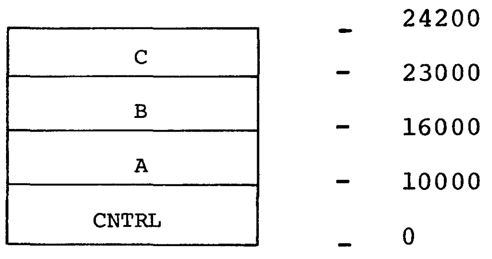
CNTRL calls A and A returns to CNTRL,
CNTRL calls B and B returns to CNTRL,
CNTRL calls C and C returns to CNTRL,
CNTRL calls A and A returns to CNTRL.

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:

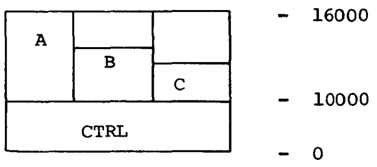
| | |
|---|---|
| CNTRL | 10000 bytes |
| A | 6000 bytes |
| B | 5000 bytes |
| C | 1200 bytes |

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:



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

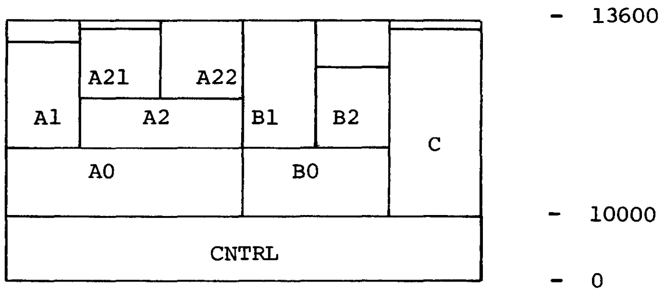
The memory allocation produced as a result of using the overlay capability and building a multi-segment task is as follows:



The multi-segment task requires 16ØØØ 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, further reduction can be made in the storage requirements for the task. Segment A is divided into a control program, AØ, and two overlays, A1 and A2. Then, A2 is further divided into a main part ,A2, and two overlays ,A21 and A22. Similarly, segment B is divided into a control module, BØ, 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 242ØØ bytes of storage. The first overlay structure reduced the requirement by 62ØØ bytes. The second overlay structure further reduced the storage requirement by 22ØØ 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, AØ, and A1 are loaded: the next such line gives memory when CNTRL, AØ, A2, and A21 are loaded: and so on.

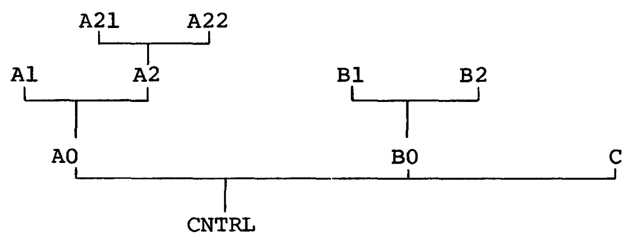
Observe also that a horizontal line can be drawn through the memory diagram to indicate segments that share the same storage. In the given diagram, the uppermost such line gives A1, A21, A22, B1, B2 and C, all of which can use the same memory; the next such line gives A1, A2, B1, B2, and C; and so on.

### 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:



The tree has a root, CNTRL, and three main branches, AØ, BØ, 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-AØ-CNTRL

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

CNTRL-BØ-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, AØ, and CNTRL; but A2 can not call A1, B1, B2, BØ or C.

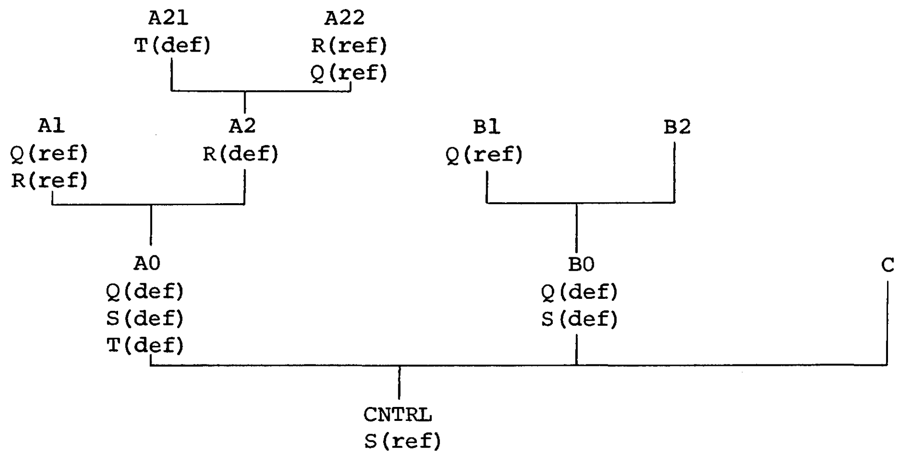
When a module calls a module in another overlay segment, the overlay segment must be in memory or must be brought into memory. The methods for loading overlays are described in the next chapter.

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.

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 (if the values differ) 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.



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 AØ and in the segment BØ. The reference to Q in segment A22 and the reference to Q in segment A1 are resolved to the definition in AØ. The reference to Q in B1 is resolved to refer to the definition of BØ. 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-AØ-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 AØ and BØ. References to S from A1, A21 or A22 are resolved to the definition in AØ and references to S in B1 and B2 are resolved to the definition in BØ. 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 AØ. Since there is a single path through the two definitions (CNTRL-AØ-A2-A21), the global symbol T is multiply defined.

5.1.2.3 Resolution of P-sections in a Multi-segment Task - Each 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.

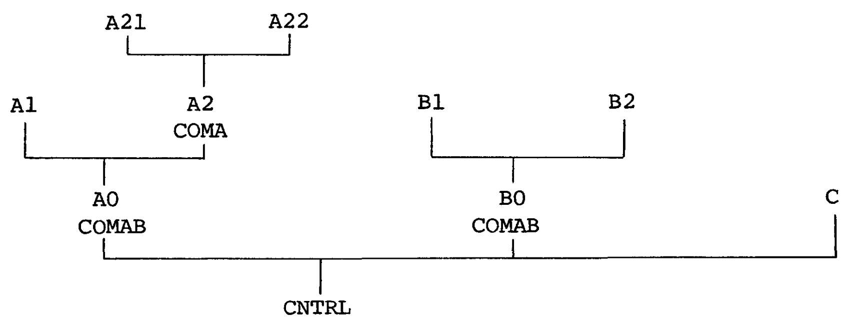
Local p-sections with the same name can appear in any number of segments. Storage is allocated for each local p-section in the segment in which it is declared. Global p-sections of the same name, however, must be resolved by the Task Builder.

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

The tree for the task TK1 including the allocation of the common blocks COMA and COMAB is:



The allocation of p-sections can be specified by the user. If AØ and BØ 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.

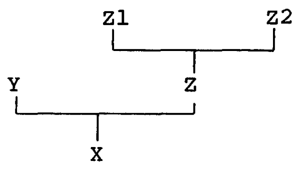
- The operator comma ´,´ appearing within parentheses indicates the overlaying of storage. For example, ´Y,Z´ means that memory can contain either Y or Z. Therefore Y and Z are share storage.

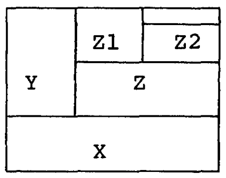
  This operator is also used to define multiple tree structures, as described in 5.1.4.

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-(AØ-(A1,A2-(A21,A22)),BØ-(B1,B2),C)
.END
```

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

```
MCR>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.

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 AØ-(A1,A2-(A21,A22))
BFCTR:    .FCTR BØ-(B1,B2)
          .END
```

The label 'BFCTR', is used in the .ROOT directive to designate the argument of the .FCTR directive, 'BØ-(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 AØ-(A1,A2FCTR)
A2FCTR:   .FCTR A2-(A21,A22)
BFCTR:    .FCTR BØ-(B1,B2)
          .END
```

The decision to use the .FCTR directive is based on considerations of space, style and readability of a complex ODL file.

5.1.3.3 <u>.NAME Directive</u> - 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 AØ-(A1,A2-(A21,A22))
BFCTR:    .FCTR BØ-(B1,B2)
          .END
```

Note that, in scanning down all .FCTR directives for a particular segment, the most recent (i.e., at deepest level) .NAME directive applied will be adopted if there is more than one .NAME directive for the segment.

5.1.3.4 <u>.PSECT Directive</u> - 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 TK1. The user was careful about the logical independence of the modules in the overlay segment, but the user failed to take into account the logical independence requirement of multiple executions of the same overlay segment.

The flow of the task TK1, as described earlier in this chapter, is 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 and 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 AØ-(A1,A2-(A21,A22))
BFCTR:    .FCTR BØ-(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.

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     XØ-(X1,X2,X3)
Y:        .FCTR     YØ-(Y1,Y2)
          .END
```

Two overlay trees are specified. A main tree containing the root segment XØ and three overlay segments and a co-tree consisting of root segment YØ and two overlay segments. The Executive loads segment XØ 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     XØ-YØ-(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 YØ.OBJ. YØ now resides in the main root.

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

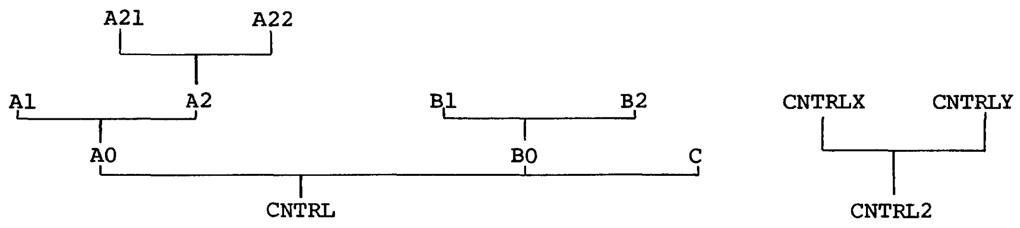
Suppose that in the task TK1, the root segment CNTRL consists of a small dispatching routine and two long modules, CNTRLX and CNTRLY. CNTRLX and CNTRLY are logically independent of each other, are approximately equal in length, and must access modules on all the paths of the main tree.

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:

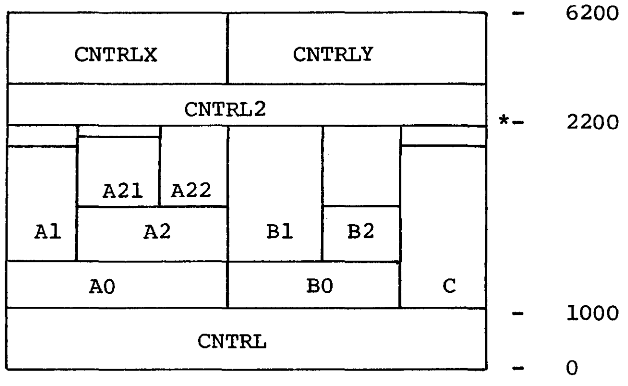
```
          .NAME CNTRL2
          .ROOT CNTRL-(AFCTR,BFCTR,C),CNTRL2-(CNTRLX,CNTRLY)
          ...
          .END
```

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:

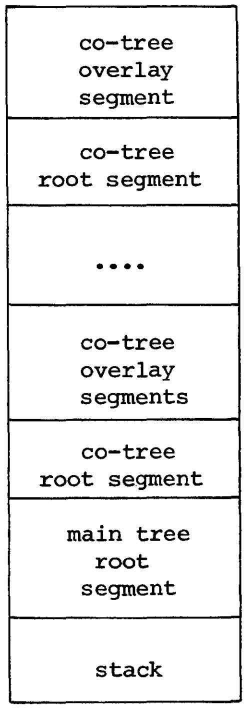


The specification of the co-tree decreases the storage allocation by 4ØØØ bytes. CNTRLX and CNTRLY can still access modules on all the paths of the main tree. The only requirement imposed by the introduction of the co-tree is the logical independence of CNTRLX and CNTRLY.

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.

