Data General

Software Documentation

Advanced Operating System (AOS) Link and Library File Editor (LFE) User's Manual

Advanced Operating System (AOS) Link and Library File Editor (LFE) User's Manual

093-000254-02

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 093-000254

Revision 02, October 1984

NOTICE

Advanced Operating System
(AOS)
Link and Library File Editor
(LFE)
User's Manual
093-000254

Revision History:	Effective with:
Original Release - December 1979	
First Revision - November 1982	
Second Revision - October 1984	AOS Rev. 4.20

This manual supersedes the *Advanced Operating System (AOS) Library File Editor User's Manual* (093-000198). A vertical bar or an asterisk in the margin of a page indicates substantive change or deletion, respectively from the previous revision.

CONTENT UNCHANGED

The content and change indicators in this revision are unchanged from 093-000254-01. This revision changes only printing and binding details.

Preface

This manual describes Link and the Library File Editor (LFE), two fundamental utilities of the Advanced Operating System (AOS).

We describe the two utilities in one manual because they play related roles in program development. Link consolidates object modules and library files into executable program files. The LFE creates, edits, and analyzes library files.

The spectrum of Link users is very broad. Some programmers require little more from this manual than the fundamental Link command line; others want to know more about Link switches, overlays, cross-linking, and partitions so that they can optimize the performance of their applications programs; some programmers want details about object block formats so that they can write compilers or assemblers that Link can handle.

We've organized the manual to comply with all these different needs:

Chapter 1	briefly describes Link's purpose and the basic Link command line. It also contains some sample programs and sample Link command lines for several AOS languages that use Link. We aimed this chapter at programmers who have never worked with Link.
Chapter 2	gives an overview of AOS and ECLIPSE® hardware. It will be particularly useful to programmers who are familiar with other operating systems and hardware, but are new to AOS and ECLIPSE hardware.
Chapter 3	discusses topics such as modular programming, absolute code and relocatable code, external symbols and entry symbols. If you want to know what a relocatable linker does, read this chapter.
Chapter 4	explains attributes, partitions, external numbers, and resource call resolution. These topics are prerequisites to understanding many of the switches in Chapter 5 and the object block formats in Appendix B.
Chapter 5	explains all Link command line options (including how to set up overlays). It also contains many figures showing how Link switches affect .PR files. All programmers will find this chapter useful.
Chapter 6	presents an overview of LFE functionality and describes several library programming strategies. It also describes all LFE function-letters. This is an essential chapter for all LFE users.
Chapter 7	explains how you can transfer object modules, libraries, or program files across operating systems. It is intended for programmers who want to use AOS Link to produce a program file that can execute on RDOS or RTOS.
Appendix A	lists all the Link error messages.
Appendix B	details object block structure. We aimed it principally at programmers writing their own compilers or assemblers.
Appendix C	lists all symbols generated by the Link utility. It is primarily for programmers writing their own compilers or assemblers, but some applications programmers will also find it helpful.

Appendix D	illustrates the structure of the AOS .SY, .PR, .ST, .OL, .DS, and .DL files and the RDOS and RTOS .SV files. Anyone programming on AOS, RDOS, or RTOS will find this appendix useful.
Appendix E	lists all the LFE error messages.
Glossary	contains definitions of many Link and LFE terms.

Related Manuals

All Link users should be familiar with AOS CLI commands. Refer to the following manuals:

- *Learning to Use Your Advanced Operating System* (069-000018)
- *Command Line Interpreter User's Manual (AOS,AOS/VS)* (093-000122)

Information on high-level language source code and on compile and Link command lines can be found in the following manuals:

- *COBOL Reference Manual (AOS)* (093-000223)
- *DG/L^{TM} Reference Manual* (093-000229)
- *FORTRAN IV User's Manual* (093-000053)
- *FORTRAN 5 Reference Manual* (093-000085)
- *FORTRAN 77 Reference Manual* (093-000162)
- *PL/I Reference Manual* 093-000204)

You can find information on language libraries in the following manuals:

- *DG/L^{TM} Runtime Library User's Manual (AOS,AOS/VS)* (093-000159)
- *FORTRAN IV Runtime Library User's Manual (ECLIPSE®)* (093-000142)
- *FORTRAN QCALLs Reference Manual (AOS)* (093-000239)

Assembly language programming is described in three books:

- *AOS Macroassembler Reference Manual* (093-000192) — details AOS assembly language pseudo-ops. Pseudo-ops are special source code directives to Link or the macroassembler.
- *AOS Programmer's Manual* (093-000120) — documents the AOS system calls and system parameters. System calls are the user program's interface to the operating system.
- appropriate manual in the Programmer's Reference Series for ECLIPSE-Line Computers — describes all ECLIPSE assembly language instructions. This manual also describes concepts specific to your ECLIPSE hardware.

If you intend to use AOS Link to create program files executable under RDOS or RTOS, you will find the following manuals useful:

- *Real-time Disk Operating System (RDOS) Reference Manual* (093-000075)
- *Real-Time Operating System (RTOS) Reference Manual* (093-000135)
- *RDOS/DOS Macroassembler User's Manual* (093-000081)
- *DG/L^{TM} Runtime Library User's Manual* (RDOS) (093-000124)
- *FORTRAN IV Runtime Library User's Manual* (NOVA®) (093-000068)
- *Library File Editor (LFE) User's Manual* (093-000074)
- *RDOS/DOS Command Line Interpreter User's Manual* (093-000109)

If you intend to run your RDOS or RTOS .SV files on a NOVA® computer, you can refer to the appropriate manual in the Programmer's Reference Series for NOVA-Line Computers.

Reader, Please Note:

We use these conventions for command formats in this manual:

COMMAND required *<optional>* ...

Where	Means
COMMAND	You must enter the command (or its accepted abbreviation) as shown.
required	You must enter some argument (such as a filename). Sometimes, we use: $\left\{\begin{matrix}\text{required}_1\\ \text{required}_2\end{matrix}\right\}$ which means you must enter *one* of the arguments. Don't enter the braces; they only set off the choice.
<optional>	You have the option of entering this argument. Don't enter the brackets; they only set off what's optional.
...	You may repeat the preceding entry or entries. The explanation will tell you exactly what you may repeat.

Additionally, we use certain symbols in special ways:

Symbol	Means
⏎	Press the NEW LINE or carriage return (CR) key on your terminal's keyboard.
□	Be sure to put a space here. (We use this only when we must; normally, you can see where to put spaces.)

In the error message listings, we use angle brackets < > to indicate the paraphrase of a value or symbol. For example, the LFE error message SWITCH NOT FOUND, <switch> means the utility will supply the actual switch string.

All numbers are decimal unless we indicate otherwise; e.g., 35_8.

Finally, in examples we use

THIS TYPEFACE TO SHOW YOUR ENTRY⏎
THIS TYPEFACE FOR SYSTEM QUERIES AND RESPONSES.

) is the CLI prompt.

&) is the AOS CLI prompt for line continuations.

Contacting Data General

- If you have comments on this manual, please use the prepaid Remarks Form that appears after the Index. We want to know what you like and dislike about this manual.
- If you need additional manuals, please use the enclosed TIPS order form (USA only) or contact your Data General sales representative.
- If you experience software problems, please notify Data General Systems Engineering.

End of Preface

Contents

Tables

Illustrations

Chapter 1
Link in Brief

This chapter is for new Link users. If you already have some experience with Link, we suggest that you refer to one of the other chapters in this manual. In particular, Chapter 5 offers a much more complete discussion of the Link command line. If you are unfamiliar with relocatable linkers, you might want to read Chapter 3, "Overview of Relocatable Linkers".

Link's job is to take assembled or compiled code and create a program file — a file that the operating system and hardware can execute. Program files that can run on AOS are called .PR files. Creating a program file is a three-step process:

1. Coding — You write instructions in a language that a particular compiler or assembler can read.
2. Compiling or Assembling — A language processor (either a compiler or an assembler) translates your code into an object module. An object module is a file, ordinarily with the file name extension .OB, consisting of two or more object blocks. Each object block consists of three or more 16-bit words arranged in a format that Link can use. (Appendix B details object block structure.) Basically, an object module contains both a program recipe and a list of program ingredients.
3. Linking — Link gathers object modules and creates an executable program file (.PR file). Basically, Link is like a chef that blends the program ingredients into a usable form by reading the program recipe.

The Fundamental Link Command Line

You can link any number of object files with the following fundamental Link command line:

```
) X LINK objectfile1 <objectfile2>... ↵
```

It is possible, however, to alter this command line with switches and overlay area delimiters (explained in Chapter 5). An object file must be the name of an .OB file or library file.

Linking Programs

The following Data General Corporation AOS languages use Link:

- FORTRAN 77
- FORTRAN 5
- FORTRAN IV
- PL/I
- COBOL
- DG/L^{TM} language
- MASM (AOS assembly language)

Although command lines for different languages differ, the process for creating a .PR file does not. In all languages you use the following procedure to create a .PR file:

1. Write one or more files of source code.
2. Separately compile (or assemble) each file of source code to create one or more .OB files.
3. Link the .OB files and the appropriate system library and language libraries to create one executable .PR file.

For many languages, the Link command line is bundled into a DGC-supplied macro. It is important to realize that this macro consists of the fundamental Link command line plus switches and object files appropriate to the language. So, even though the FORTRAN 5 and PL/I Link command lines are different, they both invoke the same utility.

In the following section, we discuss the compile and Link command lines for seven AOS languages. We also include a sample program in six of these languages.

FORTRAN 77

If you are programming in FORTRAN 77, you create a .PR file with the following procedure:

1. Write one or more files of source code.
2. Separately compile each file with the following command:

   ```
   ) F77 file ↵
   ```

   Each time you compile, the compiler creates one .OB file.
3. Link each .OB file with the Link macro:

   ```
   ) F77LINK .OBfile <.OB file>... ↵
   ```

   Link creates one .PR file.
4. If your program is not working properly, you may wish to debug it with the SWAT™ debugger. Then, you may wish to edit one or more of the files of source code, recompile the faulty source code file(s), and relink.

For instance, consider the three subprograms shown in Figure 1-1.

```
C   This is the MAIN program.  It is stored in file MAIN.F77
      PROGRAM MAIN
      IMPLICIT INTEGER (A-Z)
      PRINT *, "Enter today's high temperature. "
      READ (*,*) HIGH
      PRINT *, "Enter today's low temperature.  "
      READ (*,*) LOW

      Y = MEAN(HIGH,LOW)
      PRINT *, "Today's mean temperature is ", Y
C   MEAN(HIGH,LOW) calls out the function MEAN.

      PRINT *, "The deviation from normal is ", DEV(Y)
C   DEV(Y) calls out the function DEV.

      END
-------------------------------------------------------------------
C    This function averages the high and low.
C    It is stored in file AVRG.F77

      FUNCTION MEAN(HIGH2,LOW2)
      IMPLICIT INTEGER (A-Z)
      MEAN = (HIGH2 + LOW2) / 2  !calcuate the mean.
      RETURN                     !Return to calling subprogram
      END
-------------------------------------------------------------------
C     This function figures out the deviation from the normal temp (73)
C     It is stored in file DIFF.F77
        FUNCTION DEV(MEAN2)
        IMPLICIT INTEGER (A-Z)
        DEV = MEAN2 - 73    !Calculate the deviation from normal temp.
        RETURN              !Return to calling subprogram
        END
```

Figure 1-1. Three FORTRAN 77 Subprograms

To execute the program in Figure 1-1, you must first separately compile all three files of source code:

```
) F77 MAIN ↲

) F77 AVRG ↲

) F77 DIFF ↲
```

If there are no compilation errors, you can link the three object modules:

```
) F77LINK MAIN AVRG DIFF ↲
```

If there are no Link errors, you can execute program MAIN.PR:

```
) X MAIN ↲

Enter today's high temperature. 81 ↲

Enter today's low temperature. 61 ↲

Today's mean temperature is 71
The deviation from normal is -2
```

FORTRAN 5

If you are programming in FORTRAN 5, you create a .PR file with the following procedure:

1. Write one or more files of source code.
2. Separately compile each file with the following command:

   ```
   ) F5 file ↵
   ```

   Each time you compile, the compiler creates one .OB file.
3. Link each .OB file with the Link macro:

   ```
   ) F5LD .OBfile <.OBfile>... ↵
   ```

   Link creates one .PR file.
4. If your program is not working properly you may wish to edit one or more of the files of source code, recompile the faulty source code file(s), and relink.

For instance, consider the three subprograms shown in Figure 1-2.

```
C   This is the MAIN subprogram.  It is stored in file MAIN.FR
      IMPLICIT INTEGER (A-Z)
      EXTERNAL MEAN,DEV

      ACCEPT "Enter today's high temperature. ",HIGH
      ACCEPT "Enter today's low temperature.  ",LOW

      Y = MEAN(HIGH,LOW)
C   MEAN(HIGH,LOW) calls out the function MEAN.
      TYPE "Today's mean temperature is ",Y

      TYPE "The deviation from normal is ",DEV(Y)
C   DEV(Y) calls out the function DEV.

      END
----------------------------------------------------------------------
C    This function averages the high and low.
C    It is stored in file AVRG.FR

      INTEGER FUNCTION MEAN(HIGH2,LOW2)
      IMPLICIT INTEGER (A-Z)
      MEAN = (HIGH2 + LOW2) / 2
      RETURN
C    Return to the calling procedure

      END
----------------------------------------------------------------------
C     This function figures out the deviation from the normal temp.
C     It is stored in file DIFF.FR

        INTEGER FUNCTION DEV(MEAN2)
        IMPLICIT INTEGER (A-Z)
        DEV = MEAN2 - 73
        RETURN
C     Return to the calling procedure

        END
```

Figure 1-2. Three FORTRAN 5 Subprograms

To create an executable program file from the three subprograms shown in Figure 1-2, you must first separately compile all three files as follows:

```
) F5 MAIN ↵
) F5 AVRG ↵
) F5 DIFF ↵
```

If there are no compilation errors, you can link the three object modules together with the Link macro F5LD as shown below:

```
) F5LD MAIN AVRG DIFF ↵
```

If there are no Link errors, you can execute program file MAIN.PR with the following command line:

```
) X MAIN ↵

Enter today's high temperature. 81 ↵
Enter today's low temperature. 61 ↵
Today's mean temperature is   71
The deviation from normal is   -2
```

FORTRAN IV

If you are programming in FORTRAN IV, you create a .PR file with the following procedure:

1. Write one or more files of source code.
2. Separately compile each file with the following command:

   ```
   ) FORT4 file ↵
   ```

3. Link each .OB file with the command line:

   ```
   ) X LINK .OBfile <.OBfile>... FORT4.LB ↵
   ```

   where FORT4.LB is the name of the FORTRAN IV library. Link creates one .PR file.
4. If your program is not working properly, you may wish to edit one or more subprograms, recompile the faulty subprogram(s), and relink.

For more details, see the *FORTRAN IV User's Manual.*

PL/I

If you are programming in PL/I, you create a .PR file with the following procedure:

1. Write one or more files of source code.
2. Separately compile each file with the following command:

   ```
   ) PL1 file ↵
   ```

   Each time you compile, the compiler creates one .OB file.
3. Link each .OB file with the Link macro:

   ```
   ) PL1LINK .OBfile <.OB file2>... ↵
   ```

   Link creates one .PR file.
4. If your program is not working properly, you may wish to debug it with the SWAT debugger. Then, you may wish to edit one or more of the files of source code, recompile the faulty source code file(s) and relink.

For instance, consider the three subprograms shown in Figure 1-3.

```
/* This is the main procedure.  It is stored in file MAIN.PL1. */

main:   PROCEDURE;

/* variable declaration section */

DECLARE (high,low,mean_result,dev_result,y)     FIXED BINARY(15);
DECLARE (crt,kbd)                              FILE;
DECLARE (mean)  ENTRY(FIXED BINARY(15),FIXED BINARY(15))RETURNS(FIXED BINARY(15));
DECLARE (dev)   ENTRY(FIXED BINARY(15))RETURNS(FIXED BINARY(15));

/* end of variable declaration section */


        OPEN FILE(crt) STREAM OUTPUT PRINT TITLE("@OUTPUT");
        OPEN FILE(kbd) STREAM INPUT TITLE("@INPUT");

        PUT FILE(crt) SKIP LIST("Enter today's high temperature.");
        GET FILE(kbd) LIST(high);

        PUT FILE(crt) SKIP LIST("Enter today's low temperature. ");
        GET FILE(kbd) LIST(low);

        y = mean(high,low);
        PUT FILE(crt) SKIP LIST("Today's mean temperature is",y);

        PUT FILE(crt) SKIP LIST("The deviation from normal is",dev(y));

END; /* main procedure */
----------------------------------------------------------------------
/* This function averages the high and low.  It is stored in file AVRG.PL1 */

        mean:   PROCEDURE(high2,low2)RETURNS(FIXED BINARY(15));
                DECLARE (high2,low2)    FIXED BINARY(15);


                RETURN( DIVIDE(high2+low2,2,15) );
                /* Calculate the mean and return to the calling program */

                END;
----------------------------------------------------------------------
/* This function calculates the deviation from normal temperature. */
/* It is stored in file DIFF.PL1.  */

        dev:    PROCEDURE(mean2) RETURNS (FIXED BINARY(15));
                DECLARE mean2   FIXED BINARY(15);

                RETURN(mean2 - 73);
/* calculate the deviation from normal (73), and return to the calling procedure */

                END;
```

Figure 1-3. Three PL/I Subprograms

To execute the program in Figure 1-3, you must first separately compile all three files of source code as follows:

```
) PL1 MAIN ↵
) PL1 AVRG ↵
) PL1 DIFF ↵
```

If there are no compilation errors, you can link the three object modules with the following command line:

```
) PL1LINK MAIN AVRG DIFF ↵
```

If there are no Link errors, you can execute program MAIN.PR:

```
) X MAIN ↵
Enter today's high temperature. 81 ↵
Enter today's low temperature. 61 ↵
Today's mean temperature is 71
The deviation from normal is -2
```

DG/L Language

Programmers frequently use DG/L language for cross-development. You can write DG/L source code under RDOS, AOS, or AOS/VS. When compiling, you decide what hardware (NOVA®, ECLIPSE® or MV/Family) the program file will execute on; when linking, you decide which operating system (RDOS, RTOS, AOS, or AOS/VS) the program file will use. For complete details on the appropriate DG/L compile command line see the *DG/L™ Reference Manual.* For complete details on the DG/L cross-Link command line (that is, using AOS Link to create an executable RDOS or RTOS program file), see Chapter 7 of this manual.

You must use ECLIPSE hardware to compile DG/L source code. Use the following procedure to create a program file that will run on ECLIPSE hardware and AOS:

1. Write one or more files of source code.
2. Separately compile each file of source code with the following command:

   ```
   ) X DGL file ↵
   ```

3. Link each .OB file with the Link command line:

) X LINK/NSLS .OBfile <*.OBfile*>... [DGLIB]

   [DGLIB] is an indirect file which contains the names of DG/L libraries appropriate for AOS.

4. If your program is not working properly, you should edit one or more of the files of source code, recompile the faulty source code files and relink.

For instance, consider the three subprograms shown in Figure 1-4.

```
/* This is the main subprogram.   */
/* It is stored in file MAIN      */

BEGIN
     INTEGER   high,low,y;

     EXTERNAL INTEGER PROCEDURE mean;
     EXTERNAL INTEGER PROCEDURE dev;
     EXTERNAL STRING  PROCEDURE GETCINPUT, GETCOUTPUT;
     OPEN (1, (GETCINPUT));
     OPEN (2, (GETCOUTPUT));

     WRITE  (2,"Enter today's high temperature. ");
     READ   (1,high);

     WRITE  (2,"Enter today's low temperature. ");
     READ   (1,low);

     y := mean(high,low);
     WRITE  (2,"<NL>The mean temperature for today is ",y);

     WRITE  (2,"<NL>The deviation from normal is ",dev(y));

END;
----------------------------------------------------------------
/* This is the function that calculates the mean  */
/* It is stored in file AVRG                      */

INTEGER PROCEDURE mean(high2,low2);
       INTEGER high2,low2;
BEGIN

       mean := (high2+low2)/2;
/* Calculate the mean and return to the calling procedure */

END;
----------------------------------------------------------------
/* This is the function that calculates the deviation from normal. */
/* It is stored in file DIFF                                        */

INTEGER PROCEDURE dev(mean2);
INTEGER  mean2;

BEGIN

   dev :=  mean2 - 73;
/* Calculate the deviation from normal and return to the calling procedure */

END;
```

Figure 1-4. Three DG/L Subprograms

To create an executable program file from the three subprograms shown in Figure 1-4, you must first separately compile all three files as follows:

```
) X DGL MAIN ↲
) X DGL AVRG ↲
) X DGL DIFF ↲
```

If there are no compilation errors, you can link the three object modules together with the following Link command line:

```
) X LINK/NSLS MAIN AVRG DIFF [DGLIB] ⤶
```

If there are no Link errors, you can execute program file MAIN.PR with the following command line:

```
) X MAIN ⤶

Enter today's high temperature. 81 ⤶
Enter today's low temperature. 61 ⤶

Today's mean temperature is 71
The deviation from normal is -2
```

COBOL

If you are programming in COBOL, you create a .PR file with the following procedure:

1. Write one or more files of source code.
2. Separately compile each file with the following command:

   ```
   ) COBOL file ⤶
   ```

   Each time you compile, the compiler creates one .OB file.
3. Link each .OB file (see discussion below). Link creates one .PR file.
4. If your program is not working properly, you may wish to edit one or more source code files, recompile the faulty source code file(s), and relink.

CBIND and Link

The utility you use to link COBOL-generated .OB files depends on the COBOL revision you used to compile your source code. When you compile source code, the COBOL revision number appears in the output listing.

If you get a COBOL revision number less than 3.20, you link with a utility called CBIND. (Refer to the *COBOL Reference Manual* for details on this utility.)

If you get a COBOL revision number of 3.20 or higher, you link with the Link utility. There are two ways to invoke Link.

First, you can issue a CBIND command line. A program called CBIND.PR will automatically convert your CBIND command line into the equivalent Link command line. Then, CBIND.PR will issue the Link command line for you. In other words, if you are comfortable with CBIND, or if you have macros that invoke CBIND, you do not have to change a thing.

Second, you can issue a Link command line. The safest way to do this is to first issue a CBIND command line containing the /Q switch. CBIND.PR will convert your CBIND command line into a Link command line and store it in a .CK file; however, the /Q switch prevents CBIND.PR from issuing the Link command line. You can invoke Link by typing in the name of the .CK file, or you can edit the .CK file and then issue it. For instance, if you want to generate a Link command line from .OB files ONE.OB and TWO.OB, you would issue the following command line:

```
) CBIND/Q ONE TWO ⤶
```

CBIND.PR writes a Link command line into file ONE.CK. You can issue this Link command line straight out, or you can edit it to produce a new Link command line.

NOTE: The .CK file contains vital switches and libraries. You may create a nonexecutable file by removing the wrong element from the Link command line.

Sample COBOL Subprograms

Consider the sample COBOL program in Figure 1-5.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.     MAIN.
*This is the main program.  It is stored in file MAIN
***********************************************************************
ENVIRONMENT DIVISION.
***********************************************************************
DATA DIVISION.
WORKING-STORAGE SECTION.
01     high                     PIC 9(3).
01     low                      PIC 9(2).
***********************************************************************
PROCEDURE DIVISION.

   DISPLAY "Enter today's high temperature. " WITH NO ADVANCING.
   ACCEPT  high.
   DISPLAY "Enter today's low temperature.  " WITH NO ADVANCING.
   ACCEPT low.

   CALL "MEAN" USING high,low.
**** GO TO SUBPROGRAM MEAN. ***********

STOP RUN.


----------------------------------------------------------------------


IDENTIFICATION DIVISION.
PROGRAM-ID.     MEAN.
*This is the subprogram that calculates the mean.
*It is stored in file AVRG
***********************************************************************
ENVIRONMENT DIVISION.
***********************************************************************
DATA DIVISION.
WORKING-STORAGE SECTION.
01     mean3                    PIC 9(2).

LINKAGE SECTION.
01     high2                    PIC 9(3).
01     low2                     PIC 9(2).
***********************************************************************
PROCEDURE DIVISION USING high2,low2.

   COMPUTE mean3 = (high2 + low2) / 2.
   DISPLAY "Today's mean temperature is " mean3.
   CALL "DEV" USING mean3.
****** GO TO SUBPROGRAM DEV *******

STOP RUN.


----------------------------------------------------------------------


IDENTIFICATION DIVISION.
PROGRAM-ID.     DEV.
* This is the subprogram that calculates the deviation from normal (73).
* It is stored in file DIFF
```

Figure 1-5. Three COBOL Subprograms (continues)

```
**************************************************************************
ENVIRONMENT DIVISION.
**************************************************************************
DATA DIVISION.
WORKING-STORAGE SECTION.
01     dev2              PIC IS  S9(2) SIGN LEADING SEPARATE.

LINKAGE SECTION.
01     mean4                     PIC   9(2).
**************************************************************************
PROCEDURE DIVISION USING mean4.

   COMPUTE dev2 = mean4 - 73.
   DISPLAY "The deviation from normal is " dev2.

STOP RUN.
```

Figure 1-5. Three COBOL Subprograms (concluded)

To create an executable program file from the three subprograms shown in Figure 1-5, you must first separately compile all three files as follows:

```
) COBOL MAIN ↵
) COBOL AVRG ↵
) COBOL DIFF ↵
```

If there are no compilation errors, you can link the three object modules with the Link macro CBIND as shown below:

```
) CBIND MAIN AVRG DIFF ↵
```

If there are no Link errors, you can execute program file MAIN.PR with the following command line:

```
) X MAIN ↵
```

Enter today's high temperature. 81 ↵
Enter today's low temperature. 61 ↵
Today's mean temperature is 71
The deviation from normal is -02

MASM (Assembly Language)

If you are programming in AOS Assembly Language, you create a .PR file with the following procedure:

1. Write one or more files of source code.
2. Separately assemble each file with the following command:

) X MASM file ↵

   Each time you assemble, the assembler creates one .OB file.
3. Link each .OB file with the Link command line:

) X LINK .OBfile <*.OBfile*>... ↵

   Link creates one .PR file.
4. If your program is not working properly, you may wish to debug it with the AOS Debugger. Then, you may wish to edit one or more of the source code files, reassemble the faulty source code file(s) and relink.

For instance, consider the three subprograms shown in Figure 1-6.

```
;The following source code is stored in file MAIN

.TITLE MAIN            ;The title of this object module will be MAIN.
.EXTN  DEV  MEAN       ;DEV and MEAN are labels from other object
                       ;modules that this program will access.
                       ;MASM does not resolve externals; Link does.
                       ;Link will search through the other object
                       ;modules in the command line for entry symbols
                       ;DEV and MEAN.  (The pseudo-ops .ENT, .PENT,
                       ;.ENTO, .COMM, and .CSIZ define entry symbols.)
.ENT NORM HOME         ;.ENT serves two purposes.  First, if any
                       ;external symbols defined in other object
                       ;modules are trying to find NORM and HOME, then
                       ;Link will know that they are defined in this
.ENT RECLI HOLD1 HOLD2 ;object module.  Second, this pseudo-op allows
                       ;you to mark labels that you may want to use
                       ;during debugging.

.ZREL                  ;The following datawords will go into the ZREL
                       ;partition.
HIGH: 81.              ;81(base 10) -- Today's high temperature.
LOW:  61.              ;61(base 10) -- Today's low temperature.
NORM: 73.              ;73(base 10) -- Today's normal temperature.

.NREL 0                ;The following datawords will go into the
                       ;predefined Unshared Code partition.
MAIN:   LDA 0,HIGH     ;Load high temp into AC0.
        LDA 1,LOW      ;Load low temp into AC1.
        EJSR MEAN      ;Jump to the subroutine that starts with "MEAN".
                       ;Load return address into accumulator 3.
                       ;NOTE: The values of AC0, AC1, and AC2 do not
                       ;change during the jump.
        STA 1,HOLD1    ;Store the average at location HOLD1.
        EJMP DEV       ;Jump to the subroutine that starts with "DEV".
HOME:   STA 1,HOLD2    ;Store the deviation from normal at HOLD2.
RECLI:  SUB 2,2        ;Clear accumulator 2 for a "good" ?RETURN.
        ?RETURN        ;Halt execution and return to the CLI.
        JMP RECLI      ;If the ?RETURN call did not work then
                       ;jump back to RECLI, clear accumulator 2,
                       ;and try again.
HOLD1: 0               ;The mean is stored here.
HOLD2: 0               ;The deviation from normal is stored here.
.END MAIN              ;The instruction at the label MAIN is a
                       ;possible start address.

-------------------------------------------------------------------
;The following source code is stored in file AVRG

.TITLE AVRG            ;The title of this object module will be AVRG.
.ENT MEAN              ;Defines entry symbol MEAN.
.NREL 0                ;The following datawords will go into the
                       ;predefined Unshared Code partititon.
MEAN:   PSH 3,3        ;Push the return address onto the stack.
        ADD 0,1        ;Add the values of HIGH and LOW together.
        HLV 1          ;Divide the sum by 2, keep result in AC1
        POPJ           ;Return to calling subprogram.
.END                   ;This subprogram does not define a possible
                       ;start address.
```

Figure 1-6. Three AOS Assembly Language Subprograms (continues)

```
;The following source code is stored in file DIFF

.TITLE DIFF          ;The title of this object module will be DIFF.
.ENT DEV             ;Define entry symbol DEV.
.EXTN HOME           ;HOME is located in some part of NREL that can
                     ;only be accessed with a 32-bit instruction.
.EXTD  NORM          ;Since NORM is in ZREL, it can be accessed with
                     ;a 16-bit instruction.
.NREL 1              ;The following datawords will go into the
                     ;predefined Shared Code partition.
DEV:    LDA 2,NORM   ;Load today's normal temperature into ACC2.
        SUB 2,1      ;Today's mean - normal temp. = deviation from norm.
        EJMP HOME    ;Jump to the instruction at label HOME in
                     ;subprogram MAIN.
.END                 ;This object module does not define a possible
                     ;start address.
```

Figure 1-6. Three AOS Assembly Language Subprograms (concluded)

To execute the code in Figure 1-6, you must first separately assemble all three source code files:

```
) X MASM MAIN ⤶
) X MASM AVRG ⤶
) X MASM DIFF ⤶
```

Next, if there are no assembly errors, link the three object modules together.

```
) X LINK MAIN AVRG DIFF ⤶
```

Finally, if there are no Link errors, you can execute program MAIN.PR.