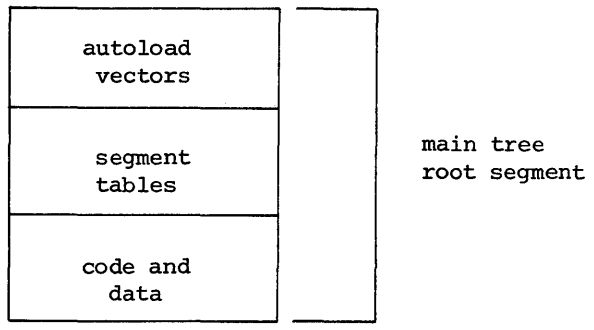
## 5.1.5 Overlay Core Image

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



The stack is described in Chapter 4.

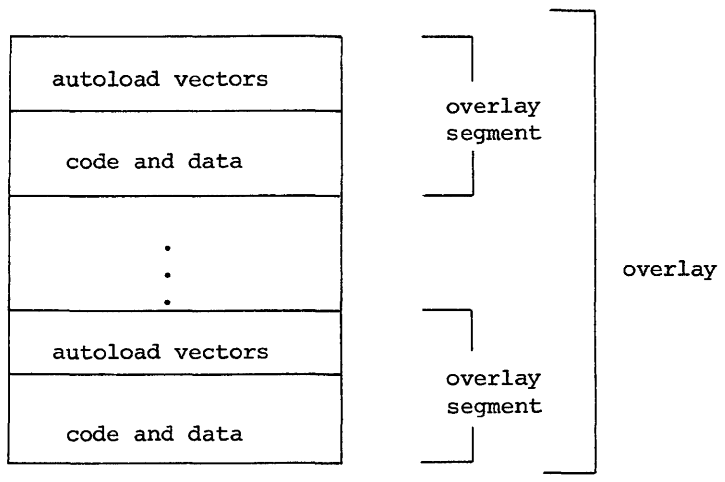
The root segment of the main tree contains all the modules that are resident in memory throughout the entire execution of the task, along with the segment tables, and if the autoload loading method is used, the autoload vectors.



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.



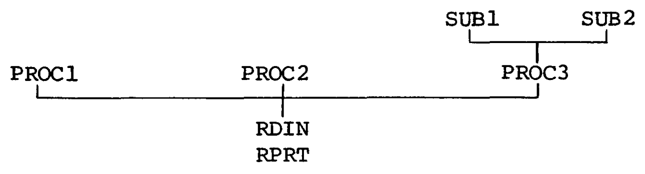
The co-tree overlay region consists of memory allocated for the overlay segments of the co-trees.

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

## 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:



### 5.2.1 Defining the ODL File

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

```
MCR>EDI
EDI>CALTR.ODL
[EDI -- CREATING NEW FILE]
INPUT
        .ROOT RDIN-RPRT-*(PROC1,PROC2,P3FCTR)
P3FCTR: .FCTR PROC3-(SUB1,SUB2)
        .END
*EX
```

**NOTE**

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

### 5.2.2 Building the Task

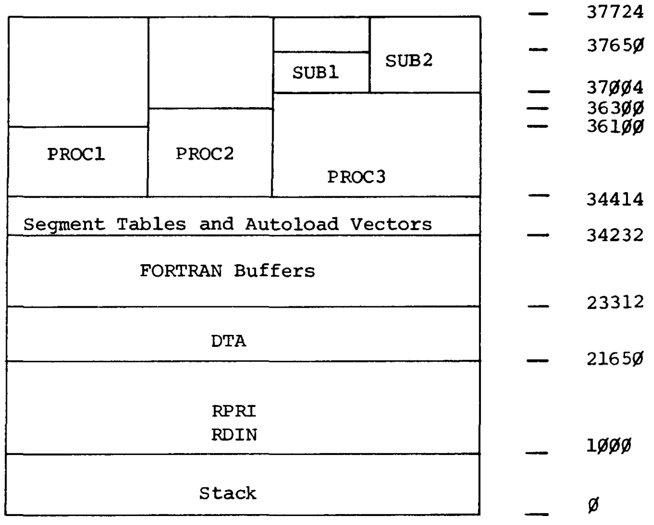
The user builds the task with the same options as in the example of Chapter 3. The names of the input files are replaced by a single filename that designates the file containing the overlay description:

```
MCR>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:



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

```
ROOT     34414
PROC1     1464
PROC2     1664
PROC3     2370
SUB1       644
SUB2       720
         -----
         44140
```

```
FILE CALC.TSK;3 MEMORY ALLOCATION MAP
THIS ALLOCATION WAS DONE ON 31-MAR-75
AT 15:47 BY TASK BUILDER VERSION R09


*** ROOT SEGMENT: RDIN


R/W MEM  LIMITS:  000000  034413  034414
STACK    LIMITS:  000000  000777  001000
DISK BLK LIMITS:  000003  000040  000036
IDENTIFICATION :  $FORT
PRG XFR ADDRESS:  001000
TASK ATTRIBUTES:  FP,NF


PROGRAM SECTION ALLOCATION SYNOPSIS:


<. BLK.>:  001000  021647  020650
<DTA   >:  021650  023311  001442
<$FIOF >:  023312  025271  001760
<$FIOI >:  025272  025675  000404
<$FIOL >:  025676  025676  000000
<$FIO1 >:  025676  027425  001530
<$FIO2 >:  027426  027463  000036
<$$ALER>:  027464  027507  000024
<$$AOTS>:  027510  030433  000724
<$$AUTO>:  030434  030567  000134
<$$DEVT>:  030570  031777  001210
<$$FSR1>:  032000  033017  001020
<$$FSR2>:  033020  033121  000102
<$$IOB1>:  033122  033325  000204
<$$IOB2>:  033326  033326  000000
<$$OBF1>:  033326  033435  000110
<$$OBF2>:  033436  033436  000000
<$$OVCT>:  033436  033711  000254
<$$OVDT>:  000000  000000  000000
<$$RESL>:  033712  034231  000320
<. ABS.>:  000000  000000  000000


*** SEGMENT: PROC1


R/W MEM  LIMITS:  034414  036077  001464
DISK BLK LIMITS:  000041  000042  000002
```

Figure 5-1
Memory Allocation File for CALC;3

```
PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: Ø34414 Ø34435 ØØØØ22
<ADTA  >: Ø34436 Ø36Ø77 ØØ1442


*** SEGMENT: PROC2

R/W MEM  LIMITS: Ø34414 Ø36277 ØØ1664
DISK BLK LIMITS: ØØØØ43 ØØØØ44 ØØØØØ2

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: Ø34414 Ø34635 ØØØ222
<ADTA  >: Ø34636 Ø36277 ØØ1442

*** SEGMENT: PROC3

R/W MEM  LIMITS: Ø34414 Ø37ØØ3 ØØ237Ø
DISK BLK LIMITS: ØØØØ45 ØØØØ47 ØØØØØ3

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: Ø34414 Ø35341 ØØØ726
<ADTA  >: Ø35342 Ø37ØØ3 ØØ1442

***SEGMENT: SUB1

R/W MEM  LIMITS: Ø37ØØ4 Ø37647 ØØØ644
DISK BLK LIMITS: ØØØØ5Ø ØØØØ5Ø ØØØØØ1


PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: Ø37ØØ4 Ø37647 ØØØ614


*** SEGMENT: SUB2

R/W MEM  LIMITS: Ø37ØØ4 Ø37723 ØØØ72Ø
DISK BLK LIMITS: ØØØØ51 ØØØØ51 ØØØØØ1


PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: Ø37ØØ4 Ø37721 ØØØ716
```

Figure 5-1 (cont.)
Memory Allocation File for CALC;3

## 5.3 EXAMPLE CALC;4

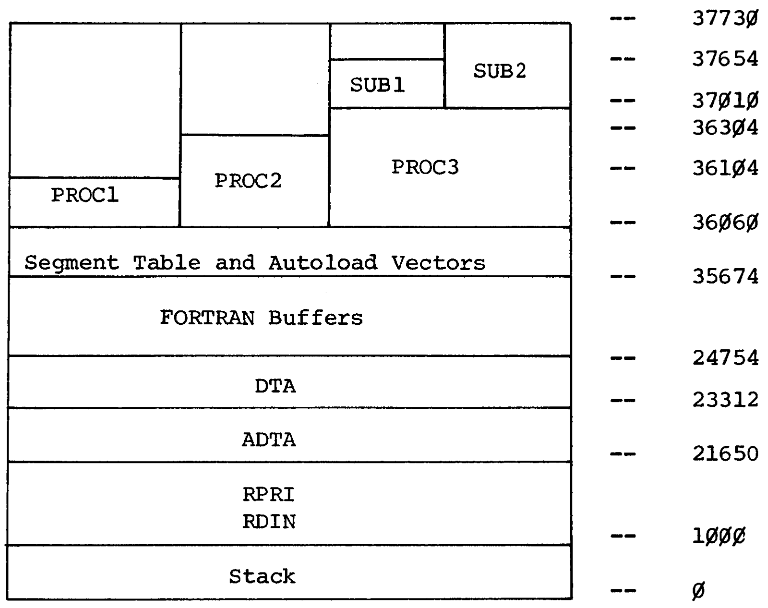
After examining the memory allocation file for CALC;3, the user observes that the Task Builder has allocated ADTA in the overlay segments PROC1, PROC2, and PROC3, since all of these segments are equidistant from the root.

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. CALTR is modified as follows:

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

The task is built as in CALC;3 and the resulting memory allocation file is represented by the following diagram:



```
FILE CALC.TSK;4 MEMORY ALLOCATION MAP
THIS ALLOCATION WAS DONE ON 31-MAR-75
AT 15:44 BY TASK BUILDER VERSION R09


***ROOT SEGMENT: RDIN

R/W MEM  LIMITS: 000000 036057 036060
STACK    LIMITS: 000000 000777 001000
DISK BLK LIMITS: 000003 000041 000037
IDENTIFICATION : $FORT
PRG XFR ADDRESS: 001000
TASK ATTRIBUTES: FP,NF

PROGRAM SECTION ALLOCATION SYNOPSIS

<. BLK.>: 001000 021647 020650
<ADTA  >: 021650 023311 001442
<DTA   >: 023312 024753 001442
<$FIOF >: 024754 026733 001760
<$FIOI >: 026734 027337 000404
<$FIOL >: 027340 027340 000000
<$FIO1 >: 027340 031067 001530
<$F102 >: 031070 031125 000036
<$$ALER>: 031126 031151 000024
<$$AOTS>: 031152 032075 000724
<$$AUTO>: 032076 032231 000134
<$$DEVT>: 032232 033441 001210
<$$FSR1>: 033442 034461 001020
<$$FSR2>: 034462 034563 000102
<$$IOB1>: 034564 034767 000204
<$$IOB2>: 034770 034770 000000
<$$OBF1>: 034770 035077 000110
<$$OBF2>: 035100 035100 000000
<$$OVCT>: 035100 035353 000254
<$$OVDT>: 000000 000000 000000
<$$RESL>: 035354 035673 000320
<. ABS.>: 000000 000000 000000
```

FIGURE 5-2
Memory Allocation File for CALC;4

```
*** SEGMENT: PROC1

R/W MEM  LIMITS: 036060 036103 000024
DISK BLK LIMITS: 000042 000042 000001

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 036060 036101 000022


*** SEGMENT: PROC2

R/W MEM  LIMITS:  036060 036303 000224
DISK BLK LIMITS:  000043 000043 000001

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 036060 036301 000222


*** SEGMENT: PROC3

R/W MEM  LIMITS: 036060 037007 000730
DISK BLK LIMITS: 000044 000044 000001

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 036060 037005 000726


*** SEGMENT: SUB1

R/W MEM  LIMITS:  037010 037653 000644
DISK BLK LIMITS:  000045 000045 000001

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 037010 037653 000644

*** SEGMENT: SUB2

R/W MEM LIMITS:  037010 037727 000720
DISK BLK LIMITS: 000046 000046 000001

PROGRAM SELECTION ALLOCATION SYNOPSIS:

<. BLK.>: 037010 037725 000716
```

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

## 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 (hyphen) and by the use of parentheses.

   The operator ´-´ 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 is required to terminate 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. If more than one .NAME directive is applied to a segment (via multiple .FCTR directives), the name encountered last in the scan down the ODL will be applied.

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 a 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 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 a 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.

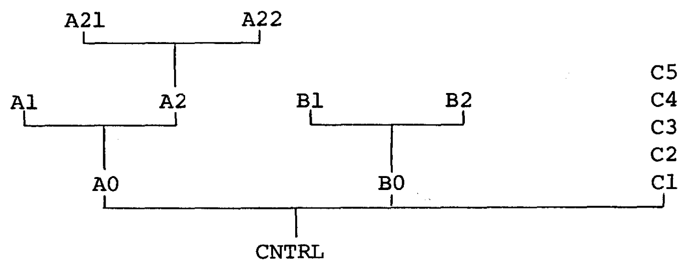
## 6.1 AUTOLOAD

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:



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 AØ-(A1,A2-(A21,A22))
BFCTR:  .FCTR BØ-(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.

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 AØ-*ASUB1-ASUB2-*(A1,A2-(A21,A22))
BFCTR:  .FCTR (BØ-(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, AØ, is made autoloadable.

*BFCTR The autoload indicator is applied to a factor label name, so the first irreducible component of that factor, (BØ-(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.

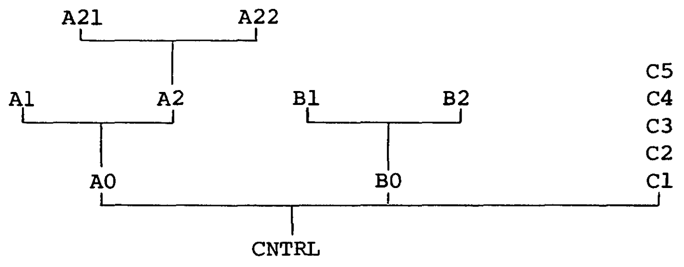
*(A1,A2-(A21,A22)) The autoload indicator is applied to a parenthesized ODL description so every name within the parentheses is made autoloadable; that is, A1, A2, A21, and A22.

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:



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.

## 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.

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 AØ
CALL A22
CALL BØ
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 AØ are loaded by path loading to A21, the autoload vectors for A2 and AØ 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 AØ-(*A1,A2-*(A21,A22))
BFCTR:    .FCTR (BØ-(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, BØ, B1, B2, and C1.

## 6.2 MANUAL LOAD

If the user decides to use the manual load method of loading segments, explicit calls to the $LOAD routine must be included in the 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.

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,RØ
          CALL    $LOAD
```

where PBLK labels a parameter block with the following format:

```
PBLK:     .BYTE   length,event-flag
          .RAD5Ø  /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 <u>RSX-11D Executive Reference Manual</u>. |
| 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,Ø,,,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.

## 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 returns 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 Ø.

## 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.

The ODL description in CALTR, is modified as follows:

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

The task is built 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 31-MAR-75
AT 15:50 BY TASK BUILDER VERSION R09


*** RDOT SEGMENT: RDIN


R/W MEM  LIMITS: 000000 036047 036050
STACK    LIMITS: 000000 000777 001000
DISK BLK LIMITS: 000003 000041 000037
IDENTIFICATION : $FORT
PRG XFR ADDRESS: 001000
TASK ATTRIBUTES: FP,NF


PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 001000 021647 020650
<ADTA  >: 021650 023311 001442
<DTA   >: 023312 024753 001442
<$FIOF >: 024754 026733 001760
<$FIOI >: 026734 027337 000404
<$FIOL >: 027340 027340 000000
<$FIO1 >: 027340 031067 001530
<$FIO2 >: 031070 031125 000036
<$$ALER>: 031126 031151 000024
<$$AOTS>: 031152 032075 000724
<$$AUTO>: 032076 032231 000134
<$$DEVT>: 032232 033441 001210
<$$FSR1>: 033442 034461 001020
<$$FSR2>: 034462 034563 000102
<$$IOB1>: 034564 034767 000204
<$$IOB2>: 034770 034770 000000
<$$OBF1>: 034770 035077 000110
<$$OBF2>: 035100 035100 000000
<$$OVCT>: 035100 035353 000254
<$$OVDT>: 000000 000000 000000
<$$RESL>: 035354 035673 000320
<. ABS.>: 000000 000000 000000
```

Figure 6-1
Root Segment of Memory Allocation
File for CALC;5

CHAPTER 7

# SHAREABLE GLOBAL AREAS

RSX-11D provides the facility for dynamic shareable global areas. This chapter describes the use and creation of shareable global areas in so far as they are related to task building.

Shareable global areas have the following characteristics.

1. They are created in a manner similar to tasks (i.e., with the Task Builder).

2. There are two types of shareable global areas.

   Read-only known (by convention) as libraries.

   Read/write known (by convention) as common blocks.

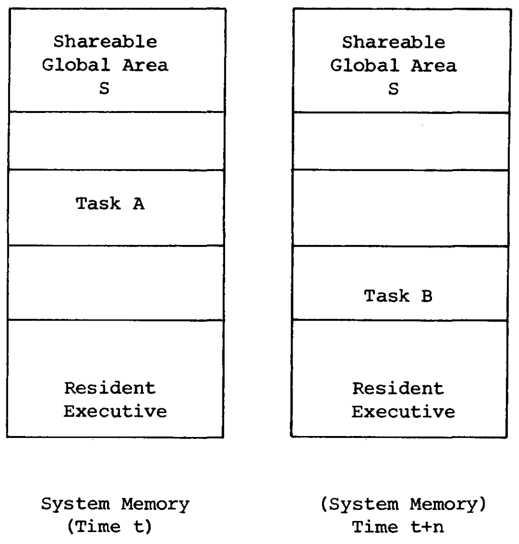
3. They occupy memory only when a referencing task is active. When all referencing tasks become inactive, the space occupied by shareable global areas is freed. Further, if the area being released was read/write, the data in the area are written back to the disk file from which the original loading occurred.

4. When a task binds to a shareable global area, the area must exist in the form of a task image and symbol table file (.TSK and .STB extensions are required) under the system UFD [1,1], on the system device.

5. At the time the binding task is to be installed, the shareable global area must have been installed.

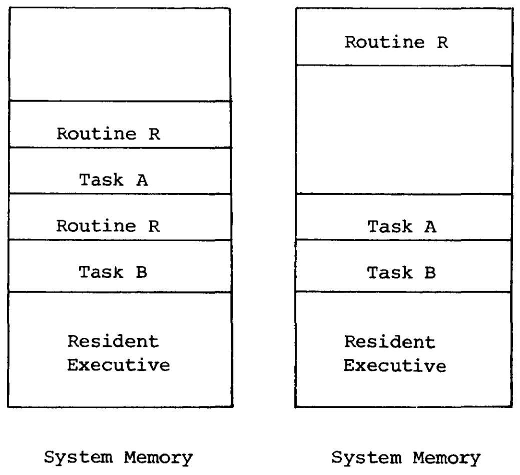
6. The /LI and /CM switches of Install are used to specify the shareable global area as a library or common block.

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 shareable global area. Tasks can communicate independent of their time of execution. This case is illustrated by the following diagram:



Task A and Task B communicate through the shareable global area, to which any number of tasks can be linked.

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 shareable global area so that a single copy is accessible to all tasks. This case is shown in the following diagram:



A task can link to as many as seven shareable global areas (SGA). If, however, the task is multi-user and has read-only sections in the root, this pure area of the root is considered as an SGA, and the number of external SGA´s which can be linked to the task is reduced to six.

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

## 7.1 USING AN EXISTING SHAREABLE GLOBAL AREA

The user can link to any of the system shareable global areas by specifying the COMMON or LIBR keyword option along with the name of the shareable global area and the type of requested access.

If the user wants to link task IMG1 to a system shareable global area named JRNAL so relevent data can be examined, the following COMMON keyword entry is used to specify the name JRNAL and read-only attribute.

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

A task can link to any shareable global area on the disk. However, before the task can be activated, any shareable global area it uses must be installed. These areas are loaded dynamically.

## 7.2 CREATING A SHAREABLE GLOBAL AREA

To create a shareable global area, the task image and symbol definition files must be built under UFD [1,1] on the system device.

In Chapter 4, runnable tasks were described. A shareable global area differs from a runnable task in that it does not have a header or a stack. Therefore, the user must specify that the header and stack are not to be produced for the task image file when a shareable global area is created. These two parameters are necessary and sufficient to identify the entity as a shareable global area to both the Task Builder and Install.

In summary, to create a shareable global area the following steps are taken:

- The task image file and symbol definition file are built under UFD [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=Ø is entered during option input to eliminate the stack.
- Although it is not mandatory, the user can save disk space by setting UNITS = Ø.

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

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

A task can now link to the shareable global area. However, before the task can be installed and activated, the shareable global area must be made known to the Executive via Install, defining the owner, non-owner access and the type of SGA. The following example illustrates a typical installation procedure (see Install command in RSX-11D User's Guide).

```
MCR>INS[1,1]ZETA/LI/UIC=[1,1]/ACC=RO
```

## 7.3 POSITION INDEPENDENT AND ABSOLUTE SHAREABLE GLOBAL AREAS

A shareable global area can be either position independent or absolute. Position independent shareable global areas can be placed anywhere in the task's virtual address space. Absolute areas must be fixed in the virtual address space.

The user must ensure that the area is position independent, if he applies the PI switch. The PI switch directs the Task Builder to treat the area as position independent, but the Task Builder can not determine whether or not the area is position independent. If the PI switch is applied to an area which is not truly position independent, the execution of a task linked to that area 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 shareable global areas because these programs do not satisfy the re-entrancy requirements necessary for shareable global areas. Refer to the RSX-11 MACRO-11 Reference Manual for further discussion of position independent coding.

FORTRAN common blocks can be included in shareable global areas. 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 shareable global area must be declared position independent. If the area is not declared position independent, the name is not retained and no FORTRAN program can link to the common block.

Absolute shareable global areas are used for code which satisfies the re-entrancy requirements for a shareable global area but is not position independent. The BASE or TOP options are used to build such images. Non-PIC shareable global areas can reference PIC shareable global areas, but PIC shareable global areas cannot reference non-PIC shareable global areas.

## 7.4 EXAMPLE: CALC;6 BUILDING AND USING A SHAREABLE GLOBAL AREA

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 shareable global area so that the two tasks can communicate.

In addition to creating the shareable global area, 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 the ODL description for the task.

### 7.4.1 Building the Shareable Global Area

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

The user first creates a separate input file for DTA:

```
MCR>EDI
EDI>DTA.FTN
[EDI -- 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:

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

Then the user builds the task image and symbol definition file for the shareable global area DTA:

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

```
TKB>STACK=Ø
TKB>UNITS=Ø
TKB>//
```

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

As required, the task image and symbol definition files are created on the system device under the User File Directory [1,1,], the switch -HD is applied to the symbol definition file to specify that the task has no header, the option STACK=Ø is entered to eliminate the stack, and Ø logical units are specified. It was necessary to specify the system device SYØ 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 shareable global area DTA now exists on the disk as an eligible candidate for inclusion in an active system. The user can now modify the task to link to that shareable global area. However, before the task can be executed, the shareable global area must be installed.

## 7.4.2 Modifying the Task to Use the Shareable Global Area

The user now modifies the task CALC. The file containing the program RDIN is edited to include the name of the reporting task in radix-5Ø format:

```
DATA RPTSK/6RGPRT  /
```

And the call to the dummy reporting routine RPRT is replaced 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(2ØØ), I
C   SET UP NAME OF REPORTING TASK IN RADIX 5Ø
      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. The .ROOT directive is changed from:

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

to:

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

Then, an indirect command file is built to include the COMMON keyword:

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

And the task is built with the single command referencing the indirect file:

```
    MCR>TKB @CALCBLD
```

The communication between the two tasks, CALC and GPRT, is now established. When the shareable global area 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 shareable global area 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 shareable global area 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 shareable global area allocated at 160000 and no longer part of the task.