```
) X MAIN ⤶
```

When you execute the program, a blank line and the AOS prompt will appear on your console to indicate program completion.

Because this program did not contain any I/O system calls, you can find the mean and deviation from normal only by entering the AOS Debugger. See the *AOS Debugger and Disk File Editor User's Manual* for details on debugging.

What to Do if You Get Link Errors

Appendix A offers remedies for some Link errors. If you get errors while linking AOS MASM-generated object modules, you may find Figure 1-6 helpful. If you get errors while linking object modules generated by high-level language compilers, you should probably refer to the reference or runtime manual for the language.

End of Chapter

Chapter 2
ECLIPSE Hardware Concepts and AOS

This chapter summarizes those parts of ECLIPSE hardware and AOS software that are pertinent to Link. We begin by defining some important AOS and ECLIPSE hardware terms. Then, we summarize system calls, hardware index-modes, system tables, reserved storage locations, and the stack. Familiarity with these topics will help you design more efficient applications programs. Finally, if you have no experience with overlays, we suggest that you read the "Overlays" section at the end of this chapter.

This chapter will give you an idea of the tools available on the AOS/ECLIPSE configuration; Chapters 4 and 5 explain how you can manipulate them.

For more information on ECLIPSE hardware, see the appropriate manual in the Programmer's Reference Series for ECLIPSE-Line Computers. For more information on the Advanced Operating System refer to the *AOS Programmer's Manual.*

Figure 2-1 illustrates the hierarchical scheme of program development on the AOS/ECLIPSE configuration. Programming languages are at the top of the hierarchy. In order for a language processor (a compiler or assembler) to work properly, source code must adhere to strict rules. Likewise, Link creates an executable program file only if the language processor generated valid object blocks. Furthermore, AOS can properly manipulate a program file only if Link has built the proper operating system components into it. Finally, at the base of the hierarchy, ECLIPSE hardware executes the program only with the preparation of AOS.

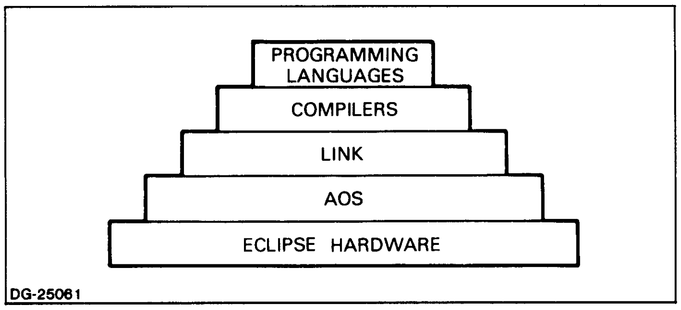


Figure 2-1. Hierarchy of Program Development

ECLIPSE Hardware and AOS Terms

The following definitions will be useful to programmers unfamiliar with 16-bit ECLIPSE hardware or AOS software.

Word	A set of 16 contiguous bits. All 16-bit ECLIPSE hardware instructions are one or two words long.
Block	A set of 400_8 or 256_{10} contiguous words.
Page	A set of four contiguous blocks. 1 page $= 1024_{10}$ words $= 2000_8$ words.
Disk	The device that AOS most frequently uses for mass file storage and retrieval.

File	A named set of logically associated blocks which may be read or modified by AOS system calls. Files in AOS are accessible by name and addressable by block, record, or byte. Link uses only disk files. That is, Link can read in only disk files and can write its output files only to disk.
.OB File	A file, usually with the extension .OB, that a language processor creates from source code. Each .OB file contains one object module. An object module consists of two or more contiguous object blocks. These object blocks must have the characteristics detailed in Appendix B.
library	A file, usually constructed by the Library File Editor, containing one or more object modules.
.PR File	An executable file that Link builds from one or more .OB files and libraries. .PR files must have the characteristics described in Appendix D. (.PR files are also called program files.)
Physical Memory	Memory stored within the ECLIPSE computer. AOS allocates physical memory in pages. In general, physical memory holds far less than disk memory.
Swapping	Copying program files from disk to physical memory or from physical memory to disk. AOS performs all swapping.
Process	The executable physical memory copy of a .PR file. For a more comprehensive definition and discussion, refer to the *AOS Programmer's Manual.*
Logical Address Space	The subset of physical memory currently occupied by an executing process. On a 16-bit ECLIPSE computer, logical address space is 100000_8 words or 32_{10} pages.
Mapping	Translating a logical address to a physical memory address. ECLIPSE computers perform mapping on a page basis: each of the 32_{10} pages in a logical address space has a physical page start address known to the mapping hardware, and all references to a given logical address are directed to the correct page without CPU intervention.

How AOS and ECLIPSE Hardware Make a Program Run

AOS begins program execution by swapping a .PR file into physical memory. Then AOS maps this process into and out of logical address space until the CPU finishes executing it. Because AOS is a time-sharing system, users compete for physical memory and logical address space; however, the competition is invisible to the user. Nevertheless, with proper use of shared and unshared pages, you can reduce system overhead in a multiuser environment, and therefore speed program execution.

Shared and Unshared Pages

Physical memory pages belong to one of two categories:

• Shared page	A page that several users of the same program file can access.
• Unshared page	A page that only one user can access.

For example, suppose that three users wanted to execute a program called KINGS.PR which occupies 12 pages in memory — 10 shared pages and 2 unshared pages. Figure 2-2 shows that since all three users can access the shared pages, AOS needs to swap in only one copy of the shared pages. Since all three users need their own copies of unshared pages, AOS swaps six unshared pages (two unshared pages each). (In reality, the pages in Figure 2-2 would not be so neatly grouped in physical memory. We did that only for clarity.)

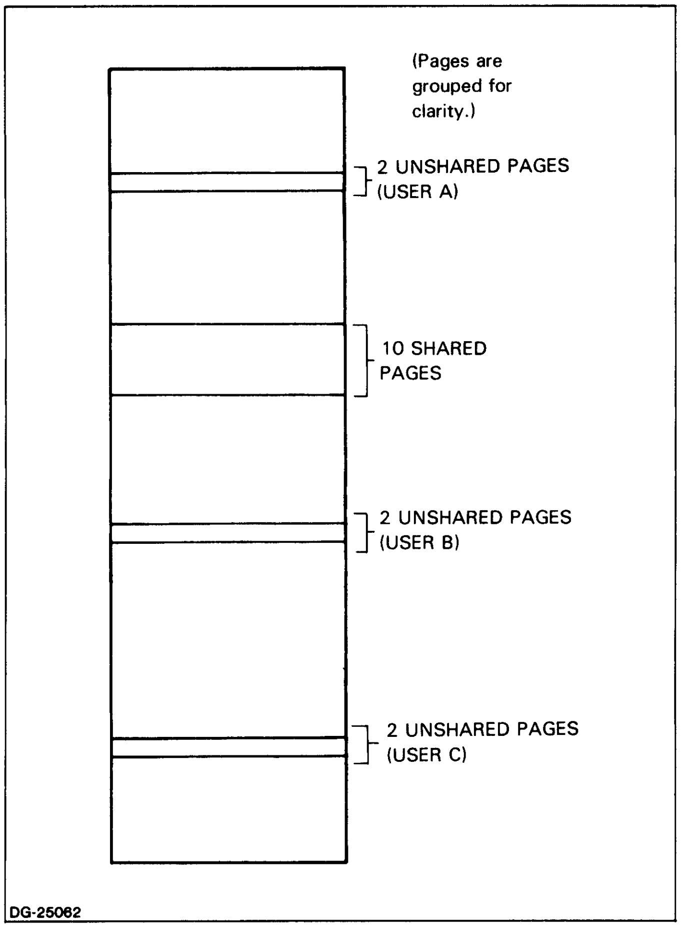

(Pages are grouped for clarity.)
2 UNSHARED PAGES (USER A)
10 SHARED PAGES
2 UNSHARED PAGES (USER B)
2 UNSHARED PAGES (USER C)
DG-25062

Figure 2-2. Physical Memory During Execution of KINGS.PR

Shared pages reduce the amount of swapping. If KINGS.PR contained 12 unshared pages (instead of ten shared and two unshared), AOS would be forced to swap in a total of 36 pages (12 pages each for all three users) instead of 16 pages.

Besides swapping, there is one other difference between shared and unshared pages:

- Shared pages are usually write protected. (Shared pages generated by Link are always write protected; however, some AOS system calls allow you to generate shared pages that are not write protected. See the *AOS Programmer's Manual* for more information on system calls.)
- Unshared pages are not write protected.

Write-protection means that the CPU will not perform any instruction that changes the contents of an address in a shared page. When the CPU detects such a change, it traps the instruction, halts program execution, and sends out an error.

If shared pages were not write-protected, then two users executing the same process and using the same input variables might get different results. For example, suppose that users JAN and JACK both want to execute a program at about the same time. Assume that the program, ALIKE.PR, consists of one unshared page and one shared page. In this case, AOS swaps in 1 shared page and 2 unshared pages (1 unshared page for each user). Suppose JAN's process gains logical address space before JACK's process. Figure 2-3 shows that unshared address 01000 contains the machine language equivalent of EDSZ 76000 — decrement (and skip on a zero result) the contents of address 76000. Initially, shared address 76000 contains "000005". If shared pages were not write-protected, the CPU would have decremented the contents of address 76000 to "000004". Now, when AOS maps JACK's process into logical address space, logical address 76000 will initially contain "000004" rather than "000005" as shown in Figure 2-4. Therefore, JACK will probably get different results than JAN.

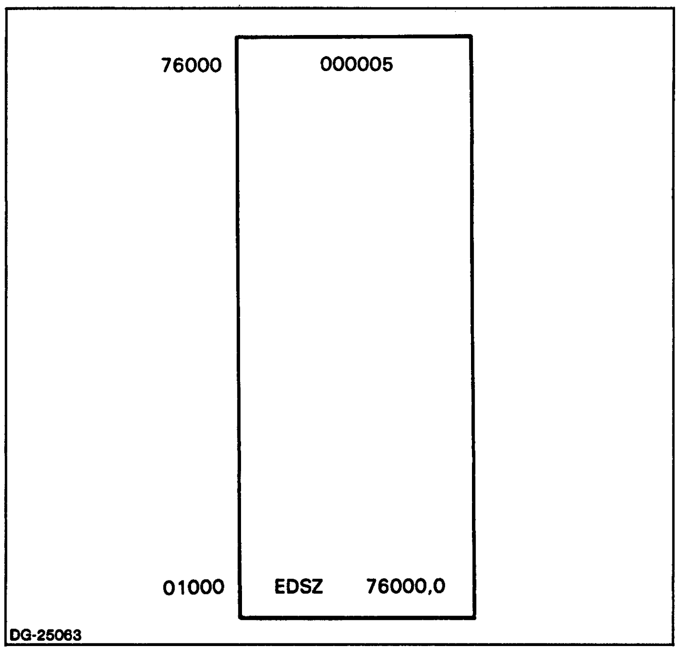


Figure 2-3. JAN's Logical Address Space Before Program Execution

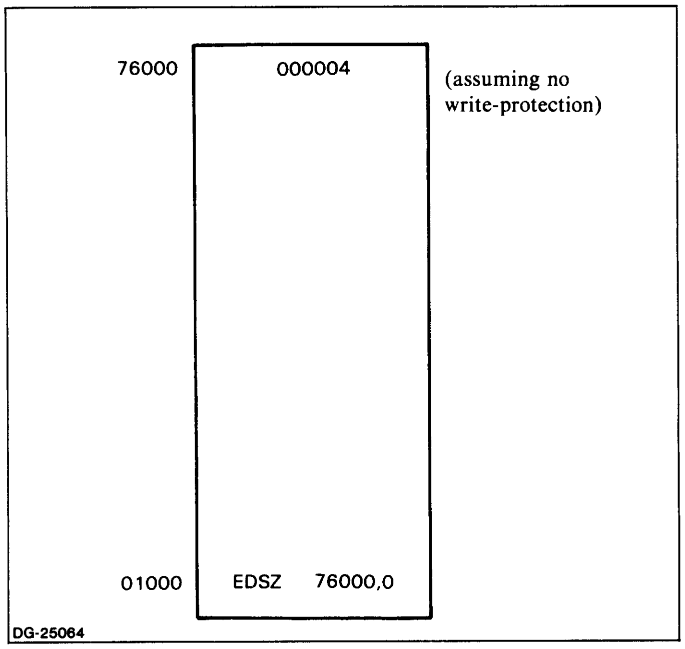


Figure 2-4. JACK's Logical Address Space Before Program Execution

To summarize, shared pages reduce system overhead, but since they are usually write protected, your program cannot attempt to change the contents of any shared address. Consider a text editing program that many people use simultaneously. The shared pages probably handle some global text editing commands; the instructions for decoding these commands are identical from one user to the next. The unshared pages could be used as an input storage buffer; the text that you enter will be different from the text that another user enters.

Logical Address Space

The logical address space of a 16-bit ECLIPSE computer consists of 32_{10} pages which is equivalent to 100000_8 logical addresses. These logical addresses are numbered from 00000_8 to 77777_8.

AOS and Link assume that unshared pages start at the lowest addresses in logical address space and "grow up"; while shared pages start at the highest addresses and "grow down." Between shared and unshared pages is a region called the unused area. Figure 2-5 shows the logical address space of a process containing 2 unshared pages, 10_{10} shared pages, and an unused area equivalent to 20_{10} pages.

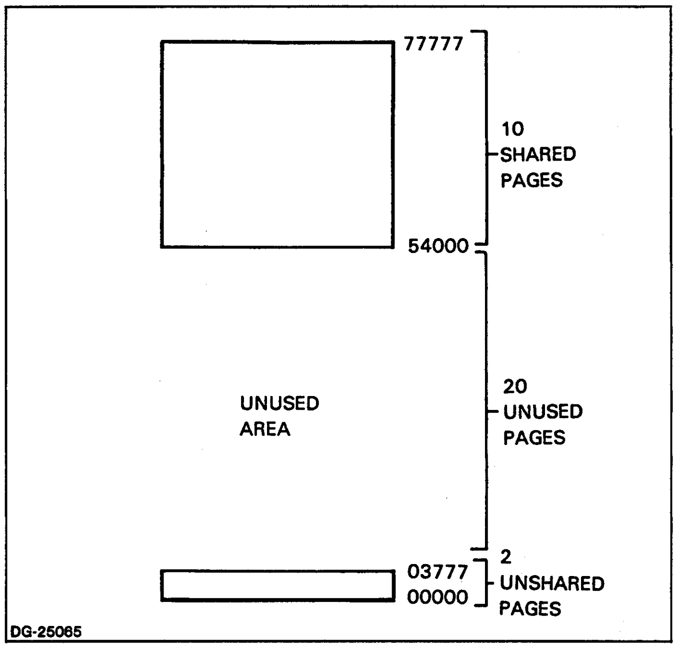


Figure 2-5. A Typical Logical Address Space

ZREL and NREL

In addition to shared and unshared pages, logical address space is also divided into *ZREL* and *NREL* addresses (or locations):

- ZREL (Addresses 00000 through 00377) — The range of addresses that can be accessed from any logical address by an instruction with an 8-bit displacement.
- NREL (Addresses 00400 through 77777) — The range of addresses that can be accessed from any logical address by an instruction with a 16-bit displacement.

A *displacement* is the part of a machine language instruction that tells the CPU which logical address it should access. Instructions with 8-bit displacements take up only one word of memory. Instructions with 16-bit displacements take up two words of memory. Intelligent use of ZREL addresses, therefore, will keep your program file smaller.

Although instructions with 8-bit displacements can always access ZREL addresses, it is also possible to use them to access NREL addresses. The ability to do this depends not only on the displacement, but also on the addressing mode (sometimes called indexing mode). Every machine language instruction that defines a displacement also defines an addressing mode. The CPU uses the addressing mode to determine how to interpret the displacement. ECLIPSE hardware distinguishes between the following four types of indexing modes:

- Index Mode 0 — absolute addressing: In this mode, displacement refers to the distance from address 00000. Therefore, an instruction using absolute addressing will access the address equivalent to the displacement. For instance, an instruction with a displacement of 100, accesses address 00100. Absolute address instructions with 8-bit displacements can access the contents of any logical address in the range 00000 to 00377 (the range of ZREL).

- Index Mode 1 — PC (Program Counter) relative addressing: In this mode, the displacement is the distance between the calling instruction and the instruction to be accessed. For instance, if an instruction at address 24200 contains a displacement of +100, then this instruction will access logical address 24300. A PC relative addressing instruction with 8-bit displacement can access the contents of any address in the range PC-200 to PC+177. For example, a PC relative instruction at address 01500 can access the contents of any address in the range 01300 to 01677.
- Index Mode 2 — AC2 relative addressing: This mode is similar to PC relative addressing, except that the hardware uses the contents of accumulator 2 in place of the PC. In other words, displacement measures the distance from the contents of accumulator 2 to the accessed instruction. For instance, if accumulator 2 contains 013500 and the displacement is -100, then the CPU accesses address 13400. ACC relative addressing instructions with 8-bit displacements can access the contents of any address in the range AC2-00200 to AC2+00177.
- Index Mode 3 — AC3 relative addressing: Same as index mode 2 except that the contents of accumulator 3 substitute for the contents of accumulator 2.

Thus, you can use instructions with 8-bit displacements if the logical address you are accessing is in ZREL or if it is sufficiently close to the PC, the value of accumulator 2, or the value of accumulator 3.

Link-Generated Information

Broadly speaking, Link builds .PR files from the following three sources:

- information generated by Link itself
- the object modules you specify in the Link command line
- the AOS system library (URT.LB)

In this section we describe information generated by Link itself, including reserved storage locations, system tables, and the stack.

Chapter 4 and Appendix B details object modules. We discuss URT.LB in Chapter 5 and in the "System Calls" section of this chapter.

Reserved Storage Locations

Logical addresses 00000 through 00047 are called *reserved storage locations.* ECLIPSE hardware uses some reserved storage locations. For instance, when the hardware detects a floating-point error, it executes a jump indirect through address 00045.

There are also some software uses for the reserved storage locations. URT.LB stores a variety of pointers within the reserved storage locations. For instance, URT.LB stores a pointer at location 00015 to a routine that handles the resource call ?RCHAIN.

The reserved storage locations can be grouped into four categories:

- Addresses 00000 to 00017. These addresses are reserved for a variety of hardware and software uses. You should not attempt to change the contents of these addresses at runtime.
- Addresses 00020 to 00037. Both 16-bit ECLIPSE and NOVA-line computers contain auto-incrementing and auto-decrementing functions at these locations, but the ECLIPSE MV/Family does not. You should avoid using these addresses if there is any chance that you will run this program on MV/Family hardware.
- Addresses 00040 to 00042. At runtime, the CPU uses these locations to define and monitor the stack and frame. Link initializes these values, though it is possible to change them at runtime.

- Addresses 00043 to 00047. These locations contain a variety of pointers to hardware fault-handling routines. By default, Link initializes these values, and URT.LB supplies the appropriate fault-handling routines; however, it is possible to change these values at runtime.

Table 2-1 lists the reserved storage locations known to Link. (For the complete list of reserved storage locations, see the appropriate manual in the Programmer's Reference Series for ECLIPSE-Line Computers.)

Table 2-1. Reserved Storage Locations Built by Link

Address	Location Name	Function
00040	Stack Pointer	Address of the current top of the stack.
00041	Frame Pointer	Address of the start of the current frame
00042	Stack Limit	Greatest address in the stack
00043	Stack Fault Address	Starting address of a routine that handles stack errors.
00045	Floating-point Fault Address	Starting address of a routine that handles floating-point errors.
00046	Commercial Fault Address	Starting address of a routine that handles commercial errors.

System Tables

Starting at the beginning of NREL (location 00400), Link builds two to four *system tables.* AOS is the primary user of the information in system tables. For instance, AOS reads an entry in the User Status Table to determine how many shared pages the .PR file contains. The number of shared pages is crucial for mapping.

In addition, some of the information in these tables is used by the process itself at execution time. For instance, the *resource manager* (a routine which handles some resource calls) accesses the information in the Resource Handler Table to find out where overlays are located.

Table 2-2 gives an overview of these tables. (Refer to the *AOS Programmer's Manual* for complete details on all system tables.)

Table 2-2. System Tables

Table Name	When Does Link Build This Table?	Table Contents
User Status Table (UST)	By default, (but /UDF switch suppresses it).	Diverse information on tasks, unshared and shared page parameters, revision numbers, and more.
Task Control Block (TCB)	By default, (but /UDF switch suppresses it).	Information for each defined task on stacks, accumulators, and starting addresses.
Overlay Directory (OLDIR)	If the .PR file contains an overlay area and if the /UDF switch is not present.	Information on overlays (the number of basic overlay areas, the number of overlays within each area, the size of overlays, etc.).
Resource Handler Table (RHT)	If the .OL file contains one or more PENTS.	Information on the position of PENTS in overlays.

Stack

The stack is the section of logical address space that stores data on a "first in-last out" basis. The CPU keeps track of the stack by monitoring addresses 00040 through 00043. (See Table 2-1 for full details.)

For instance, address 00040 always contains the stack pointer (i.e., the address of the current top of the stack). The instruction PSH 0,1 (push the contents of accumulators 0 and 1 onto the stack) increments the contents of address 00040 by 2. Thus, the next time the CPU executes a stack operation, the stack pointer will be 2 addresses greater.

The stack should be at least 36_8 addresses long. If not, then it will be impossible to perform certain system calls. Link sets the default stack length to 36_8, but you have the option of increasing its size when you link.

By default, Link starts reserving space for the stack at the first free address in unshared NREL. In other words, the stack occupies the highest addresses in unshared NREL. (See Figures 5-1 through 5-13 for further clarification of stack placement.)

System Calls

AOS creates, destroys, and monitors all processes and tasks executing on the system. AOS also manages input and output. User processes use *system calls* to communicate with AOS. System calls are the set of commands that invoke a wide variety of AOS functions. For instance, the system call ?GOPEN tells AOS that your process wants to open a file for block input and output. Another system call, ?TASK, tells AOS that your process wants to initiate one or more tasks.

System calls resolution is a very complex procedure requiring the cooperation of language processors, Link, the system library, language libraries, ECLIPSE hardware, and AOS. At runtime, some system calls are simple JSRs to routines from the system library; while other system calls force AOS to map the calling process out, map the system file in (which executes the system call), map the system file out, and map the calling process back in.

Resource calls manage overlays. Chapter 4 details resource call resolution.

If you are programming in a high-level language, the compiler usually makes system call decisions for you. However, some high-level language programmers and all assembly language programmers issue system calls in their source code. For information on system calls, refer to the *AOS Programmer's Manual*. You can find additional system call information in files SYSID.SR and PARU.SR.

Overlays

If you specify a part of logical address space as an overlay area, then two or more datawords can occupy the same logical address if AOS maps them in at different times. In general, programmers use overlay areas when the number of datawords exceeds the number of logical addresses available.

Because the 16-bit ECLIPSE computer can access no more than 100000_8 logical addresses, you will have to create overlay areas if you want to execute a program that contains over 100000_8 datawords. However, you can create overlays for smaller programs.

If you tell Link that you want your program to contain overlays areas, then Link will create two distinct files:

- .PR file — A file identical to the usual (nonoverlayed) .PR file except that it has one or more gaps in it. These gaps, called *overlay areas*, are reserved for overlays. Routines not part of any overlay are said to be in the *root*.
- .OL file — A file that contains one or more overlays. An overlay file is not executable, but the overlays inside the file become executable when mapped into an overlay area within the process.

AOS supports up to 63 overlay areas and up to 511 overlays per overlay area. Ordinarily, to create overlay areas, you tell Link, in your Link command line, which object modules you want to contribute to overlay areas and which object modules you want to contribute to the .PR file. Link then builds overlay areas and the .OL file.

Link uses two criteria to determine the size of an overlay area. First, Link determines whether the overlay area will be in a shared page or an unshared page. Overlay areas in shared pages are called *shared overlay areas*; overlay areas in unshared pages are called *unshared overlay areas*. Link builds shared overlay areas in pages, and unshared overlay areas in blocks. In other words, all shared overlay areas take up some multiple of 2000_8 logical addresses, and unshared overlay areas take up some multiple of 400_8 logical addresses.

Second, an overlay area must be large enough to accommodate the largest overlay that can occupy it. For instance, suppose that several overlays contribute to a shared overlay area. If the largest overlay takes up 5243_8 logical addresses, Link reserves 6000_8 logical addresses (the next highest page multiple) for this overlay area. If the same overlays contributed to one unshared overlay area, Link would have reserved 5400_8 logical addresses (the next highest block multiple).

For instance, suppose you wrote a program called BIG containing approximately 120000_8 datawords. Clearly, 120000_8 datawords cannot fit into the 16-bit ECLIPSE computer logical address space; therefore, overlay areas are mandated. In the Link command line, you tell Link which object modules you want to contribute to the .OL file. Assume that Link generates one shared overlay area in the .PR file, and puts three overlays in the .OL file. For convenience, we will call these overlays A (14367_8 datawords), B (17754_8 datawords), and C (3435_8 datawords).

Overlay B, the largest overlay, determines the size of this overlay area. Link will reserve 20000_8 logical addresses for this overlay area because 20000_8 is the next highest multiple of 2000_8 from 17754_8. As shown in Figure 2-6, the root takes up 60000_8 root addresses.

In Figure 2-7, overlay C is mapped into the overlay area. The code from C becomes executable as soon as it becomes part of logical address space. If the process now requests the code from overlay B, AOS will map C out and B in (as shown in Figure 2-8).

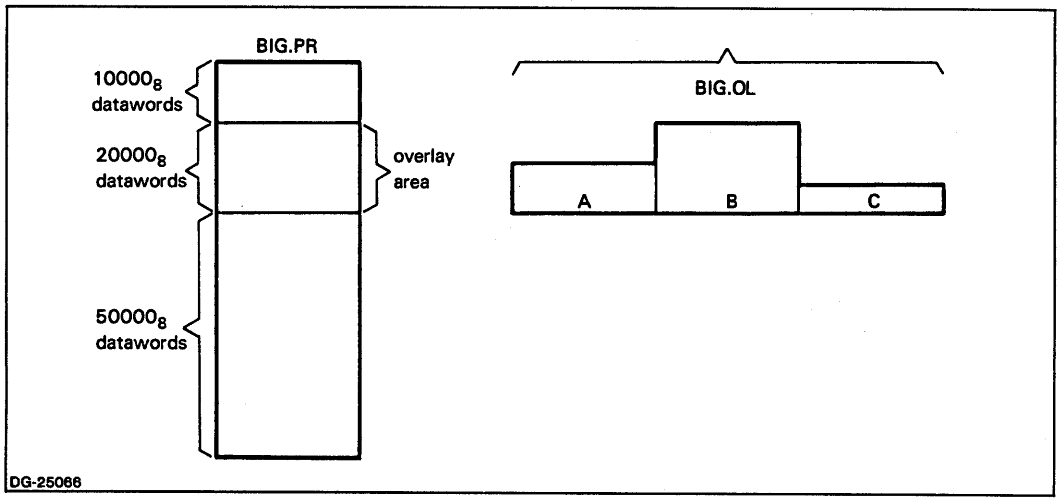


Figure 2-6. A Large Program Divided into a Program File and an Overlay File

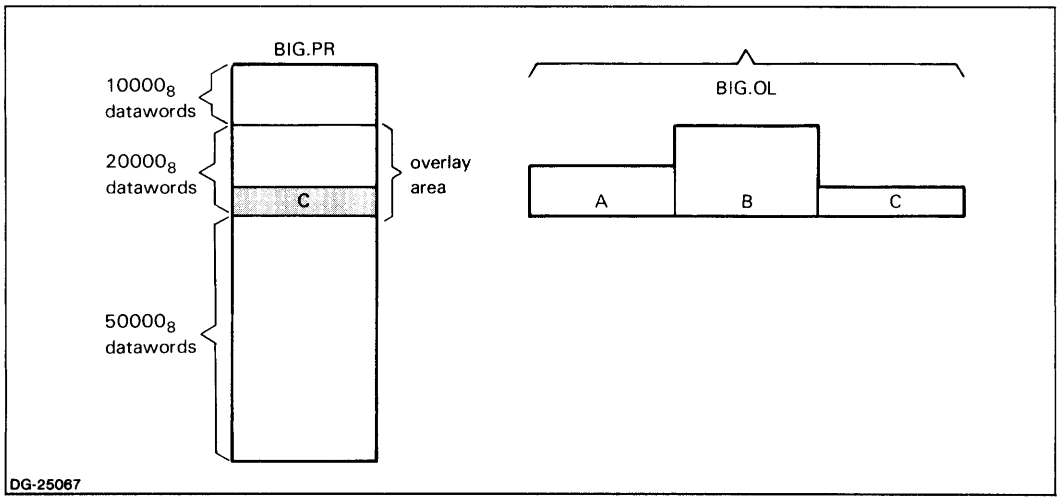


Figure 2-7. Overlay C in Logical Address Space

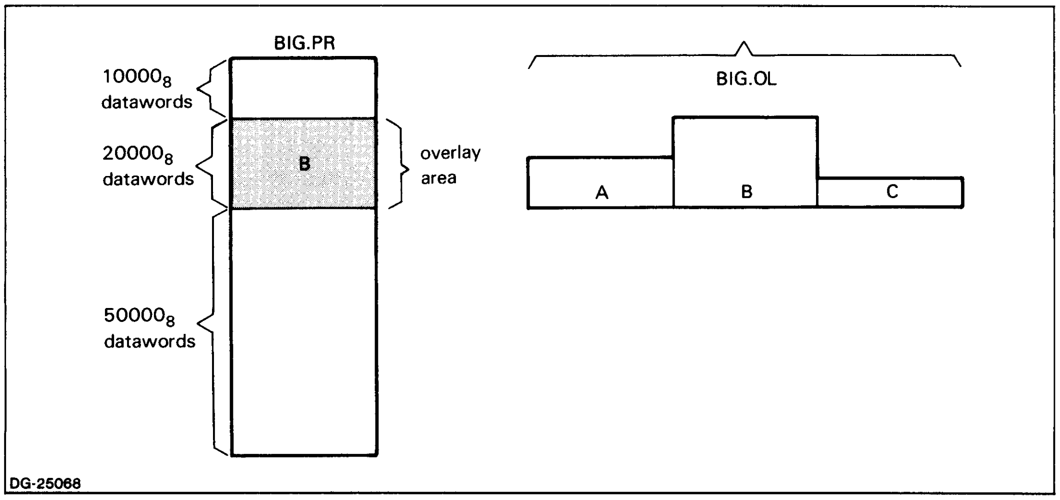


Figure 2-8. Overlay B in Logical Address Space

Multiple Basic Areas

When used properly, multiple basic areas can make your process run faster.

A *basic area* is an overlay area equal to the size of the largest overlay rounded to the next page multiple (if it is a shared basic area) or block multiple (if it is an unshared basic area). In other words, by default, a basic area is exactly the same size as the overlay area.

However, you may tell Link to build an overlay area containing *multiple basic areas.* That is, you may want Link to build an overlay area several times larger than the size of the largest contributing overlay.

For instance, in Figure 2-9 five overlays (D, E, F, G, and H) contribute to a shared overlay area. The largest overlay, F (containing 11653_8 machine language words) determines the size of the basic area (12000_8 logical addresses). However, in the Link command line, we requested multiple basic areas. More precisely, the command line told Link to create two basic areas. Thus, Link reserves 2 x $12000_8 = 24000_8$ addresses for this overlay area.

Multiple basic areas can reduce the amount of mapping, and consequently make your process execute faster. For instance, suppose overlays E and G (from Figure 2-9) contribute to the same overlay area and access each other frequently. If the overlay area consists of only one basic area, then each time E accesses G, AOS will have to map E out and G in. But, if the overlay area contains 2 basic areas, then E and G can be in the root simultaneously, thus reducing the amount of mapping.

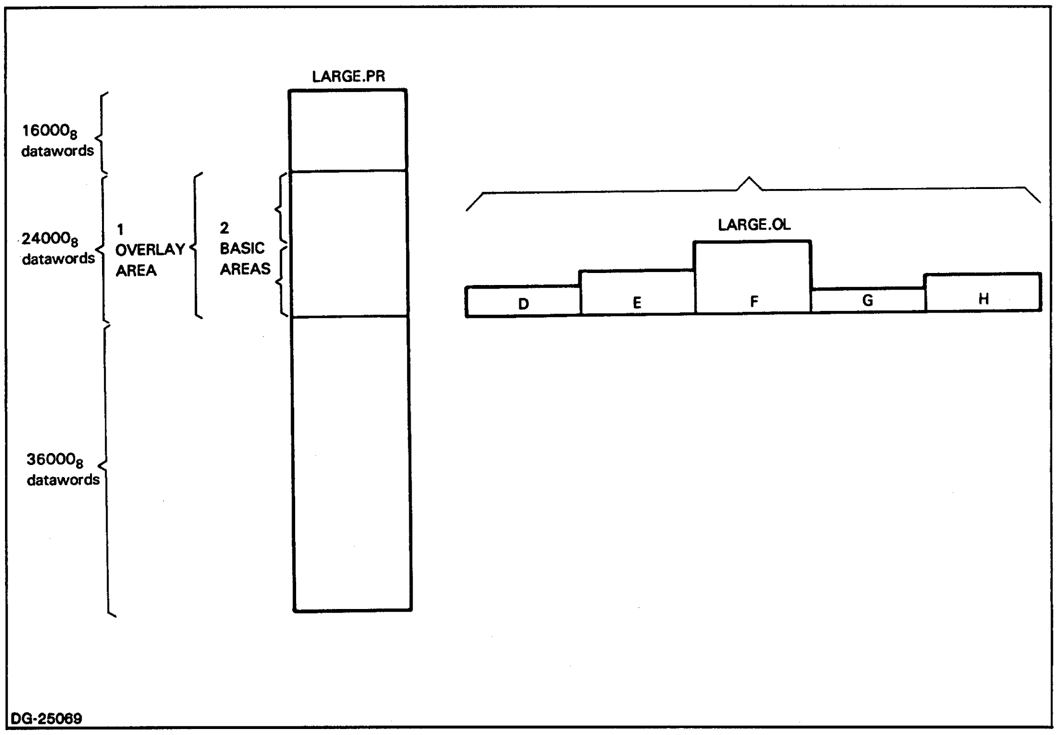


Figure 2-9. LARGE.PR Contains Two Basic Areas Within One Overlay Area

A disadvantage to using multiple basic areas is that they reduce the possible size of the root. Another disadvantage is that overlays going into multiple basic areas must contain position-independent routines. That is, routines within the overlay must be written so that it doesn't matter at which address they begin. A corollary is that position-independent machine language instructions cannot use PC relative addressing (index mode 1) to access addresses outside the overlay.

End of Chapter

Chapter 3
Relocatable Linkers and Related Concepts

Link belongs to a class of programs known as relocatable linkers. A relocatable linker is a program that allocates space in a program file for the datawords (i.e., code or data) generated by a compiler or assembler.

The primary purpose of this chapter is to introduce general concepts about relocatable linkers; Chapter 4 explores Link specifics. If you have substantial experience with other relocatable linkers, you may wish to skip this chapter and move ahead to Chapter 4.

This chapter tackles the following questions related to relocatable linkers:

- What is the difference between absolute code and relocatable code?
- What is modular programming?
- What does a relocatable linker do?
- What is the difference between a one-pass linker and a two-pass linker?
- What is a partition?
- What is cross-linking?
- What are intermodular symbols?
- What purpose do libraries serve in high-level languages?
- How does the relocatable linker prepare the debugger?

Absolute Code and Relocatable Code

Broadly speaking, source code may consist of either absolute code or relocatable code:

- Absolute code is source code that the programmer earmarks for a *specific* location in the program file. For example, in AOS, the assembly language pseudo-op .LOC 500 ultimately forces Link to put an instruction at location 00500 in the .PR file.
- Relocatable code is source code that the relocatable linker can place anywhere in the program file. For example, in AOS, the pseudo-op .NREL allows Link to decide where to put an instruction in the .PR file.

In high-level languages, the compiler usually decides whether source code will be absolute or relocatable. In assembly language, the programmer usually makes the decision.

Although there are some instances when absolute code is quite useful, relocatable code is usually more advantageous. The benefits of relocatable code are not obvious unless you understand the advantages of modular programming.

Modular Programming

A modular program consists of several separately compiled subprograms. The relocatable linker then binds these subprograms. Modular programming is practical for three reasons:

- It speeds compiling.
- It simplifies debugging.
- It is easier to write and edit several small subprograms than one large program.

The following example should demonstrate some of the benefits of modular programming:

> Your assignment is to write a complete statistical package capable of reading in data, doing 40 different complex statistical tests on a subset of the data, and presenting the results numerically or graphically. You estimate that this package will require 5000 lines of high-level language source code.

Assume that you decide not to write this program modularly. That is, you attempt to write one 5000-line program. After the first hour of programming, you compile the first subprogram (30 lines of source code); so far you are programming efficiently. By the end of the day, you have written 90 lines of source code. To make sure that things are running smoothly, you recompile. Efficiency is reduced because the compiler must recheck the first 30 (already proven) lines of source code. By the end of a few months and a few thousand lines of source code, you will probably have recompiled and rechecked hundreds of thousands of lines of source code.

Suppose though, that you decide to write the program modularly. That is, you break the program into 200 or so subprograms. You write the first subprogram (perhaps 30 lines) and compile it. Then you write the second subprogram (perhaps 50 lines). Instead of compiling 80 lines (30 + 50), you compile only the 50 lines of the second subprogram. This is a very efficient programming strategy. At the end of a few months, you will still be compiling very short subprograms rather than one huge, growing monster over and over.

After you have successfully compiled all 200 subprograms, you can use a relocatable linker to bind them into one executable program.

What a Relocatable Linker Does

A relocatable linker parcels out just enough space to fit every subprogram's memory requirements. If your subprograms contain relocatable code, the relocatable linker calculates space requirements. But if you program in absolute code, you have to make the space calculations yourself.

Compare Table 3-1 with Table 3-2. Both contain two subprograms. Because these subprograms were written in AOS assembly language, the programmers could choose whether they wanted the Macroassembler to generate absolute code or relocatable code. Both examples contain the same datawords. The programmer who created the subprograms in Table 3-1, specified absolute code; while the programmer who wrote the subprograms in Table 3-2, specified relocatable code. The programmer who wrote the subprograms in Table 3-1 was rather conservative; he left space (for future expansion or other subprograms) between addresses 00052 and 00067. Note that it is the programmer's responsibility to specify the proper addresses. For instance, if the programmer had set .LOC 50 on the second subprogram in Table 3-1, Link would have overwritten addresses 00050 and 00051.

The programmer in Table 3-2 did not have to decide where to put these numbers. She merely declared the code relocatable and let Link place the numbers sequentially (leaving no empty spaces). If she should have to expand her program, Link will have no trouble finding new addresses for the datawords.

Table 3-1. Two Absolute Code Subprograms

Source Code	Macroassembler's Instructions to Link	Address in .PR File that Link Puts This Dataword at
.TITLE ABS1		
.LOC 50	locate these next instructions sequentially starting at address 00050	
13		50
12532		51
.TITLE ABS2		
.LOC 70	locate these next instructions sequentially starting at address 00070	
145		70
030452		71
23632		72
14		73

Table 3-2. Two Relocatable Code Subprograms

Source Code	Macroassembler's Instructions to Link	Address in .PR File That Link Puts This Dataword at
.TITLE REL1		
.ZREL	locate the next instructions sequentially, anywhere within the address range 00050 — 00377	
13		50
12532		51
.TITLE REL2		
.ZREL	locate the next instructions sequentially, anywhere within the address range 00050 — 00377	
145		52
030452		53
23632		54
14		55

Absolute code can be useful in some instances; particularly when the target operating system or hardware expects values in specific logical addresses. For instance, when the ECLIPSE CPU detects a stack error, it always jumps indirect through address 00043 (which contains a pointer to a stack fault handling routine). A programmer who wants to store a pointer at address 00043 should probably use absolute code.

For most applications, relocatable code is preferable. Consider, for example, the 200 subprogram statistical package. Assuming that the compiler generates relocatable code, you do not have to calculate the memory requirements of each subprogram because the relocatable linker will do that for you. The relocatable linker will minimize the amount of empty space in the program file. If you make a mistake in one of the subprograms, you merely have to recompile the offending subprogram and rebind all the subprograms with the relocatable linker.

Partitions

A *partition* is a named contiguous area of a program file. A name allows the language processor to specify which area of a program file it wants to contribute a set of datawords to. This gives the language processor the opportunity to capitalize on hardware, operating system, or relocatable linker features. In Link, these features are called attributes.

For instance, datawords stored in the ZREL partition allow your program to take advantage of a hardware feature because they can be accessed by instructions with 8-bit displacements. If a language processor wants an instruction to be accessible by an instruction with an 8-bit displacement, it can tell Link that it wants this dataword to contribute to the ZREL partition.

One-pass Linkers and Two-pass Linkers

Relocatable linkers fall into two classes: *one-pass linkers* and *two-pass linkers*. As the name suggests, one-pass linkers scan the set of input object modules once, and two-pass linkers scan it twice.

During pass one, a two-pass linker scans all input object modules and adds up the sizes of all partitions. During pass two, the two-pass linker places datawords into the appropriate partitions.

One-pass linkers usually define only one partition. During its sole scan, the one-pass linker places datawords into this partition.

One-pass linkers are faster, but two-pass linkers allow for more sophisticated memory allocation.

Link is a two-pass relocatable linker.

Cross-Linking

Usually, a program file can execute only on a specific operating system. For instance, a program file that can execute on AOS cannot run under RDOS. Generally, the relocatable linker generates program files that can run on the host operating system. For instance, while on AOS, most programmers use Link to create program files that can run on AOS. Sometimes, however, it is desirable to cross-link.

Cross-linking means using a relocatable linker while on one operating system to create a program file that can execute on a different operating system. For instance, while on AOS, if you have the proper object modules and system libraries, you can use Link to create program files that run on RDOS or RTOS.

Cross-linking is particularly effective when you can write, compile, and link a program on a good development operating system and execute the program on a fast-executing operating system. For instance, if you use AOS Link to cross-link for RTOS, you get the flexibility of AOS text editors, language processors, and Link, plus you get RTOS's rapid execution.

Intermodular Symbols

Subprograms must be able to communicate with each other. That is, portions of some subprograms must be capable of accessing symbols or variables in other subprograms. For instance, the subprogram in the statistical package that performs t-tests may need to access the subprograms that calculate means and standard deviations. Intermodular symbols allow this communication between subprograms.

Intermodular symbols are symbols (e.g., variables) that appear in two or more subprograms. Broadly speaking, intermodular symbols fall into two classes: *external symbols* and *entry symbols*. This section explains how a relocatable linker uses external symbols and entry symbols to allow subprograms to communicate with each other.

AOS language processors are not capable of intermodular communication; they see only the subprogram that they are currently working on. It is the relocatable linker's job to put together the global picture. That is, the relocatable linker resolves intermodular symbols.

When a language processor encounters a symbol that might be defined by another subprogram, it defines that symbol as an external symbol. This tells the relocatable linker that this subprogram is searching for a symbol defined by another subprogram. The converse of an external symbol is an entry symbol. A language processor uses an entry symbol as a marker, so that the relocatable linker can match external symbols with entry symbols. Thus, an external symbol tells Link what to search for; an entry symbol tells the relocatable linker where to find it.

As an example of intermodular communication, consider the three FORTRAN 77 subprograms in Figure 1-1. While compiling subprogram MAIN.F77, the compiler does not know where the function MEAN is stored. So, it emits MEAN as an external symbol. While compiling subprogram AVRG.F77, the compiler determines that other subprograms might need to find MEAN, so it makes MEAN an entry symbol.

Link matches external symbol MEAN with entry symbol MEAN. When Link calculates the .PR file addresses of the calling procedure (MAIN) and the target function (MEAN), it can fill in the information that the FORTRAN 77 compiler could not.

Libraries and Object Modules

A library is a file consisting of a set of one or more object modules. Libraries are convenient for many reasons, including the following:

- On the Link command line, you can enter the name of a library instead of entering the names of a long set of object modules.
- Link can load object modules stored in libraries when the program file needs the routines they contain; when the program file does not need a particular routine, Link does not load the object module it is contained in.

Every free-standing object module (i.e., .OB file) on the Link command line contributes to the program file. However, Link loads an object module stored in a library only if either of the following conditions are met:

- A particular bit in the library is set to 1. This bit is called the forced load flag.
- The object module contains an entry symbol which matches an unresolved external symbol emitted by one of the other object modules in the Link command line.

Thus, Link binds a subset of a library's object modules into the program file.

In high-level languages, a compiler puts out only some of the datawords that finally appear in the program file. For instance, an object module generated by the AOS PL/I compiler might contain only 1000_8 or so datawords, yet Link will create a program file many times this size, perhaps over 20000_8 datawords long.

This broad difference between compiler output and relocatable linker output is made up for by the library files. Frequently, a compiler program leaves a certain section of source code unresolved because it determines that a routine from a library can resolve it. In such circumstances, the object module emits an external for that routine. The relocatable linker will find that routine in one of the libraries and place it in the program file.

As an example of libraries in high-level languages, suppose AOS library BLT.LB contains the routine I_O which handles line-printer output. I_O contains the entry symbol PUT_LP. The compiler emits an external symbol for PUT_LP whenever it encounters source code that requires line-printer output. When Link scans BLT.LB, it puts object module I_O into the program file. In other words, Link binds module I_O into the program file only when needed. If the program will work without it, then Link will not waste space by putting it in.

Debuggers and Relocatable Linkers

A high-level language debugger needs information about the original source code. The debugger cannot get this information from the program file, but a compiler and a relocatable linker can supply this information.

In addition to converting source code into a program file, compilers and relocatable linkers usually can keep track of what they did to the original source code. For instance, suppose your program contained the following lines of source code:

```
INT1:   NEW_PRIN = OLD_PRIN * ( 1 + INT_RATE )
ASGN:   X = NEW_PRIN + Z
```

Assume that the compiler translates source code line INT1 into 12_8 datawords and that the relocatable linker stores those 12_8 datawords at addresses 00500_8 to 00511_8 in the program file. In addition, suppose that the relocatable linker stores the value of variable NEW_PRIN at address 00104_8. Further assume that you want to use a debugger to halt the program (i.e., set a breakpoint) just after INT1, and you want to find out the value of variable NEW_PRIN.

The program file does not contain the names of variables; therefore, it does not know where NEW_PRIN and INT1 are stored. However, the compiler keeps track of source code line INT1, and the relocatable linker sends this information to debugger files. The debugger, using the information in the debugger files, halts the program at address 00511 and prints the contents of address 00104.

AOS Link can build the following debugger files:

- .ST (Symbol Table) file
- .DS (Debugger Symbols) file
- .DL (Debugger Lines) file
- Link listing files (produced with either /L or /L=filename)

See Appendix D for details on the .ST, .DS, and .DL files. See Figures 5-9 through 5-14 for sample Link listing files, and see Table 5-1 for information on the switches that produce these files.

End of Chapter

Chapter 4
Partitions and Relocation in Link

If you are writing a compiler, then this chapter is crucial. Many applications programmers will also find it helpful.

Every dataword (i.e., code or data) in a .PR file belongs to a partition. A partition is a group of datawords with the same set of attributes. At runtime, each partition corresponds to a contiguous portion of logical address space. Therefore, to build a program file, Link needs to know which partition a set of datawords should contribute to. Link gets this information from external numbers.

This chapter describes attributes, partitions, external numbers, and Resource Call resolution.

Attributes

An attribute is a characteristic of a partition. Each partition consists of a set of the following six attributes:

- absolute, ZREL, or NREL
- shared or unshared
- code or data
- normal base or common base
- alignment (0 through 12_8)
- overwrite-with-message or overwrite-without-message

The individual attributes in the six groups listed above are mutually exclusive. For instance, a partition will never have both the normal base and the common base attributes. Similarly, a partition cannot be part of absolute, ZREL, and NREL partitions at the same time.

The ZREL or NREL attribute capitalizes on an ECLIPSE hardware feature; the shared or unshared attribute takes advantage of an AOS feature. The remaining attributes are more important during linking than at runtime because they provide specific linking directives.

Absolute, ZREL, and NREL Attributes

The absolute, ZREL, and NREL attributes determine the general memory location of a partition.

A language processor assigns the absolute attribute to partitions that contain nonrelocatable datawords, that is, datawords that source code or the language processor assigned to specific addresses in the program file. Absolute partitions can reside in any area of a program file.

Relocatable datawords must have either the ZREL or the NREL attribute.

Link places partitions with the ZREL attribute between addresses 00050_8 and 00377_8. Note that there is a difference between the hardware and Link concepts of ZREL. From a hardware perspective, ZREL runs from 00000_8 to 00377_8 — addresses that can be accessed from anywhere in memory with an 8-bit displacement. However, because ECLIPSE hardware reserves the first 50_8 addresses for certain applications (see Chapter 2), Link will not allocate space for user-generated datawords until at least address 00050_8.

Partitions with the NREL attribute can reside anywhere in NREL memory; that is, anywhere from addresses 00400_8 to 77777_8. Note that Link places system tables at the beginning of NREL (unless you use the /UDF switch). Therefore, Link will not allocate space for any partitions with the NREL attribute below the space for the system tables. (For more information on NREL and ZREL, see Chapter 2.)

Shared and Unshared Attributes

Partitions having the shared attribute contain datawords that more than one process can access. At runtime, partitions with the shared attribute reside in shared pages, which are generally write-protected.

Partitions having the unshared attribute contain datawords that only one process can access. At runtime, a program's unshared partitions reside in unshared pages, which do not have write protection. (See Chapter 2 for more information on shared and unshared pages.)

Normal Base and Common Base Attributes

The normal base and common base attributes determine the way two or more object modules contribute to the same partition. In order to understand the distinction, you must first understand relocation bases and displacements.

A partition's *relocation base* is its lowest address. Link builds a partition from the relocation base "up". *Displacement* is the distance "up" from the partition relocation base. (If you are not familiar with partition relocation bases, you might want to generate a Link listing with the switch /MAP. To find the partition relocation base, read down from "ADDRESS".)

The total length of a normal base partition is the sum of each object module's contributions to that partition. Link assigns a unique displacement to every object module's partition contributions.

The total length of a common base partition is not affected by the amount of contributions to it. Instead, total length is *defined* by the object block that establishes the common base partition. That is, when you establish a common base partition (with either a named common block, unlabeled common block, or partition definition block), you define the length of the block. If two or more object blocks define the same common base partition, Link sets the total length of the partition equal to the highest length.

Each object module that contributes to a common base partition has a displacement of 0 relative to that partition.

For instance, suppose that the Link command line includes several object modules which make the following three contributions to the same partition:

- first contribution — 2 words long
- second contribution — 4 words long
- third contribution — 10_8 words long

If the partition has the normal attribute, Link allocates 16_8 addresses for this partition. Link allocates space for the first contribution at the partition relocation base, for the second at the partition relocation base+2, and for the third at the partition relocation base+2+4. (See Figure 4-1.)

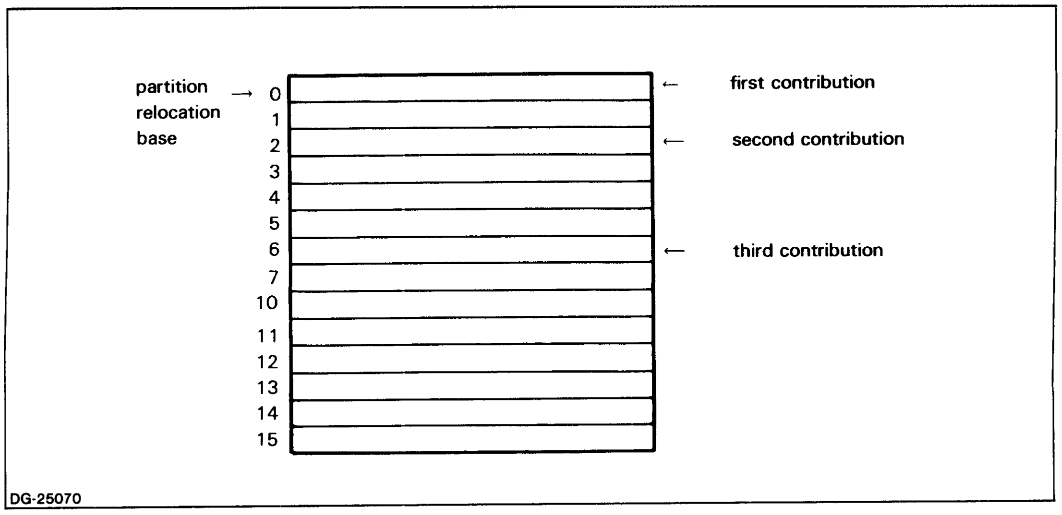


Figure 4-1. Allocation of a Partition with the Normal Base Attribute

Suppose that these same three contributions were destined for a common base partition. Further assume that two object modules define the length of this partition. If one object block defines a length of 11_8 for this partition, and another object module defines a length of 13_8, Link sets the total length of the partition equal to 13_8. All three contributions have a displacement of 0. (See Figure 4-2.)

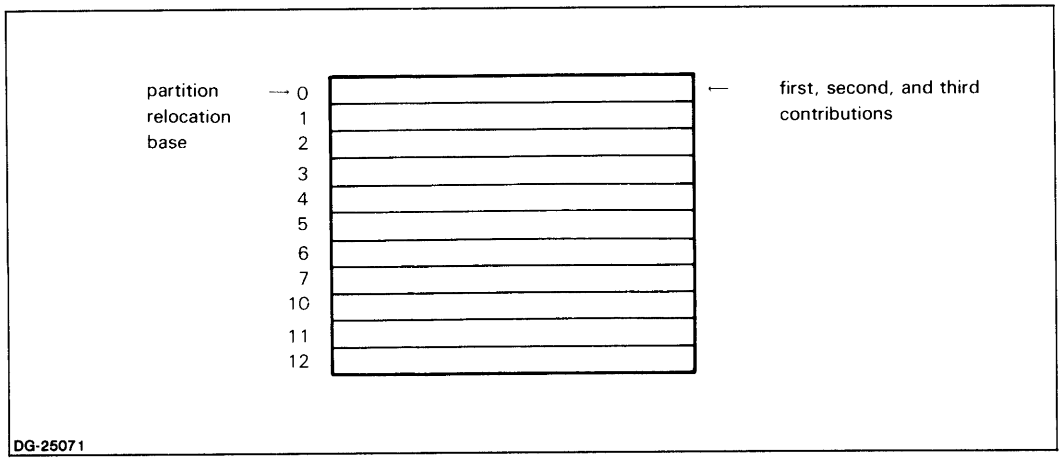

partition relocation base
0
1
2
3
4
5
6
7
10
11
12
first, second, and third contributions
DG-25071

Figure 4-2. Allocation of a Partition with the Common Base Attribute

The purpose of the common attribute is to allocate a certain area of memory and give it a name. (For further information, see the pseudo-op .COMM in the *AOS Macroassembler Reference Manual.*)

Alignment (0-12_8) Attribute

A partition's alignment is a number between 0 and 12_8 which Link uses to determine the partition relocation base. For an alignment of X, Link sets the relocation base to:

$N * 2^X$ (where N is an integer, and X is any integer from 0 to 12_8)

The default partition alignment is 0. Since 2^0 equals 1, a partition alignment of 0 permits Link to set any relocation base. There are two exceptions to the default partition alignment:

- The first shared partition has an alignment of 12_8. (Note that the first shared partition varies from program file to program file.)
- All shared overlay areas have an alignment of 12_8.

An alignment of 12 forces Link to set the relocation base to the start of a page. (See Chapter 2 for more information on pages.)

To demonstrate alignment, suppose that Link allocates space for unshared code from 03000_8 through 03056_8. Next, Link chooses to allocate space for Partition Z. If Z has an alignment of 0, then Link will set Z's relocation base to 03057_8. If Z's alignment had been 12, Link would have set Z's relocation base to some multiple of 2000_8. Therefore, Link would have set Z's relocation base to address 04000_8.

Code and Data Attributes

The code and data attributes determine whether a partition can reside in an .OL file. In this context, the accepted definitions of "code" as machine instructions and "data" as strings, constants, etc. do not apply.

Link puts partitions with the data attribute inside the program file. Data partitions can not reside in an .OL file.

Predefined partitions with the code attribute may reside in either the program file or the .OL file. (See the "Partition Types" section for a definition of predefined partitions.)

Overwrite-with-Message and Overwrite-without-Message Attributes

An *overwrite* means that Link placed a different value at an address that already contained a nonzero value. That is, if an address contains a nonzero value (e.g., 000423) and Link places a different value (e.g., 035423) at this address, then an overwrite has occurred. Note that if Link had written the same value (e.g., 000423) into this address, then an overwrite would not have occurred.

The *overwrite-with-message* attribute directs Link to send out an error message if it overwrites. The *overwrite-without-message* attribute forces Link to suppress overwrite error messages.

Partition Types

All partitions fall into one of two categories:

- predefined partitions
- user-defined partitions

The predefined partitions represent those combinations of partition attributes most frequently used in object modules. User-defined partitions may consist of any set of attributes.

Predefined Partitions

Currently, there are eight predefined partitions. Table 4-1 lists the predefined partitions, their attributes, and their *external numbers*. Object modules use external numbers to tell Link which partition a group of datawords should contribute.

External numbers 2 are 3 are seldom used; 2 resolves to the relocation base of the User Status Table, and 3 resolves to the relocation base of the unlabeled common area. (We detail external numbers later in this chapter.)

Table 4-1. Predefined Partitions

External Number	Partition Name	Attributes
0	Absolute	absolute, common base, overwrite-with-message, data, alignment=0
1	ZREL	ZREL, unshared, normal base, overwrite-with-message, data, alignment=0
2	User Status Table	not applicable. (Seldom used.)
3	Unlabeled Common Area	not applicable. (Seldom used.)
4	Unshared Code	NREL, unshared, normal base, code, overwrite-with-message, alignment=0
5	Shared Data	NREL, shared, normal base, data, overwrite-with-message, alignment=0
6	Unshared Data	NREL, unshared, normal base, data, overwrite-with-message, alignment=0
7	Shared Code	NREL, shared, normal base, code, overwrite-with-message, alignment=0

User-Defined Partitions

In addition to the predefined partitions, you may define any number of additional partitions. There are two general reasons for creating user-defined partitions.

First, you may want to generate several partitions having the same attributes but different names. Second, you may want to use a partition that has a different set of attributes than any of the predefined attributes. For instance, suppose you want some words to go into a partition that has the following attributes: NREL, unshared, normal base, code, overwrite-without-message, alignment=12. Since none of the predefined partitions has this set of attributes, you will have to generate a user-defined partition.

Generating User-Defined Partitions

If you are writing a compiler, then you should know that three object blocks can generate user-defined partitions: the partition definition block, the address information block, and the named common block. Table 4-2 explains which attributes the three blocks allow you to define.

Table 4-2. Attributes That Object Blocks Can Define

Attribute	Partition Definition Block	Address Information Block	Named Common Block
Common Base	YES		YES
Normal Base	YES	YES	
Code/Data	YES	YES	YES
Shared/Unshared	YES	YES	YES
Local/Global	YES		
Ov.-With-Mess./Ov.-Without-Mess.	YES		
Alignment (0-12_8)	YES		

The partition definition block allows you to generate a partition containing any permutation of attributes. The other two object blocks are not nearly as flexible. For instance, if you wanted to generate a partition having an alignment of 12, then you would have no choice but to use the partition definition block.

User-defined partitions generated with the partition definition block can have a *global* or a *local* name. If partition definition blocks in two or more object modules define a global name, then Link creates one partition with this name. If, however, the same partition definition blocks defined local names, Link would have created a partition for each partition definition block. A local partition can only be accessed by the object module that defined it. Link renames local partitions. Therefore, when writing your source code (or writing a language processor) keep in mind that the name of a local partition might change.

For instance, suppose two object modules, A and B, each contain partition definition blocks defining a partition named BLUE. If both of these partition definition blocks define global names, then Link will create one partition named BLUE. If, however, both of these partition definition blocks define local names, then Link will create a partition named ?UDP00 for A's contributions to BLUE and a second partition named ?UDP01 for B's contributions to BLUE.

When two or more partition definition blocks or named common blocks define the same name, Link sends out an error if their attributes do not match. Then, Link substitutes the first set of attributes for all subsequent sets of attributes.

Source Code and Partitions

If you are programming in a high-level language, the compiler ordinarily decides which partition should contain a particular group of datawords. Some high-level languages, however, do permit some user control.

If you are programming in AOS assembly language, then you can use certain pseudo-ops (see the *AOS Macroassembler Reference Manual*) to determine which partition will contain your source code. Table 4-3 shows four assembly language pseudo-ops and the associated predefined partitions.

Table 4-3. AOS Macroassembler Pseudo-ops and Partitions They Generate

Assembly Language Pseudo-op	Predefined Partition Containing the Results
.LOC expression	Absolute (assuming that "expression" is nonrelocatable.)
.ZREL	ZREL
.NREL or .NREL 0	Unshared Code
.NREL 1	Shared Code

You can override a language processor's partition decisions with certain switches in the Link command line (see Chapter 5). For instance, suppose you have two object modules: A.OB and B.OB. If you want to move *all* instructions destined for the predefined Unshared Code partition into the predefined Unshared Data partition, use the following Link command line:

```
) X LINK/UC=UD   A.OB   B.OB ↲
```

If, however, you want to move only A.OB's predefined Unshared Code contributions into the predefined Unshared Data partition:

```
) X LINK   A.OB/UC=UD   B.OB ↲
```

External Numbers

A language processor uses external numbers to send information to Link about partitions and external symbols. Many types of object blocks define external numbers (see Appendix B). For instance, the external number in the relocation entry of a data block tells Link which partition the datawords should contribute to; while the external numbers in the relocation dictionary entries allow Link to associate a dataword with an external symbol.

If you are writing a language processor, then you must generate Link-compatible external numbers. Each object module must follow this external numbering scheme:

- External numbers 0 – 7 refer to the predefined partitions (see Table 4-1).
- User-defined partitions come next in the external numbering scheme. Assign external number 10_8 to the first partition defined by a partition definition block and assign external numbers in ascending order to each subsequent partition defined by a partition definition block.
- Finally, assign external number $10_8 + n$ (where n is the number of partitions defined by this object module's partition definition blocks) to the first external symbol, and assign external numbers in ascending order to subsequent external symbols.

When you use the above external numbering scheme, external numbers in different object modules need not symbolize the same partition or external symbol. For instance, in the following example NEWPART has external number 10_8 in object module A.OB and external number 11_8 in object module B.OB, but Link knows that both refer to NEWPART.

Example

The following example demonstrates the external numbering scheme:

Assume that you have two files of source code A and B, and that a language processor uses these files to generate object files A.OB and B.OB.

Assume that object module A.OB contributes to the predefined ZREL and Unshared Code partitions and the following user-defined partitions (generated by the Partition Definition Block):

Partition Name	**Attributes**
NEWPART	NREL, shared, common base, alignment=0, data, overwrite-with-message, global
TJ	NREL, unshared, normal base, alignment=0, data, overwrite-with-message, local
ARF	NREL, shared, normal base, alignment=1, code, overwrite-with-message, global

Object Module A.OB also contains the following external symbols:

FOO
FOO1

Assume that object Module B contributes to the predefined Shared Code and Shared Data partitions and the following user-defined partitions (generated by the Partition Definition Block):

Partition Name	**Attributes**
ZZZ	NREL, shared, common base, alignment=0, code, overwrite-with-message, local
NEWPART	NREL, shared, common base, alignment=10, data, overwrite-with-message, global.

The language processor that generated A.OB and B.OB must use the following external numbering scheme in order to be Link-compatible:

For Object Module A.OB		**For Object Module B.OB**	
0		0	
1		1	
2		2	
3	predefined partitions	3	predefined partitions
4		4	
5		5	
6		6	
7		7	
10_8	NEWPART	10_8	ZZZ
11_8	TJ	11_8	NEWPART
12_8	FOO		
13_8	FOO1		

If you link A.OB and B.OB, Link knows that external number 10_8 in A.OB and external number 11_8 in B.OB both refer to the same user-defined partition.

External Numbers in Named Common Blocks and AIBs

User-defined partitions generated by named common blocks are not ordinarily part of the external numbering scheme. Rather, the names of these partitions are a special form of entry symbol. If you want to include the names of these partitions in the external numbering scheme, you must additionally define them as an external symbol.

Link treats external numbers generated by address information blocks (AIBs) differently than those generated by other object blocks. If you choose to generate partitions this way, then external numbers can take on any value above 7. However, unlike external numbers generated by partition definition blocks or external symbols blocks, Link matches external numbers across object modules.

For instance, suppose an AIB in object module A generates external number 15, and that an AIB in object module B also generates external number 15. Since Link matches external numbers across object modules, Link will send out an error if these two AIBs define partitions with different attributes. Suppose, the AIB in A defines a shared partition and the AIB in B defines an unshared partition. If A precedes B on the Link command line, Link will send out an error. Link then assigns the first set of attributes to subsequent AIBs which generated the same external number. So, datawords in B that were destined for an unshared partition will actually contribute to a shared partition.

Overlays

The Resource Calls ?RCALL, ?KCALL, and ?RCHAIN are system calls that load and release overlays under AOS. (For a description of Resource Call functionality, see the *AOS Programmer's Manual.*) The *resource manager* is a subroutine from URT.LB that helps execute resource calls at runtime.

This section describes how the language processor, Link, and the resource manager work together to resolve Resource Calls. This cooperative resolution is transparent to the user; therefore, we aimed this section mainly at programmers who are writing their own compilers. However, if you use overlays frequently, you should be aware of the Resource Call optimization switches described later in this chapter.

Generating Overlays

To generate overlays, you write a Link command line that contains overlay area delimiters. It is up to you to decide which object modules you want to contribute to overlay areas and which you want to stay in the root. Your decision should be based on variables like the size of object modules and the frequency that the code in them will be executed. Chapter 5 details the Link command line; Chapter 2 discusses overlay concepts.

Calling Overlays

If your .PR file contains overlay areas, you will need some method for calling overlays from the .OL file into an overlay area at runtime. There are two ways to do this:

- Primitive Overlay Calls
- Resource Calls

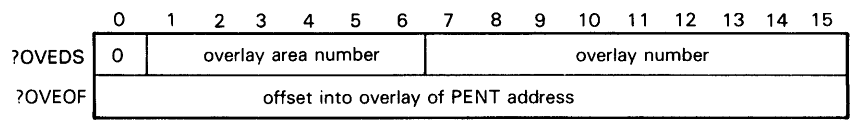
Most programmers prefer to use Resource Calls.

If you use Primitive Overlay Calls, you have to explicitly manage the loading and releasing of overlays, while Resource Calls let the resource manager handle much of the loading and releasing for you. For instance, suppose you want to load an overlay and transfer program control to it. If you use Primitive Overlay Calls, you have to pass the number of the overlay (through an ENTO symbol) and the name of an ENT symbol that you want to jump to. If you use Resource Calls, you need only pass the name of the symbol you want to jump to and the resource manager will load it (if necessary) and transfer control to it.

In short, Resource Calls are easier to use then Primitive Overlay Calls.

Link and Resource Calls

As mentioned earlier, Resource Calls take one argument: a procedure entry symbol (PENT). PENTs and standard entry symbols (ENTs) are functionally almost identical. Both are defined by an entry symbols block (see Appendix B), and Link uses both for intermodular communication. But unlike an ENT, if Link gives a PENT a value in an overlay, then Link builds the following two word entry in the Resource Handler Table (RHT):



DG-15271

(See the *AOS Programmer's Manual* for more information on system tables.)

If Link builds one or more of these RHT entries, then it also loads the resource manager from URT.LB. At runtime, the resource manager uses the information in the RHT to locate PENTs with overlay values. Once the resource manager has this information, it can execute the Resource Call.

Link never defines ENTs in the RHT. Therefore, the resource manager will not be able to find ENTs that are in an overlay. However, as we shall demonstrate later, if an ENT has a root value, it is still possible to use a Resource Call to load or release it.

In assembly language, there are two ways to pass an argument to the Resource Call:

- in line — The argument immediately follows the Resource Call. For instance, if you want to ?RCALL the overlay containing PENT "B1", then issue the command ?RCALL B1.
- on the stack — The argument is the top word on the stack when the Resource Call is made. (Refer to the *AOS Macroassembler Reference Manual* for information on the pseudo-op .PTARG.)

In order to be Link-compatible, a language processor must translate both the Resource Call and its argument into two datawords. The first dataword is the call word, and the second is the target word. For arguments passed in line, a language processor must generate the call word and target word shown in Figure 4-3. For arguments passed on the stack, a language processor must generate the call word and target word shown in Figure 4-4.

	?KCALL	?RCALL	?RCHAIN
Call Word	006013	006014	006015
Target Word	000000	000001	000002

DG-15269

Figure 4-3. Call Word and Target Word for Argument Passed In Line

	?KCALL	?RCALL	?RCHAIN
Call Word	006013	006014	006015
Target Word	000000	000000	000000

DG-15270

Figure 4-4. Call Word and Target Word for Argument Passed on the Stack

Once the call word and target word are in place, the language processor must define the proper relocation operations. For arguments passed in line, the language processor should define relocation operation 10 for the call word and relocation operation 13 for the target word. If the argument is passed on the stack, the language processor should not define relocation operations for either the call word or the target word since Link cannot know what will be on the stack when the call is issued. Instead of using relocation operations on the call word or target word, you may choose to use relocation operation 13 on an argument. Then, at runtime, you can push the resolved argument onto the stack immediately prior to issuing the resource call. (See the .PTARG pseudo-op in the *AOS Macroassembler Reference Manual.*)

Four variables affect Link's resolution of the call word and target word:

- The type of Resource Call (i.e., was the Resource Call ?RCALL, ?KCALL, or ?RCHAIN?)
- The Resource Call optimization switches (i.e., does the Link command line include the switches /NRC, /NRP, or /WRL? The next section details these switches.)
- The location the Resource Call will be made from (i.e., will the call be issued from the root or from an overlay?)
- The location of the argument (i.e., will the target PENT be in the root or in an overlay area? If the target PENT is in an overlay area, is it the same overlay area that the resource call will be issued from or some other overlay area?)

Tables 4-4 through 4-9 shows how Link resolves the call and target words for all possible combinations of these four variables. For instance, suppose that your assembly language source code contains the following command:

```
VARJM:  ?RCALL ARF  ;ARF is a PENT
```

Suppose that Link places ARF in an overlay file, and VARJM in the root. Further assume that the Link command line did not contain any Resource Call optimization switches. Thus, the Resource Call will travel from "Root" to "Other Overlay". Table 4-4 shows that Link converts the call word to a JSR@ 14 command and the target word to a RHT address. If, however, the Link command line included the switch /NRC, then Table 4-7 shows that Link would have changed the call word to EJSR and the target word to the address (within the root) of the PENT.

Resource Call Optimization Switches

Resource Call resolutions (shown in Tables 4-4 through 4-9) really fall into two categories:

- EJSR commands — At runtime, the CPU jumps direct to the target subroutine.
- JSR@ commands — At runtime, the CPU jumps indirect to the resource manager, which finds the target subroutine, loads it into logical address space (if necessary), and passes control to it.

Although the resource manager is quick, it is obvious that an EJSR command is quicker.

The Resource Call optimization switches (/NRC, /NRP, and /WRL) allow Link to substitute 106070 (i.e., an EJSR command) for 006014 (i.e., a JSR @14 commands) in certain ?RCALL situations. Without any optimization switches set, Table 4-4 shows that Link resolves 4 of the 5 possible ?RCALL situations with JSR@ commands. With /NRP set (Table 4-5), the number is reduced to 3; while /WRL (Table 4-6) further diminishes it to 2. /NRC (Table 4-7) converts every call word into an EJSR command. In fact, when you include /NRC in your Link command line, Link does not even load the resource manager into your .PR file.

Table 4-10 details the terms used in Tables 4-4 through 4-9.

The following restrictions apply to the three Resource Call optimization switches:

/NRP
1. Do not pass the target word on the stack. (Pass the target word in line.)
2. Overlay areas cannot have more than one basic area. (That is, you cannot put /MULT=n and /NRP in the same Link command line.)

/WRL
1. Do not pass the target word on the stack. (Pass the target word in line.)
2. Overlay areas cannot have more than one basic area. (You cannot put /MULT=n and /WRL in the same Link command line.)
3. At runtime, this process should not make the following sequence of calls: ROOT → OVERLAY → ROOT → DIFFERENT OVERLAY.

/NRC
1. At runtime, this process cannot issue resource calls. (To manage overlays, you must use primitive overlay calls. Use /NRC when some of your object modules contain call and target words, but you are linking for a target operating system, such as RDOS or RTOS, that does not support Resource Calls.)
2. Your program cannot issue system calls ?WALKBACK or ?UNWIND.

Table 4-4. ?RCALL Resolution if No Resource Call Optimization Switches Are in Link Command Line

To / From	Root	Same Overlay	Other Overlay
Root	EJSR Root Address		JSR @14 RHT Address
Overlay	JSR @14 Root Address	JSR @14 RHT Address	JSR @14 RHT Address

Table 4-5. ?RCALL Resolution if /NRP Is in Link Command Line

To / From	Root	Same Overlay	Other Overlay
Root	EJSR Root Address		JSR @14 RHT Address
Overlay	JSR @14 Root Address	EJSR Root Address	JSR @14 RHT Address

Table 4-6. ?RCALL Resolution if /WRL Is in Link Command Line

To / From	Root	Same Overlay	Other Overlay
Root	EJSR Root Address		JSR @14 RHT Address
Overlay	EJSR Root Address	EJSR Root Address	JSR @14 RHT Address

Table 4-7. ?RCALL Resolution if /NRC Switch Is in Link Command Line

To / From	Root	Same Overlay	Other Overlay
Root	EJSR Root Address		EJSR Root Address
Overlay	EJSR Root Address	EJSR Root Address	EJSR Root Address

Table 4-8. ?KCALL Resolution

To / From	Root	Same Overlay	Other Overlay
Root	EJSR Root Address		JSR @13 RHT Address
Overlay	JSR @13 Root Address	JSR @13 RHT Address	JSR @13 RHT Address

Table 4-9. ?RCHAIN Resolution

To / From	Root	Same Overlay	Other Overlay
Root	JSR @15 Root Address		JSR @15 RHT Address
Overlay	JSR @15 Root Address	JSR @15 RHT Address	JSR @15 RHT Address

Table 4-10. Abbreviations Used in Tables 4-4 through 4-9

	Call Words	
Abbreviation	**Machine Language**	**Comments**
JSR @13	0 | 0 | 6 | 0 | 1 | 3	Location 13 points to a routine in the resource manager that performs the ?KCALL.
JSR @14	0 | 0 | 6 | 0 | 1 | 4	Location 14 points to a routine in the resource manager that performs the ?RCALL.
JSR @15	0 | 0 | 6 | 0 | 1 | 5	Location 15 points to a routine in the resource manager that performs the ?RCHAIN.
EJSR	1 | 0 | 6 | 0 | 7 | 0 0 | displacement	Transfers control to address equal to value of PENT or without going through resource manager. Since this is an index mode 0 instruction, displacement is the actual address of the PENT or ENT.

	Target Words	
Root Address	0 (bit 0) | Address (not less than ?USTA) of PENT (bits 1–15)	
Comment:	If Link resolves the call word to an EJSR, then it resolves the target word to a root address. This address forms the displacement of the EJSR instruction.	
RHT Address	0 (bit 0) | Address of Resource Handler Table entry for this PENT (bits 1–15)	
Comment:	When the resource manager wants to know which overlay contains this PENT, it looks it up in the Resource Handler Table. An EJSR can not precede a RHT address.	

End of Chapter

Chapter 5
Link Command Line

You communicate with Link through the AOS Command Line Interpreter (CLI). The CLI command XEQ (abbreviated X) followed by the argument LINK invokes the Link utility. This chapter explores the many options available in the Link command line.

Command Line Without Overlays

If you do not want any overlay areas in the .PR file, use the following format to execute Link:

```
) X LINK</GLOBALswitch...> objectfile</OBswitch...>... &
&) <partition or symbol/PARTSYMswitch...> ↵
```

where:

GLOBAL switch	is one or more of the switches listed in Table 5-1.
object file	is the name of an .OB file or a library file.
OB switch	is one or more of the switches listed in Table 5-2, that affect a particular object file.
partition or symbol/PARTSYM switch	is the name of a partition or symbol defined in one of the object files in the Link command line followed by one or more of the switches listed in Table 5-3. You cannot use a partition or symbol name on the Link command line without affixing at least one PARTSYM switch to it. A PARTSYM switch affects only the partition or symbol it is attached to.

Use one or more spaces, tabs, or commas, to separate each object module name. All switches must be flush against the argument they act on. For example, if you use a GLOBAL switch, make sure there are no spaces between LINK and the switch; e.g., X LINK/L. Similarly, if you modify the same element with more than one switch, make sure there are no spaces between the switches. For example, X LINK/L/E=MISTAKES, and X LINK GREEN.OB/OVER/MAIN.

Filenames listed on the Link command line must reside in either your working directory or one of the directories in your search list. If the file is in neither, you must supply the pathname of the file.

If you put an unextended filename (e.g., SUM) on the Link command line, Link initially scans for the filename with the .OB extension (e.g., SUM.OB). If it fails to find the filename with this extension, it searches for the filename without the extension (e.g., SUM). If Link cannot find the unextended filename, it puts out an error message. Most programmers find it convenient to leave off the .OB extension.

If you put a filename with the extension .OB or .LB on the Link command line, Link searches only for the filename with this extension. For instance, if you include SUM.LB on the Link command line, Link puts out an error message if it cannot find SUM.LB.

Sample Link Command Lines

The following are examples of acceptable Link command lines:

```
1.  ) X LINK SUM.OB ↵  or  ) X LINK SUM ↵
2.  ) X LINK/REV=4.2  RED/ALIGN=10  SUM.OB  MULT.OB ↵
3.  ) X LINK/V/L/MODMAP  SUM.OB  MULT.OB/MAIN  INTEGRAL.LB ↵
4.  ) X LINK  MVT.OB/START/ZR=UC  DIFFER.LB/OVER  INTEGRAL.LB ↵
```

Example 1 links .OB file SUM.OB to form SUM.PR and SUM.ST.

Example 2 links .OB files SUM.OB and MULT.OB. to form SUM.PR and SUM.ST. /REV=4.2 is a GLOBAL switch. RED/ALIGN=10 demonstrates a partsym/PARTSYM switch sequence. Note that RED must be the name of a partition or symbol in SUM.OB or MULT.OB.

Example 3 links .OB files SUM.OB and MULT.OB with a subset of the files in library INTEGRAL.LB to form SUM.PR and SUM.ST. The example contains three GLOBAL switches /V, /L, and /MODMAP, and one OB switch /MAIN.

Example 4 links .OB file MVT.OB with library files DIFFER.LB and INTEGRAL.LB to create MVT.PR and MVT.ST. The example contains two OB switches (/START and /ZR=UC) affixed to MVT.OB, and one OB switch affixed to library DIFFER.LB.

Command Line With Overlays

In addition to the options discussed above, you can also set up one or more overlay areas with a Link command line. You must use the following format to set up an overlay area:

!*<*/OVswitch...*> objectfile<*/OBswitch...*> <*!*> <*objectfile*<*/OBswitch...*>*...*> *!

where:

!* *!	are overlay area delimiters. They force Link to create zero, one, or two overlay areas. The left overlay area delimiter !* begins an overlay area and the right overlay area delimiter *! ends it.
OV switch	is one or more of the switches listed in Table 5-4 affixed to a left overlay area delimiter !*. OV switches affect the object modules specified within the appropriate overlay area delimters.
!	is an overlay delimiter. It separates overlays. Note that the overlay area delimiters also separate overlays.
object file	is either an .OB file or a library.
OB switch	is one or more of the object file switches listed in Table 5-2.

The following is the complete format for a Link command line:

```
) X LINK</GLOBALswitch...> objectfile</OBswitch...>... &
&) <partsym/PARTSYMswitch...> <!*></OVswitch...> &
&) <objectfile</OBswitch...>...> <!> <objectfile</OBswitch...>...> <*!> ↵
```

NOTE: You do not have to put the overlay section of the Link command line on a separate line. We included the CLI line continuation mark (&) for clarity only.

How Object Modules Contribute to Overlay Areas

Object modules outside overlay delimiters contribute to root partitions only. Object modules inside overlay area delimiters contribute to the root and/or an overlay area according to the following scheme:

- Contributions to the predefined Absolute, ZREL, Unshared Data, and Shared Data partitions and all contributions from user-defined partitions go into the root.
- Contributions to the predefined Unshared Code and Shared Code partitions go into overlay areas.

For example, the following list shows eight object modules and the partitions that they can contribute to:

Object Modules	**The partitions they can contribute to**					
A.OB	ZR		UC	SD		
B.OB			UC			
C.OB	ZR	UD	UC			
D.OB			UC	SD		
E.OB					SC	
F.OB	ZR			SD	SC	
G.OB					SC	BLUE (a user defined partition)
H.OB		UD			SC	

If we link four of these object modules with the following Link command line:

```
) X LINK A.OB !* B.OB ! C.OB ! D.OB *! ↵
```

Link produces one overlay area containing three overlays. The following list shows that only the contributions from the predefined Unshared Code partition end up in the .OL file, while the remaining partitions reside in the .PR file:

Object Module	**What it Contributes to the .PR File**				**What it Contributes to the .OL file**	**Overlay Area Number Link Assigns to This Contribution to the .OL file**
A.OB	ZR		UC	SD	---	---
B.OB					UC	0
C.OB	ZR	UD			UC	0
D.OB				SD	UC	0

Usually it is desirable to fit several object modules inside the same overlay. For instance, the following command line puts the Unshared Code contributions from C.OB and D.OB into the same overlay:

```
) X LINK A.OB !* B.OB ! C.OB D.OB *! ↵
```

A pair of overlay area delimiters generates one, two, or no overlay areas depending on the following conditions:

- If the object modules inside the overlay area delimiters can contribute to either Shared Code or Unshared Code but not to both, then Link generates one overlay area.
- If the object modules inside overlay area delimiters can contribute to both Shared Code and Unshared Code, then Link generates two overlay areas.
- If the object modules inside overlay area delimiters can not contribute to Shared Code or Unshared Code, then Link generates no overlay areas.

It is a common programming practice to put all the object modules that contribute to Unshared Code into one overlay area and all the object modules that contribute to Shared Code in a second overlay area. For instance, the following Link command line generates two overlay areas:

```
) X LINK A.OB !* B.OB ! C.OB ! D.OB *! !* E.OB ! F.OB ! G.OB ! H.OB *! ↵
```

As the following information shows, this Link command line creates one unshared overlay area and one shared overlay area:

Object Module	What it contributes to the root					What it Contributes to the .OL file	Overlay Area Number Link Assigns to This Contribution to the .OL file
A.OB	ZR		UC	SD		---	---
B.OB		---				UC	0
C.OB	ZR	UD				UC	0
D.OB				SD		UC	0
E.OB		---				SC	1
F.OB	ZR			SD		SC	1
G.OB					BLUE	SC	1
H.OB		UD				SC	1

The following Link command line also contains two sets of overlay area delimiters:

```
) X LINK A.OB !* B.OB ! C.OB *! !* D.OB ! E.OB ! F.OB ! G.OB ! H.OB *! ↵
```

However, the second set of delimiters generates two overlay areas because modules inside it contribute to both Shared Code and Unshared Code. Therefore, Link generates three overlay areas — two unshared and one shared as shown below.

Object Module	What it contributes to the root					What it Contributes to the .OL	Overlay Area Number Link Assigns to This Contribution
A.OB	ZR		UC	SD		---	---
B.OB		---				UC	0
C.OB	ZR	UD				UC	0
D.OB				SD		UC	1
E.OB		---				SC	2
F.OB	ZR			SD		SC	2
G.OB					BLUE	SC	2
H.OB		UD				SC	2

Sample Link Command Lines

The following are acceptable Link command lines:

1.
```
) X LINK POWER.OB !* FORCE.OB ! NEWTON.OB *! LAGRANGE.OB ↵
```

2.
```
) X LINK/L/MAP/MODMAP   REAL   !*   EXP   !   INT/UD=UC   ! &
&) CHAR.LB/SD=SC   !   I_O.LB   *!   TEST_IO   SYS_INT/OVER ↵
```

3.
```
) X LINK   MY_PROG.OB   !*/ALIGN=10R8   COLL   !   COMB &
&) STR   !   SUBSTR   *! !*/MULT=3   I_O1   I_O2   !   I_O3   ! &
&) CHOICE_IO I_O.LB *! INIT/START ↵
```

Example 1 loads OB files POWER.OB and LAGRANGE.OB into POWER.PR. FORCE.OB and NEWTON.OB can contribute to POWER.PR and/or POWER.OL.

Example 2 loads .OB files REAL.OB, TEST_IO.OB, and SYS_INT.OB into the REAL.PR. The four object files inside the overlay area delimiters may contribute to REAL.PR and/or REAL.OL. This command line contains a variety of GLOBAL switches (/L=, /MAP, and /MODMAP) and OB switches (/UD=UC, /SD=SC, /OVER.) Note that the programmer left off the .OB extensions on four input object files; presumably, these are .OB files.

Example 3 loads object files MY_PROG.OB and INIT.OB into MY_PROG.PR. The remaining object files can contribute to MY_PROG.PR and/or MY_PROG.OL. This command line contains two sets of overlay area delimiters. The first set contains four .OB files, and the second set contains four .OB files and a library. The OV switch /ALIGN=10R8 affects the first set of object files; while /MULT=3 affects the second set. The OB switch /START is attached to the .OB file INIT.OB.

Linking High-Level Language Object Modules

In high-level languages the Link command line is usually bundled into a Link macro. Although these macros vary between languages, they invoke the same Link utility and must obey the same Link command line rules. These Link macros serve a two-fold purpose:

- They bind the input object modules to the proper language libraries.
- They contain the appropriate Link switches.

Switches in High-Level Language Macros

High-level language Link macros permit you to use most of the switches listed in this chapter. Sometimes, to link object modules, you can simply substitute the name of the Link macro in place of X LINK. For instance, the following command line links object modules generated by the AOS Macroassembler:

```
) X  LINK/L=COMPLEX.MAP/MAP  REAL.OB  IMAGINARY.OB/UC=UD ⌋
```

Suppose though that you wrote NICK and NORA in FORTRAN 77 and you compiled them with the FORTRAN 77 compiler. You can link REAL.OB and IMAGINARY.OB with the Link macro for FORTRAN77 (F77LINK):

```
) F77LINK/L=COMPLEX.MAP/MAP  REAL.OB  IMAGINARY.OB/UC=UD ⌋
```

Refer to the appropriate high-level language manual for more information.

Library Files

As already mentioned, you can use a library file in the Link command line anywhere that you can use an .OB file. A library file, or simply a library, consists of one or more object modules grouped together by the library file editor. For a complete description of library files, see Chapter 7.

Link automatically loads free-standing object modules into the .PR or .OL files. However, if an object module is part of a library, Link uses the following process to decide whether to load it:

- Is the forced load flag for the object module on? If yes, link it into the .PR (or .OL) file. If not, proceed to the next test.
- Does the object module satisfy any outstanding (unresolved) external symbols referred to by the other modules in the command line? If so, load it to the .PR (or .OL) file; otherwise, do not load it into the .PR (or .OL) file.

Refer to Chapter 6 for more information about the forced load flag; refer to Chapter 3 for more information about external symbol resolution.

The Systems Libraries

Link automatically scans the appropriate system library to resolve any outstanding external symbols referred to by the object modules. The AOS system library is called URT.LB and it contains a variety of object modules. (The *AOS Programmer's Manual* explains how you can get an updated list of these object modules.) In effect, when you enter the following command line:

```
) X LINK KINE.OB STAT.OB ↵
```

AOS Link actually links the following command line:

```
) X LINK KINE.OB STAT.OB URT.LB ↵
```

By default, Link scans URT.LB; however, you may change this default with the GLOBAL switch /SYS=n (shown in Table 5-1). If n, which stands for the target operating system, is RDOS, then Link scan SYS.LB instead of URT.LB. (See Chapter 7 for more information about SYS.LB.)

To suppress a system library scan entirely, use either the /NSLS or /UDF GLOBAL switches. (Note that /UDF produces other effects as well.) Table 5-1 describes both /UDF and /NSLS.

Starting Address

The *starting address* of a .PR file is the address of the first instruction that the hardware will execute at runtime. In other words, the starting address is the initial contents of the PC. Link stores the starting address of a .PR file in the Task Control Block. (See the *AOS Programmer's Manual* for more information about system tables.)

Two factors control the program file's starting address:

- possible starting addresses stored within object modules.
- the /START switch.

Object modules may optionally define a possible starting addresses. (For more information, see the "End Block" section in Appendix B.) Therefore, within a Link command line, several object modules may have possible starting addresses. By default, Link sets the starting address of the .PR file equal to the *last* possible starting address it encounters among the object modules on the command line.

If the /START switch is attached to an .OB file that has a possible starting address, then Link sets the .PR file starting address equal to this object module's possible starting address. If the /START switch is attached to an .OB file that does not have a possible starting address, then Link sends out the error message NO START ADDRESS HAS BEEN SPECIFIED.

For example, assume that DIVID.OB and SUM.OB have possible starting addresses, but ADD.OB does not:

Link Command Line	**.PR file Starting Address**
) X LINK ADD.OB SUB.OB DIVID.OB ↵	starting address of DIVID.OB
) X LINK ADD.OB DIVID.OB SUB.OB ↵	starting address of SUB.OB
) X LINK ADD.OB DIVID.OB/START SUB.OB ↵	starting address of DIVID.OB
) X LINK ADD.OB/START DIVID.OB SUB.OB ↵	error

The /MAIN switch forces Link to define a ENT called ".MAIN". This ENT has the same value and relocation properties as the possible starting address of the object module it is attached to.

Radix

By default, Link interprets any numeric values on the command line as decimal values. To specify an alternate radix, append the designator Rn to numerical values, where n represents a radix from 2 through 9. For example, the switch /STACK=100 increases the size of the stack to 100_{10} which is equal to 144_8. If you had used /STACK=100R8 instead, then Link would have increased the stack size to 100_8.

By default, all addresses in Link output listings (i.e., listings produced or associated with the /L or /L=filename switch) are in octal. The /HEX switch changes output listings to hexadecimal.

Examples in This Chapter

Figures 5-1 through 5-18 show the logical address spaces associated with Link-generated .PR files. The logical address space in Figure 5-1 was generated by a Link command line without switches, so you can use this figure for comparison.

Link generated the .PR files shown in Figures 5-1 through 5-18 by linking the following object modules:

- Six user-supplied object modules — three object modules in .OB files (DAY.OB, YOUNG.OB, and YIN2.OB), and three object modules from library OLIB.LB (NIGHT, OLD, and YANG.)
- Eight object modules from the User Runtime Library — CALL, CALLS, GCRB, RSLOA, WAIT, DUMMY, URTSC, and SCALL.

Table 5-1. GLOBAL switches

Switch	Description

/<ZR,UC,UD,SC,SD>= <ZR,UC,UD,SC,SD>	Diverts all datawords destined for the partition on the left side of the equal sign into the partition on the right side of the equal sign. For example:
	`) X LINK/UC=SD   MICRO.OB   MACRO.OB ↓`
	In this example, Link diverts all contributions to the predefined Unshared Code (UC) partition into the predefined Shared Data (SD) partition; furthermore, these datawords will have the attributes of the Shared Data partition.
	NOTE: Do not use the same partition on both sides of the equal sign. For instance, this command line generates a Link error:
	`) X LINK/UC=UC   MICRO.OB   MACRO.OB ↓`
	NOTE: This switch is valid only for the five predefined relocatable partitions.

/ALPHA	Produces an alphabetically sorted list of all symbols and their values. You must use /L or /L=pathname with this switch. For example, `) X LINK/L/ALPHA   ASSETS_INPUT   FINANCE.LB ↓` (Figure 5-9 shows a sample /ALPHA listing.) NOTE: Link gives undefined symbols the value of ?UNDF. (Refer to symbol ?UNDF in Table C-1.)
/BUILDSYS	Produces an AOS system (.SY) file. Only AOSGEN uses this switch. (See *How to Load and Generate Your AOS System.*)
/CHANNELS=n	Generates the symbol ?CHAN which some languages use as a channel directive. When you use this switch in combination with /SYS=RDOS or /SYS=RTOS, Link will place n in offset USTCH of the User Status Table. If you use the /SYS=RDOS or /SYS=RTOS switch, but do not use the /CHANNELS=n switch, then Link will put the value 10_8 in offset USTCH.
/DEBUG	Directs Link to create a .DL file if any object module in the Link command line contains one or more debugger lines blocks or lines title blocks. This switch also directs Link to create a .DS file if any object module in the Link command line contains one or more debugger symbols blocks. (See Appendix B for details on object blocks.) The /DEBUG switch also causes Link to emit the external symbol "DEBUG" and to place the value of that symbol in offset USTDA in the User Status Table (UST). (See the *AOS Programmer's Manual* for information about system tables.)
/E=pathname	Sends Link errors to pathname. Without this switch, Link errors go to @OUTPUT.
/HEX	Converts all numbers in Link output listings from the default (octal) to hexadecimal.
/KTOP=n	Limits the logical address space to n_{10} pages (where 1 page = 2000_8 addresses). The default value of n is 32_{10}. (See Figure 5-3.)
/L	Sends Link information to @LIST. By default, this information includes the titles of all input object modules (and their revision numbers) and the values of the basic memory parameters. Also, Link sends error messages to this file. For example, if your @LIST is file MLF, then the following command line sends Link information to file MLF: `) X LINK/L   FOURIER   TAYLOR   POLY2 ↓`
/L=pathname	Same as /L except that Link information goes to pathname rather than @LIST. For example, this command line sends Link information to file LOW: `) X LINK/L=LOW   FOURIER   TAYLOR   POLY2 ↓`

(continues)

Table 5-1. GLOBAL switches

Switch	Description
/MAP	Lists the name, length, lowest and highest address of all partitions in the .PR and .OL file. You must use /L or /L=pathname with this switch. For example: `) X LINK/L=CALC.MAP/MAP   CALC   TRIG   GEOM.LB ↵` (See a sample /MAP listing in Figure 5-10.)
/MODMAP	Produces a more detailed version of the MAP (see /MAP above). The MODMAP reports the name, length, highest and lowest addresses of each object module's contributions to each partition. You must use /L or /L=pathname with this switch. For example: `) X LINK/L/MODMAP SOLUBILITY.OB ↵` (See the sample /MODMAP listing in Figure 5-11.)
/MODSYM	Lists the .PR file addresses of the following symbol types on an object module-by-object module basis: · entry symbols (ENTs) · accumulating symbols (ASYMs) · overlay entry symbols (ENTOs) · local symbols (LOCALs) · procedure entry symbols (PENTs) · system overlay symbols (SOENTOs) You must use /L or /L=pathname with this switch. For example: `) X LINK/L=RATE.MM/MODMAP   RATE   MULTIPLIER ↵` (Also, see Figure 5-12.)
/N	Suppresses the creation of .PR, .ST, .OL, .DS, and .DL output files, but does not suppress Link output listings (e.g., information stored with /L, /L=pathname, or /E). This switch is valuable when you are not ready to execute or debug a program, but you want to know how Link will allocate space in the .PR file.
/NBOT=n	Changes the lowest NREL address from 00400_8 (the default address) to n_{10}. (See Figure 5-4.) NOTE: Under AOS, unless n equals 400, the resulting .PR file will be nonexecutable.
/NRC (No resource calls)	Converts all ?RCALLs into EJSRs. See "Resource Call Optimization Switches" in Chapter 4.
/NRP (No Resource Passing)	Converts ?RCALLs in certain situations into EJSRs. (See "Resource Call Optimization Switches" in Chapter 4.)
/NSLS (No System Library Scan)	Suppresses Link from scanning the appropriate system library. If you do not use this switch, Link automatically scans either the AOS system library URT.LB or the RDOS system library SYS.LB.
/NTOP=n	Sets ?NTOP, the highest logical address, to n_{10}. The default value of n is 32767_{10}. (See Figure 5-3.) NOTE: /NTOP is similar to /KTOP. The only difference is that /NTOP takes an address as an argument, and /KTOP takes the number of pages as an argument.
/NUMERIC	Lists the same information as /ALPHA except the information is sorted by symbol value rather than alphabetically. You must use /L or /L=pathname with this switch. For example: `) X LINK/L/NUMERIC   LANG_LIST.OB   THREE.OB ↵` (Also, see Figure 5-13.)

(continued)

Table 5-1. GLOBAL switches

Switch	Description
/O=pathname	Forces Link to name your program file pathname.PR. Without the switch, Link names the file after the first .OB file in your Link command line. Compare the following two Link command lines:) X LINK ONE.OB TWO.OB ↵ *LINK REVISION 04.20 ON 8/30/82 AT 11:24:02* *=ONE.PR CREATED*) X LINK/O=SEQUENCE ONE.OB TWO.OB ↵ *LINK REVISION 04.20 ON 8/30/82 AT 11:24:38* *=SEQUENCE.PR CREATED*
/OBPRINT	Produces an octal dump of every object block on an object module-by-object module basis. This switch is useful for examining object block structure; however, it usually generates voluminous output. You must use /L or /L=pathname with this switch.
/OVER	Suppresses overwrite error messages. By default, Link sends out an error if it overwrites an address. (See the "Overwrite-with-Message" and "Overwrite-without-Message Attributes" section in Chapter 4.)
/PRSYM	Generates an RDOS-style symbol table in the .PR file. (See Chapter 7.)
/REV=n1<.n2>	Sets offset USTRV in the User Status Table to n1<.n2> where n1 and n2 are base 10 integers between 0 and 255 inclusive. (Refer to the *AOS Programmer's Manual* for information on system tables.) If you do not use the /REV switch, Link sets USTRV to the first module revision number greater than -1 that it encounters. If Link does not encounter a revision number greater than -1, it sets USTRV to -1. For example, suppose that object module ELASTICITY has revision number 1.1 and that object module DEMAND has revision number 4.2. The following Link command line sets offset USTRV to 10.0:) X LINK/REV=10.0 ELASTICITY DEMAND ↵ The following Link command line sets offset USTRV to 1.1:) X LINK ELASTICITY DEMAND ↵ The following Link command line sets offset USTRV to 4.2:) X LINK DEMAND ELASTICITY ↵ The CLI command:) REVISION pathname [major number.minor number] ↵ also sets (or displays) offset USTRV. For more information about revision numbers, refer to the "Title Block", "Revision Block" and "Module Revision Block" sections in Appendix B.
/SRES=n	Reserves n shared pages (1 page = 2000_8 addresses), starting at ?SBOT, before the existing shared pages. (See Figure 5-5; also refer to Table C-1 for information on ?SBOT.)
/STACK=n	Changes the stack length from 30_{10} to n_{10} addresses. (See Figure 5-6.) NOTE: Because Link does not build a stack for RDOS or RTOS program files, you cannot use this switch in a command line that contains either /SYS=RDOS or /SYS=RTOS.

(continued)

Table 5-1. GLOBAL switches

Switch	Description
/SUPST	Suppresses Link's creation of the .ST file. .ST files are useful during debugging, but they do not affect program execution.
/SYS=n	Specifies the target system the output program file should be built to execute on. By default, Link builds a program file that can run on AOS. With this switch, Link can build a program file that can run on AOS, RDOS, or RTOS. For example: `) X LINK/SYS=RDOS FUNC1.OB FUNC2.OB CALCF.LB ↵` Here, Link creates a program file (FUNC1.SV) that can run on RDOS out of three AOS object files. NOTE: This switch will not work properly unless Link has access to the appropriate system library(s). For more information, see Chapter 7.
/TASKS=n	Informs Link that the .PR file will contain n potential tasks. (Task specification affects construction of the system tables.) If you use this switch and the object modules already contain task blocks, Link compares n and the information in the task blocks, and takes the maximum task specification.
/TEMP=pathname pointer	Sends Link's temporary files to the directory given by the pathname pointer. Link creates and deletes several files during the course of linking. By default, Link stores these temporary files in your working directory. This switch forces Link to create these files in a different directory; for instance, a directory stored on a faster disk. In some circumstances, this switch reduces the amount of time required for linking. A *pathname pointer* can be either a valid AOS pathname followed by a colon, or it can be one or more up arrows ↑. For instance, if you are in directory :UDD:RB:1:2:3:4 and you want Link to send its temporary files to directory :UDD:RB, you can issue either of the following command lines: `) X LINK/TEMP=:UDD:RB: MICRO.OB MACRO.OB A.LB ↵` or `) X LINK/TEMP=↑↑↑↑ MICRO.OB MACRO.OB A.LB ↵` NOTE: This switch does not affect the directory that Link sends its output files to. (See the /O switch.)
/UDF	Builds a nonexecutable file (UDF). A UDF file is a program file without system tables, a stack, or any routines from URT.LB.
/ULAST=n	Places contents of partition n in the highest unshared portion of the .PR file (just below the stack). n must be the name of a predefined or user-defined partition. If it is neither, Link ignores the switch and sends out an error message. (See Figure 5-7.)
/V	Reports the full pathname of all input .OB files and library files. Without this switch, Link reports only the titles of input object modules. You must use /L or /L=pathname with this switch.
/WRL	Converts ?RCALLs in certain situations into EJSRs. (See the "Resource Call Optimization Switches" section in Chapter 4.)

(continued)

Table 5-1. GLOBAL switches

Switch	Description
/XREF (Cross Reference)	Like /ALPHA, this switch directs Link to write an alphabetically sorted list of symbols and their values. In addition, /XREF produces a list of every object module that referred to this symbol as an external. Also, /XREF lists the .PR file address of every dataword that accessed this symbol. Refer to Figure 5-14. Notice that some listings do not contain a symbol type. For instance .GCRB is an ENTRY symbol, but calls does not have an associated TYPE. Words under the heading NAME that do not have a TYPE are actually the titles of object modules that access the entry symbol above it. Therefore, we know that one or more instructions in object module CALLS accesses symbol .GCRB. If you want to find the value of an entry symbol, look under the heading ADDRESS in an entry symbol's row. To find the address(es) of the instructions that accessed this entry symbol, look under the heading ADDRESS in an object module's row. For instance, the value of .GCRB is 76031. An instruction from object module CALLS accessed it and that instruction ended up at address 75665 in the .PR file. For a more advanced example, refer to page 5-32 and find the ENTRY symbol ?URTB. This symbol has a value of 00073. Four object modules (CALLS, WAIT, DUMMY, and URTSC) accessed it. Five instructions in object module CALLS accessed it. These instructions occupy addresses 75641, 75672, 75717, 75745, and 76015 in DAY.PR. Also in DAY.PR, at address 76221, an instruction from object module WAIT accessed ?URTB. Address 76316 in DAY.PR contains an instruction from object module DUMMY that accessed ?URTB. In addition, object module URTSC contributed two instructions (at 76532 and 76541) that accessed ?URTB. NOTE: This switch produces voluminous output.
/ZBOT=n	Changes ?ZBOT, the lowest ZREL address, from 00050_8 (the default) to n_{10}. See Figure 5-8. NOTE: When you use the /UDF switch, Link sets ?ZBOT to 00000 unless you use the /ZBOT=n switch. If n is lower than 50_8 Link will overwrite the reserved storage locations. If you set n too high, Link will overwrite the system tables that ordinarily begin at address 00400_8. Avoid both extremes if you plan to execute this file.

(concluded)

Table 5-2. OB switches

Switch	Description

/<ZR,UC,UD,SC,SD>= <ZR,UC,UD,SC,SD>	Same as the GLOBAL switch except that it acts on one object module (or library) rather than every object module in the Link command line. For example:
	`) X LINK   ONE.OB/UC=SC   TWO.OB   THREE.OB ↵`
	Link diverts ONE.OB's Unshared Code (UC) contributions into the Shared Code (SC) partition. Unshared Code contributions from TWO.OB and THREE.OB are not affected. (For a second example, refer to Figure 5-15.)
	NOTE: Unlike the equivalent GLOBAL switch, the OB switch allows you to put the same partition name on both sides of the equal sign. This OB switch can override the GLOBAL switch. For instance, consider the following Link command line:
	`) X LINK/UC=SC   FL1   FL3/UC=UC   FL5 ↵`
	The GLOBAL switch /UC=SC forces Link to divert all UC contributions into SC, but the OB switch /UC=UC overrides the GLOBAL switch's effect on object file FL3. Therefore, FL1 and FL5 can not contribute to UC, but FL3 can.

/<ZR,UC,UD,SC,SD>=n	Diverts contributions from one of the predefined relocatable partitions into an absolute partition with relocation base n. For example: `) X LINK   FL2.OB   FL4.OB/UC=2000R8   FL6.OB ↵` Without the switch, Link would have allocated FL4.OB's UC contributions immediately after it had allocated FL2.OB's UC contributions. Instead, Link will begin allocating FL4.OB's UC contributions at address 2000_8. For a second example, see Figure 5-16. NOTE: Improper placement of absolute partitions will cause overwriting.
/LOCAL	Directs Link to place an object module's local symbols (as defined in a local symbols block) into the .ST file (if the target operating system is AOS) or PRSYM (if the target operating system is RDOS). (Refer to the "Local Symbols Block" section in Appendix B.) This switch does not affect the .PR file or program execution; however, it may simplify debugging.
/MAIN	Directs Link to create ENT symbol ".MAIN". Link sets the value of this symbol equal to this object module's possible starting addresses. (See the "Starting Address" section in this chapter.) For example, assume that TWO.OB has a possible starting address: `) X LINK   ONE.OB   TWO.OB/MAIN   THREE.LB ↵` Suppose that Link sets TWO's starting address at 00565_8 in the .PR file. In this case, Link also sets the value of .MAIN to 00565.
/OVER	Suppresses Link from sending out the *OVERWRITE PREVIOUS <old contents> PRESENT <new contents>* error message. (This switch is the local equivalent of the GLOBAL switch /OVER.)
/START	Directs Link to take the possible starting address of this object module as the starting address of the .PR file. Without this switch, Link takes the last possible object module starting address it encounters as the starting address of the .PR file. (See the "Starting Address" section in this chapter for more information.)

Table 5-3. PARTSYM switches

Switch	Description
partition/ALIGN=n	Changes the relocation base of a partition to an integral multiple of 2^n where n is an integer between 0 and 12_8 inclusive. For instance, suppose that without the switch the relocation base of the predefined Unshared Data partition is 02745_8. `) X LINK   UD/ALIGN=10   NODE.OB   WIND.LB ↲` The switch forces Link to change the relocation base of UD from 02745_8 to an integral multiple of 2^{10} (02000_8). Link will set UD's relocation base to 04000_8, which is the next highest multiple of 2000_8. (For another example, see Figure 5-17. In this example, note that PUT is the name of the common base partition abbreviated COMM.)
partition/SHARED	Changes a partition's unshared attribute to shared. Suppose object module ORDER.OB contains a user-defined partition called BUBBLE. `) X LINK   BUBBLE/SHARED   ORDER.OB ↲` The switch changes partition BUBBLE from an unshared partition into a shared partition and allocates space for it in a shared page. (For another example, see Figure 5-18.) NOTE: This switch only affects the shared/unshared attribute.
accumulating symbol/VAL=n	Creates accumulating symbol and assigns the value n to it. For example, suppose that object module LPIO.OB defined a symbol Y having value 100_8. Assume that Y has symbol type ASYM. If we link: `) X LINK Y/VAL=40R8 LPIO.OB ↲` Link defines Y as an accumulating symbol and assigns the value 140 ($140 = 100_8 + 40_8$) to it. If LPIO.OB had not defined Y, then the switch would have created Y, assigned it symbol type ASYM, and set its value to 40_8. If LPIO.OB defined Y with a symbol type other then ASYM, this switch would have caused a MULTIPLY DEFINED SYMBOL ERROR.

Table 5-4. OV switches

Switch	Description
/<ZR,UC,UD,SC,SD>= <ZR,UC,UD,SC,SD>	Same as the GLOBAL switch except that it acts only on object modules in the overlay area. For example: `) X LINK   A.OB   !*/SD=SC   B.OB   !   C.OB   !   D.OB   *! ↲` Because of this switch, Link diverts contributions destined for the predefined Unshared Data partition into a shared overlay. This switch does not affect A.OB.
/ALIGN=n	Changes the relocation base of an overlay area to an integral multiple of 2^n where n is an integer between 0 and 12_8 inclusive. Link automatically sets $n=12_8$ for shared overlays. Therefore, all shared overlays begin on a page boundary (i.e, the first word of a new page.) The default value of n for unshared overlays is 0 which means that unless you use the /ALIGN switch, they start at the first available unshared address. (See the "Aligned Attribute" section in Chapter 4.)
/MULT=n	Increases the size of an overlay area by a factor of n. (Size of an overlay area = size of basic overlay area * n.) See the "Multiple Basic Areas" section in Chapter 2.

*/OVER OV switch is no longer supported.

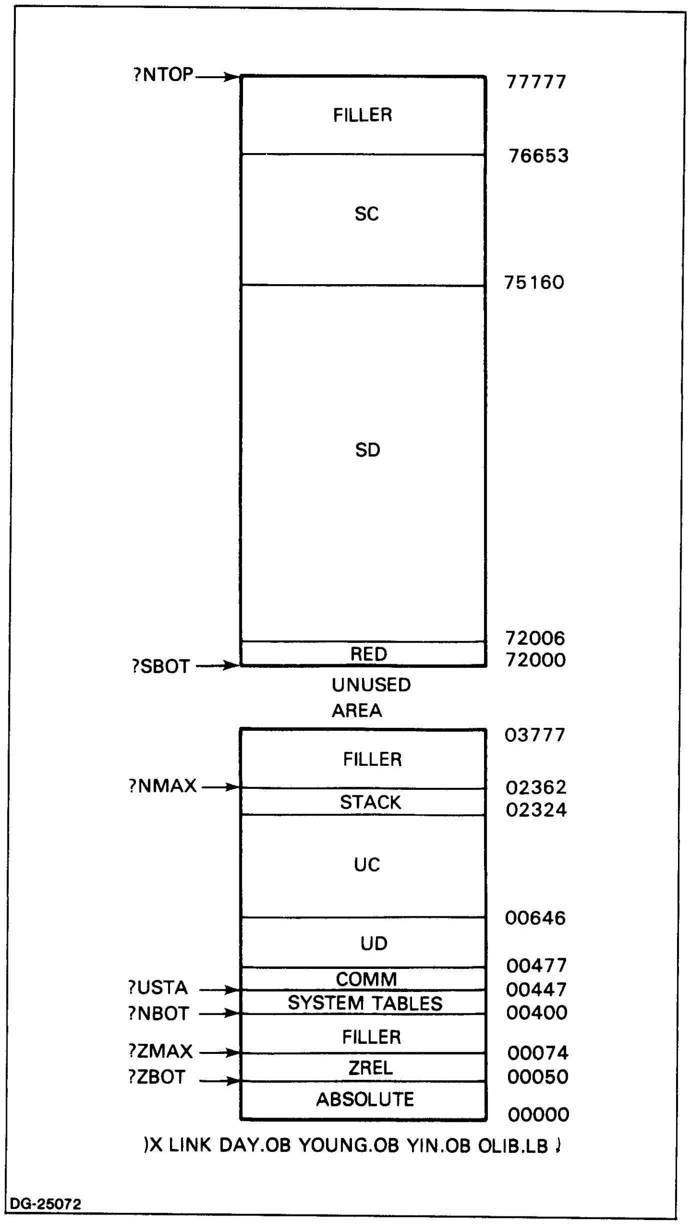


Figure 5-1. A Logical Address Space

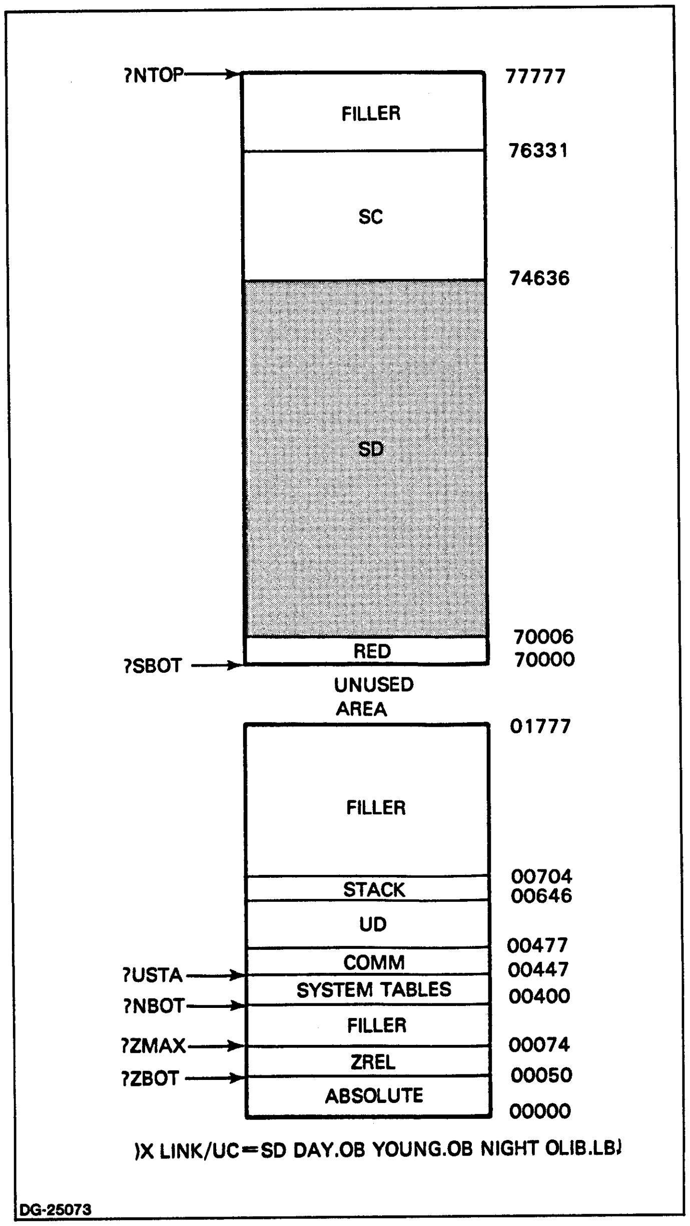


Figure 5-2. Example Showing Effect of /<ZR,UC,UD,SC,SD>= <ZR,UC,UD,SC,SD> GLOBAL Switch

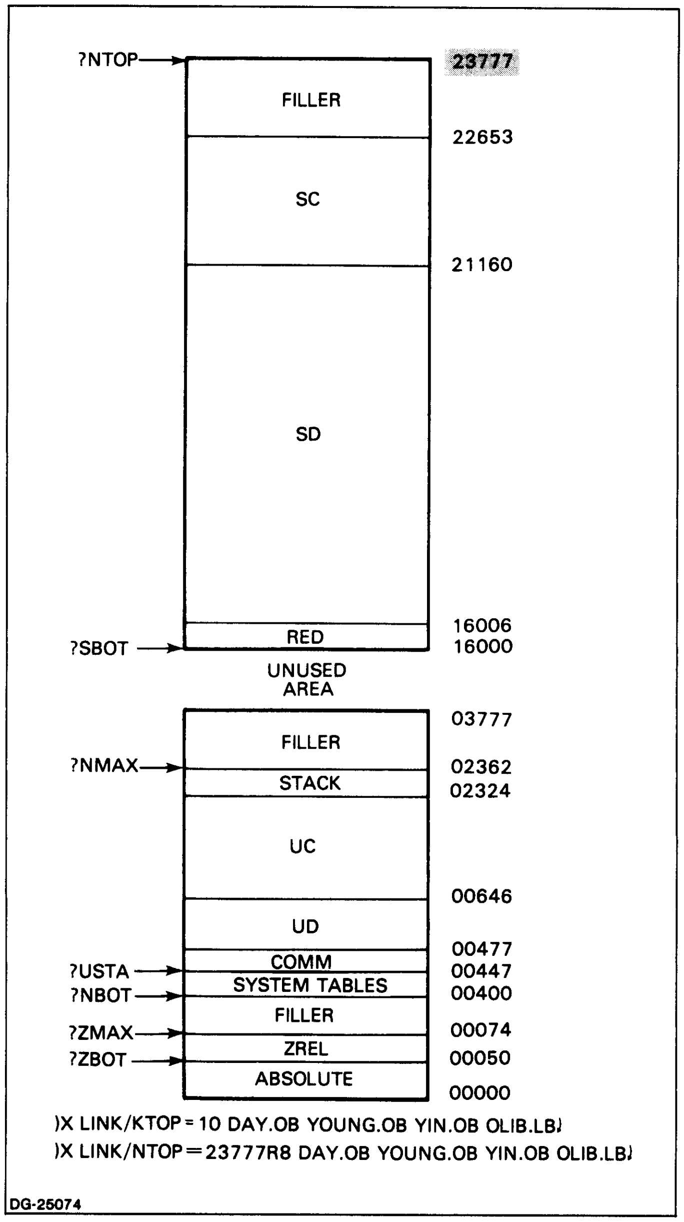
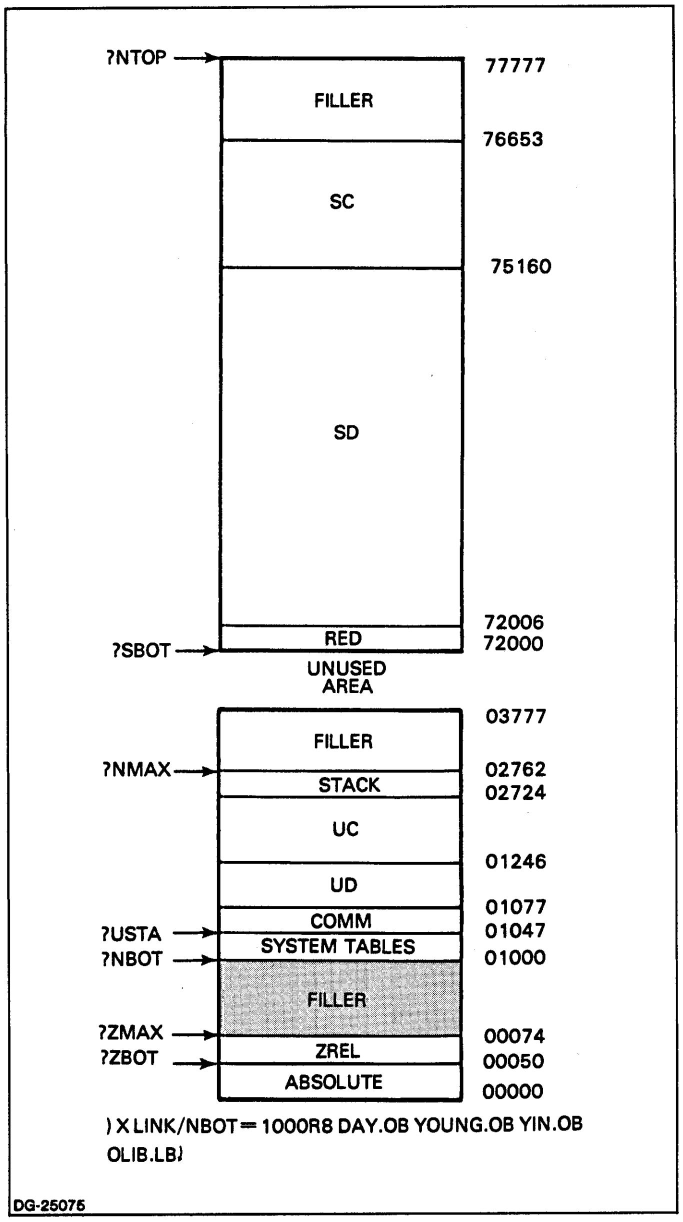


Figure 5-3. Example Showing Effect of /KTOP=n or /NTOP=n Switch



```
) X LINK/NBOT= 1000R8 DAY.OB YOUNG.OB YIN.OB
OLIB.LB)
```

Figure 5-4. Example Showing Effect of /NBOT=n Switch

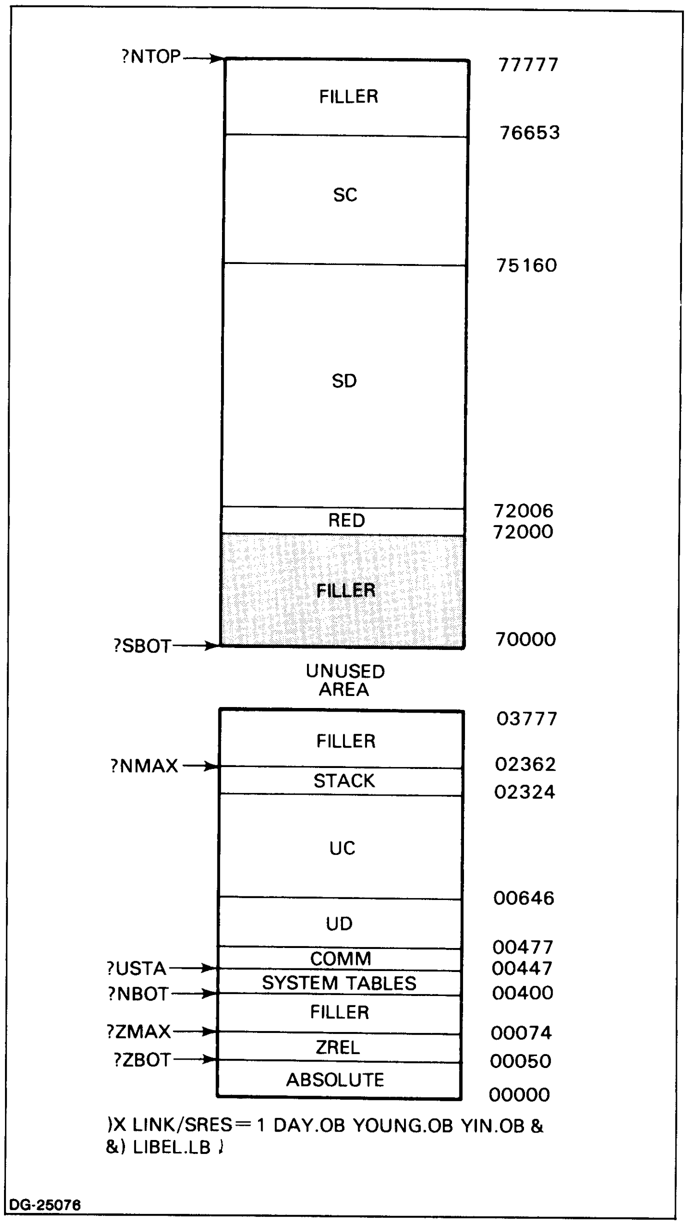
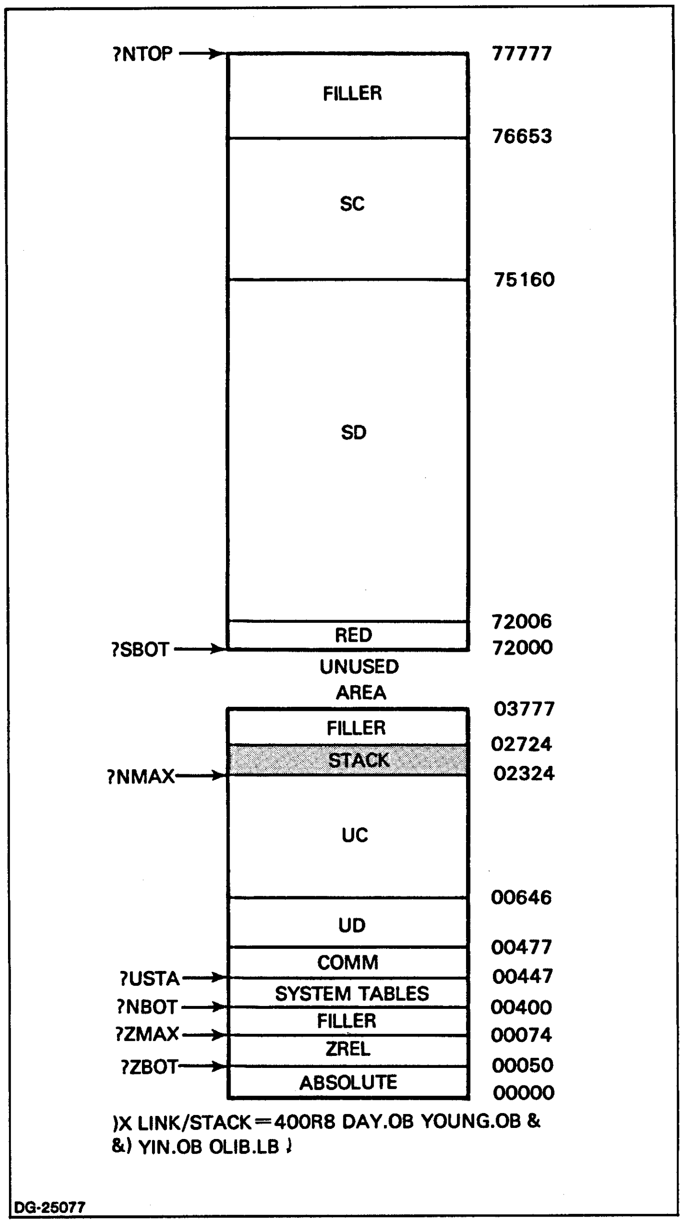


Figure 5-5. Example Showing Effect of /SRES=n Switch



```
)X LINK/STACK=400R8 DAY.OB YOUNG.OB &
&) YIN.OB OLIB.LB ↵
```

Figure 5-6. Example Showing Effect of /STACK=n Switch

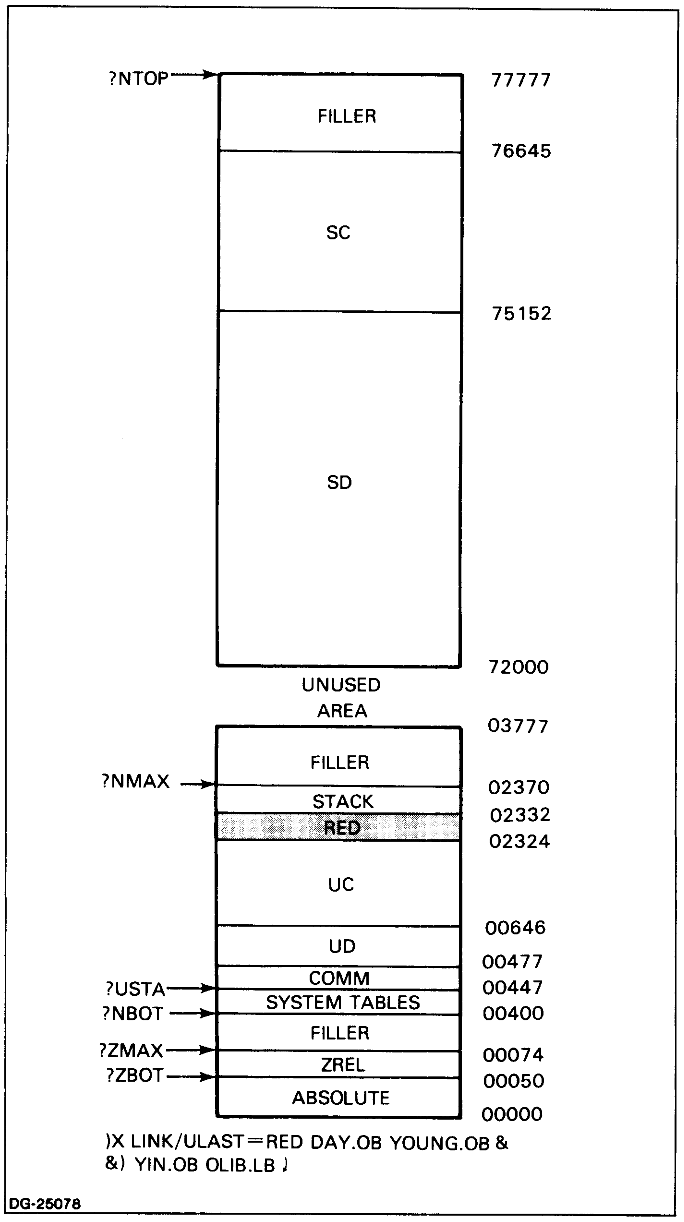


Figure 5-7. Example Showing Effect of /ULAST=n Switch

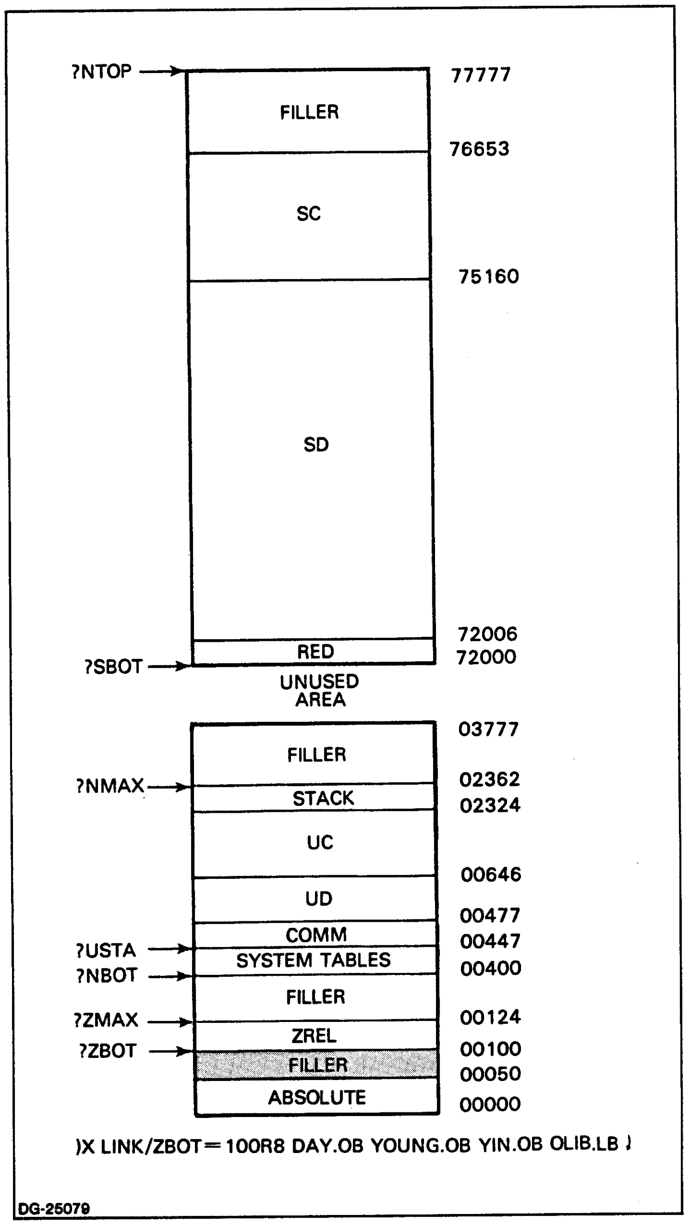


Figure 5-8. Example Showing Effect of /ZBOT=n Switch

```
LINK REVISION 04.20 ON 9/17/82 AT 13:10:24

  DAY 11.05
  YOUNG
  YIN2
  NIGHT 11.06
  OLD
  YANG
  CALL
  CALLS
  GCRB
  RSLOA
  WAIT
  DUMMY
  URTSC
  SCALL

ZBOT:           000050
ZMAX:           000074
NBOT:           000400
USTA:           000447
NMAX:           002362
SBOT:           072000
NTOP:           077777
STACK SIZE:     000036 (OCTAL)

ALPHABETIC SYMBOL LISTING

  TYPE     NAME              ADDRESS      LENGTH       END
  -------------------------------------------------------
  ENTRY    .GCRB             076031
  ENTRY    .KILL             076444
  ENTRY    ??KCA             006013
  ENTRY    ??RCA             006014
  ENTRY    ??RCH             006015
  ENTRY    ?BOMB             076314
  ENTRY    ?CLOC             177777
  ENTRY    ?CSZE             000000
  ENTRY    ?DEAD             076251
  ENTRY    ?EXRS             002321
  ENTRY    ?INOV             002323
  ENTRY    ?LBOT             177777
  ENTRY    ?LODO             076305
  ENTRY    ?NBOT             000400
  ENTRY    ?NMAX             002362
  ENTRY    ?NTOP             077777
  ENTRY    ?OFND             076362
  ENTRY    ?OVAV             076305
  ENTRY    ?RCMH             076027
  ENTRY    ?RCML             075625
  ENTRY    ?RSLO             076073
  ENTRY    ?RSRE             076142
  ENTRY    ?SBOT             072000
  ENTRY    ?SCHE             006072
  ENTRY    ?SLFN             076305
  ENTRY    ?SLLO             076305
  ENTRY    ?SLRE             076305
  ENTRY    ?SRES             000000
  ENTRY    ?TERC             076354
  ENTRY    ?UKIL             076360
  ENTRY    ?UNDF             177777
  ENTRY    ?UNWA             076230
```

Figure 5-9. Sample /ALPHA Listing (continues)

```
  ENTRY    ?URTB         000073
  ENTRY    ?USTA         000447
  ENTRY    ?UTSK         076361
  ENTRY    ?WAIT         076203
  ENTRY    ?XLD          076632
  ENTRY    ?XSV          076632
  ENTRY    ?ZBOT         000050
  ENTRY    ?ZMAX         000074
  ENTRY    AHEAD         000654
  ENTRY    CFALT         076637
  PENT     CITY          072006
  ENTRY    FAR           075160
  ENTRY    FFALT         076634
  ENTRY    FICUS         002311
  ENTRY    K?CAL         075625
  ENTRY    KNSAS         075170
  ENTRY    LIGHT         072012
  ENTRY    LOW           000051
  ENTRY    LUNCH         000501
  PENT     MOVIE         000655
  ENTRY    ONE           002314
  COMM UD  PUT           000447      000030      000476
  ENTRY    Q             000050
  ENTRY    R?CAL         075703
  ENTRY    R?CHA         075776
  PART SD  RED           072000      000006      072005
  ENTRY    SCALL         076544
  ENTRY    SCHE1         076446
  ENTRY    SCHED         076451
  ENTRY    SCNUL         076530
  ENTRY    SFALT         076643
  PENT     SLEEP         000650
  ENTRY    STAR          000646
  ENTRY    STEP          075165
  ENTRY    TREE          002313
  ENTRY    WORK          000477
=DAY.PR CREATED
```

Figure 5-9. Sample /ALPHA Listing (concluded)

```
LINK REVISION 04.20 ON 09/18/82 AT 11:19:55

  DAY 11.05
  YOUNG
  YIN2
  NIGHT 11.06
  OLD
  YANG
  CALL
  CALLS
  GCRB
  RSLOA
  WAIT
  DUMMY
  URTSC
  SCALL

ZBOT:          000050
ZMAX:          000074
NBOT:          000400
USTA:          000447
NMAX:          002362
SBOT:          072000
NTOP:          077777
STACK SIZE:    000036 (OCTAL)