```
FILE DTA.TSK;1 MEMORY ALLOCATION MAP
THIS ALLOCATION WAS DONE ON 31-MAR-75
AT 15:54 BY TASK BUILDER VERSION R09

*** ROOT SEGMENT: DTA

R/W MEM LIMITS: 000000 001443 001444
DIS BLK LIMITS: 000002 000003 000002
IDENTIFICATION:
TASK ATTRIBUTES:FP,NF,PI,NH

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 000000 000000 000000
<DTA   >: 000000 001441 001442
<. ABS.>: 000000 000000 000000
```

Figure 7-1
Memory Allocation File for the Shareable Global Area DTA

```
FILE CALC.TSK:6 MEMORY ALLOCATION MAP
THIS ALLOCATION WAS DONE ON 31-MAR-75
AT 16:20 BY TASK BUILDER VERSION R09

*** ROOT SEGMENT: RDIN

R/W MEM  LIMITS: 000000 034613 034614
STACK    LIMITS: 000000 000777 001000
DISK BLK LIMITS: 000003 000040 000036
IDENTIFICATION : $FORT
PRG XFR ADDRESS: 001000
TASK ATTRIBUTES: FP,NF

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 001000 022057 021060
<ADTA  >: 022060 023521 001442
<DTA   >: 160000 161441 001442
<$FIOF >: 023522 025501 001760
<$FIOI >: 025502 026105 000404
<$FIOL >: 026106 026106 00000
<$FIO1 >: 026106 027635 001530
<$FIO2 >: 027636 027673 000036
<$$ALER>: 027674 027717 000024
<$$AOTS>: 027720 030643 000724
<$$AUTO>: 030644 030777 000134
<$$DEVT>: 031000 032207 001210
<$$FSR1>: 032210 033227 001020
<$$FSR2>: 033230 033331 000102
<$$IOB1>: 033332 033535 000204
<$$IOB2>: 033536 033536 000000
<$$OBF1>: 033536 033645 000110
<$$OBF2>: 033646 033646 000000
<$$OVCT>: 033646 034121 000254
<$$OVDT>: 000000 000000 000000
<$$RESL>: 034122 034441 000320
<. ABS.>: 000000 000000 000000
```

FIGURE 7-2
Memory Allocation File for CALC;6

# APPENDIX A

# 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.

Ø. 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 file is required for a task build.

3. ILLEGAL SWITCH
   invalid-line

   The invalid line printed contains an illegal switch or switch value.

4. NO DYNAMIC STORAGE AVAILABLE

   The Task Builder needs additional symbol table storage and cannot obtain it. The input has exceeded the Task Builder´s capability.

5. ILLEGAL ERROR/SEVERITY CODE

   System error. (No recovery.)

6. COMMAND I/O ERROR

   I/O error on command input device. (Device may not be online or possible hardware error.)

7. INDIRECT FILE OPEN FAILURE
   invalid-line

   The invalid-line contains a reference to a command input file which could not be located.

8. INDIRECT COMMAND SYNTAX ERROR
   invalid-line

   The invalid-line printed contains a syntactically incorrect indirect file specification.

9. MAXIMUM INDIRECT FILE DEPTH EXCEEDED
   invalid-line

   The invalid-line printed gives the file reference that exceeded the permissible indirect file depth (2).

10. I/O ERROR ON INPUT FILE file-name

11. OPEN FAILURE ON FILE file-name

12. SEARCH STACK OVERFLOW ON SEGMENT segment-name

    The segment segment-name is more than 16 branch segments from the root segment.

13. PASS CONTROL OVERFLOW AT SEGMENT segment-name

    The segment segment-name is more than 16 branch segments from the root segment.

14. FILE file-name HAS ILLEGAL FORMAT

    The file file-name contains an object module whose format is not valid.

15. MODULE module-name AMBIGUOUSLY DEFINES P-SECTION p-sect-name

    The p-section p-sect-name has been defined in two modules not on a common path and referenced ambiguously.

16. MODULE module-name MULTIPLY DEFINES P-SECTION p-sect-name

    1. The p-section p-sect-name has been defined in the same segment with different attributes.

    2. A global p-section has been defined in more than one segment along a common path with different attributes.

17. MODULE module-name MULTIPLY DEFINES XFR ADDR IN SEG segment-name

    This error occurs when more than one module comprising the root has a start address.

18. MODULE module-name ILLEGALLY DEFINES XFR ADDRESS p-sect-name addr

    The module module-name is in an overlay segment and has a start address. The start address must be in the root segment of the main tree.

19. P-SECTION p-sect-name HAS OVERFLOWED

    A section greater than 32K has been created.

20. MODULE module-name AMBIGUOUSLY DEFINES SYMBOL sym-name

    Module module-name references or defines a symbol sym-name whose definition cannot be uniquely resolved.

21. MODULE module-name MULTIPLY DEFINES SYMBOL sym-name

    Two definitions for the relocatable symbol sym-name have occurred on a common path. Or two definitions for an absolute symbol with the same name but different values have occurred.

22. SEGMENT seg-name HAS R-O SECTION

    An attempt has been made to allocate a read-only p-section in an overlay segment. The build continues with R-W attribute forced.

23. SEGMENT seg-name HAS ADDR OVERFLOW: ALLOCATION DELETED

    Within a segment, the program has attempted to allocate more than 32K. A map file is produced, but no task image file is produced.

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 INDENTIFIER
    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.

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. ILLEGAL BASE OR TOP ADDRESS OFFSET

The task is too large to fit into the space allowed by BASE= or TOP= keywords.

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. ILLEGAL NUMBER OF ACTIVE FILES SPECIFIED

51. ILLEGAL BASE OR TOP BOUNDARY VALUE
invalid line

52. ILLEGAL POOL USAGE NUMBER SPECIFIED
invalid line

The pool request is greater than 255 or it is zero.

53. ILLEGAL DEFAULT PRIORITY SPECIFIED
invalid-line

The invalid-line printed contains a priority greater than 25Ø.

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. ILLEGAL DEVICE/VOLUME

invalid line

The device/volume string is too long.

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 UFD.

59. INCOMPATIBLE REFERENCE TO A LIBRARY P-SECTION p-sect-name

A task has attempted to reference more storage in a shareable global area than exists in the shareable global area definition.

6Ø. ILLEGAL REFERENCE TO LIBRARY P-SECTION p-sect-name

A task has attempted to reference a p-sect-name existing in a resident library (shareable global area) 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 shareable global areas have been specified.

2. The same shareable global area has been specified more than once.

3. Shareable global areas whose memory allocations overlap

have been specified.

4. BASE or TOP specifications conflict.

62. LOOKUP FAILURE RESIDENT LIBRARY FILE
invalid-line

No symbol table or task image file found for the shareable global area on SYØ under UFD [1,1].

63. INVALID ACCESS TYPE
invalid-line

Requested access to shareable global area was not RW or RO.

64. ILLEGAL PARTITION/COMMON BLOCK SPECIFIED
invalid-line

User defined base or length not on 32 word bound or user defined length = Ø.

65. NO MEMORY AVAILABLE FOR LIBRARY library-name

Insufficient virtual memory available to cover total memory needed by referenced shareable global areas.

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 Ø-7.

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 shareable global area.

69. LIBRARY REFERENCES UNDEFINED LIBRARY

A shareable global area specified by LIBR or COMMON references another area which is undefined.

7Ø. not used.

71. INVALID APR RESERVATION

APR specified on a LIBR or COMMON keyword for an absolute shareable global area.

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. Possibly an attempt to extend the file when no more space is available on the volume.

74. LOOKUP FAILURE ON SYSTEM LIBRARY FILE

    The Task Builder cannot find the system Library (SYØ:[1,1]SYSLIB.OLB) file to resolve undefined symbols.

75. UNABLE TO OPEN WORK FILE

    Work file device is not mounted or has not been initialized as Files-11, or there is no space on the volume.

76. NO VIRTUAL MEMORY STORAGE AVAILABLE

    Maximum permissible size of the work file exceeded (no recovery).

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-5Ø character.

79. LIBRARY FILE filename HAS INCORRECT FORMAT

    A module has been requested from a library file that has an empty module name table.

8Ø. RESIDENT LIBRARY IMAGE HAS INCORRECT FORMAT
    invalid-line

    The invalid-line specifies a shareable global area that has one of the following problems:

    1. The library file image has a header.

    2. The shareable global area references another shareable global area with invalid address bounds (i.e., not on 4K boundary).

    3. The shareable global area has invalid address bounds.

81. PARTITION partition-name HAS ILLEGAL MEMORY LIMITS

    The user has attempted to build a privileged task whose length exceeds 16K.

82. not used

83. ABORTED VIA REQUEST
input-line

The input-line contains a request from the user to abort the task build.

84. not used

85. END OF FILE REACHED BEFORE .END DIRECTIVE IN file-name

The overlay description file named in this message does not contain a .END directive as required.

# 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-11D 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.

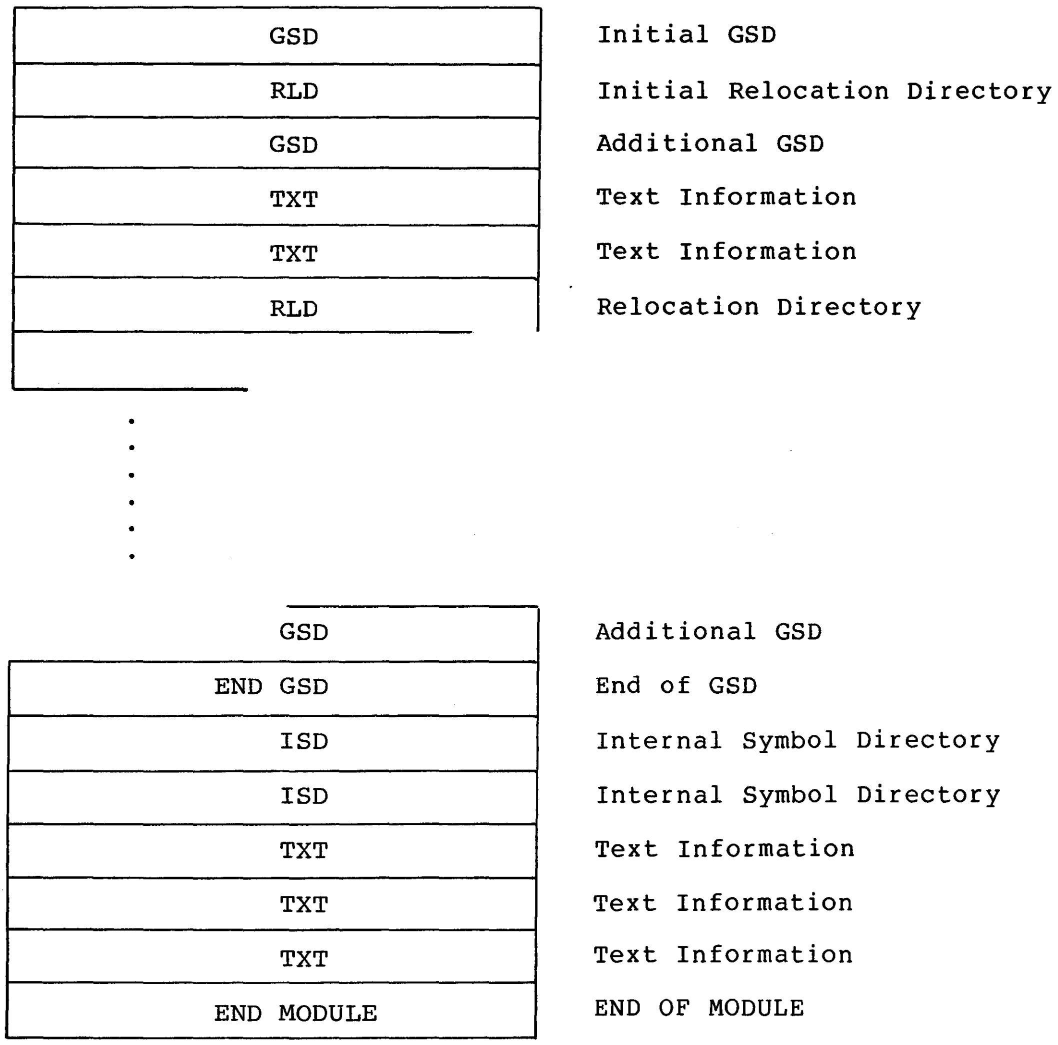


Figure B-1
General Object Module Format

## 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).

GSD records contain seven types of entries:

- Type Ø - 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-5Ø 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.

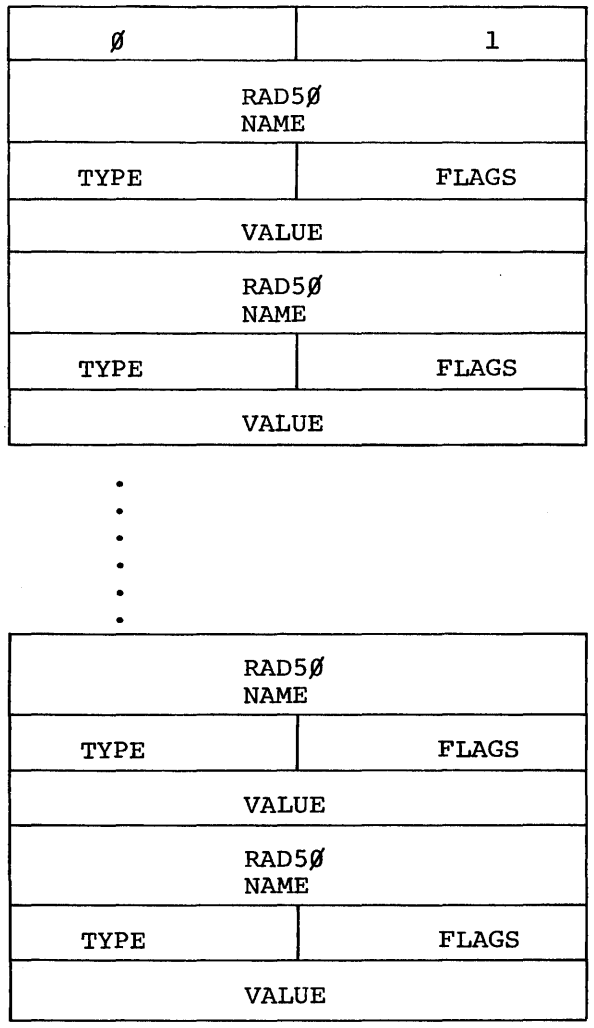


Figure B-2
GSD Record And Entry Format

## 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.

| | |
|---|---|
| MODULE NAME | |
| Ø | Ø |
| Ø | |

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-11D 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

## 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.

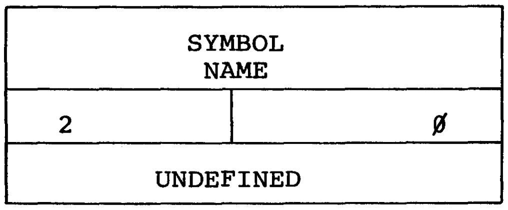


Figure B-5
Internal Symbol Name Entry Format

## 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 ØØØØØ1 relative to the default absolute P-section (. ABS.) must be specified. See Figure B-6.

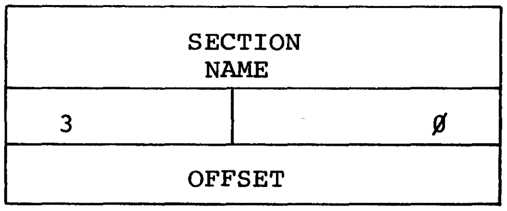


Figure B-6
Transfer Address Entry Format

NOTE

If the P-section is absolute, then OFFSET is the actual transfer address if not ØØØØØ1.

## 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 Ø - 2 - Not used.

Bit 3 - Definition

Ø = Global symbol references.

1 = Global symbol definition.

Bit 4 - Not used

Bit 5 - Relocation

Ø = Absolute symbol value.

1 = Relative symbol value

Bit 6 - 7 - Not used.

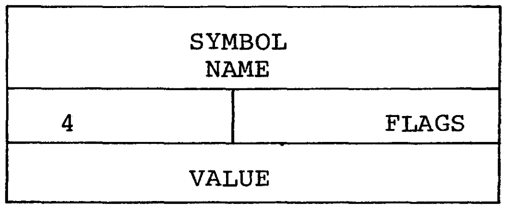


Figure B-7
Global Symbol Name Entry Format

## 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, the next 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 shareable global area.

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

Ø = 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

Ø = The P-section contains instruction (I) references.

1 = The P-section contains data (D) reference Identification

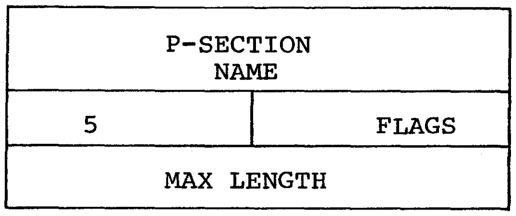


Figure B-8
P-Section Name Entry Format

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 | Ø |
| Ø | |

Figure B-9
Program Version Identification Entry Format

## B.2 END-OF-GLOBAL-SYMBOL-DIRECTORY

The end-of-global-symbol-directory record (Figure B-1Ø) 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.

| | |
|---|---|
| Ø | 2 |

Figure B-1Ø
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 Section B.4). 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.

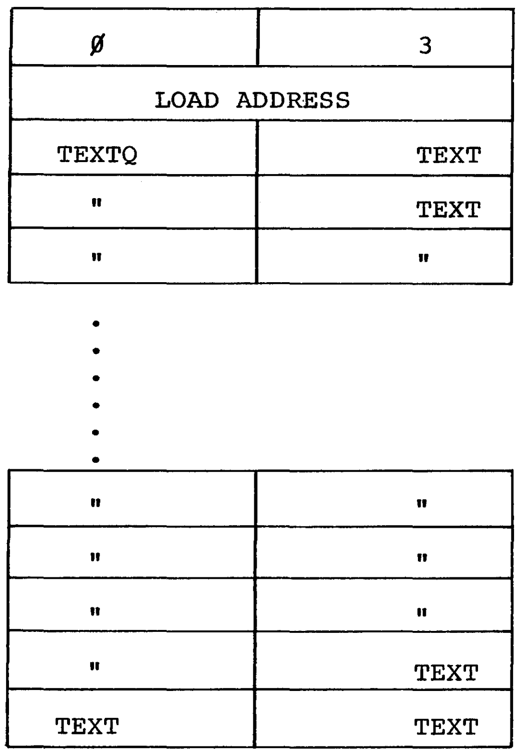


Figure B-11
Text Information Record Format

## 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 1Ø - 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), a displacement byte, and the information required for the particular type of entry, in that order. The displacement byte, when added to the value calculated from the load address of the previous text information record, (see Section G.3) yields the virtual address in the image that is to be modified. The command byte of each entry has the following bit assignments.

Bits Ø - 6 Specify the type of entry. Potentially 128 command types may be specified although only 15(decimal) are implemented.

Bit - 7 Modification

Ø = 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.

| | |
|---|---|
| Ø | 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

## 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,RØ

        or

        .WORD   A
```

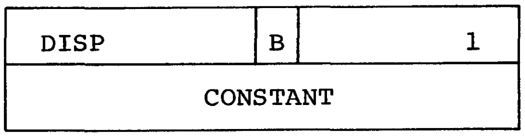


Figure B-13
Internal Relocation Command Format

## 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,RØ

                or

        .WORD   GLOBAL
```

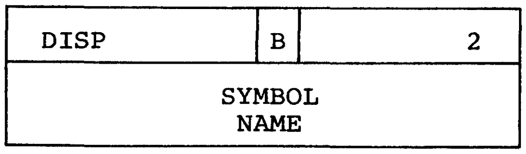


Figure B-14
Global Relocation

## 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 sectiion. 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     17755Ø

                or

        MOV     17755Ø,RØ
```

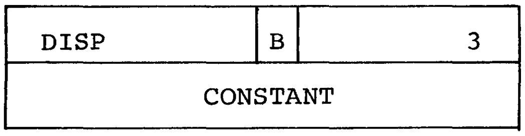


Figure B-15
Internal Displaced Relocation

## B.4.4 Global Displaced Relocation

This type of entry (Figure B-16) relates a relative reference to a global symbol. The definition of the global symbol is obtained and the address plus 2 that the relocated value is to be written into is subtracted from the definition value. This value is then written into the task image file at the calculated address.

Example:

```
        CLR     GLOBAL

        or

        MOV     GLOBAL,RØ
```

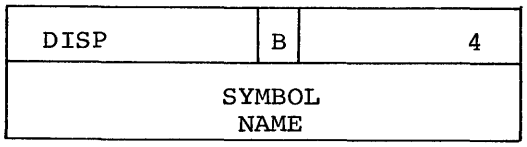


Figure B-16
Global Displaced Relocation

## 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
```

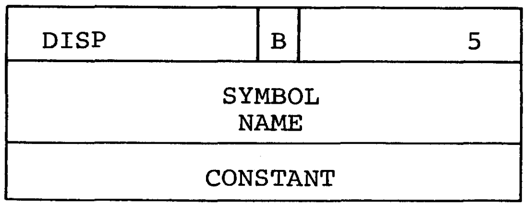


Figure B-17
Global Additive Relocation

## 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
```

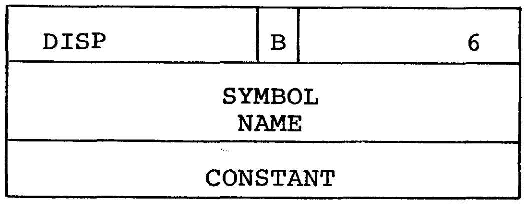


Figure B-18
Global Additive Displaced Relocation

## 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.

| | | |
|---|---|---|
| Ø | B | 7 |
| SECTION NAME | | |
| CONSTANT | | |

Figure B-19
Location Counter Definition

## B.4.8 Location Counter Modification

This type of entry (Figure B-2Ø) 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
```

| | | |
|---|---|---|
| Ø | B | 1Ø |
| CONSTANT | | |

Figure B-2Ø
Location Counter Modification

### 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.4.1Ø 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,RØ

           or

        .WORD   B
```

| | | |
|---|---|---|
| DISP | B | 12 |
| SECTION NAME | | |

Figure B-22
P-Section Relocation

### 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,RØ
```

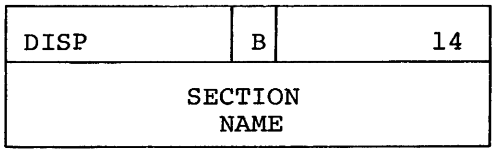


Figure B-23
P-Section Displaced Relocation

### 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.

Example:

```
        .PSECT  A
B:
        .
        .
        .
        .
        .
C:      .
        .
        .
        .
        PSECT   D
        MOV     #B+1Ø,RØ
        MOV     #C,RØ

        or

        .WORD   B+1Ø
        .WORD   C
```

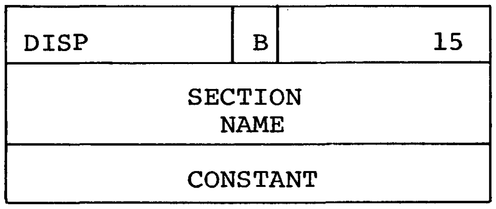


Figure B-24
P-Section Additive Relocation

## 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.

Example:

```
        .PSECT  A
B:
        .
        .
        .
C:      .
        .
        .
        .
        .
        .PSECT  D

        MOV     B+1Ø,RØ
        MOV     C,RØ
```

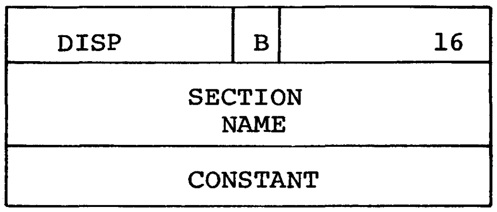


Figure B-25
P-Section Additive Displaced Relocation

## B.4.14 Complex Relocation

This type of entry (Figure B-26) 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.

Ø - No operation

1 - Addition (+)

2 - Subtraction (-)

3 - Multiplication (*)

4 - Division (/)

5 - Logical AND (&)

6 - Logical inclusive OR (!)

10 - Negation (-)

11 - Complement (^C)

12 - Store result (command termination)

13 - Store result with displaced relocation (command termination)

16 - Fetch global symbol. It is followed by four bytes containing the symbol name in RADIX-50 representation.

17 - Fetch relocatable value. It is followed by one byte containing the sector number, and two bytes containing the offset within the sector.

20 - Fetch constant. It is followed by two bytes containing the constant.

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.

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:
          .
          .
          .
          MOV      #A+B-G1/G2&<^C<17712Ø!G3>>,R1
```

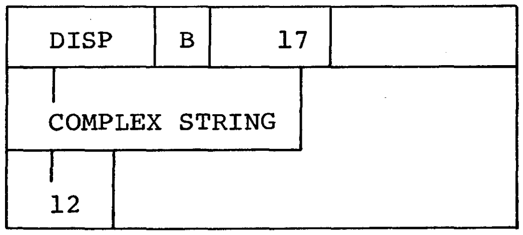


Figure B-26
Complex Relocation

## B.5 INTERNAL SYMBOL DIRECTORY

Internal symbol directory records (Figure B-27) 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.