  TYPE     NAME              ADDRESS      LENGTH       END
  ------------------------------------------------------------
  COMM UC  AB                000000       000020       000017
  PART UC  ZR                000050       000024       000073
  COMM UC  UST               000400       000023       000422
  COMM UC  TCB               000423       000024       000446
  COMM UD  PUT               000447       000030       000476
  PART UD  UD                000477       000147       000645
  PART UC  UC                000646       001456       002323
  COMM UC  STACK             002324       000036       002361
  PART SD  RED               072000       000006       072005
  PART SD  SD                072006       003152       075157
  PART SC  SC                075160       001473       076652
=DAY.PR CREATED
```

Figure 5-10. Sample /MAP Listing

```
LINK REVISION 04.20 ON 9/17/82 AT 13:09:51

  DAY 11.05
  YOUNG
  YIN2
  NIGHT 11.06
  OLD
  YANG
  CALL
  CALLS
  GCRB
  RSLOA
  WAIT
  DUMMY
  URTSC
  SCALL
  TYPE     NAME              ADDRESS      LENGTH       END
  -------------------------------------------------------
DAY
  PART UD  UD                000477       000147       000645
YOUNG
  PART SD  SD                072006       003152       075157
  PART UC  ZR                000050       000001       000050
YIN2
  PART SD  RED               072000       000006       072005
  PART UC  UC                000646       000002       000647
NIGHT
  PART UC  UC                000650       001441       002310
OLD
  PART UC  UC                002311       000003       002313
YANG
  PART UC  UC                002314       000005       002320
  PART SC  SC                075160       000445       075624
  PART UC  ZR                000051       000021       000071
CALL
CALLS
  PART UC  UC                002321       000001       002321
  PART SC  SC                075625       000203       076027
GCRB
  PART SC  SC                076030       000040       076067
RSLOA
  PART SC  SC                076070       000113       076202
WAIT
  PART SC  SC                076203       000102       076304
DUMMY
  PART UC  UC                002322       000002       002323
  PART SC  SC                076305       000141       076445
URTSC
  PART SC  SC                076446       000076       076543
  PART UC  ZR                000072       000001       000072
SCALL
  PART SC  SC                076544       000107       076652
  PART UC  ZR                000073       000001       000073
ZBOT:           000050
ZMAX:           000074
NBOT:           000400
USTA:           000447
NMAX:           002362
SBOT:           072000
NTOP:           077777
STACK SIZE:     000036 (OCTAL)
=DAY.PR CREATED
```

Figure 5-11. Sample /MODMAP Listing

```
LINK REVISION 04.20 ON 9/17/82 AT 13:10:06

  DAY 11.05
  YOUNG
  YIN2
  NIGHT 11.06
  OLD
  YANG
  CALL
  CALLS
  GCRB
  RSLOA
  WAIT
  DUMMY
  URTSC
  SCALL

  TYPE     NAME              ADDRESS     LENGTH      END
  -------------------------------------------------------
DAY
  ENTRY    LUNCH             000501
  ENTRY    WORK              000477
YOUNG
  PENT     CITY              072006
  ENTRY    LIGHT             072012
  ENTRY    Q                 000050
YIN2
  ENTRY    AHEAD             000654
  ENTRY    STAR              000646
NIGHT
  PENT     MOVIE             000655
  PENT     SLEEP             000650
OLD
  ENTRY    TREE              002313
  ENTRY    FICUS             002311
YANG
  ENTRY    LOW               000051
  ENTRY    STEP              075165
  ENTRY    KNSAS             075170
  ENTRY    FAR               075160
  ENTRY    ONE               002314
CALL
  ENTRY    ??RCH             006015
  ENTRY    ??RCA             006014
  ENTRY    ??KCA             006013
CALLS
  ENTRY    ?EXRS             002321
  ENTRY    ?RCMH             076027
  ENTRY    ?RCML             075625
  ENTRY    R?CHA             075776
  ENTRY    K?CAL             075625
  ENTRY    R?CAL             075703
GCRB
  ENTRY    .GCRB             076031
RSLOA
  ENTRY    ?RSRE             076142
  ENTRY    ?RSLO             076073
WAIT
  ENTRY    ?DEAD             076251
  ENTRY    ?UNWA             076230
  ENTRY    ?WAIT             076203
```

Figure 5-12. Sample /MODSYM Listing (continues)

```
DUMMY
  ENTRY    ?INOV              002323
  ENTRY    ?SLFN              076305
  ENTRY    ?EXRS              002321
  ENTRY    ?SLLO              076305
  ENTRY    ?SLRE              076305
  ENTRY    ?LODO              076305
  ENTRY    ?OVAV              076305
  ENTRY    ?RSRE              076142
  ENTRY    ?OFND              076362
  ENTRY    ?UKIL              076360
  ENTRY    ?UTSK              076361
  ENTRY    ?TERC              076354
  ENTRY    .KILL              076444
  ENTRY    ?BOMB              076314
URTSC
  ENTRY    SCHE1              076446
  ENTRY    SCNUL              076530
  ENTRY    SCHED              076451
  ENTRY    ?SCHE              006072
SCALL
  ENTRY    ?URTB              000073
  ENTRY    SFALT              076643
  ENTRY    CFALT              076637
  ENTRY    FFALT              076634
  ENTRY    ?XSV               076632
  ENTRY    ?XLD               076632
  ENTRY    SCALL              076544

ZBOT:          000050
ZMAX:          000074
NBOT:          000400
USTA:          000447
NMAX:          002362
SBOT:          072000
NTOP:          077777
STACK SIZE:    000036 (OCTAL)
=DAY.PR CREATED
```

Figure 5-12. Sample /MODSYM Listing (concluded)

```
LINK REVISION 04.20 ON 9/17/82 AT 13:10:41

  DAY 11.05
  YOUNG
  YIN2
  NIGHT 11.06
  OLD
  YANG
  CALL
  CALLS
  GCRB
  RSLOA
  WAIT
  DUMMY
  URTSC
  SCALL

ZBOT:           000050
ZMAX:           000074
NBOT:           000400
USTA:           000447
NMAX:           002362
SBOT:           072000
NTOP:           077777
STACK SIZE:     000036 (OCTAL)

NUMERIC SYMBOL LISTING

  TYPE     NAME              ADDRESS     LENGTH      END
  ------------------------------------------------------
  ENTRY    ?CSZE             000000
  ENTRY    ?SRES             000000
  ENTRY    ?ZBOT             000050
  ENTRY    Q                 000050
  ENTRY    LOW               000051
  ENTRY    ?URTB             000073
  ENTRY    ?ZMAX             000074
  ENTRY    ?NBOT             000400
  ENTRY    ?USTA             000447
  COMM UD  PUT               000447      000030      000476
  ENTRY    WORK              000477
  ENTRY    LUNCH             000501
  ENTRY    STAR              000646
  PENT     SLEEP             000650
  ENTRY    AHEAD             000654
  PENT     MOVIE             000655
  ENTRY    FICUS             002311
  ENTRY    TREE              002313
  ENTRY    ONE               002314
  ENTRY    ?EXRS             002321
  ENTRY    ?INOV             002323
  ENTRY    ?NMAX             002362
  ENTRY    ??KCA             006013
  ENTRY    ??RCA             006014
  ENTRY    ??RCH             006015
  ENTRY    ?SCHE             006072
  ENTRY    ?SBOT             072000
  PART SD  RED               072000      000006      072005
  PENT     CITY              072006
  ENTRY    LIGHT             072012
  ENTRY    FAR               075160
  ENTRY    STEP              075165
```

Figure 5-13. Sample /NUMERIC Listing (continues)

```
  ENTRY    KNSAS            075170
  ENTRY    ?RCML            075625
  ENTRY    K?CAL            075625
  ENTRY    R?CAL            075703
  ENTRY    R?CHA            075776
  ENTRY    ?RCMH            076027
  ENTRY    .GCRB            076031
  ENTRY    ?RSLO            076073
  ENTRY    ?RSRE            076142
  ENTRY    ?WAIT            076203
  ENTRY    ?UNWA            076230
  ENTRY    ?DEAD            076251
  ENTRY    ?LODO            076305
  ENTRY    ?OVAV            076305
  ENTRY    ?SLFN            076305
  ENTRY    ?SLLO            076305
  ENTRY    ?SLRE            076305
  ENTRY    ?BOMB            076314
  ENTRY    ?TERC            076354
  ENTRY    ?UKIL            076360
  ENTRY    ?UTSK            076361
  ENTRY    ?OFND            076362
  ENTRY    .KILL            076444
  ENTRY    SCHE1            076446
  ENTRY    SCHED            076451
  ENTRY    SCNUL            076530
  ENTRY    SCALL            076544
  ENTRY    ?XLD             076632
  ENTRY    ?XSV             076632
  ENTRY    FFALT            076634
  ENTRY    CFALT            076637
  ENTRY    SFALT            076643
  ENTRY    ?NTOP            077777
  ENTRY    ?CLOC            177777
  ENTRY    ?LBOT            177777
  ENTRY    ?UNDF            177777
=DAY.PR CREATED
```

Figure 5-13. Sample /NUMERIC Listing (concluded)

```
LINK REVISION 04.20 ON 03/25/82 AT 11:04:04

  DAY 11.05
  YOUNG
  YIN2
  NIGHT 11.06
  OLD
  YANG
  CALL
  CALLS
  GCRB
  RSLOA
  WAIT
  DUMMY
  URTSC
  SCALL

ZBOT:          000050
ZMAX:          000074
NBOT:          000400
USTA:          000447
NMAX:          002362
SBOT:          072000
NTOP:          077777
STACK SIZE:    000036 (OCTAL)

CROSS-REFERENCED ALPHABETIC SYMBOL LISTING

  TYPE     NAME              ADDRESS      LENGTH       END
  -----------------------------------------------------
  ENTRY    .GCRB             076031
           CALLS             075665
  ENTRY    .KILL             076444
  ENTRY    ??KCA             006013
  ENTRY    ??RCA             006014
  ENTRY    ??RCH             006015
  ENTRY    ?BOMB             076314
           GCRB              076067
           RSLOA             076202
           WAIT              076304
  ENTRY    ?CLOC             177777
  ENTRY    ?CSZE             000000
  ENTRY    ?DEAD             076251
           RSLOA             076136
  ENTRY    ?EXRS             002321
           RSLOA             076132
           WAIT              076212
  ENTRY    ?INOV             002323
           WAIT              076252
  ENTRY    ?LBOT             177777
  ENTRY    ?LODO             076305
           RSLOA             076122
  ENTRY    ?NBOT             000400
  ENTRY    ?NMAX             002362
  ENTRY    ?NTOP             077777
  ENTRY    ?OFND             076362
           GCRB              076052
  ENTRY    ?OVAV             076305
           RSLOA             076167
  ENTRY    ?RCMH             076027
  ENTRY    ?RCML             075625
```

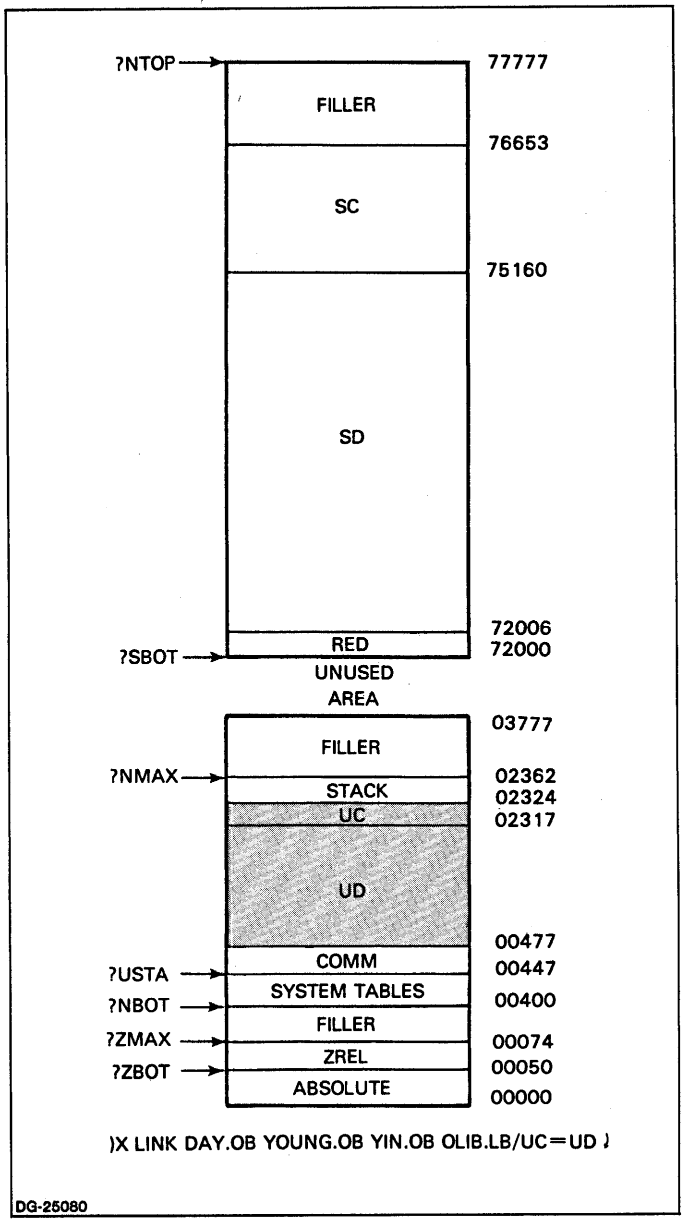
Figure 5-14. Sample /XREF Listing (continues)

```
ENTRY    ?RSLO          076073
         CALLS          075727      075765      075773
ENTRY    ?RSRE          076142
         CALLS          075647      075723      075751
                        076021
ENTRY    ?SBOT          072000
ENTRY    ?SCHE          006072
         CALLS          075656      075667      075742
                        075767
         WAIT           076227
         DUMMY          076357
ENTRY    ?SLFN          076305
         RSLOA          076071
ENTRY    ?SLLO          076305
         RSLOA          076103
ENTRY    ?SLRE          076305
         RSLOA          076150
ENTRY    ?SRES          000000
ENTRY    ?TERC          076354
ENTRY    ?UKIL          076360
ENTRY    ?UNDF          177777
ENTRY    ?UNWA          076230
ENTRY    ?URTB          000073
         CALLS          075641      075672      075717
                        075745      076015
         WAIT           076221
         DUMMY          076316
         URTSC          076532      076541
ENTRY    ?USTA          000447
         GCRB           076040
         RSLOA          076113      076156
ENTRY    ?UTSK          076361
ENTRY    ?WAIT          076203
ENTRY    ?XLD           076632
ENTRY    ?XSV           076632
ENTRY    ?ZBOT          000050
ENTRY    ?ZMAX          000074
ENTRY    AHEAD          000654
ENTRY    CFALT          076637
PENT     CITY           072006
         NIGHT          000656
ENTRY    FAR            075160
ENTRY    FFALT          076634
ENTRY    FICUS          002311
         YOUNG          072011
ENTRY    K?CAL          075625
         CALL           000013
ENTRY    KNSAS          075170
ENTRY    LIGHT          072012
ENTRY    LOW            000051
         DAY            000500
ENTRY    LUNCH          000501
PENT     MOVIE          000655
ENTRY    ONE            002314
COMM UD  PUT            000447      000030      000476
         NIGHT          000652
ENTRY    Q              000050
ENTRY    R?CAL          075703
         CALL           000014
ENTRY    R?CHA          075776
         CALL           000015
PART SD  RED            072000      000006      072005
```

Figure 5-14. Sample /XREF Listing (continued)

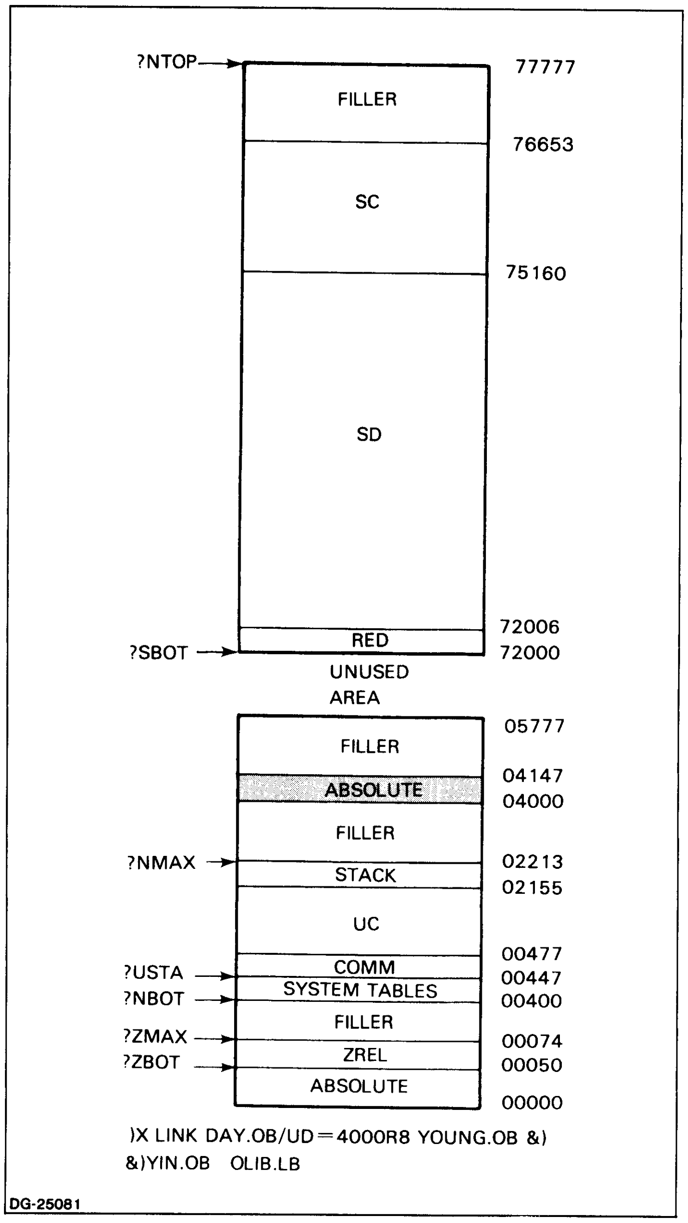
```
  ENTRY   SCALL          076544
  ENTRY   SCHE1          076446
  ENTRY   SCHED          076451
  ENTRY   SCNUL          076530
  ENTRY   SFALT          076643
  PENT    SLEEP          000650
          DAY            000502
  ENTRY   STAR           000646
  ENTRY   STEP           075165
  ENTRY   TREE           002313
  ENTRY   WORK           000477
=DAY.PR CREATED
```

Figure 5-14. Sample /XREF Listing (concluded)



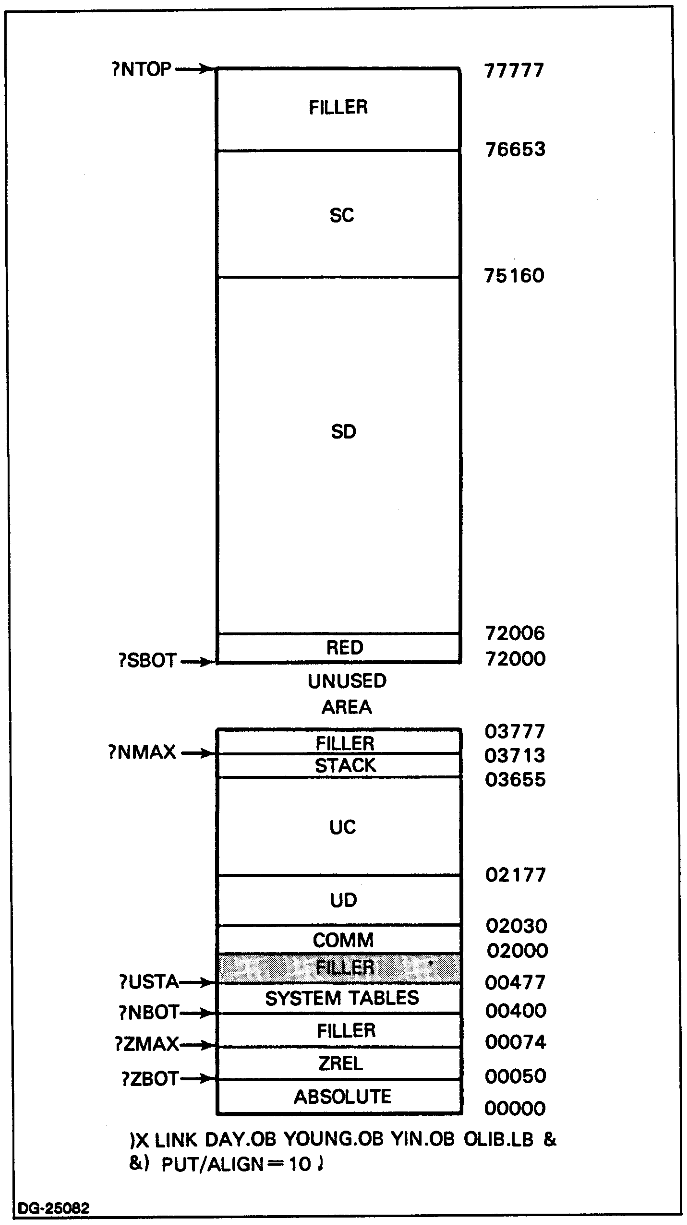
```
)X LINK DAY.OB YOUNG.OB YIN.OB OLIB.LB/UC=UD ↲
```

Figure 5-15. Example Showing Effect of /<ZR,UC,UD,SC,SD>=<ZR,UC,UD,SC,SD> OB Switch



```
)X LINK DAY.OB/UD=4000R8 YOUNG.OB &)
&)YIN.OB  OLIB.LB
```

Figure 5-16. Example Showing Effect of /<ZR,UC,UD,SC,SD>=n Switch



```
)X LINK DAY.OB YOUNG.OB YIN.OB OLIB.LB &
&) PUT/ALIGN=10 ⏎
```

Figure 5-17. Example Showing Effect of /ALIGN=n Switch

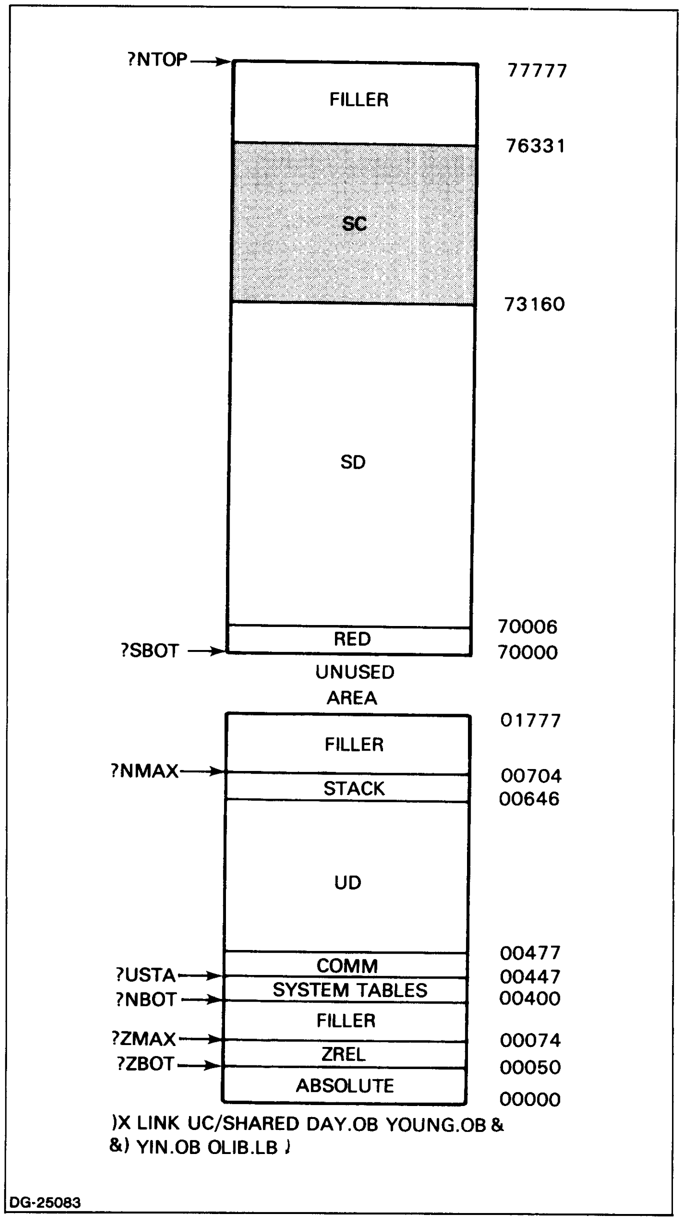


Figure 5-18. Example Showing Effect of partition/SHARED Switch

End of Chapter

Chapter 6
Introduction to the Library File Editor (LFE)

This chapter describes the AOS Library File Editor (LFE) — the utility you use to create, edit, and analyze library files. A library file, or more simply, a library, is a set of object modules preceded by a library start block and terminated by a library end block. (Refer to Appendix B for details on both of these object blocks.) By default, library filenames have the extension .LB.

LFE works in parallel with Link, since you usually use it to create library files for Link input. (Throughout this chapter, the term "library" or "library file" refers only to unshared libraries. AOS Link does not support shared libraries.)

Appendix E details all LFE errors and warnings.

Functional Overview

In addition to building libraries, LFE can also:

- analyze a library's contents
- delete, insert, or replace object modules in a library
- extract object modules from a library
- list the titles of all object modules in a library
- merge two or more libraries into a new library

LFE commands are called *function-letters*. The end of this chapter describes the LFE command line in detail.

Link loads an object module within a library if either of the following tests are true:

- If the forced load flag is set. (The library start block contains one forced load flag for each object module within a library. You can set or clear this flag with the LFE.)
- If the object module contains an entry symbol which matches an external symbol emitted by some other object module on the Link command line.

Therefore, if you want Link to load a particular object module from a library, you can either set the module's forced load flag (for unconditional loading) or define the appropriate entry symbols in the object module (for conditional loading). You can set or clear the forced load flag with function-letters I, M, N, or R. The next section of this chapter explains how Link resolves intermodular symbols (i.e., entry symbols and external symbols) in libraries.

Examples: Link and Libraries

Since Link scans a command line sequentially from left to right and does not rescan it, external symbols should be emitted before their entry symbol counterparts.

For instance, in Figure 6-1 object module TRIGR.OB defines external symbol COSIN, which object module TRIG3 defines as an entry symbol. Given the Link command line:

```
) X LINK TRIGR.OB MATH.LB ↲
```

Link loads object modules A and TRIG3 into the .PR file because external symbol COSIN precedes entry symbol COSIN. However, if the command line in Figure 6-1 were

```
) X LINK MATH.LB TRIGR.OB ↲
```

Link would load none of MATH.LB's object modules. Moreover, this command line produces a Link error, because Link will not be able to satisfy external symbol COSIN.

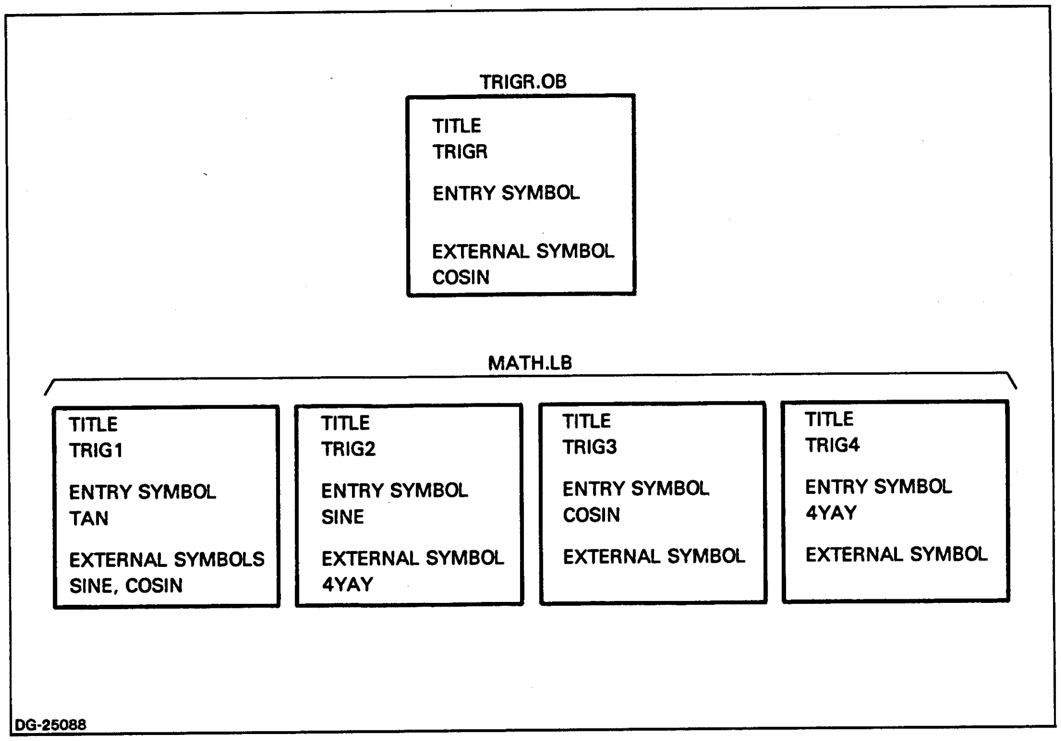


Figure 6-1. An Object Module and a Library

Link also scans object modules within libraries sequentially. For instance, suppose that TRIG1's forced load flag is set; therefore, Link automatically loads TRIG1. Then, Link satsifies external symbols SINE and COSIN, by loading object modules TRIG2 and TRIG3. Now Link must also satisfy any external symbols emitted by TRIG2 and TRIG3. TRIG2 defines external symbol 4YAY, and object module TRIG4 satisfies it. Therefore, Link must also load TRIG4.

Suppose that TRIG1's forced load flag is set, but TRIG1 came after TRIG2 in MATH.LB. In this case, Link would send out an error message because it would be unable to resolve external symbol SINE. Object module order within libraries can be critical.

It is sometimes desirable to put the same entry symbol in more than one library object module. In such a case, the program usually uses the repeated entry symbol for intermodular communication purposes, and uses a different symbol to choose which object module Link will load.

For instance, Figure 6-2 shows that entry symbol UM is defined twice by two different object modules in PERIPH.LB. Suppose that LP_1 and LP_2 are routines that serve the same purpose, but for different operating systems. If you issue the following Link command line:

```
) X LINK DECID.OB PERIPH.LB ↲
```

DECID.OB emits the external symbol that causes Link to load the appropriate object module. In this example, DECID emits external symbol AUM which forces Link to load LP_1. In either case, external symbol GET forces Link to load CALL_LP also. Now that Link has loaded the appropriate object modules, it can use UM as an intermodular communication symbol, so that CALL_LP can access datawords or routines in LP_1.

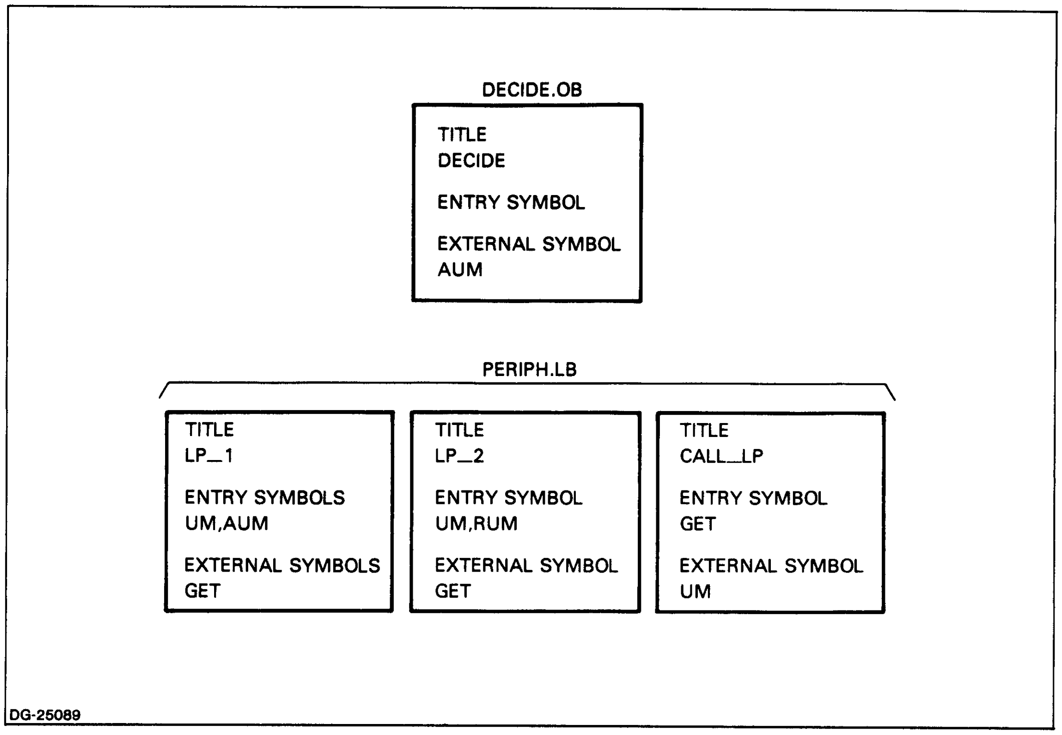


Figure 6-2. An Object Module and A Library

Operating the Library File Editor (LFE)

The remainder of this chapter details the LFE command lines that allow you to create, modify, or analyze a library file. You invoke the Library File Editor (LFE) by typing the AOS CLI command XEQ (abbreviated X) followed by LFE and the appropriate function-letter, LFE switches, and object files.

A function-letter is an LFE command. You use only one function-letter per command line. Table 6-1 categorizes the LFE function-letters. The rest of this chapter details each function-letter individually.

Table 6-1. LFE Function-letters

Category	Function-letter	Meaning

Analysis	A(nalyze)
	T(itle)
		analyze one or more object files list the title of each object module in a library, and list certain library start block information

Creation	M(erge)
	N(ew)
		create library from existing libraries and/or .OB files
		create library from existing .OB files and/or libraries

Editing	D(elete)
	I(nsert)
	R(eplace)
		delete one or more object modules from a library
		insert one or more object files into an existing library
		replace a library's object module with one or more object files

Extraction	(e)X(tract)	extract one or more object modules from a library

A

Analyzes one or more object files

Formats

X LFE/L<=*filename*></*globalswitch*...> A objectfile</*LOCAL*...>...

X LFE</*globalswitch*...> A filename/L objectfile</*LOCAL*...>...

X LFE</*globalswitch*...> A objectfile</*LOCAL*...>...

where:

/L<=*filename*> or filename/L	sends the analysis to filename. If you do not specify filename, LFE sends the analysis to @LIST. If you do not include /L on the command line, LFE sends the analysis to @OUTPUT.
global switch	is one or more of the following switches:

/F	puts the analysis of each object module on a separate page in the listing file.
/DECIMAL	sets the radix in the analysis to 10. The default output radix is 8.
/HEX	sets the radix in the analysis to 16. The default output radix is 8.
/XREF	generates an alphabetically sorted cross-referenced list of the following:

the names and symbol types of all symbols defined by the object file.

predefined and user-defined partitions defined by the object file, and the potential number of datawords the object file can contribute to them.

debugger files (.DS or .DL) the object file can contribute to, and the potential contribution to these files that the object file can make.

A	is the function-letter.
object file	is one or more library files or .OB files that you want LFE to analyze.
/LOCAL	is an optional switch attached to an object file. By default, LFE analysis ignores local symbols, but if you use this switch, LFE includes the object file's local symbols in the analysis.

Description

The A (analyze) function-letter directs LFE to analyze one or more libraries or .OB files. The default output analysis includes:

- the name and symbol types of all defined symbols.
- the number of datawords that each object module can contribute to each defined partition.
- the total number of datawords all object modules can contribute to all defined partitions.

- the size of each debugger block (debugger symbols block, debugger lines block, or lines title block) defined by each object module.
- the total number of datawords all object file(s) can contribute to either the .DS or .DL file.
- flags marking undefined, previously defined, and multiply defined symbols. These flags do not mark errors, rather they caution you about possible flaws in library logic which may cause problems when you link. An explanation of each flag follows:

  * MULTIPLY DEFINED SYMBOL — the same entry symbol has been defined in more than one object module.

  ^ PREVIOUSLY DEFINED SYMBOL — this external symbol appears after its matching entry symbol.

  ? UNDEFINED SYMBOL — LFE could not find a matching entry symbol within the object file for this external symbol.

To the right of all flagged symbols, LFE writes object module titles. LFE uses the following rules when it writes these titles:

- If the flagged symbol is an ENT, PENT, or ENTO, LFE lists the titles of all object modules referring to or defining the symbol.
- If the flagged symbol is an external symbol, LFE lists the titles of object modules defining this symbol.
- If the flagged symbol is neither of the above, LFE does not list the titles of object modules referring to or defining the symbol. If you want to find out which other object modules referred to or defined this symbol, use the /XREF switch.

For function-letter A, LFE initially assumes that unextended object files are libraries, and it searches for filenames with the .LB extension. If it can't find the appropriate .LB file, LFE then searches for a filename with the .OB extension. If LFE cannot find the appropriate .OB file, it searches for the unextended filename.

NOTE: In other versions of LFE, if you wanted to analyze a particular library object module, you had to include the name of the library and the title of the object module. This revision allows you to analyze .OB files and .LB files, but it does not accept object module titles as input.

Examples

1.
   ```
) X LFE/L=CONSULT A MATH.LB EUCLID.OB ⤶
```

2.
   ```
) X LFE A CONSULT/L MATH.LB EUCLID.OB ⤶
```

3.
   ```
) X LFE/DECIMAL/XREF/F A TRIG.LB CALC.LB ADD.OB ⤶
```

4.
   ```
) X LFE/L A SUB/LOCAL ⤶
```

Examples 1 and 2 are functionally identical. Both analyze MATH.LB and EUCLID.LB, and send the analysis to file CONSULT.

Example 3 analyzes TRIG.LB, CALC.LB and ADD.OB, and sends the analysis to @OUTPUT. Because of /XREF, the analysis will include a symbolic cross-reference. /DECIMAL forces LFE to write the analysis in base 10, and /F tells LFE to separate each object module's analysis with a form feed.

A (continued)

Example 4 analyzes only one object file. Because there is no extension name, LFE first tries to find SUB.LB in your working directory or in one of the directories on your search list. If it can't find SUB.LB, it searches for SUB.OB. If it can't find SUB.OB, it searches for SUB. If it can't find SUB, it sends out the error message "FILE DOES NOT EXIST SUB". Assume that SUB.LB is in your working directory. In this case, LFE sends the analysis to @LIST. /LOCAL forces LFE to include all local symbols in the analysis.

Sample Analysis

Assume that library 12.LB contains two object modules. This command line

```
) X LFE/L=@LPT A 12.LB ⤶
```

sends the following analysis to the line printer:

```
  TITLE   OECON
  COMM    BUFF    15
* PENT    SUPP   *TWO
  ENTRY   MACRO   TWO
  ENTRY   MICRO
? EXT     INT_1
? EXT     CROSS
^ EXT     BUFF
  PART    ZR      04
  PART    UC      22

  TITLE   TECON
  PENT    M1
* ENTRY   SUPP   *ONE
^ EXT     MACRO  ^ONE
  PART    ZR      06
  PART    SC      51

    TOTAL BUFF    15
    TOTAL SC      51
    TOTAL UC      22
    TOTAL ZR      12

  -------------------------

* MULTIPLY DEFINED SYMBOL
^ PREVIOUSLY DEFINED SYMBOL
? UNDEFINED SYMBOL
```

The analysis shows that library 12.LB contains two object modules titled OECON and TECON. The first object module can contribute four datawords to the predefined ZREL partition and 22 to the predefined Unshared Code partition; while the second module can contribute six datawords to ZREL and 51 datawords to the predefined Shared Code partition. The first object module also defined a 15 word common base partition named BUFF. The bottom of the analysis lists the partition totals for the entire library.

This analysis also contains all three warning flags. Both object modules defined SUPP as an entry symbol, so LFE marked this symbol with the * flag. Since LFE could not find a matching entry symbol for external symbols INT_I and CROSS, these symbols received the ? flag. Finally, because entry symbol MACRO precedes external symbol MACRO, LFE marks the external symbol with a ^ flag. To the right of all flagged symbols, LFE places the title of the object module defining the conflicting symbol.

D

Deletes one or more object modules from a library

Formats

```
X  LFE/I=library/O=library</DELETE>  D  title...
X  LFE</DELETE>  D  library/I  library/O  title...
X  LFE</DELETE>  D  library  library/O  title...
```

where:

library/I or /I=library	is the input library. If you use the third format (as demonstrated in Example 3) the /I is optional.
library/O or /O=library	is the output library (a reduced version of the input library).
/DELETE	is an optional switch which forces LFE to delete the output library before creating a new one. You should use this switch when you want the output library to have the same name as the input library.
D	is the function-letter
title	is the title(s) of one or more object modules in the input library that you want to delete.

Description

The D (Delete) function-letter deletes one or more object modules from the input library to produce a new library.

Note that a *title* is a symbol defined within the library start block. The title does not necessarily have the same name as the .OB file that it was originally contained in.

Examples