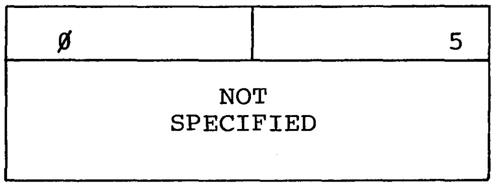


Figure B-27
Internal Symbol Directory Record Format

## B.6 END OF MODULE

The end-of-module record (Figure B-28) 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.

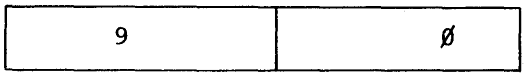


Figure B-28
End-Of-Module Record Format

# APPENDIX C

# TASK IMAGE FILE STRUCTURE

The task image as it is recorded on the disk appears in Figure C-1.

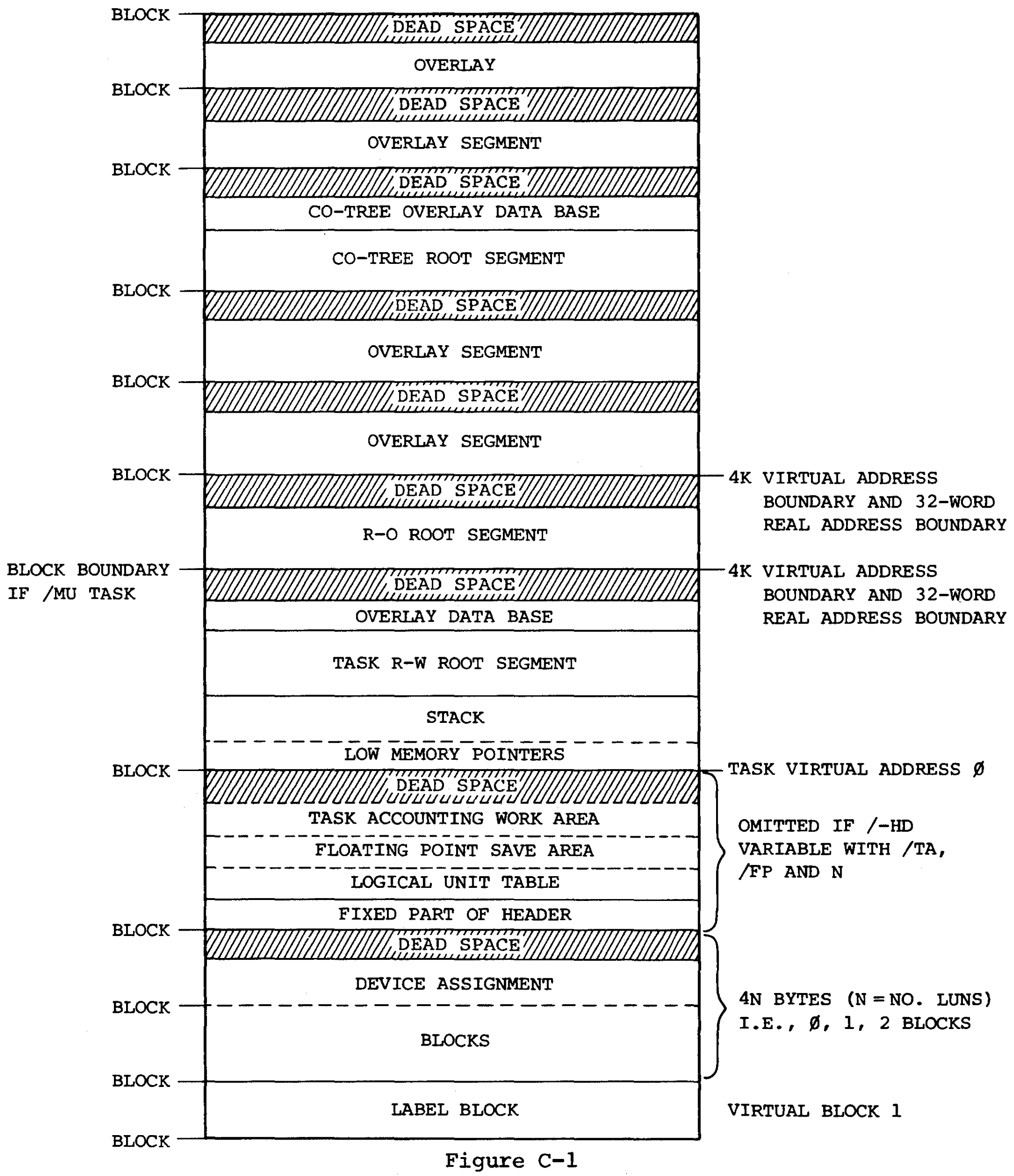
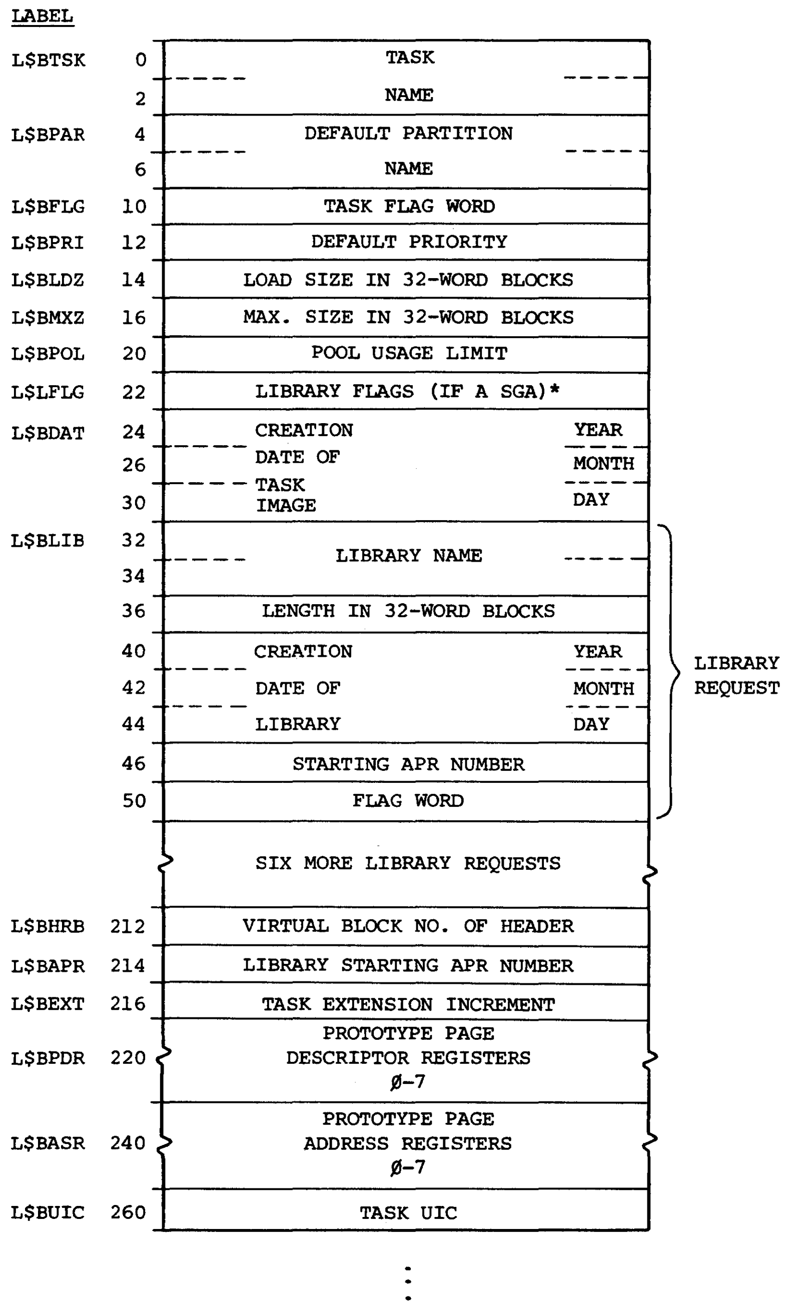


Figure C-1
Task Image on Disk

## 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.


LABEL
L$BTSK 0 TASK
2 NAME
L$BPAR 4 DEFAULT PARTITION
6 NAME
L$BFLG 10 TASK FLAG WORD
L$BPRI 12 DEFAULT PRIORITY
L$BLDZ 14 LOAD SIZE IN 32-WORD BLOCKS
L$BMXZ 16 MAX. SIZE IN 32-WORD BLOCKS
L$BPOL 20 POOL USAGE LIMIT
L$LFLG 22 LIBRARY FLAGS (IF A SGA)*
L$BDAT 24 CREATION YEAR
26 DATE OF MONTH
30 TASK IMAGE DAY
L$BLIB 32 LIBRARY NAME
34
36 LENGTH IN 32-WORD BLOCKS
40 CREATION YEAR
42 DATE OF MONTH
44 LIBRARY DAY
46 STARTING APR NUMBER
50 FLAG WORD
LIBRARY REQUEST
SIX MORE LIBRARY REQUESTS
L$BHRB 212 VIRTUAL BLOCK NO. OF HEADER
L$BAPR 214 LIBRARY STARTING APR NUMBER
L$BEXT 216 TASK EXTENSION INCREMENT
L$BPDR 220 PROTOTYPE PAGE DESCRIPTOR REGISTERS Ø-7
L$BASR 240 PROTOTYPE PAGE ADDRESS REGISTERS Ø-7
L$BUIC 260 TASK UIC

*Shareable Global Area

Figure C-2
Label Block Group

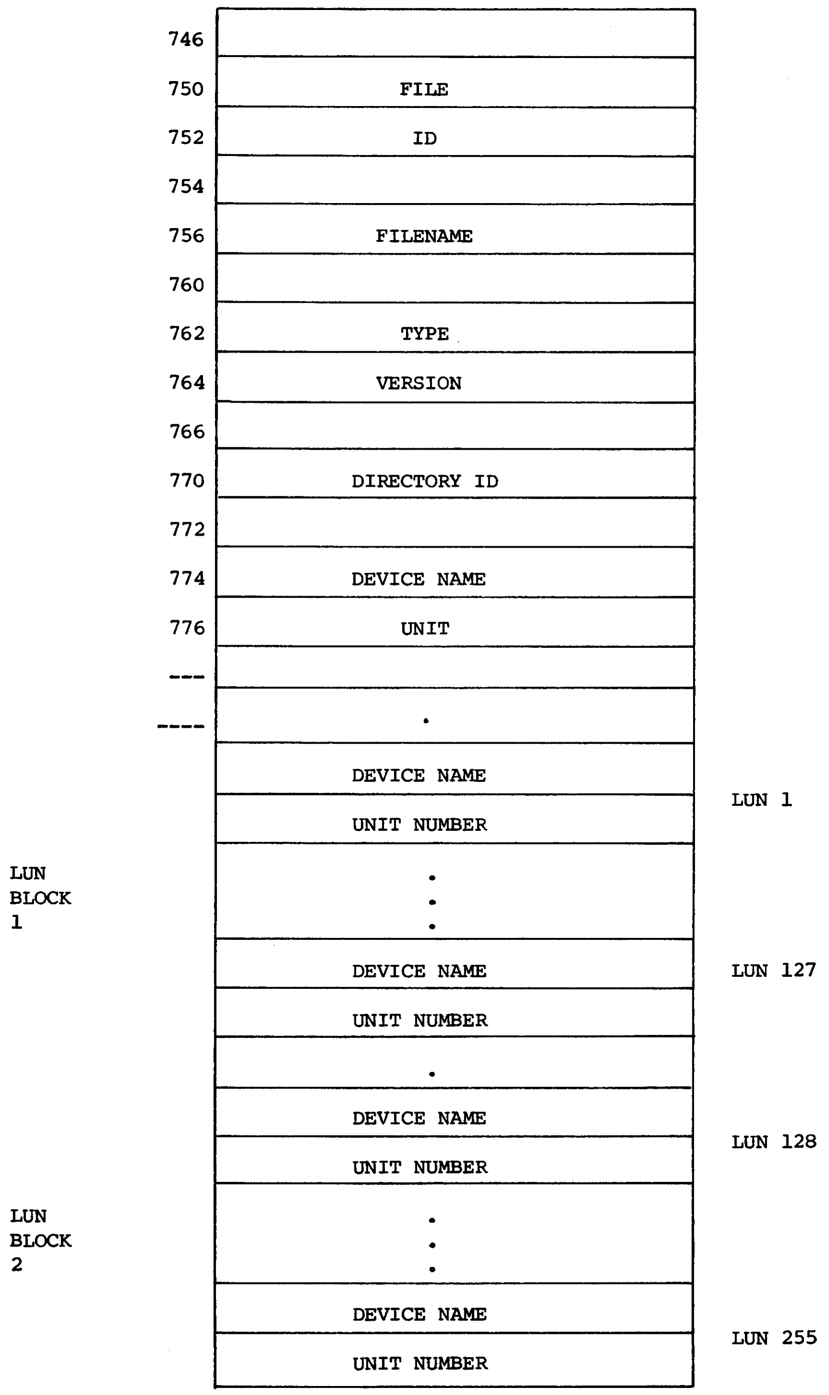


Figure C-2 (Cont.)
Label Block Group

## 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 (STD) entry for the task, and in linking the task to shareable global areas.

| | |
|---|---|
| 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 Set=1 |
|---|---|---|
| SF.MU | 6 | Task is multi-user (/MU) |
| SF.PT | 7 | Task is privileged (/PR) |
| SF.XA | 11 | Task is not abortable (/-AB) |
| SF.XD | 12 | Task is not disableable (/-DS) |
| SF.XF | 13 | Task is not fixable (/-FX) |
| SF.XC | 14 | Task is not checkpointable (/-CP) |

| | |
|---|---|
| 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$BMXZ is used by Install to verify that the task fits into the specified partition. |
| L$BPOL | Pool usage limit indicating maximum number of pool nodes that can be used simultaneously by the task. The default is 40 (decimal) which is overridden by the POOL keyword. |
| L$LFLG | Flags word for the image of a shareable global area. |

| | Bit | Interpretation if Set=1 |
|---|---|---|
| LF$PIC | 0 | Image is position independent (/PI) |
| LF$NHD | 1 | Image has no header (/-HD) |

| | |
|---|---|
| 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

The following paragraphs describe components of the Shareable Global Area Name Block. An 8-word block is generated for each shareable global area referenced by the task. Because shareable global areas need not be resident in the system, the Task Builder builds the block from the area's disk image, using information in the label blocks of that image.

| | |
|---|---|
| Library Name | A 2-word Radix-5Ø name specified in the LIBR or COMMON keyword. |
| Creation Date | Obtained from the creation date in the shareable global area disk image label block. |
| Starting Address | First address used to map the Shareable Global Area into the task addressing space. |

Flag Word Bits 2 and 15 are used as follows:

| | Bit | Value | Meaning |
|---|---|---|---|
| LD$REL | 2 | 1 | Global area is PIC.set if value of LF$PIC in the library image flags word (L$FLG) is =1. |
| | | 0 | Global area is absolute. Cleared if LF$PIC in L$LFLG of global area image is Ø. |
| LD$ACC | 15 | 1 | Read/Write access request. Set if RW specified in LIBR or COMMON option. |
| | | 0 | Read-only ACCESS request. Cleared if RO specified in LIBR or COMMON option. |

L$BHRB Virtual block number of the task header. Between 2 and 4 depending on number of LUNS, as follows:

```
UNITS = Ø  virtual block 2
UNITS = 1-128  virtual block 3
UNITS = 129-255  virtual block 4
```

L$BAPR Starting APR number if this image is a shareable global area. Calculated from BASE or TOP keywords.

L$BEXT The default number of words by which the memory allocated to a task at install time will be increased. This value is overridden by INSTALL/INC keyword. Value is set with EXTTSK keyword of TKB.

L$BPDR The eight prototype page descriptor registers for the task. INSTALL copies these into task header where necessary modifications can be performed.

L$BASR The eight prototype page address registers for the task. INSTALL copies these into the task header where necessary modifications can be performed.

| | |
|---|---|
| L$BUIC | The UIC with which the task is built. Set by UIC keyword. |

Since the R/W and R-O parts of the root segment are each contiguous blocks on the disk, each can be loaded with a single disk read. If the task is not multi-user, only a single read is required.

## C.2 HEADER

The task is read into main memory starting at the base of the Header. Figure C-3 illustrates the format of the fixed part. As shown in Figure C-1, the variable part consists of the Logical Unit Table, Floating Point Save Area and the Task Accounting Work Area. The Logical Unit Table identifies to the Executive which device is assigned to which LUN. The Floating Point Save Area is storage for the PDP-11/45 floating point registers when this option is requested. The task accounting work area is used by the Executive if accounting is enabled.

The Header is always a multiple of 32-word blocks. This insures that the root segment code starts on a 32-word boundary, a requirement for the allocation of a APR pair of relocation registers. The Task Header is not covered by a task relocation register, and is therefore, not part of the virtual address space of the task.

NOTE

Privileged tasks may access the Header via symbolic offsets, which are obtained by linking the file EXEC.STB. Executive defined symbolic offsets are indicated in the figure, to the right of the corresponding Header element and Task Builder reference labels are those to the left. The real load address of a task header can be found from the tasks Active Task List entry.

TASK IMAGE FILE STRUCTURE

| LABEL | | | | OFFSETS |
|---|---|---|---|---|
| H$DFLP | Ø | FLOATING POINT SAVE POINTER | | H.CR1 |
| H$DPDR | 2 | INSTALL TIME PAGE DESCRIPTOR REGISTERS | | H.PDØ |
| | 2Ø | | | H.PD7 |
| H$DPAR | | INSTALL TIME PAGE ADDRESS REGISTERS | | H.PAØ |
| | 4Ø | | | H.PA7 |
| H$DPWA | 42 | RUN TIME PAGE ADDRESS REGISTERS | | H.PWØ |
| | 6Ø | | | |
| H$DPS | 62 | CURRENT PS WORD | | H.TPS |
| H$DPC | 64 | CURRENT PC WORD | | H.TPC |
| H$DRØ | 66 | INITIAL RØ | | |
| H$DR1 | 7Ø | INITIAL R1 | | |
| H$DR2 | 72 | INITIAL R2 | | |
| H$DR3 | 74 | INITIAL R3 | | |
| H$DR4 | 76 | INITIAL R4 | | |
| H$DR5 | 1ØØ | INITIAL R5 | | |
| H$DSP | 1Ø2 | CURRENT SP | | H.TSP H.CR2 |
| H$DIPS | 1Ø4 | INITIAL PS WORD | | H.ISP |
| H$DPIC | 1Ø6 | INITIAL PC WORD | | H.IPC |
| H$DISP | 11Ø | INITIAL SP | | H.ISP |
| H$DDSV | 112 | ODT SST VECTOR ADDRESS | | H.DSV |
| H$DTSV | 114 | TASK SST VECTOR ADDRESS | | H.TSV |
| H$DTVT H$DDVT | 116 | TASK VECTOR LENGTH | ODT VECTOR LENGTH | H.DV7 (116) H.TVZ (117) |
| H$DPHN | 12Ø | POWERFAIL AST. | | H.PUN |

Figure C-3
Task Header Fixed Part

| LABEL | | | OFFSETS |
|---|---|---|---|
| H$DFEN | 122 | FLOATING POINT EXCEPTION ADDRESS | H.FEN |
| H$DDUC | 124 | DEFAULT UIC | H.DUI |
| H$DUIC | 126 | CURRENT UIC | H.UIC |
| H$DSIZ | 13Ø | SIZE OF HEADER IN 32-WORD BLOCKS | H.HSZ |
| H$DFZI | 132 | FILE SIZE INDICATION | H.FZI |
| | 134 | RECEIVE AST NODE ADDRESS | H.REC |
| | 136 | SPARE | |
| | 14Ø | EXECUTIVE VALIDATION WORD | H.CHK |
| | 142 | SPARE | |
| | 144 | SPARE | |
| H$IOQ | 146 | I/O QUEUE | H.IOQ |
| | 15Ø | LISTHEAD* | |
| H$TACC | 152 | ACCOUNTING WORK AREA POINTER | H.AC |
| | 154 | EXIT ACTION FLAGS | H.EAF |
| H$DLUT | 156 | NUMBER OF UNITS | H.LUT |

*Used only by handler tasks but reserved in all.

Figure C-3 (Cont.)
Task Header Fixed Part

## C.2.1 Logical Unit Table Entry

Each entry in the Logical Unit Table has the form shown in Figure C-4.

| |
|---|
| PUD POINTER |
| WINDOW BLOCK POINTER |

Figure C-4
Logical Unit Table Entry

The first word contains the address of the device physical unit directory (PUD) 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 PUD 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 and is modified by the re-assign MCR function.

The window block pointer is set when a file is opened on the device whose PUD address is specified by word 1. The window block pointer is cleared when the file is closed.

## C.2.2 Floating Point Save Area

If a task is built with the FP attribute, 25 (decimal) words are allocated in the header immediately following the Logical Unit Table. A pointer to this area is set in the fixed part of the header at offset H.CRl. The Executive uses this area to save and restore the registers of the PDP 11/45 floating point unit upon context switching.

## C.2.3 Task Accounting Work Area

If a task is built accountable (with the TA switch), 160 (decimal) words are allocated following the floating point save area. A pointer (ASR 3 relative) is stored in the fixed part of the header at offset H.AC. When the task is run in a system on which task accounting is enable, the Executive accumulates the pertinent accounting information in this area.

## C.3 LOW MEMORY POINTERS

Several locations at the beginning of a task's virtual address space are reserved for system dependent information. These locations are as follows:

| | Address (Virtual) | Usage |
|---|---|---|
| 0 | $DSW | Directive Status Word. The Executive returns the completion code in this word for every system directive issued by the task. |
| 2. | .FSRPT | File Control Services work area and buffer pointer. |
| 4. | $OTSV | FORTRAN OTS work area pointer (i.e., address of $OTSVA). |
| 6. | N.OVPT | Overlay Run Time system work area pointer. |

The last three of these locations contain addresses of work areas. These addresses are needed to provide reentrancey capability to the associated system routines when these areas are required by shareable global areas or multi-user tasks.

Note that it is possible for a task to destroy these pointers if a stack overflow occurs.

## C.4. TASK R/W ROOT SEGMENT

The low memory pointers, stack space and all R/W p-sections of the task root segment are concatenated by the Task Builder to form the R/W part of the root segment.

Following the user specified p-sections, the Task Builder allocates space for the overlay run time system data base, which is described below. Finally the R/W area is rounded-up to a 32-word boundary for APR allocation. However, if the task is defined multi-user, a further rounding is performed. This time the R/W area is rounded to a block boundary.

## C.5 TASK R-O ROOT SEGMENT

If the root segment of the task has R-O p-sections, the R-O p-sections are allocated contiguous space for mapping on a separate APR.

Since the R-O root segment is the shareable global area of a multi-user task, it is loaded separately, and only once, by the Executive no matter how many versions of the task are invoked. Consequently, the R-O root segment must be on a block boundary for a multi-user task.

## C.6 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-5. 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.

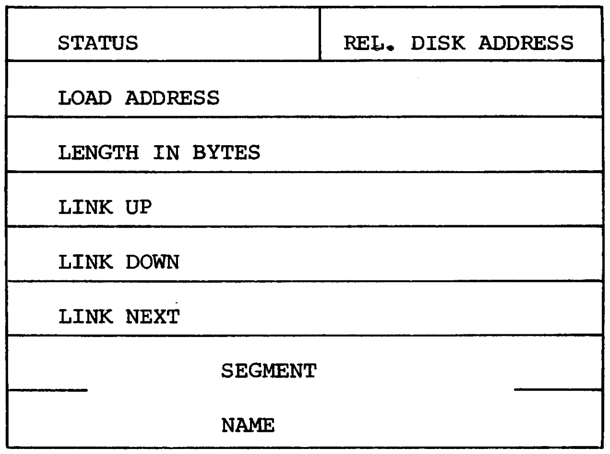


Figure C-5
Segment Descriptor

### C.6.1 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.6.2 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.6.3 Load Address

The load address contains the address into which the loading of the overlay segment starts.