```
1.  ) X LFE  D  MATH.LB/I  NEWMATH.LB/O  LOG  EXP ↵
2.  ) X LFE/I=MATH.LB/O=NEWMATH.LB  D  LOG  EXP ↵
3.  ) X LFE  D  MATH.LB  NEWMATH.LB/O  LOG  EXP ↵
4.  ) X LFE/I=MATH.LB/O=MATH.LB  D  LOG  EXP ↵
5.  ) X LFE/I=MATH.LB/O=MATH.LB/DELETE  D  LOG  EXP ↵
```

The LFE command lines in examples 1, 2, and 3 are functionally equivalent. All three command lines create a new library, NEWMATH.LB, from the input library MATH.LB. NEWMATH.LB contains all of MATH.LB's object modules except LOG and EXP. MATH.LB is unchanged.

The LFE command line in example 4 causes the AOS system error FILE NAME ALREADY EXISTS because the input and output files have the same name. The /DELETE switch in example 5 prevents this error.

I

Inserts one or more object modules or libraries into a library

Formats

X LFE/I=library/O=library*</DELETE>* I title/A or title/B objectfile*</F or /C>...*

X LFE*</DELETE>* I library/I library/O title/A or title/B objectfile*</F or /C>...*

X LFE*</DELETE>* I library library/O title/A or title/B objectfile*</F or /C>...*

where:

library/I or /I=library	is the input library. If you use the third format (as demonstrated in Example 3) the /I is optional.
library/O or /O=library	is the output library (an expanded version of the input library containing the inserted object module(s)).
/DELETE	is an optional switch which forces LFE to delete the output library before creating a new one. You should use this switch when you want the output library to have the same name as the input library.
I	is the function-letter.
title	is the title of an object module in the input library. title must be followed by either /A or /B. title/A directs LFE to insert the object file(s) immediately after title in the library. title/B directs LFE to insert the object file(s) immediately before title in the library.
object file	the .OB file(s) and/or library(s) to be inserted. You may optionally append one of the following switches to an object file:

/F	turns on the forced load flag associated with this object file's object module(s).
/C	turns off the forced load flag associated with this object file's object module(s).

Description

The I (Insert) function-letter inserts one or more object files into the input library (library/I) to form a new output library (library/O). The input library remains unchanged.

Earlier versions of LFE accepted only .OB files for insertion; this revision allows you to insert any combination of .OB files and libraries.

Note the distinction between *title* and *object file*. A title is the name of an object module; while an object file is a file name. (For more information on object module titles, refer to the "Title Block" and "Library Start Block" sections in Appendix B.)

Examples

```
1.  ) X LFE/I=MATH.LB/O=NEWMATH.LB  I  LOG/A  LN.OB ↵
2.  ) X LFE  I  MATH.LB/I  NEWMATH.LB/O  LOG/A  LN.OB ↵
3.  ) X LFE  I  MATH.LB  NEWMATH.LB/O  LOG/A  LN.OB ↵
4.  ) X LFE/I=MATH.LB/O=NEWMATH.LB  I  EXP/B  CALC.LB/F ↵
```

5.
   ```
   ) X LFE/I=MATH.LB/O=NEWMATH.LB  I  SIN/A  ARCSIN.OB/C  COS/A & ↲
   &) ARCCOS.OB/C  TAN/A  ARCTAN.OB/C ↲
   ```

6.
   ```
   ) X LFE/I=MATH.LB/O=NEWMATH.LB  I  LOG/A  MATRIX.LB  JACOBIAN.OB ↲
   ```

The LFE command lines in examples 1, 2, and 3 produce the same result. First, LFE creates NEWMATH.LB by copying MATH.LB. Then, LFE inserts the object module contained in LN.OB after the object module titled LOG. LFE does not modify MATH.LB. Figure 6-3 illustrates this transformation.

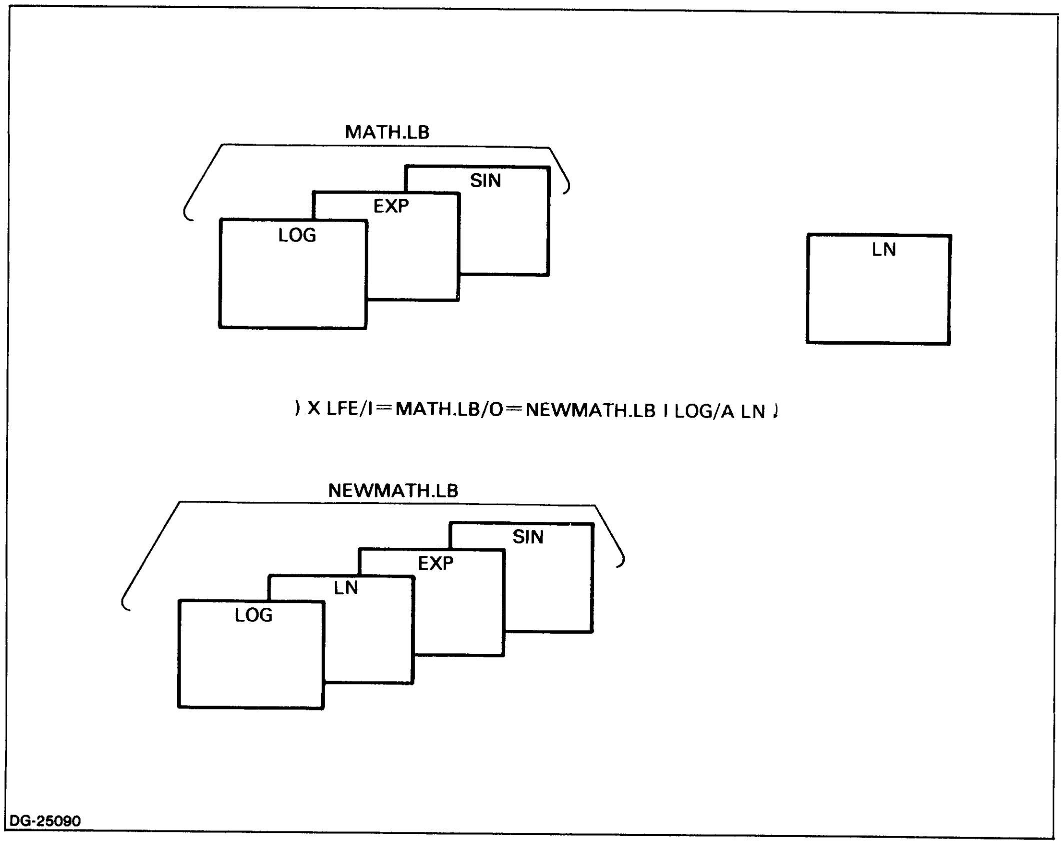


Figure 6-3. Inserting Object Modules Into a Library

I (continued)

The LFE command line in example 4 inserts all the object modules in CALC.LB just before the object module titled EXP. The /F switch appended to CALC.LB tells LFE to set the forced load flag of every object module in CALC.LB. In other words, whenever NEWMATH.LB appears on a Link command line, Link will automatically load every object module originally contained in CALC.LB.

Example 5 shows three separate insertions into NEWMATH.LB. First, LFE inserts the object module contained in ARCSIN.OB just after the module titled SIN. Second, LFE inserts the object module contained in ARCCOS.OB immediately after COS. Finally, LFE inserts the object module contained in ARCTAN.OB after TAN. The /C switch appended to the input .OB files forces LFE to clear the forced load flags associated with these object modules. In other words, Link will load the object modules originally contained in ARCSIN.OB, ARCCOS.OB, and ARCTAN.OB only if they satisfy an unresolved external symbol.

Example 6 creates NEWMATH.LB from MATH.LB and inserts the object modules in MATRIX.LB and JACOBIAN.OB immediately after the object module titled LOG. For function-letter I, when two or more object files follow a title, LFE inserts them in the same order that they had on the command line. Since MATRIX.LB precedes JACOBIAN.OB on the command line, all the object modules in MATRIX.LB will precede the object module in JACOBIAN.OB in NEWMATH.LB. If /B had been used instead of /A, then LFE would have inserted the object modules in MATRIX.LB and the object module in JACOBIAN.OB immediately before the object module titled LOG.

M & N

Merges Existing Libraries and/or .OB Files Into a New Library

Formats

X LFE<*/globalswitch...*> M library/O objectfile<*/F or /C*>...

X LFE<*/globalswitch...*> N library/O objectfile<*/F or /C*>...

X LFE/O=library<*/globalswitch...*> M objectfile<*/F or /C*>...

X LFE/O=library<*/globalswitch...*> N objectfile<*/F or /C*>...

where:

global switch	is one or more of the following switches:	
	/DELETE	is an optional switch which forces LFE to delete the output library before creating a new one. If you do not use this switch, and if the output filename already exists, LFE reports the error (to @OUTPUT) and terminates.
	/REV=aa.bb	forces LFE to generate an object module consisting only of a title block and end block. The object module title is always REVISION and the revision number, defined by the title block, is aa.bb. LFE then inserts the object module at the beginning of the library and sets its forced load flag.
M or N	are function-letters.	
library<*/O*> or /O=library	is the output library (the name you want the new library to have).	
object file	is either an .OB file or a library file. You may optionally append one of the following switches to an object file:	
	/F	turns on the forced load flag associated with this object file's object module(s).
	/C	turns off the forced load flag associated with this object file's object module(s).

Description

The M (Merge) and N (New) function-letters merge existing libraries and/or .OB files to form a new output library. The input libraries and .OB files remain unchanged.

M & N (continued)

NOTE: Function-letters M and N are functionally identical except in the way that LFE scans object files. When you use M, LFE first assumes that object files are library files; when you use N, LFE initially assumes that object files are .OB files. For instance, suppose that your working directory contains a library file named TRIG.LB and an .OB file named TRIG.OB. The following LFE command line merges MATH.LB and TRIG.LB:

```
) X LFE M NEW.LB/O MATH.LB TRIG ↵
```

However, because the function-letter N tells LFE to initially assume that all unextended filenames are .OB files, the following LFE command line merges MATH.LB and TRIG.OB:

```
) X LFE N NEW.LB/O MATH.LB TRIG ↵
```

Of course, by specifying extensions in the LFE command line, M and N become functionally identical.

Examples

```
1.  ) X LFE M MATH.LB/O TRIG.LB CALC.LB EUCLID.OB ↵
2.  ) X LFE/O=MATH.LB M TRIG.LB CALC.LB EUCLID.OB ↵
3.  ) X LFE N MATH.LB/O TRIG.LB CALC.LB EUCLID.OB ↵
4.  ) X LFE/O=MATH.LB N TRIG.LB CALC.LB EUCLID.OB ↵
5.  ) X LFE/O=MATH.LB/DELETE/REV=2.18 M TRIG.LB CALC.LB EUCLID.OB ↵
6.  ) X LFE/O=MATH.LB N TRIG.LB CALC.LB PLANE_TRUTHS.OB/F ↵
```

Examples 1, 2, 3, and 4 are functionally identical. All merge two libraries and one .OB file to form library MATH.LB. If MATH.LB had been in your working directory prior to the merger, LFE would have sent out an error message and terminated.

Example 5 deletes MATH.LB (if present) and merges TRIG.LB, CALC.LB, and EUCLID.OB to form a new MATH.LB. In addition, LFE inserts an object module titled REVISION (containing revision number 2.18) at the very beginning of MATH.LB.

Example 6 merges libraries TRIG.LB and CALC.LB along with the object module stored in PLANE_TRUTHS.OB to form library MATH.LB. The /F switch sets the forced load flag on the object module stored in PLANE_TRUTHS.OB. Therefore, if MATH.LB is included in a Link command line, Link will automatically load the object module originally stored in PLANE_TRUTHS.OB.

R

Replaces a library object module with an object module or a library

Formats

X LFE/I=library/O=library*</DELETE>* R title objectfile*</F or /C>...*

X LFE*</DELETE>* R library/I library/O title objectfile*</F or /C>...*

X LFE*</DELETE>* R library library/O title objectfile*</F or /C>...*

where:

library/I or /I=library	is the name of the input library file. If you use the third format (as demonstrated in Example 3) the /I is optional.
library/O or /O=library	is the name of the output library (a modified version of the input library).
/DELETE	is an optional switch which forces LFE to delete the output library before creating a new one. You should use this switch when you want the output library to have the same name as the input library.
R	is the function-letter.
title	is the title of the object module you want to replace.
object file	is one or more .OB file(s) or library(s) you want to replace the object module with. You may optionally append one of the following switches to an object file:

/F	turns on the forced load flag associated with this object file's object module(s).
/C	turns off the forced load flag associated with this object file's object module(s).

Description

The R (Replace) function-letter replaces object modules in the input library with new object modules to form a new library. This function-letter does not change the input library.

Earlier versions of LFE allowed you to replace an object module only with an .OB file. This version allows you to replace an object module with either an .OB file or a library file.

On the command line, there must be a one-to-one correspondence between titles and replacement object files.

R (continued)

Examples

1. `) X LFE R MATH.LB/I NEWMATH.LB/O EXP EXP2.OB/F ⤶`
2. `) X LFE/I=MATH.LB/O=NEWMATH.LB R EXP EXP2.OB/F ⤶`
3. `) X LFE R MATH.LB NEWMATH.LB/O EXP EXP2.OB/F ⤶`
4. `) X LFE/I=MATH.LB/O=NEWMATH.LB R SIN FOURIER.LB LOG SYN.LB COS COS2.OB ⤶`

Examples 1, 2, and 3 all do the same thing — they force LFE to copy NEWMATH.LB from MATH.LB and to replace the object module titled EXP with the object module stored in EXP2.OB. The /F switch tells LFE to set the forced load flag in NEWMATH.LB associated with EXP2.OB. MATH.LB is unaffected.

Example 4 replaces three object modules in MATH.LB to form a new, expanded library called NEWMATH.LB. First, LFE replaces the object module titled SIN with all the object modules in FOURIER.LB. Then, LFE replaces the object module titled LOG with all the object modules in SYN.LB. Finally, LFE replaces the object module titled COS with the object module stored in COS2.OB.

T

Lists the title of each object module in a library, and certain library start block information

Formats

```
X LFE/L<=filename></globalswitch...>  T  library...
X LFE</globalswitch...>  T  filename/L  library...
X LFE</globalswitch...>  T  library...
```

where:

/L=*filename* or filename/L	sends the analysis to filename. If you do not include a filename, LFE sends the output to @LIST. If you do not specify any list file, LFE sends the analysis to @OUTPUT.
global switch	is one or more of the following switches:
	/F — puts the analysis of each library on a separate page in the listing file.
	/DECIMAL — sets the listing radix to 10. The default output radix is 8.
	/HEX — sets the listing radix to 16. The default output radix is 8.
T	is the function-letter.
library	is the name of one or more library files to be analyzed.

Description

The T (title) function-letter causes LFE to list the following information:

- the title of every object module in the library
- the starting position of every object module's *OB descriptor.*
- the starting position of every object module's title block.
- the size (in words) of every object module.
- an asterisk in front of any object module in which the forced load flag is set.

The function-letter A produces a much more detailed analysis of a library than T; however, T is much faster. LFE scans an entire library to produce the full analysis A; however, LFE scans only the library start block to produce the title analysis T.

Examples

```
1.  ) X LFE/L=FAST  T  MATH.LB  CALC.LB ↵
2.  ) X LFE  T  FAST/L  MATH.LB  CALC.LB ↵
3.  ) X LFE/HEX/F  T  TRIG  GEOM  INTEGRAL ↵
```

Examples 1 and 2 are functionally identical. Both command lines produce title analysis of libraries MATH.LB and CALC.LB.

Example 3 produces title analysis of libraries TRIG.LB, GEOM.LB, and INTEGRAL.LB. Because of the /HEX switch, the listing will be in base 16; the /F switch ensures that the title analysis of each library starts on a fresh page.

X

Extract one or more object modules from a library

Formats

```
X LFE/I=library X title...
X LFE X library/I title...
X LFE X library title...
```

where:

X	is the function-letter.
library/I or /I=library	is the input library (from which you will extract the object module(s)). If you use the third format (as demonstrated in Example 3) the /I is optional.
title	is the title of one or more object modules you want to extract from the input library.

Description

The X (Extract) function-letter extracts one or more object modules from the input library and builds each into a freestanding .OB file. This command does not change the input library.

If your working directory already contains an .OB file of the same name, then the command fails and LFE sends out an error message.

Be careful not to confuse the title of an object module with the name of the .OB file it was originally stored in.

Examples

```
1.  ) X LFE/I=PRIMARY.LB X RED YELLOW BLUE ↵
2.  ) X LFE PRIMARY.LB/I X RED YELLOW BLUE ↵
3.  ) X LFE X PRIMARY.LB RED YELLOW BLUE ↵
```

Command lines 1, 2, and 3 are all functionally identical. All three command lines force LFE to extract copies of the object modules titled RED, YELLOW, and BLUE from library file PRIMARY.LB. LFE creates RED.OB, YELLOW.OB, and BLUE.OB, but does not change PRIMARY.LB in any way.

End of Chapter

Chapter 7
Developing RDOS and RTOS Programs on AOS

If you intend to execute a program file under RDOS or RTOS, and if you also have access to AOS, you may find it advantageous to develop your program under AOS. Using AOS Link to create an RDOS or RTOS .SV file is called cross-linking. Program development across two systems usually follows one of these two strategies:

A. 1. On RDOS, write source code.
   2. On RDOS, create .RB files by compiling or assembling your source code.
   3. Move .RB files from RDOS to AOS.
   4. On AOS, convert your .RB files to .OB files.
   5. On AOS, link your .OB files to form an RDOS or RTOS .SV file.
   6. Move your .SV file to RDOS or RTOS and execute it.

B. 1. On AOS, write source code.
   2. On AOS, compile or assemble the source code to produce .OB files.
   3. On AOS, link your .OB files to form an RDOS or RTOS .SV file.
   4. Move your .SV file to RDOS or RTOS and execute it.

These methods of program development currently apply only if source code is written in assembly language or DG/L language.

This chapter describes both methods of program development.

RDOS, RTOS, and NOVA Overview

This section briefly describes some aspects of RDOS and RTOS that are relevant to Link. A full description of these operating systems is outside the scope of this chapter, so we suggest that you refer to the following manuals:

- *Real-time Disk Operating System (RDOS) Reference Manual*
- *Extended Relocatable Loaders User's Manual*
- *Real-Time Operating System (RTOS) Reference Manual*
- *Programmer's Reference Series: NOVA® Line Computers*

RDOS

RDOS is a disk-based operating system that runs on a variety of mapped or unmapped NOVA or ECLIPSE computers. Because of the different hardware configurations possible, there are several different versions of RDOS.

As in AOS, Link places RDOS system tables at the beginning of NREL. The RDOS .SV file contains both a User Status Table and one or more Task Control Blocks. If the .SV file contains one or more overlay areas (also called overlay nodes) Link constructs an Overlay Directory. RDOS supports primitive overlay calls, but it does not support resource calls (?RCALL, ?KCALL, and ?RCHAIN); consequently, .SV files do not contain a Resource Handler Table (RHT). (Appendix D contains an illustration of the RDOS .SV file.)

RDOS does not support shared pages. Consequently, NREL extends from NBOT to NMAX. When you cross-link for RDOS, the shared/unshared attribute is irrelevant. Link gives the Shared Code and Unshared Code partitions different relocation bases (starting addresses); however, both of these partitions (despite their names) have the unshared attribute.

When you Link for a target system of AOS, Link sets the initial stack parameters; however, when you cross-link for a target system of RDOS, Link does not set the stack parameters. If you want to perform stack manipulation, you must set up initial stack parameters within your object modules. On Data General hardware, reserved storage locations define the stack parameters. Refer to the appropriate hardware manual for more information on stacks.

Like AOS, RDOS supports overlays. For more information, refer to the "RDOS and RTOS Overlays" section in this chapter.

RTOS

RTOS is a real-time memory-resident subset of the RDOS system. It is compatible with RDOS and has the same general design. Unlike RDOS, it cannot support text editors or the CLI. Therefore, you will have to develop RTOS programs on RDOS, AOS, or AOS/VS.

When you build RTOS programs, you load in the operating system routines needed for execution. In effect, an RTOS program contains both user code and the operating system itself. All the code for executing system calls is actually inside the .SV file. When an RTOS program issues a system call, no remapping takes place. Consequently, system calls execute much faster under RTOS than under RDOS or AOS.

The same rules for stack initialization and the shared attribute apply to RTOS and RDOS. (See the "RDOS" section.)

RTOS supports overlays but to a different extent then RDOS. (See the "RDOS and RTOS Overlays" section later in this chapter.)

NOVA Hardware

The NOVA hardware instruction set is a subset of the ECLIPSE hardware instruction set. So, a hardware instruction that works on a NOVA will work on an ECLIPSE; though, the converse is not always true.

Writing RDOS Source Code Under AOS

As mentioned earlier, there are two ways to create .OB files. One way is to generate .RB files under RDOS, move them over to AOS, and convert them to .OB files. The next section details this method. This section explains how you can write and compile (or assemble) RDOS source code while you are on AOS.

Programming in DG/L Language

DG/L language is ideally suited for cross-development because you can defer decisions about the target hardware until you compile. In other words, the same file of DG/L source code can be compiled and linked for either a NOVA or ECLIPSE computer. When you compile your source code, you specify the target hardware system. For instance, while under AOS, you write DG/L source code into a file called ACCOUNTS. If you intend to execute this program on a NOVA computer, you would use the /CODE=N switch:

```
) X DGL/CODE=N ACCOUNTS ⤶
```

If you intend to execute the same program on an ECLIPSE computer running RDOS, you would use the /CODE=E switch:

```
) X DGL/CODE=E ACCOUNTS ↲
```

(Your compile command line may require other switches. Refer to the *DG/L^{TM} Reference Manual* for further information.)

Regardless of the target hardware, either command line produces an .OB file. If the appropriate system library is on your AOS system, then you are ready to cross-link, and you should refer to the "Cross-Linking Command Line" section later in this chapter. If the appropriate system library is not on your AOS system, then read the "Preliminary Steps Before Linking" section later in this chapter.

Programming in Assembly Language

If you intend to assemble RDOS assembly language source code while under AOS, you first have to build an RDOS macroassembler symbol table in one of your AOS directories. This involves moving some source files from RDOS to AOS and using MASM to combine them into a symbol table.

A macroassembler symbol table associates standard Data General assembly language commands (such as LDA) with their machine language counterparts. The macroassembler refers to this table when it assembles source code. This symbol table is often called MASM.PS for AOS source code, and MAC.PS for RDOS source code.

To build a macroassembler symbol table that will allow you to assemble RDOS assembly language source code while under AOS, use the four-step procedure listed below. Remember, you need only do this procedure once:

1. Under RDOS, dump (usually to a tape) the files that comprise a MAC.PS for your target version of RDOS. The *RDOS/DOS Macroassembler User's Manual* explains which files you need. For instance, if you intend to execute programs on a mapped NOVA 830 running RDOS, use the following RDOS command line:

   ```
   R
   DUMP/V MT0:0 NBID.SR OSID.SR ODOS.SR PARU.SR ↲
   ```

2. Take the tape to an AOS system.
3. Under AOS, issue the RDOS LOAD command which translates RDOS files into AOS files. Because these are text files, use the /C switch to change carriage returns (acceptable on RDOS) to new lines (acceptable on AOS). For instance:

   ```
   ) X RDOS LOAD/V @MTA0:0 NBID.SR/C OSID.SR/C ODOS.SR/C PARU.SR/C ↲
   ```

4. Under AOS, use the MASM /S switch to build an AOS macroassembler symbol table. For instance:

   ```
   ) X MASM/S NBID.SR OSID.SR ODOS.SR PARU.SR ↲
   ```

By default, AOS assigns the new symbol table the name MASM.PS. Since the symbol table that handles AOS assembly language source code is also named MASM.PS, there could be some confusion. Either of the two methods listed below ensure that your RDOS assembly language source code gets assembled correctly:

- Do all your RDOS assembly language development in the same directory in which you stored the RDOS MASM.PS.
- Rename the RDOS MASM.PS and use the /PS=pathname switch on the MASM CLI command. For instance, if you intend to assemble file SHARK, which contains RDOS assembly language source code, you could issue the following commands:

  ```
  ) RENAME MASM.PS MAC.PS ↲
  ) X MASM/PS=:UDD:KNIFE:MAC.PS SHARK ↲
  ```

If you have the appropriate system library on AOS, then you are ready to link; refer to the "Cross-Linking Command Line" section in this chapter. If you do not have the proper system library on AOS, refer to the "Preliminary Steps Before Linking" section.

Writing RDOS Source Code Under RDOS

As an alternative to writing and compiling (or assembling) RDOS source code while on AOS, you may write and compile (or assemble) your RDOS source code while on RDOS. If you choose this strategy, you probably will not have to build a special macroassembler table (since it is probably already on your RDOS system). But there is also a disadvantage. If you compile or assemble under RDOS, but intend to Link while under AOS, then you will have to change your .RB files to .OB files. The RDOS-extended relocatable loader uses .RB files, but Link can use only .OB files. It is not difficult to change .RB files to .OB files, but you will have to do it for each object module that you intend to cross-link.

Assume that you are on RDOS, and that you have produced .RB files either by compiling DG/L source code or assembling RDOS assembly language source code. Follow these steps to convert your .RB files to .OB files:

1. Under RDOS, dump (probably to tape) every .RB file you intend to link. For instance:

   ```
   R
   DUMP/V MT0:0 TANH.RB SINH.RB ↵
   ```

2. Take the tape to an AOS system.
3. Under AOS, issue the RDOS LOAD command. This command converts RDOS files to AOS files. For instance:

   ```
   ) X RDOS LOAD/V @MTA0:0 (TANH.RB SINH.RB) ↵
   ```

4. Under AOS, issue the CONVERT command. This command translates .RB files into .OB files. For example, the following command line converts TANH.RB and SINH.RB into TANH.OB and SINH.OB:

   ```
   ) X CONVERT TANH SINH ↵
   ```

Preliminary Steps Before Linking

Link cannot create an executable program file unless it can scan the appropriate system library. To produce an executable AOS program file (.PR file), Link must scan the AOS system library URT.LB. Likewise, to produce an executable RDOS program file (.SV file), Link must scan the RDOS system library SYS.LB. Furthermore, to produce an executable RTOS program file, Link must scan RTOS1.LB, RTOS2.LB, and the RTOS trigger module (which is an .RB file created by RTOSGEN).

All AOS systems come with URT.LB; however, because RDOS and RTOS system libraries vary with hardware configurations, their system libraries do not come on AOS systems. Therefore, if you intend to cross-Link, you first have to bring the appropriate system library(s) to AOS from your target system.

SYS.LB, RTOS1.LB, and RTOS2.LB perform similar functions to URT.LB. That is, they all contain vital routines for system call resolution.

If you are programming in DG/L, Link must scan not only the appropriate system library, but also the appropriate DG/L language libraries. AOS DG/L always comes with the DG/L language libraries for AOS, RDOS on a NOVA computer, and RDOS on an ECLIPSE computer. Therefore, you do not have to transfer any DG/L language libraries to AOS, (though you may still have to bring the appropriate system library(s) over).

Transferring an RDOS Library to AOS

Since SYS.LB is such an essential library, we use it as an example of how to convert an RDOS library to an AOS library. Note that you can use the same procedure to convert other RDOS libraries to AOS libraries. Basically, the procedure involves breaking the RDOS library into its component .RBs, dumping the .RBs to tape, loading the .RBs onto AOS, converting the .RBs to .OBs, and recombining the .OBs into an AOS library.

Some of the steps in the following procedure are rather lengthy; however, you need only perform this procedure once:

1. Under RDOS, use the Library File Editor to list the title of every object module in SYS.LB. The RDOS LFE sends the list to an output file. For instance, you can send the listing to the line printer with the following command line:

   ```
   R
   LFE T SYS.LB $LPT/L ↵
   ```

   Keep the list handy. You will need it in steps 2 and 6. (For more information see the *Library File Editor User's Manual.*)

2. Under RDOS, use the Library File Editor to extract every object module on the list. You may find it easiest to spread the extraction process over several LFE command lines. (In this example, we are using the SYS.LB from a mapped NOVA 830.)

   ```
   R
   LFE X SYS.LB BFPKG IOPC OPCOM TRDOP TWROP OPMSG TOVLD OVREL ↵
   .
   (verification)
   .
   R
   LFE X SYS.LB OVKIL OVEX TOVLY QTSK DQTSK TQTAS AKILL ASUSP ARDY ↵
   .
   (verification)
   .
   ```

3. Under RDOS, dump the extracted .RB files to tape:

   ```
   R
   INIT MT0 ↵
   R
   DUMP/V MT0:0 -.RB ↵
   ```

   Note that this command dumps every file in your directory having the extension .RB. If you are not careful, you may accidentally dump files that were not part of SYS.LB.

4. Take the tape to an AOS system.

5. Under AOS, use the RDOS LOAD command to change RDOS files to AOS files:

   ```
   ) X RDOS LOAD/V @MTA0 ↵
   ```

6. Under AOS, use the CONVERT command to change .RB files to .OB files. There are several ways to do this. First, you can invoke CONVERT directly as in the following example:

```
) X CONVERT (BFPKG.RB IOPC.RB OPCOM.RB) ↵ (etc. for every .RB file.)
```

In the second method, you type the names of all the .RB files into a file, and then invoke CONVERT indirectly. For instance:

```
) CRE/I INDFL ↵
)) BFPKG IOPC OPCOM TRDOP TWROP OPMSG TOVLD OVREL OVKIL & ↵
)) OVEX TOVLY QTSK DQTSK TQTAS AKILL ASUSP ARDY TACAL XMT & ↵
)) XMTW REC IXMT TXMT TIDS TIDR TIDK TIDP TIDC PRI TPRI & ↵
)) DRSCH ERSCH TRSCH ABORT TABT SUSP TPEND MTUMO TSVRS SINGL & ↵
)) MULTI SMTSK TASK KILL KILAD MTCBM TMIN NSAC3 MDEBU DUMMY ↵
))) ↵
) X CONVERT ([INDFL]) ↵
```

In the above example, we created file INDFL containing the names of all .RB files that we wanted to convert to .OB files. (Note the CLI line continuation mark & appended to each line. Also note that the .RB files are in exactly the same order as in the original RDOS library.) Then, by enclosing INDFL in square brackets and enclosing the square brackets in parentheses we ensured that CONVERT would read INDFL as an indirect file.

7. At this point you should have 50 or so .OB files. To complete the cycle, you must recombine them to form an AOS library. There are two methods to this, both of which invoke the AOS LFE function-letters M or N. In both methods, the .OB files must be in exactly the same order as the .RB files were in the RDOS library. In the first method, you enter the names of all the .OB files directly. For instance:

```
) X LFE/O=SYS.LB M BFPKG IOPC OPCOM TRDOP TWROP OPMSG TOVLD & ↵
&) OVREL OVKIL OVEX TOVLY QTSK DQTSK TQTAS AKILL ASUSP ARDY & ↵
&) TACAL XMT XMTW REC IXMT TXMT TIDS TIDR TIDK TIDP TIDC PRI & ↵
&) TPRI DRSCH ERSCH TRSCH ABORT TABT SUSP TPEND MTUMO TSVRS & ↵
&) SINGL MULTI SMTSK TASK KILL KILAD MTCBM TMIN NSAC3 MDEBU & ↵
&) DUMMY ↵
```

In the other method, you use an indirect file as an argument to LFE. You can use the indirect file created in step 6. For instance:

```
) X LFE/O=SYS.LB M <[INDFL]> ↵
```

NOTE: Throughout the manual we use angle brackets < and > to denote optional arguments; however, in this case, the angle brackets themselves must appear on the command line. That is, the name of the indirect file must be enclosed by square brackets and the square brackets must be enclosed by angle brackets.

Cross-Link Command Line

Table 7-1 shows what you must have on AOS if you intend to cross-link for either RDOS or RTOS. For instance, if you intend to cross-link assembly language programs for RDOS, Link must have access to your .OB files, your libraries, and SYS.LB.

Table 7-1. Requirements For Executable .SV Files

	Assembly Language	DG/L
RDOS	your .OB files your libraries (optional) SYS.LB	your .OB files your libraries (optional) SYS.LB either [DGLIBN] for NOVA RDOS or [DGLIBE] for ECLIPSE RDOS
RTOS	your .OB files your libraries (optional) trigger module RTOS1.LB RTOS2.LB	your .OB files your libraries (optional) trigger module either edited [DGLIBN] for NOVA RDOS or edited [DGLIBE] for ECLIPSE RDOS

Assuming that Link has access to the proper program elements, you cross-link by using either the /SYS=RDOS or /SYS=RTOS switch on the Link command line. These switches tell Link to create either a program file that can execute on RDOS or a program file that can execute on RTOS. Note that some Link switches work differently (or not at all) when the /SYS switch is present. (See the "Incompatible Switches" section.)

If you intend to execute your program in the foreground of an unmapped RDOS system, you should read the "Cross-linking for Unmapped Foreground" section.

RDOS

The /SYS=RDOS switch is the only real difference between cross-linking assembly language programs for RDOS and linking assembly language programs for AOS. You should not explicitly name SYS.LB on the cross-link command line because Link scans it anyway. For instance, to create an RDOS .SV file from ONE.OB and TWO.OB (generated by a macroassembler), you would use the following cross-link command line:

```
) X LINK/SYS=RDOS ONE.OB TWO.OB ↵
```

If you are cross-linking DG/L programs, Link must scan the appropriate DG/L language libraries. Instead of entering the names of all these libraries, you can simply enter the name (enclosed in brackets) of an indirect file. This file contains the names of the appropriate DG/L language libraries. If you intend to execute your DG/L program under RDOS on NOVA hardware, the indirect file is called DGLIBN. If you intend to execute your DG/L program under RDOS on ECLIPSE hardware, the indirect file is called DGLIBE. For instance, suppose you want to cross-link MEAN.OB and MODE.OB (both generated by the DG/L compiler) and execute the resulting .SV file on a NOVA running RDOS. In that case, use the following cross-link command line:

```
) X LINK/SYS=RDOS/NSLS MEAN.OB MODE.OB [DGLIBN] ↵
```

Since SYS.LB is included in DGLIBN, you should use the /NSLS switch because it prevents Link from scanning SYS.LB twice.

RTOS

If you use the /SYS=RTOS switch, Link does not automatically scan a system library, so you will have to find some way of including RTOS1.LB and RTOS2.LB on the cross-link command line. In addition, Link must scan TRIGR.OB (the RTOS trigger module you created during RTOSGEN).

If you are cross-linking assembly language programs for RTOS, you must explicitly name TRIGR.OB, RTOS1.LB, and RTOS2.LB on the cross-link command line. For instance, if you are cross-linking ONE.OB and TWO.OB (created by a macroassembler) for RTOS, you could issue the following cross-link command line:

```
) X LINK/SYS=RTOS TRIGR.OB ONE.OB TWO.OB RTOS1.LB RTOS2.LB ↵
```

Before you can cross-link DG/L programs for RTOS, you have to create a special version of either DGLIBN or DGLIBE. To do this, you should first copy DGLIBN (if your target is NOVA hardware) or DGLIBE (if your target is ECLIPSE hardware). Then, edit (with a text editor) your copy by replacing SYS.LB with RTOS1.LB and RTOS2.LB.

For instance, suppose you intend to cross-link DG/L programs and execute the .SV file under RTOS on NOVA hardware. First, you copy DGLIBN to another file, say DGLIBNRT. Then, you edit DGLIBNRT so that it contains the names of the following libraries:

```
DGLNTASK  RTOS1.LB  RTOS2.LB  DGLNMATH.LB  DGLNOPSYS.LB  DGLNENV.LB
DGLNINIT.LB/I & ↵
```

In addition to your edited indirect file, you must also include TRIGR.OB and the .OB files created by the DG/L compiler on the cross-link command line. TRIGR.OB should be the first argument on the cross-link command line, then your .OB files, and finally the name of the indirect file.

For instance, suppose you generated MACRO.OB and MICRO.OB with the DG/L compiler. In this case, you use the following cross-link command line:

```
) X LINK/SYS=RTOS TRIGR.OB MACRO.OB MICRO.OB [DLIBNRT] ↵
```

Incompatible Switches

The following switches have no effect during cross-linking:

GLOBAL switches

/KTOP=n /NTOP=n /SRES=n	logical address space under RDOS and RTOS does not extend past ?NMAX (/KTOP, /NTOP) and neither system supports shared pages (/SRES).
/NRP /WRL	neither system supports resource calls (/WRL,/NRP).
/STACK=n	Link does not know what the target hardware configuration is (/STACK).

OV switch

/MULT=n	neither system supports multiple basic overlay areas.

PARTSYM switch

/SHARED	neither system supports shared pages.

Different Switch Meanings

The following switches have different meanings when cross-linking:

/DEBUG
/PRSYM
/UDF

The GLOBAL switch /DEBUG, which normally emits the external DEBUG for high-level language debugging, emits the same external under a cross-link, but for use by the RDOS debugger utility. When you use this switch, Link resolves the debug symbol and forces a copy of the RDOS Debugger into the .SV file. If the modules contain one or more debugger symbols blocks or debugger lines blocks, Link also produces a .DS or .DL file. RDOS does not use these files, so you may want to delete them. (See the "Debugger Symbols Block" and "Debugger Lines Block/Lines Title Block" sections in Appendix B. Also, see Appendix D for more information on the .DS and .DL files.)

If you intend to debug your RDOS or RTOS .SV file, you probably should include the /PRSYM switch in your cross-link command line. This switch will allow you to use on-line symbols during a debugging session.

When linking for AOS, the /UDF switch suppresses Link's default reading of URT.LB; when cross-linking for RDOS, the /UDF switch suppresses Link's default reading of SYS.LB. In either case, if you include the /UDF switch on a Link command line, Link produces a nonexecutable file.

Cross-Linking For Unmapped Foreground in RDOS

If you intend to execute an .SV file in the foreground of an unmapped RDOS system, then your cross-link command line should contain both the /ZBOT=n and the /NBOT=n switches.

By default, AOS Link builds system tables at the beginning of NREL (address 00400_8). In addition, by default, the relocation base for the ZREL partition is address 00050. However, if you intend to execute an .SV file in unmapped foreground, system tables cannot begin at 00400, and the predefined ZREL partition cannot begin at 00050.

The /NBOT=n switch forces Link to begin building the system tables at n. The /ZBOT=n switch forces Link to make ZREL's relocation base n. Therefore, these switches allow you to produce a .SV file that can execute in unmapped foreground.

For instance, Figure 7-1 shows an unmapped RDOS logical address space. Foreground takes up ZREL addresses 00225 through 00377. To ensure that your .SV file ends up in foreground, issue the following cross-link command line:

```
) X LINK/SYS=RDOS/ZBOT=225R8/NBOT=24000R8 FORMAT.OB DATA.OB ⮐
```

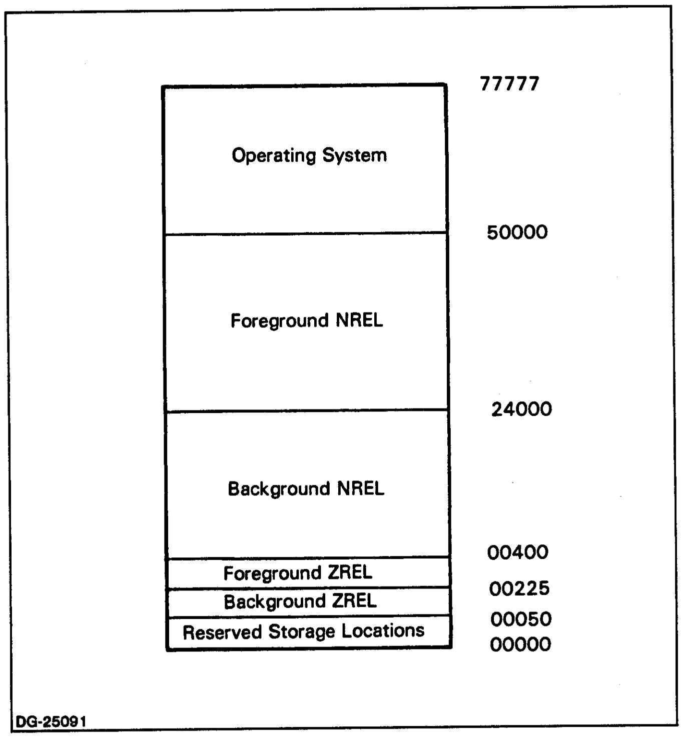


Figure 7-1. Typical Unmapped RDOS Logical Address Space

RDOS and RTOS Overlays

RDOS supports two kinds of overlays: conventional overlays (which are similar to AOS overlays) and virtual overlays. RTOS supports virtual overlays only.

Conventional Overlays

Conventional overlays reside in an .OL file on disk. If your cross-link command line requests conventional overlays, then Link builds *overlay areas* into your .SV file. Overlay areas, very similar to overlay areas in AOS .PR files, are gaps in the .SV file. When your .SV file issues a primitive overlay call, RDOS first swaps the appropriate overlay from the .OL file to main memory. Then, RDOS maps this overlay into the logical address space of the appropriate overlay area. RDOS does not support resource calls.

If you are cross-linking, you create an RDOS conventional overlay the same way you create overlays for an AOS .PR file. Thus, if you want your .SV file to contain overlay areas for conventional overlays, you must include overlay area delimiters (!* and *!) on the cross-link command line. Within overlay area delimiters, an overlay delimiter (!) separates overlays.

A pair of overlay area delimiters generates one overlay area if any of the object modules inside the overlay area delimiters contain datawords destined for the predefined Unshared Code partition. If the object modules can not contribute to the predefined Unshared Code partition, then Link will not create an overlay area.

For example:

```
) X LINK/SYS=RDOS A.OB B.OB !* C.OB ! D.OB ! E.OB F.OB *! ↵
```

The object modules in A.OB and B.OB contribute to the .SV file; while the object modules in C.OB, D.OB, E.OB, and F.OB can contribute to both the .SV file and the .OL file. Datawords in C.OB, D.OB, E.OB, and F.OB destined for the predefined Unshared Code partition will contribute to the .OL file; other datawords in C.OB, D.OB, E.OB, and F.OB will contribute to the .SV file. The .OL file will contain three overlays.

RDOS supports up to 124_{10} overlay areas within a .SV file. You can define up to 256_{10} overlays within each overlay area. All overlay areas are unshared.

Link pads conventional overlay areas to an integral multiple of a block (one block contains 400_8 or 256_{10} words). The size of an overlay area is determined by the largest overlay within it. For instance, if the largest overlay within a particular overlay area is 1100_8 words long, then Link must build an overlay area big enough to contain it, and it must be a multiple of 400_8. Therefore, Link builds an overlay area 1400_8 words long.

RDOS does not support multiple basic areas.

Virtual Overlays

Virtual overlays reside in extended memory — within physical memory, but outside your process's logical address space. When your process loads (with a primitive overlay call) a virtual overlay, the operating system maps it into a virtual overlay area in your logical address space.

To declare virtual overlays from the cross-link command line, append the /VIRTUAL switch to the !*. For instance:

```
) X LINK/SYS=RDOS   A.OB   B.OB   !*/VIRTUAL   C.OB   !   D.OB   E.OB   *! ↵
```

With one exception, Link uses the same rules for building virtual overlays and for building conventional overlays. The one exception is that virtual overlay areas are built in page multiples rather than block multiples. In other words, Link pads a virtual overlay area with zeros so that its length will equal a multiple of 2000_8 words. Link also sets the alignment of a virtual overlay area to 12_8. Because 2^{12} equals 2000_8, the first address of a virtual overlay area is always a multiple of 02000_8.

A computer running unmapped RDOS does not have any extended memory. Therefore, unmapped RDOS does not support virtual overlays.

NOTE: If you want to create a .SV file supporting both virtual and conventional overlays, the virtual overlays must precede the conventional overlays on the cross-Link command line. For instance, the following cross-Link command line is acceptable:

```
) X LINK/SYS=RDOS A.OB !*/VIRTUAL C.OB ! D.OB *! !* E.OB ! F.OB *! ↵
```

However, the next cross-Link command line is unacceptable:

```
) X LINK/SYS=RDOS A.OB !* E.OB ! F.OB *! !*/VIRTUAL C.OB ! D.OB *!
```

After Cross-linking

After cross-linking, you must bring the .SV file (and the .OL file, if present) over to your target system. This is a relatively easy procedure.

Use the following procedure to transfer an .SV file from AOS to RDOS:

1. Under AOS, dump your .SV file (and .OL file if present) with the RDOS DUMP command. This command changes the file(s) to an RDOS format. For instance, to dump ACCTS.SV, use the following command line:

   ```
   ) X RDOS DUMP/V @MTA0:0 ACCTS.SV ↵
   ```

2. Take the tape to an RDOS system.
3. Under RDOS, load the .SV file (and the .OL file, if present) into one of your directories. For instance:

   ```
   R
   INIT MT0 ↵
   R
   LOAD MT0:0 ACCTS.SV ↵
   R
   RELEASE MT0 ↵
   ```

4. You can now execute the program by typing the name of the .SV file. For instance:

   ```
   R
   ACCTS ↵
   ```

There are several methods for transferring an .SV file from AOS to RTOS. Here is one:

1. Copy an RTOS bootstrap loader onto file 0 of a tape. This bootstrap loader is stored on all AOS systems and has the pathname :TBOOT.

   ```
   ) COPY @MTA0:0 :TBOOT ↵
   ```

   By using the CLI command COPY rather then DUMP, AOS copies only the contents of the file. (The DUMP command forces AOS to copy the contents of the file plus AOS file information.)

2. COPY your .SV file to tape. The boot must precede the .SV file on tape. Do not put the .SV file on file 0 of the tape. For instance, suppose you want to put TAXES.SV on tape:

   ```
   ) COPY @MTA0:1 TAXES.SV ↵
   ```

3. Take the tape to an RTOS system.
4. Boot the tape up. (See the *Real-Time Operating System (RTOS) Reference Manual* for details on booting.) For example, on an ECLIPSE C/350, you could use the following procedure:

   a. Throw the STOP switch (on the front panel) to halt the CPU.

   b. Throw the RESET switch.

   c. Set the data switches to the device code for the tape drive (usually 100022).

   d. Throw the PROGRAM LOAD switch. This will load TBOOT into main memory. The following message should appear at console 0:

   ```
   FROM MT-0:
   ```

   TBOOT wants to know which tape file you stored the program on.

   e. Enter "1". TBOOT will load the contents of MT0:1 (TAXES.SV) into memory and transfer control to it.

End of Chapter

Appendix A
Link Error Message

Link reports errors as they occur, and sends them to one or possibly both of thе following files:

- the error file — the default error file is @OUTPUT, but you can override the default by using the /E=filename switch. If you use the /E=filename switch, Link opens *filename* even if there are no errors.
- the list file — if the Link command line contains either the /L or /L=filename switch, then Link also sends errors to this file.

There are two classes of errors: fatal and nonfatal. Upon detecting a fatal error, Link terminates immediately, signals FATAL LINK ERROR, and sends the appropriate error message to the list file or error file (assuming that the error was not due to Link's inability to open one of these files). Any of the following conditions can cause a fatal error:

- an unexpected error return from some system calls (primarily file system errors)
- user input errors (e.g., a null command line)
- Link internal errors (e.g., symbol table overflow, or a bad pointer into the symbol table)

Nonfatal errors do not force Link to terminate. When the utility encounters a nonfatal error, it writes the appropriate error message to the error file and list file. Then, after executing the command line, Link signals normal termination and returns LINK ERROR to the calling process. The following conditions can cause nonfatal errors:

- expected error returns from some system calls (e.g., object module does not exist, illegal pathname, etc.)
- user input errors (e.g., invalid object block, relocation errors)

The general format of the error messages is:

object module title BLOCK <number> PC <address> specific error

Since Link emits errors as it detects them, it may not include the PC in the error message (since it cannot calculate actual addresses until the second pass). The specific error may also contain information such as filenames, overwrite contents, symbol names, and so on. If Link detects an error that cannot be blamed on a particular object block, it prints out only the specific error.

Angle brackets <> indicate the paraphrase of a value Link supplies. For example, <switch name> means Link returns the name of the switch that caused the error.

NOTE: The following messages are not errors, but default output messages that Link passes to @OUTPUT:

Link revision <revision number> on <date> at <time>

and

<program name> file created

The remainder of this appendix contains all AOS Link errors as well as some AOS system errors that occur as a result of linking. (Occasionally, you might incur AOS errors not listed. If you do, refer to the *AOS Programmer's Manual.*) Since some messages change with each new release of Link, we advise you to check the current AOS Release Notice for any enhancements or changes.

Error Messages During Linking

Attempt to load data outside file limits

If your program contains both shared and unshared partitions, you tried to load datawords beyond ?NTOP; if your program contains only unshared partitions, you tried to load datawords beyond ?NMAX. This error probably resulted from using absolute code. To correct the error, either convert the absolute code to relocatable code or change the relocation base of the absolute code.

Attempt to load outside common partition <name>

A data block in one of your object modules is trying to store datawords (i.e., code or data) at an address in a common base partition; however, this address is outside the defined boundaries of the partition.

Block sequence number <block sequence number> out of sequence.

An object block in one of the input object modules does not have the correct sequence number. The sequence number is the second word in every object block. The title block of every object module must have sequence number 1. Subsequent object blocks must have a sequence number one greater than the preceding object block.

Circularly Defined Externals

An entry symbol's relocation value is defined in terms of itself. For instance, if an object module sets the value of entry symbol A with A=B+5, and another object module sets the value of entry symbol B with B=A+4, Link will not be able to resolve either A's or B's values. Link also emits this error when an entry symbol tries to define its value in terms of itself; e.g., A=A+4.

Command line is null[1]

You did not pass any arguments on the Link command line. Sometimes this error results from a missing & in a CLI macro. (Refer to the *Command Line Interpreter's User's Manual (AOS,AOS/VS)* for information about CLI macros.)

Control point directory max size exceeded[1 3]

Link is using a directory which does not have enough free space for the files it creates. To correct, send Link-generated files to a different directory with the /TEMP=pathname prefix or /O=pathname switches, or move your object files to a different directory before linking them, or create a larger control point directory.

Directory access denied[1 3]

Link does not have write access to this directory. Therefore, it cannot create its temporary files or its output files. To correct, either change the directory's ACL or set the /TEMP=pathname prefix and /O=pathname switches to a directory where you have write access.

End of file[3]

An object module does not have an end block, or an object module is zero length.

Extended relocation error using operation <number>[2]

An object block containing an extended relocation entry or extended dictionary relocation entry was not formatted correctly. (Refer to Appendix B for information on both kinds of entries.)

File does not exist <filename>[3]

Filename is not in your working directory or one of the directories on your search list. If the input filename has no extension, it means that Link found neither the unextended filename nor the filename with extension .OB. For instance, the error message "FILE DOES NOT EXIST THREE" tells you neither THREE nor THREE.OB are in your working directory or one of the directories on your search list.

Filename already exists[1] [3]

Link tried to create filename, but a permanent file with the same name already exists in the target directory. To prevent this error, turn OFF the permanence attribute of the appropriate .PR, .ST, .DS, .DL, .SV, or .OL file.

File too large for address space

Your program file contains more datawords (in NREL) than there are logical addresses to contain them. When this happens, the shared partitions of your program file will start to overwrite the unshared partitions. Wise use of overlay areas will prevent this from happening; use the /MAP or /MODMAP to gauge memory requirements. Then, refer to the "Command Line With Overlays" section in Chapter 5.

Invalid block size <block size>[2]

An object block in one of the input object modules was either smaller than 3 words or larger than 2000_8 words. Sometimes this error results when Link interprets a file that is not an object file as an object file.

Invalid block type <block number>[2]

An object block contains a block type (the right byte of the first word of every object block) which is not supported by this revision of Link. (See Appendix B for information on block types; check the AOS Release Notices for Link revision information.)

Invalid dictionary offset <number>[2]

An offset within a dictionary relocation entry does not point to an entry in the data block. Check the revision format of the data block.

Invalid or missing switch value /<switch name> = <switch value>

Link expected a switch value with this switch name and you did not provide one or you provided a faulty value. Tables 5-1 through 5-4 explain the proper syntax for switches.

Invalid overlay size <overlay size>

Either an overlay was larger than the entire logical address space, or, more probably, it was zero length. Link creates an overlay 000000 words long if any modules within overlay area delimiters fail to contribute to either the predefined Unshared Code or Shared Code partitions. For instance, suppose object module A.OB contributes only to Unshared Data and to ZREL. If you put A.OB within overlay area delimiters, Link will emit this message. Note, however, that your program file is still executable.

Invalid overlay syntax

You used the overlay area delimiter !* without using a matching *!, or you used *! without a matching !*. Unless object files are within properly paired overlay area delimiters, Link contributes them to the root. Chapter 5 details the proper overlay area syntax.

Invalid relocation operation <relocation operation>

You specified a relocation operation not supported by your revision of the Link utility.

Invalid switch /<switch name>

You used an illegal combination of switches. For instance, RDOS does not support multiple basic overlay areas, so the Link command line X LINK/SYS=RDOS A !*/MULT=2 B ! C *! would cause this error.

Invalid switch value /ULAST=<partition name>

You used a partition name as an argument to the /ULAST switch, but this partition was not defined by any object modules.

Link revision no. <revision number> conflicts with rev no. <revision number>[2]

A revision block in one of the input object modules contained a revision of Link newer than you are now using. In other words, one of your input object modules contained a revision number in a revision block that was higher then the revision number in the Link utility. To correct, get a newer revision of Link.

No start address has been specified

None of the object modules contributing to your program file had a possible starting address in their end block.

NOTE: The /START switch cannot correct this error. /START generates a starting address only if the object module it is attached to already has a possible starting address in its end block.

Not enough contiguous blocks[1 3]

A Link-generated file cannot fit on a device because the file's element size is greater than the number of free contiguous blocks on the device. Note that the element size of an RDOS overlay file is equal to the file size (in blocks); therefore, on a nearly filled disk, it is sometimes difficult to find enough free contiguous blocks to hold a large RDOS overlay file.

/NTOP,/KTOP switch conflict

You used both /NTOP=n and /KTOP=n switches in a Link command line. Although these switches produce similar results, you may not use both of them in the same Link command line.

Null switch /<switch name> = <switch value>

You used the GLOBAL switch /<ZR,UD,UC,SD,SC> = <ZR,UD,UC,SD,SC> incorrectly. You cannot put the same partition name on both sides of the equal sign. For instance, the Link command line X LINK/UC=UC A.OB causes this error.

NOTE: When using the equivalent OB switch, you can put the same partition name on both sides of the equal sign.

Overwrite previous <old contents> present <new contents>

Link tried to place two datawords (i.e., code or data) at the same address, or a data block tried to place datawords in a system table. In some cases overwriting is desirable, and there are a variety of ways to suppress these messages. (See the "Overwrite-with-message, Overwrite-without-Message Attribute" section in Chapter 4, and see "/OVER" in Table 5-1 and Table 5-2.) In other cases, overwriting is not desirable — check your absolute code for conflicting addresses. Many times, this error message goes hand in hand with the "File too large for address space" error message.

Partition <name> definition attribute <bits> conflict

Two object modules each defined a partition with the same name, but the partitions have different attributes. For instance, Link will emit this error if object module A and object module B both define a partition named RED, but A's RED has the unshared attribute and B's RED has the shared attribute. Link emits this error regardless of the object block used to generate the conflicting partitions. For example, even if a named common block generated A's RED, and a partition definition block generated B's RED, Link still would have emitted this error. To find out which partition attributes conflict, refer to the object block structure of the partition definition block (Figure B-20) and match up the bits in the error message with the bits in the first word of a partition descriptor. For instance, suppose the error message reports a "bit 14 conflict". Bit 14 in the first word of a partition descriptor is the shared/unshared attribute.

Reference to undefined symbol <external symbol>

Link could not resolve an *external symbol.* Make sure that you included all the necessary object files on the Link command line. Also, make sure that the appropriate system library is in either your working directory or one of the directories on your search list.

Reference to unknown external <external number>[2]

An object module contains an external number not defined by a predefined partition, partition definition block, external symbols block, or address information block. (See Chapter 4 for information about the external numbering scheme.)

Relocation overflow from <symbol name>

Link could not reference <symbol name> with the relocation operation given. The usual cause is that Link calculated a displacement (from the relocation operation) which could not fit into the displacement field. For instance, the assembly language command LDA 1,X,2 produces a 16-bit instruction with an 8-bit displacement field. If Link calculates a displacement greater than $+177_8$ or less than -200_8, Link will not be able to fit it into the displacement field. Sometimes an extended instruction (e.g., ELDA) prevents the error. This error also occurs when ZREL overflows (i.e., ?ZMAX exceeds ?NBOT) and you can no longer access ZREL locations with index mode 0 instructions. Finally, Link puts this error out if one of your datawords tries to access an address higher than 77777_8.

Resource Handler Table overflows 16K

The Resource Handler Table (RHT) went beyond address 37777_8. To reduce the size of the RHT, define fewer PENTs.

Symbol <name> invalid name length^2

An object block defined a name length greater than 32 or less than 1. Link truncates symbol names to 32 characters if they exceed the limit. Thus, entry symbol ACCOUNTS_RECEIVABLE_KANSAS_CITY_MO and external symbol ACCOUNTS_RECEIVABLE_KANSAS_CITY_KN match.

Symbol <name> is multiply defined

Two object modules are attempting to define an entry symbol with the same name.

NOTE: A library sometimes contains two object modules which both define the same entry symbol. This does not produce an error.

Symbol <symbol name> is not an accumulating symbol

Symbol name has a symbol type other than accumulating symbol, but you tried to redefine it as an accumulating symbol. There are two ways to cause this error. First, you included the PARTSYM switch symbol/VAL=n, but an object module already defined symbol as a different symbol type. Second, one object module defined symbol as an accumulating symbol and a second object module defined it as a different symbol type.

Symbol <symbol name> is not an EXTC

An object block specified relocation operation 20_8 for symbol name, but symbol name is not a chain external. (See the "External Symbols Block" section in Appendix B.)

Symbol <symbol name> is not a PENT

Link can't resolve a target word and call word because the target argument is in an overlay, but it is not a PENT. (See the "Resource Call Resolution" section in Chapter 4.)

Symbol <external symbol> is undefined

Link could not find a matching entry symbol for external symbol. The undefined external symbol was either emitted by your object modules or by Link itself. If one of your object modules emitted the external symbol, make sure that all the necessary object files are in the Link command line. If Link emitted the undefined external symbol, it means that Link is not reading the appropriate system library. Make sure that SYS.LB or URT.LB (depending on the target operating system) is either in your working directory or one of the directories on your search list.

NOTE: You will cause this error if you use the /NSLS switch without including the name of the appropriate system library on the Link command line.

Symbol table overflow

The .ST file is larger than 65536_{10} bytes. This does not affect the .PR file, but it means that debuggers using the .ST file may not be able to reference symbols stored beyond the 65536th byte.

Too many overlay areas

The root contains more overlay areas than the target operating system allows. For AOS, this number is 100_8; for RDOS, this number is 174_8.

Too many overlays in area <overlay area number>

There are more overlays in overlay area number than the target operating system allows. For AOS, this number is 1000_8; for RDOS this number is 400_8.

Unknown symbol type <name>$_1$

An entry block contained a symbol type other than 0 (ENT), 4 (ENTO), or 6 (PENT).

User may not define <entry symbol>

You defined an entry symbol which only Link can define. You will cause this error if your object modules or your Link command line tries to define any of the following: ?CLOC, ?CSZE, ?NMAX, ?SBOT, ?TTOP, ?TBOT, ?USTA, ?ZMAX, and USTAD.

User may not align <name>

You tried to align name, but since it was not a partition, Link could not align it. This error can also occur when an object module or switch attempts to increase a normal base partition's alignment factor, but other object modules have already contributed to the partition.

UST TCB or Overlay Directory overflows 1K

AOS and mapped versions of RDOS require the UST (User Status Table), TCBs (Task Control Blocks) and Overlay Directory to fit within the first page of the program file. Therefore, Link emits this error if it builds any part of one of these tables at an address above 01777_8. The size of these system tables depends on several factors; refer to the *AOS Programmer's Manual* (or Chapter 2 of this manual) for information on system tables. This restriction does not apply to .SV files that will run on unmapped RDOS.

ZREL overflow

?ZMAX is greater than 00377_8; your object modules contributed too many datawords (i.e., code or data) to the ZREL partition.

1	Fatal error.
2	If you are certain that this error was not due to a faulty command line or some other user error, please report it to your system manager or the appropriate Data General Corporation compiler/assembler support personnel.
3	AOS system error. Refer to the *AOS Programmer's Manual* for more information.

End of Appendix

Appendix B
Object Block Structure

Each object module consists of a series of two or more object blocks. An object block is a set of three or more words produced by a language processor as it interprets your source code. Currently, Link supports 21_{10} different types of object blocks.

This appendix shows the structure of the object blocks Link recognizes. If you're linking object modules produced by the AOS Macroassembler or by an AOS language compiler, you may find this chapter a useful reference. If you're linking object modules produced by your own language processor, the information in this chapter is essential, since the structure of the object blocks you create must conform to Link's expectations.

Object Block Restrictions

An object module need not contain all of the blocks listed in Table B-1. However, Link imposes several restrictions on the order of the object blocks in a module, as Table B-2 indicates.

Table B-1. Object Block Types

Object Block	Block Type	Object Block	Block Type
data block	0	accumulating symbol block	15
title block	1	debugger symbols block	16
end block	2	debugger lines block	17
unlabeled common block	3	lines title block	20
external symbols block	4	library end block	21
entry symbols block	5	(reserved)	22
local symbols block	6	partition definition block	23
library start block	7	(reserved)	24
address information block (AIB)	10	(reserved)	25
shared library block (not supported)	11	revision block	26
task block	12	filler block	27
limit block (not supported)	13	module revision block	30
named common block	14	alignment block	31

Table B-2. Object Block Order

Object Block	Order
title block	must be the first object block in every object module.
revision block	if used, must be the second object block in the object module.
address information block(s) (AIB)	if used, must appear before any external symbols block or partition definition block. Must also appear before any other object block that refers to a partition defined by it.
partition definition block(s)	if used, must appear after any AIBs. Must also appear before any object block that refers to a partition defined by it.
alignment block	if used, must appear after an AIB, partition definition block, or named common block that it refers to. Must also appear before any object block that refers to a partition defined by an AIB, partition definition block, or named common block.
external symbols block(s)	if used, must appear after all AIBs and partition definition blocks. Must also appear before any data block that refers to an external number defined by the external symbols block.
data block(s) local symbols block(s) entry symbols block(s) named common block(s) accumulating symbol block(s) debugger symbols block(s) debugger lines block(s)	if used, must appear after any external symbols block, AIB, or partition definition block that it refers to. In other words, these object blocks cannot use an external number until an external symbols block, AIB, or partition definition block defines it
lines title block	if used, must be the next to last object block in the object module. That is, the end block comes immediately after it.
end block	must be the last object block in every object module.

The object blocks in an object module must be contiguous, that is, there can be no extraneous information between the last word of one block and the first word in the next block. No object block, except the library start block, can contain more than 2000_8 words.

Standard Object Block Header

Except for the library start block, every object block begins with the standard three-word header shown in Figure B-1.

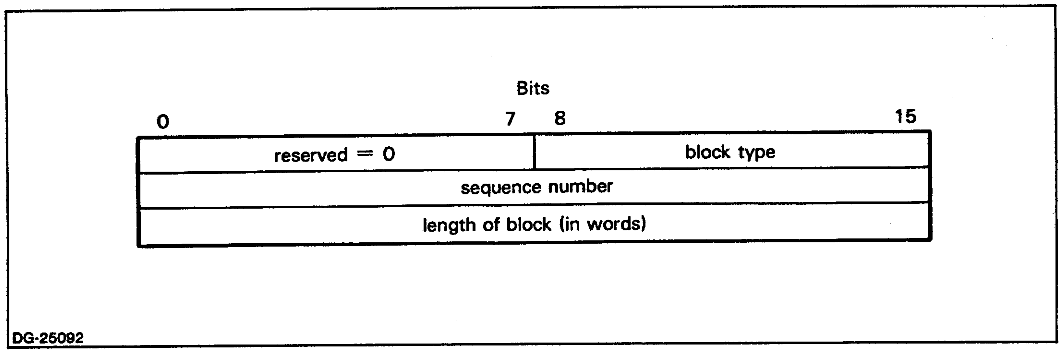


Figure B-1. Object Block Header

The left byte in the first header word must be set to 0 in all object blocks except the data block. (Refer to the "Data Block" section.)

Block type identifies the object block by its type number. Refer to Table B-1 for a list of the block types and the corresponding object blocks.

Sequence number, the second header word, indicates the position or sequence of the object block relative to the other object blocks in the object module. For example, the title block always has a sequence number of 1 because it must be the first object block in the object module. The next object block must contain sequence number 2, and so on.

Block length, the third header word, contains the total number of 16-bit words in this object block, including the header words. For example, the task block always contains four words, including the header, so its block length is 4.

Relocation Entries and Relocation Dictionary Entries

If you scan the object block formats shown in this appendix, you'll notice that many contain one or more *relocation entries*. Different object blocks use relocation entries differently, but in most cases, relocation entries help Link calculate symbol values.

Four object blocks contain *relocation dictionary entries*. Link uses a relocation dictionary entry to resolve the value of an individual dataword in a data block, debugger symbols block, debugger lines block, or lines title block.

Although their formats differ, both relocation dictionary entries and relocation entries consist of an offset, an external number, and a relocation operation.

The function of the offset depends on the object block it is contained in. It may determine the length of a common base partition, the value of a symbol, or the position of a group of datawords within a partition.

An external number usually tells Link which partition or external symbol this relocation entry or relocation dictionary entry refers to.

A relocation operation is a number corresponding to the function shown in Table B-3. Usually, there are two arguments to these functions: data and relocation base.

Data, in Table B-3, might be either the contents of a particular dataword or it might be the offset itself. It depends on the object block that defines the relocation entry or relocation dictionary entry.

The value of *relocation base* in Table B-3 depends on block type and on the external number in the relocation entry or relocation dictionary entry. If the external number refers to an external symbol, then the value of the relocation base is equal to the value of the matching entry symbol. If the external number refers to a normal base partition, then the value of the relocation base equals the partition relocation base plus the displacement of previous object modules' contributions to this partition. If the external number refers to a common base partition, then the relocation base equals the partition relocation base. (Chapter 4 describes the common base and normal base attributes.)

Table B-3. Relocation Operations

Relocation Operation	Function	Comments
0	absolute relocation: result = offset	
1	word relocation: result = relocation base + signed 16-bit data.	$-200000_8 <= result <= +177777_8$
2	byte relocation: (2 * relocation base + dataword)	$-200000_8 <= result <= +177777_8$
3	if dataword's bits 1 - 15 equal zero, then word relocation: result = relocation base + 16-bit data.	$-200000_8 <= result <= + 177777_8$
3	if dataword's bits 6 and 7 are zero, then index mode 0 16-bit displacement relocation: intermediate result = 8-bit relocation base + unsigned 8-bits data; final result = intermediate result + high 8-bits of data.	$0 <=$ intermediate result $<= +377_8$
3	if either bit 6 or bit 7 in the dataword is 1, then relative 16-bit displacement relocation: intermediate result = 8-bit relocation base + signed 8-bit data; final result = intermediate result + high 8-bits of data.	$-200_8 <=$ intermediate result $<= +177_8$
4	subtraction relocation: intermediate result = relocation base - signed 15-bit data; final result = intermediate result + 16-bit data (where bit 0 is the indirect bit)	$-200000_8 <=$ intermediate result $<= +177777_8$
5	primitive overlay relocation: dataword is replaced by a 0 in bit 0, the overlay area number in bits 1 through 6, and the overlay number in bits 7 through 15.	
6	multiplication relocation: result = relocation base * unsigned 16-bit data	
7	debugger symbols chain-link: dataword replaced by previous chain-link.	creates a reverse chain of addresses in the .DS file. (See the "Debugger Symbols Block" section in Appendix B.)
10	call word relocation:	resolves the call word of a resource call. (See Chapter 4 for details on resource call resolution.)
11	GREF (global reference): intermediate result = base + signed 15_bit data. final result = intermediate result + 16_bit data (including the indirect bit).	$-200000_8 <=$ intermediate result $<= +177777_8$
12	15-bit PC relative relocation: intermediate result = relocation base + signed 15-bit data minus the address that Link will place dataword at; final result = intermediate result + unsigned 16-bit data (including the indirect bit).	$-200000_8 <=$ intermediate result $<= +177777_8$
13	target relocation:	resolves the target word of a resource call. (See Chapter 4 for details on resource call resolution.)

(continues)

Table B-3. Relocation Operations

Relocation Operation	Function	Comments
14	16-bit PC relative relocation: result = relocation base + signed 16-bit (data) minus the address that Link will place dataword at.	$-200000_8 <= \text{result} <= +177777_8$
16	bit-field relocation dictionary entry designator (not an actual relocation operation).	
17	extended relocation designator (not an actual relocation operation).	
20	.PR file chain-link: dataword replaced by previous chain-link.	Creates a reverse chain of addresses in the .PR file; used to resolve EXTC external symbols.
21	Offset: result = 16-bit unsigned offset + 16-bit data	$-200000_8 <= \text{intermediate result} <= +177777_8$
22	Subtraction relocation (type 2): intermediate result = 15-bit signed data minus the relocation base; final result = 15-bit intermediate result + 16-bit data (including the indirect bit).	$-200000_8 <= \text{intermediate result} <= +177777_8$
23	Bit relocation: result = (4 * relocation base) + 16-bit data.	$0 <= \text{result} <= 200000_8$

(concluded)

Figures B-2 and B-3 show the format for standard relocation entries and standard relocation dictionary entries. These formats are quite similar, the one difference being that relocation dictionary entries always point to a specific dataword; while relocation entries can point to one dataword, an entire set of datawords, or no datawords at all.

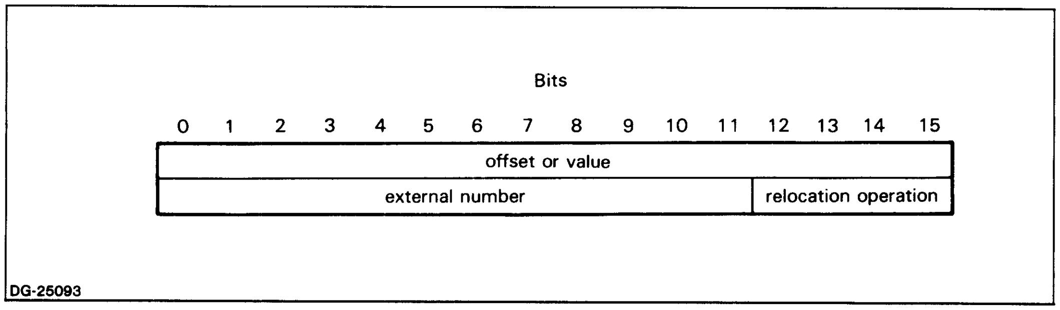


Figure B-2. Standard Relocation Entry

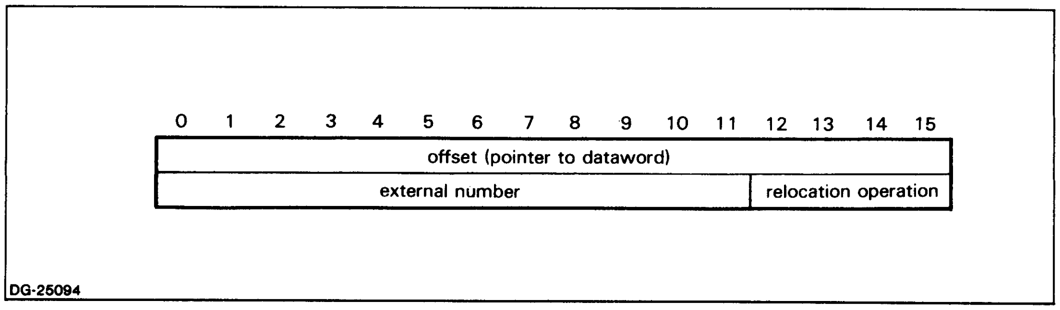


Figure B-3. Standard Relocation Dictionary Entry

Extended Formats

Relocation operations greater than 17_8 cannot fit within the 4 bits of the second word; however, they can fit in an extended format. Figures B-4 and B-5 illustrate extended relocation entries and extended relocation dictionary entries. You must use these formats when the relocation operation is greater than 17_8; you have the option of using these formats for relocation operations 17_8 or less.

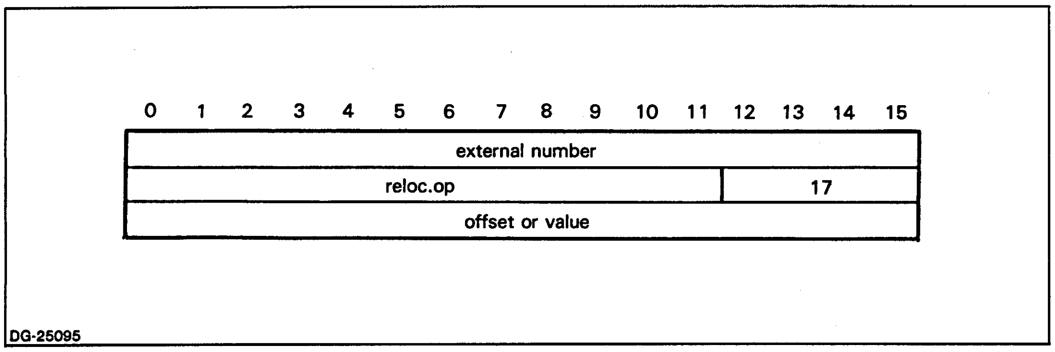


Figure B-4. Extended Relocation Entry

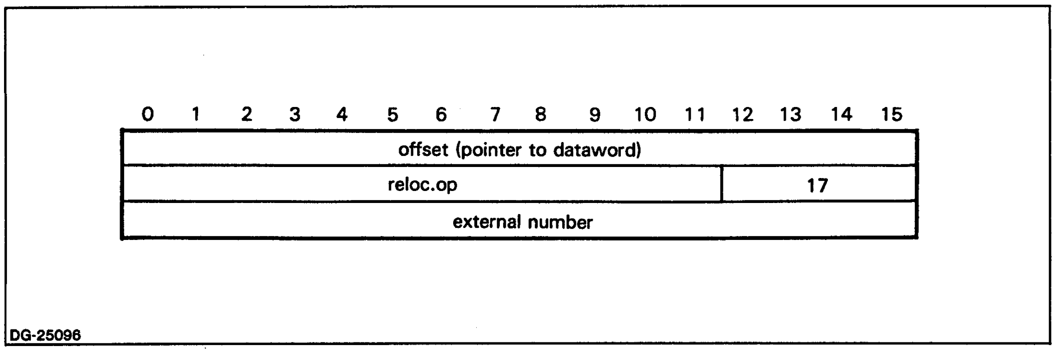


Figure B-5. Extended Relocation Dictionary Entry

Bit Field Relocation Dictionary Entry

A bit field relocation dictionary entry permits partial word resolution. A bit field relocation dictionary entry is similar to an extended relocation dictionary entry; however, Link uses an extended relocation dictionary entry to resolve a word or a byte, and Link uses a bit field relocation dictionary entry to resolve any contiguous string of bits in a dataword. You can use a bit field relocation dictionary entry in place of a standard or extended relocation dictionary entry.

As Figure B-6 shows, a bit field relocation dictionary entry consists of 4 words. The 16 in bits 12 through 15 of the second word tells Link that this is a bit field relocation dictionary entry. The offset, relocation operation, and external number in a bit field relocation dictionary entry perform the same function as in an extended relocation dictionary entry. In both types of entries, the offset points to a particular dataword which serves as "data" for the relocation operation. In an extended relocation dictionary entry, Link substitutes a new value for the dataword. In a bit field relocation dictionary entry, Link calculates a new value for the dataword, and then uses the *start of bit field* and and *width of bit field - 1* as a sort of bit mask. Link resolves only the bits within this range. Link does not resolve bits to the left of the start of bit field or to the right of the start of the bit field plus width of bit field - 1.

For instance, consider the 16-bit dataword 1111111111111111. Suppose that the offset points to this dataword and that the relocation operation calculates a new value for this dataword — 0000000000000000. If this is an extended relocation dictionary entry, Link would substitute 0000000000000000 for 1111111111111111. Assume though, that this is a bit field relocation dictionary entry with start of bit field equal to 6, and width of bit field - 1 equal to 3. Therefore, bits 6 through 9 are affected by the relocation operation, but bits 0 through 5 and 10 through 15 are not affected, and Link substitutes 1111110000111111 for 1111111111111111.

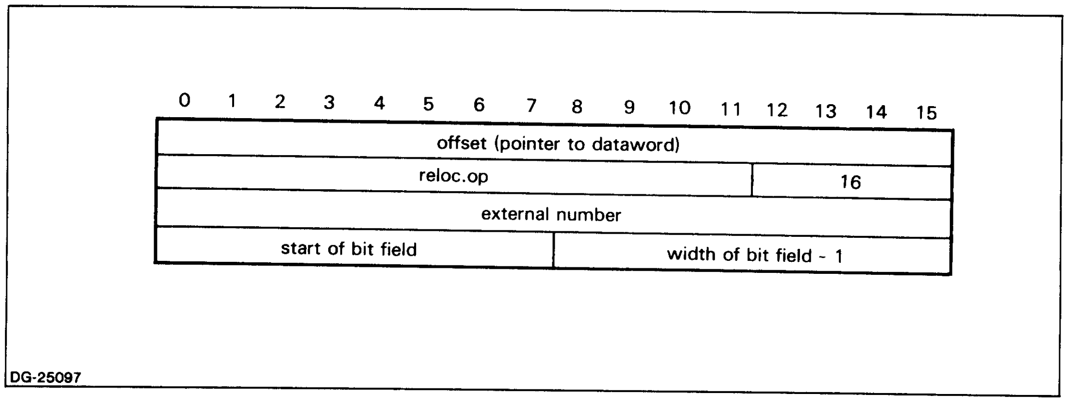


Figure B-6. Bit Field Relocation Dictionary Entry

Examining Object Blocks

You can not examine object blocks with text editors, but there are several utilities you can use:

DEDIT	the disk file editor allows you to examine and change the contents of an OB file. See the *AOS Debugger and File Editor User's Manual.*
DISPLAY	the CLI command that produces an octal dump of any file. See the *Command Line Interpreter User's Manual (AOS,AOS/VS)* for details.
LFE	the function-letter A analyzes a library file or .OB file. The listing summarizes partition contributions and names all entry and external symbols defined in an object file. See Chapter 6 of this manual for details.
Link	the global Link switch /OBPRINT produces an octal dump, grouped by object block, of every object module that Link loads into a program file or overlay file.

Note: In an OBPRINT listing, the title of the object module is spelled out immediately after the title block.

Object Block Rules

Many object blocks require the information shown below. Link expects object blocks to obey the following rules:

- *Byte pointers* identify the beginning of a byte string. Use the following formula to calculate the value of a byte pointer:

  byte pointer = (2 x number of words preceding the beginning of the byte string) + (1 if byte string begins on a right byte or 0 if byte string begins on a left byte)

  For example, a byte pointer with a value of 0 refers to the left byte of the first word in an object block; a value of 1 points to the right byte of the first word. A byte pointer with a value of 2 marks the left byte of the second word, and so on.

- *Symbol names* are packed two ASCII characters per word. Link truncates symbol names longer than 40_8 characters to 40_8 characters.

Examples Used in This Appendix

This appendix contains sample object blocks for some of the more complex object block formats. Basically, these sample object blocks show the OBPRINT listing you would get if you used the specifications described in the example. The numbers at the top and left side of these sample listings are to aid you in finding the proper word. Note that in our numbering scheme, the first word in an object block is Word 0.

Data Block

The data block contains the actual datawords that comprise a .PR or .OL file. In addition, the data block tells Link which partition these datawords will contribute to and where within this partition they should be allocated. Finally, data blocks help Link resolve the contents of individual datawords. Figure B-7 shows the structure of the data block.

Bit 4 of Word 0 is an overwrite flag. Overwriting means that Link reallocated the contents of a previously filled address. If this bit has a value of 1, Link suppresses overwrite messages. If this bit has a value of 0 and if overwriting occurs, Link sends an error message to the appropriate error files.

Bit 7 of Word 0 is a code flag. When bit 7 is set, it means that the data block contains code (as opposed to data). Link ignores this bit's value, but some utilities use it.

Each data block must contain one relocation entry. This relocation entry defines an external number usually corresponding to a partition, but occassionally corresponding to an external symbol (see pseudo-op .GLOC in the *AOS Macroassembler Reference Manual*). The offset and the relocation operation tell Link how the entire set of datawords in the data block should be allocated. Usually, the relocation operation is 1 — word relocation. If the external number refers to a common base partition, and the relocation operation is 1, Link allocates datawords beginning at the address equal to the partition relocation base plus the offset. If the external number refers to a normal base partition, the beginning allocation address depends on the amount of datawords other data blocks have already contributed to that partition. Assuming that the relocation operation is 1, Link begins allocating datawords at the partition relocation base plus the number of addresses already filled by contributions to the partition plus the offset.

For instance, suppose two object modules each contain one data block that contributes to a normal base partition. Suppose the first data block contributes 100_8 datawords to the partition. Therefore, the beginning allocation address for the second object block is the partition relocation base + 100_8. If the relocation operation of the second data block's relocation entry is 1, then Link begins allocating the second data block's datawords at the partition relocation base + 100_8 + offset.

After the relocation entry, the data block contains the datawords that will comprise the .PR (or .OL) file.

A data block may contain zero or more relocation dictionary entries. A relocation dictionary entry affects only the dataword that it points to. The external number in a relocation dictionary entry corresponds to either a partition or an external symbol. The relocation operation tells Link what it should do with the dataword once it sets a value for the partition or matching entry symbol.

Sometimes, a relocation dictionary entry helps Link resolve the displacement field of a memory reference instruction. For example, suppose your language processor chooses to generate the machine language equivalent of LDA 2,TAX, but symbol TAX is not defined within this object module. In this case, the language processor can write the machine language equivalent of LDA 2, and leave the displacement field 000 (i.e., 024000). Then, the language processor can put TAX in an external symbols block and generate a relocation dictionary entry pointing to this LDA command. Given the relocation operation within the relocation dictionary entry, Link will be able to write the correct displacement into the displacement field as soon as it knows TAX's .PR file address.

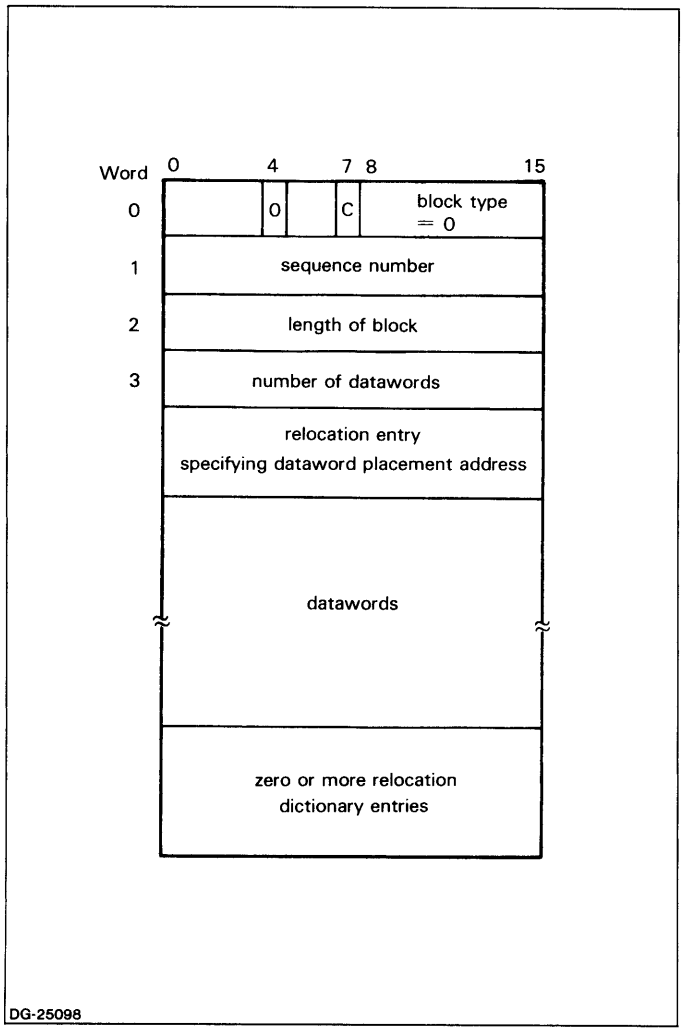


Figure B-7. Data Block

Data Block Formats 0, 1, and 2

Currently, Link defines three formats for data blocks: 0, 1, and 2. The distinction between Revision 0 and Revision 1 is very slight, the only difference is the way Link interprets the offset of a relocation dictionary entry. If the format revision is 0:

relocation dictionary entry offset = relocation entry offset + offset within this data block

If the format revision is 1:

relocation dictionary entry offset = offset within this data block

For example, consider an object module containing two data blocks, both of which contribute to the predefined Unshared Data partition. Assume that the first data block contributes 5 datawords to this partition. Therefore, you set the offset in the relocation entry of the second data block to 5. Suppose you want a relocation dictionary entry pointing to the third dataword of the second data block. If you are using a revision 0 data block, set the offset of the relocation dictionary entry to 7 (= 5 + 2). For a revision 1 data block, set the offset of the relocation dictionary entry to 2.

Format revision 2 is useful for initializing large arrays. Although AOS Link supports it, revision 2 is primarily used in 32-bit programs.

A revision 2 data block contains an extra two words between the "number of datawords" and the "relocation entry specifying dataword placement address". This double word is called the *repetition count*. As the name implies, the repetition count tells Link how many times it should repeat allocating the datawords in this data block. The first word defines the high-order portion of the repetition count and the second word (which can only be used by AOS/VS object modules) defines the low-order portion of the repetition count.

For instance, suppose you wanted to initialize the contents of each word in a 5000 word array to -1. To accomplish this, you can set the repeat count to 50 and generate 100 datawords containing -1. Since 50 * 100 = 5000, Link will initialize each word in the 5000 word array to -1.

Revision 0 is the default format. Use a revision block to change this default. (See the "Revision Block" section in this chapter.)

Example