## C.6.4 Segment Length

The segment length contains the length of the overlay segment in bytes and is used to construct the disk read.

## C.6.5 Link-Up

The link-up is a pointer to a segment descriptor away from the root.

## C.6.6 Link-Down

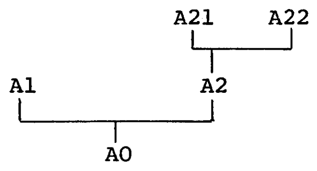
The link-down is a pointer to a segment descriptor nearer the root.

## C.6.7 Link-Next

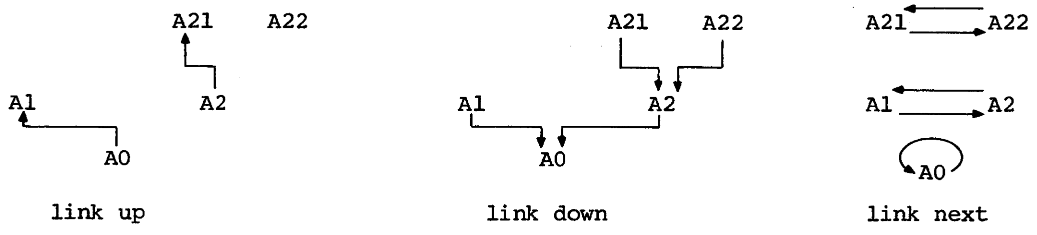
The link-next is a pointer to the adjoining segment descriptor. When a segment is loaded, the loading routine follows the link-next to determine if a segment in memory is being overlaid and should therefore be marked out-of-memory.

The link-next pointers are linked in a circular fashion:

Consider the tree:



The segment descriptors are linked in the following way:



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.7 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-6.

| |
|---|
| JSR PC |
| $AUTO |
| SEGMENT DISCRIPTOR ADDR. |
| ENTRY POINT ADDRESS |

Figure C-6
Autoload Vector Entry

## C.8 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 input device. |
| .ODTL2 | Logical unit number for the ODT output device. |
| $OTSV | The address in low memory of the FORTRAN OTS work area ($OTSVA defined by the FORTRAN OTS). |

The definition of a reserved p-section causes that p-section to be

* 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. Refer to RSX-11 MACRO-11 Reference Manual for additonal discussion of .ASECTS and .CSECT directives.

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 |
|---|---|
| $$DEVT | The extension length (in bytes) is calculated from the formula<br><br>EXT = <S.FDB+52>*UNITS<br><br>Where the definition of S.FDB is obtained from the root segment symbol table and UNITS is the number of logical units used by the task, excluding the Message Output, Overlay , and ODT units. |
| $$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

# CROSS REFERENCE TASK

The RSX-11D Task Builder is capable of providing a global cross reference file in addition to its other files. A global cross reference is an alphabetical listing of all global symbols, their values, and the names of all modules that refer to them. The global cross reference is appended to the memory allocation map of the task by the cross reference utility.

A global cross reference is requested by a Task Builder command string that includes the /CR switch with the memory allocation (MAP) file specification. The /CR switch causes the Task Builder to create two files:

1. A .MAP file that is the traditional memory allocation file,

2. A .CRF file that contains records to be processed by the cross reference task.

Upon completion of a task build operation, the Task Builder issues a SEND AND REQUEST directive to the cross reference task to provide it with the name of its input (.CRF) file.

The cross reference task processes the input file, appending its output to the .MAP file of the same name under the same UFD. When the cross reference task has completed its processing, the .CRF input file is deleted.

The cross reference task is an independent task having [11,1]CRF.TSK as a file name and CRF... as a task name. Therefore, the Task Builder or MCR reprompts at the terminal before the cross reference text has been completely appended to the memory allocation file.

If the memory allocation file specification to the Task Builder includes both a /-SP switch to inhibit spooling and the /CR switch, the user should wait until the .CRF file has been deleted (that is, until the cross reference task has exited) before queuing the memory allocation file for printing. CRF appends the cross reference information to the memory allocation file.

# 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)

The following points must be taken into consideration when using debugging aids on a task (particularly ODT):

1. Breakpoints cannot be set in R-O p-sections

2. Care must be used if settings breakpoints in overlay branches.

# APPENDIX G

# RSX-11D 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 SYMBOL - | A symbol whose definition is known outside the defining module. |
| MAIN TREE - | An overlay tree whose root segment is loaded by the Monitor when the task is made active. |
| 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. |
| 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 STRUCTURE - | A structure containing a main tree and optionally one or more co-trees. |
| OVERLAY TREE - | A tree structure consisting of a root segment and optionally one or more overlay segments. |

| | |
|---|---|
| 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 shareable global areas. |
| 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. |
| SHAREABLE GLOBAL AREA - | Code and/or data bound together by the Task Builder, in such a way that tasks can be bound to them to reference data and/or routines by symbol. Many tasks can share one copy of such areas and the area is resident only when a referencing task is active. |
| SEGMENT - | A group of modules and/or p-sections that occupy memory simultaneously and that can be loaded by a single disk access. |
| SYMBOL DEFINITION FILE - | The output file created by the Task Builder that contains the global symbol definitions and values in a format suitable for reprocessing by the Task Builder. Symbol definition files are used to link tasks to shared regions. |
| TASK IMAGE FILE - | The output file created by the Task Builder that contains the executable portion of the task. |

# APPENDIX H

# VIRTUAL SYMBOL TABLE

## H.1 ADJUSTMENT AND PLACEMENT OF VIRTUAL MEMORY

The symbol table constructed by the Task Builder is only part of a dynamic, virtual memory system. This virtual memory is partially resident and partially a disk workfile. The maximum size of this virtual memory is 65,534 words. The ratio of resident storage to workfile storage is a parameter that can be adjusted by re-building the Task Builder. This can be done in RSX-11D by altering the directive EXTSCT = FRSIZ1: ..... in the build command file of the Task Builder. Note that the extension of this memory must not cause the Task Builder to become larger than the maximum size for any task - namely 32,736. (32K-32) words.

Increasing the proportion of dynamic memory that is resident reduces the amount of I/O necessary for access to the Task Builder's internal data structures. Once the resident memory has been filled, the data structures overflow into a temporary work file on the device assigned to the workfile logical unit number. This logical unit number (W$LUN) is specified in the build command file; it may be advantageous to assign this unit number to a device other than the system device(e.g., a fixed head disk).

Displacement of pages to the workfile is done on a least recently used basis. The workfile will be automatically extended as necessary to hold all pages displaced. The parameter W$KEXT is provided in the build command file of the Task Builder and defines the file extension properties. A negative value indicates that the extension is non-contiguous; a positive value indicates a contiguous extend. If the workfile remains contiguous, a higher access rate can be obtained; however, this is advisable only when it is known that contiguous space is always available.

## H.2 CONTENT OF VIRTUAL MEMORY

It is not possible to state exactly how many symbols the Task Builder can process, because there are many data structures included in virtual memory. Following is a list of the structures that are stored in the virtual memory. All the sizes given are approximations; the size varies with the characteristics of the task being built and may vary from release to release of the operating system.

| Structure Name | Description | Approx. Size (WORDS) |
|---|---|---|
| Segment Descriptor | Contains listheads sizes, the pointers defining the overlay tree, the segment name, Part of this structure becomes the segment descriptor in the resultant task image. | 60. |
| P-section descriptor | Contains the name, address, size and attributes of a p-section. | 10. |
| Symbol Descriptor | Contains symbol name, value, flags and pointers to defining segment and p-section descriptors. | 8. |
| Element Descriptor | Contains module name, ident, filename, count of p-sections and some flags. | 8.-18. |

The maximum usage of virtual memory occurs during phase three of the Task Builder, when the symbol table is built. However, phase one makes significant use of virtual memory when an overlayed task is being built. It is at this point that all the segment descriptors are allocated and that element descriptors are allocated for each filename encountered during the parsing of the tree description. In addition, a P-section descriptor is produced for every .PSECT directive encountered in the overlay description.

The parsing of the overlay description also makes use of dynamic memory during the processing of each directive. This memory is released upon completion of the analysis, but during the analysis, the whole tree description must fit into the resident portion of storage. If sufficient storage cannot be obtained in the resident dynamic memory, the error message 'NO DYNAMIC STORAGE AVAILABLE' is produced. The method of increasing the ratio of dynamic storage to virtual memory can be applied to allow a task with a large overlay description to be built.

The amount of memory required during analysis depends on:

1. Number of directives

2. Length of .FCTR lines

3. Number of operators (i.e., commas, dashes, parentheses and asterisks).

4. Number of filenames encountered.

The resident portion of the virtual memory specified on the released version of the Task Builder is sufficient to handle the overlay description file of the Task Builder itself, as well as Filex. These overlay descriptions are suggested as guidelines for constructing complex overlay trees.

## H.3 REDUCTION OF VIRTUAL MEMORY REQUIREMENTS

There are a number of ways to reduce the amount of virtual memory required during the build of a specific task. Reduction of the data structures in virtual memory increases the speed of searching the tables and reduces the amount of paging to the workfile. The following metods will reduce virtual memory:

1. Extraction of object modules by name (i.e., LIBRY/LB:MOD1,MOD2 type constructs) from relocatable object libraries. This technique requires smaller element descriptors and fewer filename descriptors; it is faster because there are fewer files to open and close.

2. Use of concatenated object modules.

3. Use of shareable global areas (resident libraries and common areas) for language and overlay run time systems and file control services. This means that symbols and P-sections are defined once rather than on multiple branches of the tree.

4. Use of common segments. Modules that occur on parallel branches of the tree should be placed on a common (i.e., closer to root) segment for the same reasons as 3 above.

5. Use of the /SS switch on symbol table files (.STB) that describe absolute symbol definition. This means that only those symbols that are referenced will be extracted from the module.

6. Minimizing the number of segments and keeping the tree balanced. For example, if one segment is very long, there is no value in puting a tree structure in parallel unless creation of one segment in parallel would be longer.

## H.4 ERROR MESSAGES

There are four error messages associated with the virtual memory system:

1. NO DYNAMIC STORAGE AVAILABLE. This error is produced when there is insufficient resident storage for creating a data structure. As much as possible (all unlocked pages) of the data already allocated has been paged to the workfile but there is still not enough free space in the resident area. Such a situation may arise during the analysis of the overlay description, early in the task build run and particularly if it is a complex tree. The recommended recovery procedures are to reduce the ODL and extend the Task Builder memory allocation.

2. UNABLE TO OPEN WORKFILE. The probable causes of this error are:

   1. Device assigned to the logical unit W$KLUN of the Task Builder does not have a volume mounted.

   2. The volume does not support FILES-11.

   3. There is no space on the volume.

   4. The device is offline, not ready, write locked or faulty.

   5. There is no such device.

   The MCR function LUN ...TKB can be used to determine which device the Task Builder is attempting to use.

3. WORKFILE I/O ERROR. The most probable causes of this error are:

   1. Hardware error - e.g., parity errors on the disk.

   2. Device is not ready, is dismounted or write locked.

   3. An extend failure has occurred (e.g., there is insufficient contiguous space in this area of the disk when W$KEXT is positive); the disk is full. In these cases, try allocating a contiguous 256 block file and deleting it just before running the task build.

4. NO VIRTUAL MEMORY STORAGE AVAILABLE. The addressable limit of the virtual memory has been reached. There is no recovery other than to reduce the virtual memory requirements of the task being built.

INDEX

Please cut along this line.

RSX-11D
Task Builder
Reference Manual
DEC-11-OXDLA-D-D

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 require a written reply, please check here. ☐

Fold Here

Do Not Tear - Fold Here and Staple

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Communications
P. O. Box F
Maynard, Massachusetts 01754

digital

digital equipment corporation

Printed in U.S.A.