```
          0      1      2      3      4      5      6      7
 0   000000 000003 000023 000011 000000 000101 137000 030000
10   133000 106470 000004 152400 006017 100012 000775 000001
20   000223 000004 000212
```

NOTE: Assume that this is a revision 0 data block and that this is the first data block in an object module.

The sample data block shown above defines 11 (see Word 3) datawords beginning at Word 6 and continuing through Word 16. Because the external number of the relocation entry (bits 0 through 11 of Word 5) is 4, these datawords will contribute to the predefined Unshared Code partition. Because the offset is 0 (see Word 4), Link will allocate these datawords at the object module relocation base for Unshared Data.

This data block defines two relocation dictionary entries. The first relocation dictionary entry occupies Words 17 and 20. It points to dataword 1 (which corresponds to Word 7) is an LDA instruction. Link uses relocation operation 3 to calculate the displacement field of the LDA instruction.

The second relocation dictionary entry occupies Words 21 and 22. It points to dataword 4 (which corresponds to Word 12) which is the displacement field of an EJSR instruction. Link uses relocation operation 12 to calculate the displacement field of the EJSR instruction.

Title Block

The title block defines the name and revision number of one object module. Figure B-8 shows the structure of the title block.

Bit 4 of Word 0 is a forced load flag. Link ignores this bit; however, LFE does not. If bit 4 is set, LFE also sets the forced load flag in the appropriate OB descriptor within the library start block. Note that force-load decisions made in title blocks are not irrevocable; you can set or clear the real forced load flag (the one in the library start block) when you use LFE.

Note that the sequence number (Word 1) must always be 1. That is, the title block must always be the first object block in an object module.

The left byte of Word 3 contains the major revision number, while the right byte contains the minor revision number. These numbers serve two purposes. First, you can use them as mnemonic markers. Second, they define a possible .PR file revision number.

Link stores the .PR file revision number in offset USTRV of the User Status Table (UST). The value of the .PR file revision number is equal to the value of the first valid revision number Link encounters in either a title block or a module revision block. (See the "Module Revision Block" section.) Any revision number other than 255.255 is valid.

The AOS CLI command REV also sets (or displays) the contents of offset USTRV.

The GLOBAL switch /REV=n overrides revision information in both the title block and the module revision block.

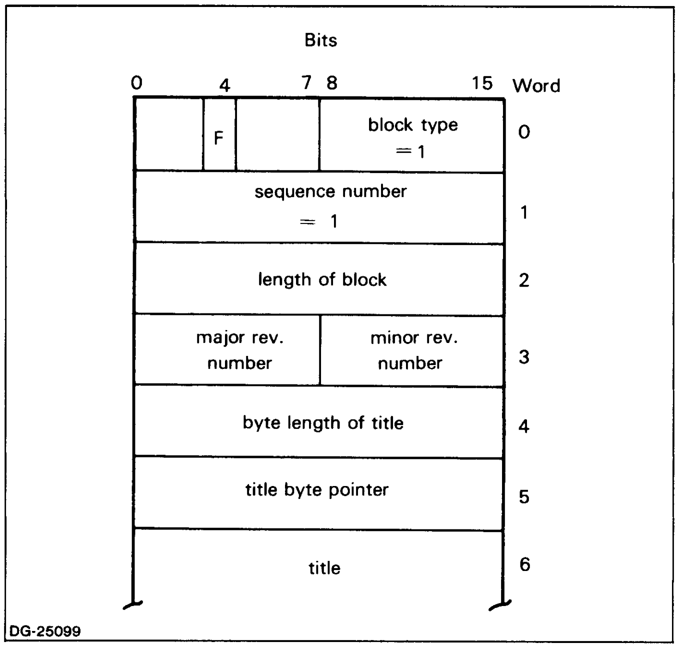


Figure B-8. Title Block

Sample Title Block

```
  0      1      2      3      4      5      6      7
000001 000001 000010 006420 000003 000014 051105 042000
```

The above listing shows a title block with the name "RED" and the revision number "13.16"

End Block

The end block defines the end of the object module and, optionally, a potential starting address for the .PR file. Figure B-9 shows the structure of the end block.

All object modules must contain one end block and it must be the last object block (i.e., it must have the highest block sequence number).

The relocation entry in an end block optionally defines a *possible starting address* for the .PR file. A *starting address* is the address of the first instruction in the program file that the CPU will execute at runtime.

If more than one object module defines a possible starting address, Link sets the .PR file starting address to the last valid possible starting address it encounters on the Link command line. If the OB switch /START is present, Link sets the .PR file starting address to the possible starting address of the object module the switch was attached to. If none of the object modules on the Link command line define a possible start address, then Link sends a "No start address has been specified" message to the appropriate error files.

If the OB switch /MAIN is present, Link sets the value of symbol .MAIN to the possible starting address of the object module.

Usually, the relocation operation for the end block relocation entry is "1". In this case, the offset points to a particular dataword in the object module. The possible starting address is simply the address in the .PR file that this particular dataword will occupy. If you do not want this object module to define a possible starting address, set the offset to a negative number.

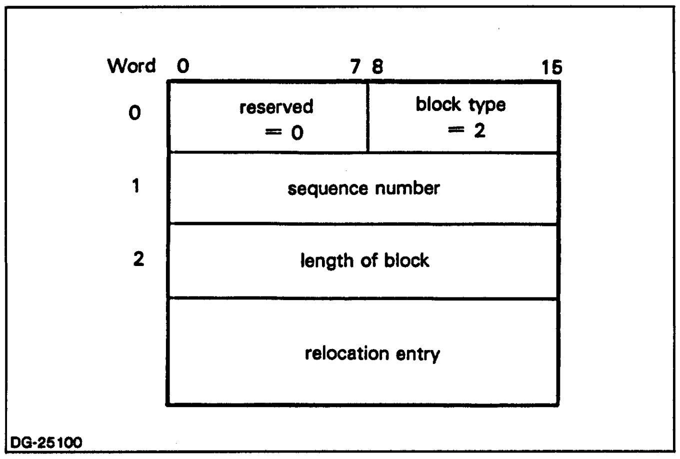


Figure B-9. End Block

Unlabeled Common Block

Link generates an unlabeled common area if any of the object modules on the Link command line contain an unlabeled common block. Under RDOS, the user allocates the unlabeled common area at runtime. Under AOS, Link always builds the unlabeled common area in unshared NREL just above the system tables.

Figure B-10 shows the format for the unlabeled common block. The size of an unlabeled common area is defined by the relocation entry's offset.

If Link encounters several unlabeled common blocks, it sets the size of the unlabeled common area to the largest offset it encounters.

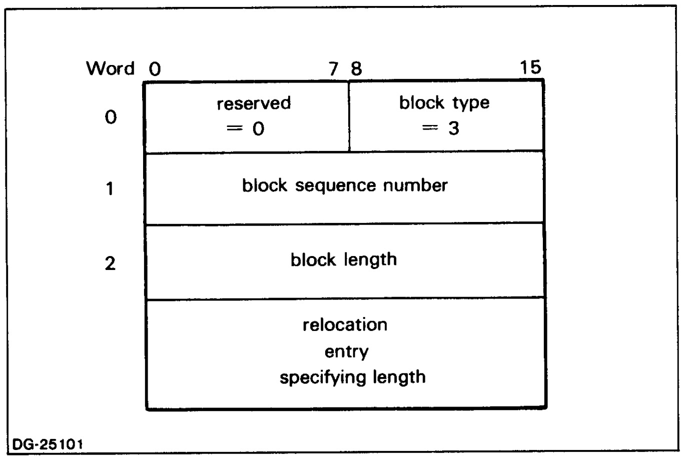


Figure B-10. Unlabeled Common Block

External Symbols Block

An external symbols block defines one or more external symbols. An external symbol is a symbol (e.g., label, variable) whose value is defined by another object module. Figure B-11 shows the structure of the external symbols block.

When you declare an external symbol, Link searches other object modules for an entry symbol with the same name. For instance, if you define symbol PUT in an external symbol block, Link searches in entry symbols blocks, named common blocks, and accumulating symbols blocks for an entry symbol named PUT. Link can resolve the external symbol if it can find a matching entry symbol; if Link can't find a match, it sends an error message to the appropriate files.

Figure B-11 shows the structure of the external symbols block. Notice that this object block contains no relocation information (i.e., no relocation operations, no external numbers). Instead, Link gets relocation information from the matching entry symbol.

Word 3 contains the number of external symbols defined in the object block. After Word 3, you must define one *external symbol descriptor* for each external symbol.

If external mnemonic in an external symbol descriptor has the value 0, Link considers the symbol a standard external symbol (symbol type EXT).

If external mnemonic in an external symbol descriptor has the value 1, Link considers it a *chain external* (symbol type EXTC). A chain external allows you to build a reverse symbol chain across two or more object modules. Link records the name and value of each chain external in the .ST file. If the symbol already exists in the .ST file, but has a different symbol type, then Link generates an error.

Link sets symbol ?LBOT equal to the value of the first chain external it encounters. When it encounters subsequent references to a chain external, Link creates a reverse chain of addresses. At execution time, a reference to a chain external is actually one or more indirect addressing instructions where the final instruction leads to the value of ?LBOT.

Chain externals must be referred to by relocation operation 20.

If external mnemonic in an external symbol descriptor has the value 2, Link considers the symbol a *suppressed external* (symbol type EXTS). Link allows suppressed externals to remain undefined without issuing an undefined symbol error. That is, suppressed externals do not cause Link errors when Link can't find a matching entry symbol.

Link assigns the value ?UNDF to all unresolved external symbols. ?UNDF has a default value of -1, but you may change this default.

For more information on ?UNDF and ?LBOT, see Appendix C.

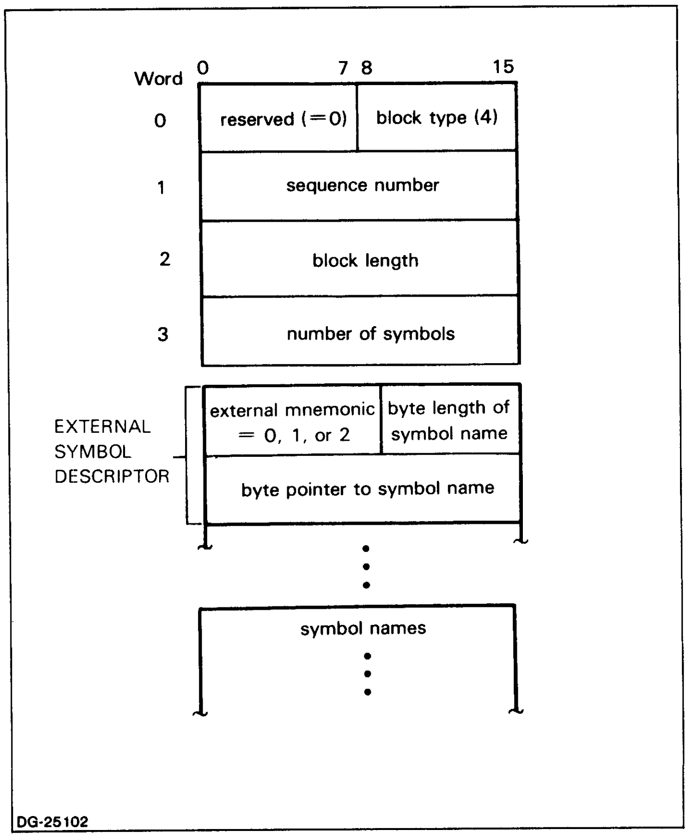

Word
0
7 8
15
0
reserved (=0)
block type (4)
1
sequence number
2
block length
3
number of symbols
EXTERNAL SYMBOL DESCRIPTOR
external mnemonic = 0, 1, or 2
byte length of symbol name
byte pointer to symbol name
symbol names
DG-25102

Figure B-11. External Symbols Block

Example

```
     0      1      2      3      4      5      6      7
0  000004 000003 000014 000002 000003 000020 001004 000023
10 050125 052123 044516 042400
```

The external symbols block shown above defines 2 (see Word 3) external symbols. The first external symbol is 3 bytes long (see right byte of Word 4) and begins at byte 20 (see Word 5). Byte 20 corresponds to the left byte of Word 10. Therefore, the first external symbol is named "PUT".

The second external symbol is a 4 character (see right byte of Word 6) chain external (see bit 6 of Word 6.) Byte 23 corresponds to the right byte of Word 11. Therefore, the second external symbol is named "SINE".

Entry Symbols Block

An entry symbols block defines one or more entry symbols. Entry symbols serve two purposes:

- Link matches external symbols with entry symbols to perform intermodular communication.
- Link stores the values of all entry symbols in the .ST file where you can use them to debug your program.

Figure B-12 shows the format of an entry symbols block.

Word 3 tells Link how many entry symbols the object block defines. You must build one "entry symbols descriptor" for each entry symbol defined in the object block.

"Symbol type" must contain one of the following numbers:

- 0 if the entry symbol is an ENT
- 4 if the entry symbol is an ENTO
- 6 if the entry symbol is a PENT

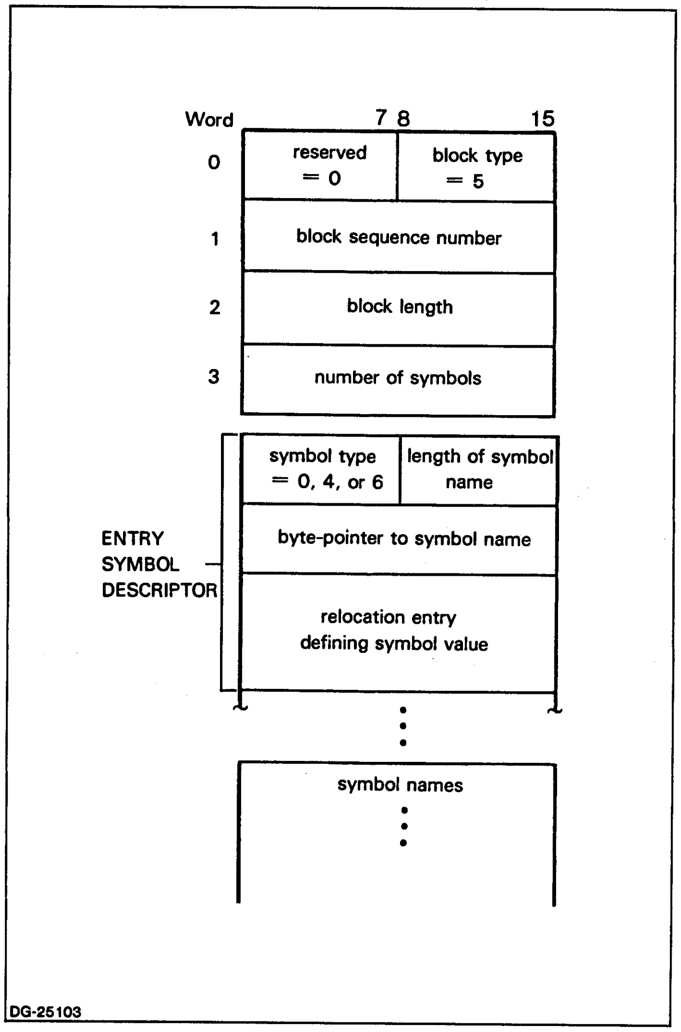


Figure B-12. Entry Symbols Block

(Refer to Appendix C for more information on symbol types; refer to Chapter 4 for details on ENTs and PENTs.)

Each entry symbol descriptor contains a relocation entry. This relocation entry allows Link to define a value for this symbol. Once Link knows the value of a symbol, it can resolve external symbols.

After the last entry symbol descriptor, an entry symbols block contains the names of all entry symbols packed two characters to a word.

Example

```
       0      1      2      3      4      5      6      7
 0   000005 000002 000021 000002 003007 000030 000000 000161
10   000004 000037 000015 000101 046501 054111 046525 046501
20   053122 043400
```

The sample entry symbols block shown above defines 2 entry symbols (see Word 3). The first entry symbol is a PENT (see left byte of Word 4). Its name is 7 characters long (see right byte of Word 4) and begins at byte 30 (see Word 5). Byte 30 is equivalent to the left byte of Word 14. Therefore, the symbol is named MAXIMUM. Words 6 and 7 form a standard relocation entry. Bits 0 through 11 of Word 7 contain the external number 7 (which corresponds to the predefined Shared Code partition.) The relocation operation (see bits 12 through 15 in Word 7) is 1 which tells Link to set MAXIMUM's value to the offset plus the beginning address of the object module's contributions to SC. Since the offset is 000000 (Word 6), Link will give MAXIMUM a value equal to the beginning address of the object module's contributions to SC.

The second entry symbol is a 4-character ENT named AVRG. Its external number is 4 (which corresponds to the predefined Unshared Code partition). The offset 15 and the relocation operation 1 tell Link that symbol AVRG is to have a value equal to 15 plus the beginning address of the object module's contributions to UC.

Local Symbols Block

A local symbols block defines one or more local symbols.

Like entry symbols, local symbols are often used in debugging; unlike entry symbols local symbols can't be used for intermodular communication. That is, Link does not satisfy external symbols with local symbols.

Although Link automatically puts entry symbols in the .ST file, it puts an object module's local symbols in the .ST file only if you affix the OB switch /LOCAL to the object module.

Aside from block type (which must be 6), there is only one difference between a local symbols block and an entry symbols block (detailed in Figure B-12). While an entry symbols block can have a symbol type of 0, 4, or 6, a local symbols block must have a symbol type of 0.

Library Start Block

A library consists of a library start block, a set of object modules, and a library end block. The library start block must be the first object block in a library. The Library File Editor generates this block. Figure B-13 shows the structure of the library start block.

There is a slight deviation from the ordinary object block header. Unlike other blocks, library start blocks can exceed 2000_8 words in length. Also, if bit 0 in Word 1 is set, Link reads the block length as a double word. That is, if bit 0 has a value of 1, then Word 1 contains the high-order portion of the block length and Word 2 the low-order.

A library start block contains one *object module descriptor* for each object module in the library. In essence, these object module descriptors allow Link to scan a library's vital details (entry symbols and forced load flags) without having to scan the entire library.

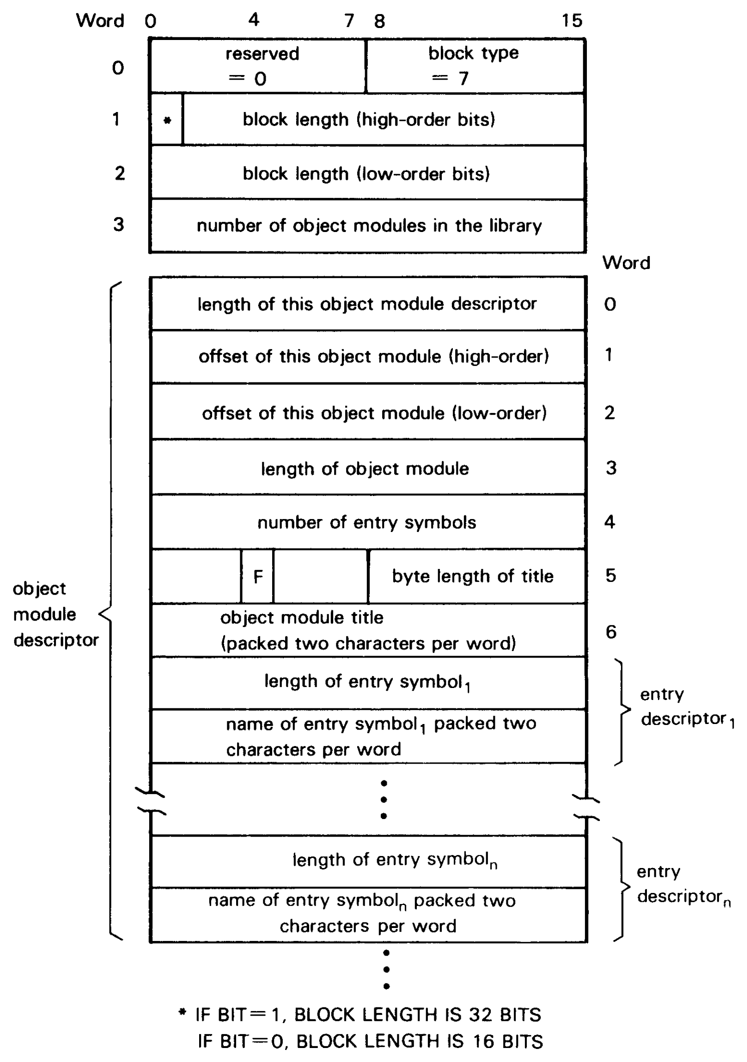
In an object module descriptor, Words 1 and 2 contain a number equal to the distance from the beginning of the first object module in the library. In other words, the offset of the first object module is always 0. The offset of the second object module is equal to the length of the first module. The offset of the third object module is equal to the sum of the lengths of the first and second object modules. Because a library may exceed 100000_8 words in length, the offset is always a double word.

Word 3 in an object module descriptor defines the length of the object module (in words).

Bit 5 of Word 5 in an object module descriptor is the forced load flag. If this bit has a value of 1, Link will automatically load the object module. If this bit has a value of 0, Link will load the object module only if one of its entry symbols resolves an outstanding external symbol. Note that there is a forced load flag in the title block as well; however, Link only reads the forced load flag in the object module descriptor. In other words, if the forced load flag in the title block is set (i.e., equal to 1), but the forced load flag in the corresponding object module descriptor is clear (i.e., equal to 0), Link will not force load this object module.

Object module descriptors define the title of the appropriate object module. "Title" means the symbol stored in the title block of the corresponding object module. The title must be packed two characters per word beginning at Word 6 of an object module descriptor.

A library start block must contain one *entry descriptor* for each entry symbol (ENT, PENT, or ENTO) defined in the corresponding object module. The first word of an entry descriptor defines the the number of characters in the symbol. Then comes the symbol itself packed two characters per word.


Word
0
4
7
8
15
0
reserved = 0
block type = 7
1
*
block length (high-order bits)
2
block length (low-order bits)
3
number of object modules in the library
Word
length of this object module descriptor
0
offset of this object module (high-order)
1
offset of this object module (low-order)
2
length of object module
3
number of entry symbols
4
F
byte length of title
5
object module title (packed two characters per word)
6
object module descriptor
length of entry symbol$_1$
name of entry symbol$_1$ packed two characters per word
entry descriptor$_1$
length of entry symbol$_n$
name of entry symbol$_n$ packed two characters per word
entry descriptor$_n$
* IF BIT=1, BLOCK LENGTH IS 32 BITS
IF BIT=0, BLOCK LENGTH IS 16 BITS

DG-25104

Figure B-13. Library Start Block

Example

```
         0      1      2      3      4      5      6      7
0     100007 000001 000040 000002 000015 000000 000000 000076
10    000001 004005 051105 040504 051400 000005 051524 040522
20    052000 000017 000000 000076 000047 000002 000004 043114
30    047527 000004 043111 041501 000005 050101 054522 046000
```

The library start block listed above was taken from a library generated by LFE. LFE created the library from 2 (see Word 3) object modules. The first object module descriptor begins at Word 4, and the second begins at Word 21. Words 5 and 6 contain the first object module's offset which is, of course, 0; while Word 7 contains the length of the object module (76). Logically, the second object module should have an offset of 76 (previous offset + length of previous object module) and words 22 and 23 verify that fact.

The first object module title is READS (see Words 12 and 13 and the left byte of Word 14) and it contains one entry symbol: START (see Words 16 and 17, and the left byte of Word 20). READS's forced load flag (bit 4 in Word 11) is set. Therefore, Link will automatically load READS if this library is included in a Link command line.

The second object module title is FLOW (see Words 27 and 30) and it contains two entry symbols: FICA (see Words 32 and 33) and PAYRL (see Words 35 and 36 and the left byte of Word 37). The force-load flag (bit 4 in Word 26) is clear. Therefore, if this library is included in a Link command line, Link will load FLOW only if FICA or PAYRL resolve an outstanding external symbol.

Address Information Block

You can use address information blocks (AIBs) to generate overlays and user-defined partitions. AIBs can only define partitions that have the same set of attributes as the predefined NREL partitions. Also, the external numbering scheme for AIBs is inflexible. Instead of using AIBs, we recommend generating user-defined partitions with the partition definition block, and we suggest that you set up overlays by using overlay area delimiters and overlay delimiters on your Link command line. (See Chapter 4 for a more detailed comparison of AIBs, partition definition blocks, and named common blocks.) Figure B-14 shows the structure of the address information block.

Word 3 of an AIB defines the number of user-defined partitions you want to set up with the AIB. You must then define a one word entry called a partition descriptor for each user-defined partition in Word 3. The right byte of a partition descriptor contains a predefined partition's external number. This number tells Link which attributes the partition defined by the AIB will contain. For instance, if this byte contains external number 4, Link defines a partition which has the same attributes as the predefined Unshared Code partition. You may use external numbers 4, 5, 6, or 7 only.

The left byte of each partition descriptor defines the external number that Link will use to establish a relocation base for this partition. Chapter 4 describes the external numbering scheme and how AIBs fit into it.

After the last partition descriptor, an AIB may optionally define one or more overlays. If you do not want to define overlays with this AIB, set the number of overlay descriptors to 0. If you place a nonzero number here, you must set up a corresponding number of overlay descriptors. Each overlay descriptor consists of two words.

The right byte of the first word of an overlay descriptor is an external number which tells Link which predefined partition the overlay should inherit its attributes from. Since partitions with the data attribute cannot contribute to an overlay area, you must use either 4 (for the predefined Unshared Code partition) or 7 (for the predefined Shared Code partition).

The left byte of the first word of an overlay descriptor is the external number that Link will use to define this overlay's relocation base.

The second word of an overlay descriptor defines the relative overlay area number and overlay number for the code associated with the external number. The first overlay descriptor in each AIB must define an overlay area number of 0 and an overlay number of 0. Subsequent overlay descriptors must be numbered consecutively. The overlay information you use in these AIBs allows Link to set up overlays, but Link treats these as local values. That is, when Link builds overlays in the .OL file, it does not necessarily use the same set of overlay area numbers and overlay numbers that you assigned in the AIB.

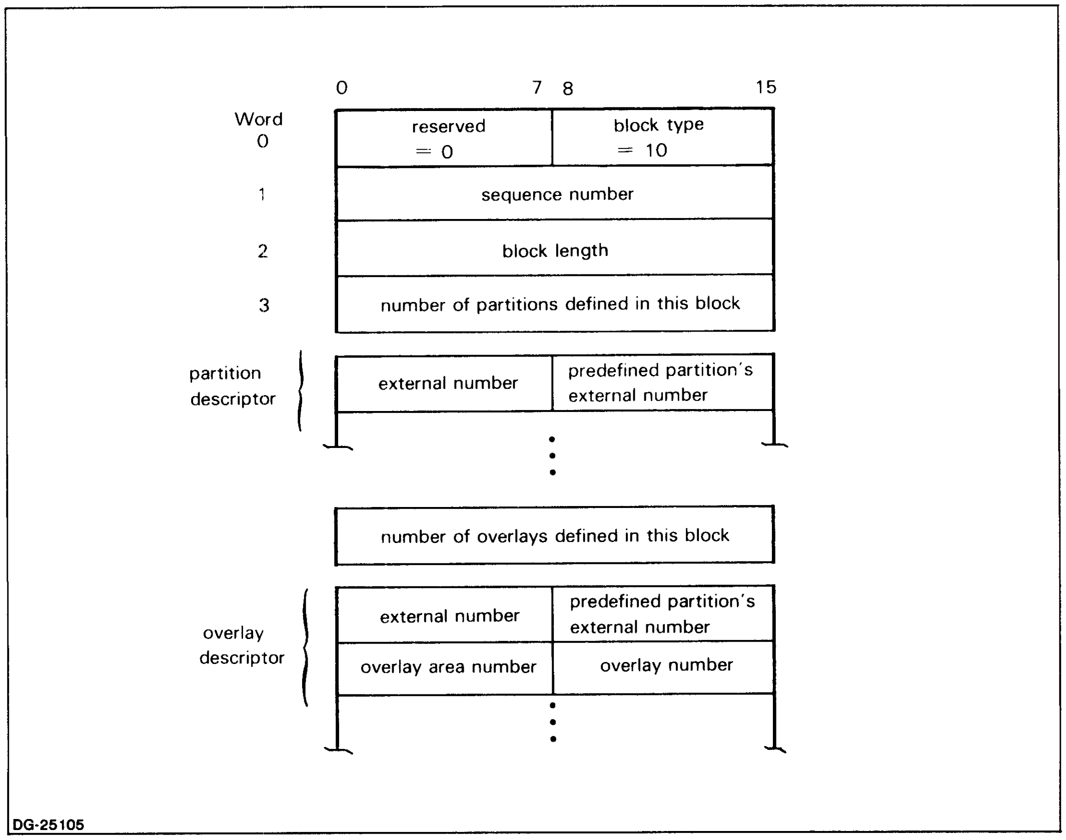


Figure B-14. Address Information Block

Example

```
     0      1      2      3      4      5      6      7
0   000010 000020 000015 000002 004004 004405 000003 005004
10  000000 005404 000001 006007 000400
```

Shown above is a sample address information block. Word 3 indicates that this AIB defines 2 partitions. The first partition will have the same attributes as the predefined Unshared Code partition (right byte of Word 4) and the second partition will have the same attributes as the predefined Shared Data partition (right byte of Word 5).

Word 6 indicates that this AIB defines 3 overlays. The overlay area number of 0 (see left bytes of Words 10 and 12), indicates that the first two overlays contribute to the same overlay area. The third overlay descriptor defines an overlay area number of 1 (see left byte of Word 14), which means that this overlay contributes to a second overlay area. The first overlay area is unshared because the right byte of Word 7 and Word 11 contains external number 4 (which corresponds to the predefined Unshared Code partition). The second overlay area is shared because the right byte of Word 13 contains a 7 (which corresponds to the predefined Shared Code partition.)

The left bytes of Words 4, 5, 7, 11, and 13 define external numbers beginning with 10 and increasing sequentially to 14.

Task block

Link uses task blocks and/or the GLOBAL switch /TASKS=n to determine the maximum number of tasks a process may initiate. Link stores this number in the User Status Table. Figure B-15 shows the structure of the task block.

For a target system of AOS or RDOS, Link builds one Task Control Block (TCB) for each potential task. For a target system of RTOS, Link does not build TCBs since they are built at runtime. (See Chapter 2 or the *AOS Programmer's Manual* for more information on AOS system tables.)

Link always uses the maximum task specification, whether you define it with task blocks or with the GLOBAL switch /TASKS=n in the Link command line. For example, given three object modules with task block specifications of 2, 3, and 4, Link sets the maximum number of tasks to 4. If the Link command line for the same object modules specifies /TASKS=5, Link sets the maximum number of tasks to 5.

If you do not use the /TASKS=n switch, and none of the input object modules contains a task block, then Link assumes that the .PR file will support only one task, the initial task.

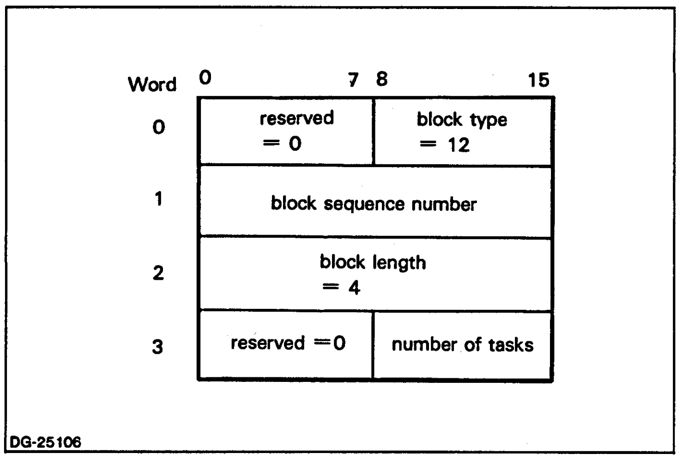


Figure B-15. Task Block

Named Common Block

Named common blocks force Link to generate partitions with the common base attribute. If two or more object modules contain named common blocks with the same name, Link builds only one common base partition having this name. Otherwise, Link generates one common base partition for each (uniquely named) named common block it encounters. Figure B-16 shows the structure of the named common block.

Each named common block contains two relocation entries. The first relocation entry defines the size (in words) of the partition. Link ignores the relocation operation and external number of this relocation entry.

The second relocation entry defines the partition that the common base partition will inherit its attributes from. For instance, if this relocation entry defines external number 6, Link creates a partition with the almost the same attributes as the predefined Unshared Data partition. The only difference being that this partition will have the common base rather than the normal base attribute. The external number in this relocation entry must correspond to one of the NREL predefined partitions; that is, the external number must be 4, 5, 6, or 7. Link ignores both the offset and relocation operation of the second relocation entry.

If two or more named common blocks define the same name, their external numbers must match; however, their sizes (offsets) may be different. If their sizes are different, Link sets the size of the partition equal to the largest of the offsets. For instance, suppose two object modules each contain named common blocks. Further assume that each named common block defines the partition name MATRXA. If the first named common block defines a length of 100 and the second a length of 150, then Link builds the partition 150 words long.

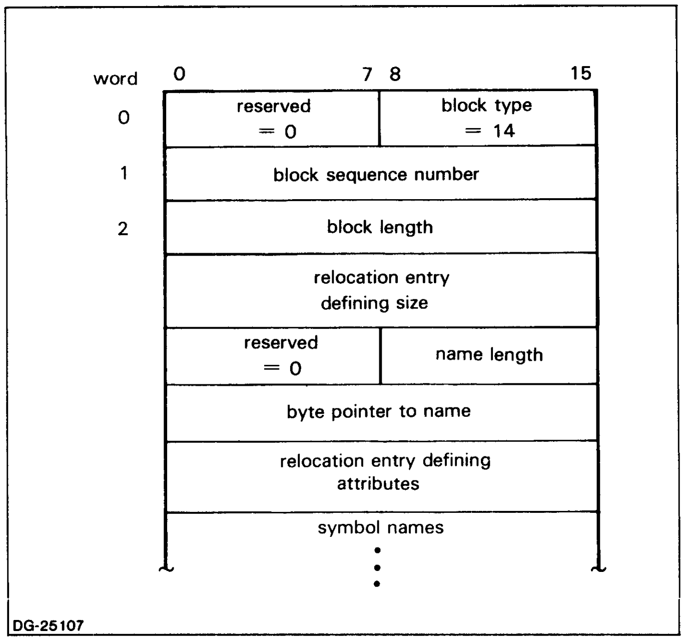


Figure B-16. Named Common Block

Example

```
         0      1      2      3      4      5      6      7
 0  000014 000007 000014 000052 000000 000005 000022 000000
10  000140 052101 041114 042400
```

The named common block shown above defines a 52 word long common base partition (see Word 3). Because the external number is 6 (see bits 0 through 11 of Word 10), Link will generate a partition with attributes unshared, data, common-base, overwrite-with-message, NREL, alignment=1. Word 5 indicates that the partition name consists of 5 characters, and Word 6 tells Link that the name begins at byte 22. Since byte 22 corresponds to the left byte of Word 11, the partition name is TABLE.

Accumulating Symbols Block

An accumulating symbols block defines one accumulating symbol. Figure B-17 shows the structure of the accumulating symbols block.

The value of an accumulating symbol is equal to the sum of the values of all accumulating symbols with the same name.

For instance, suppose three object modules A.OB, B.OB, and D.OB each define an accumulating symbols block with the name Y. Assume that A.OB defines Y's value as 10, B.OB defines Y's value as 12, and D.OB defines Y's value as 20. Further assume that object module C.OB contains an ELEF 1,Y instruction in one of its data blocks. For the command line:

```
) X LINK A B C D ↵
```

since 10+12+20 equals 42, Link sets the value of Y to 42. Link converts ELEF 1,Y to ELEF 1,42.

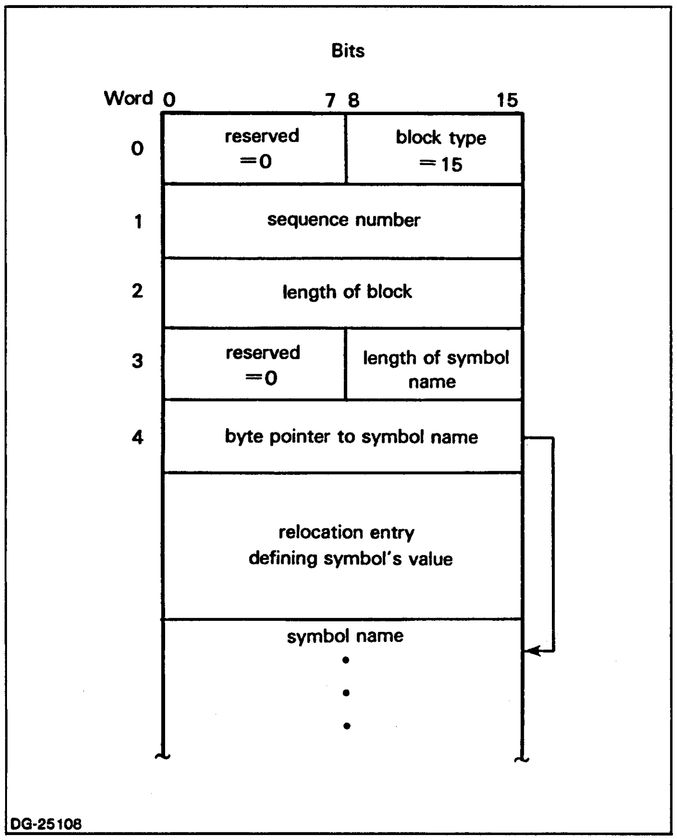


Figure B-17. Accumulating Symbols Block

Debugger Symbols Block

If Link encounters one or more debugger symbols blocks and the GLOBAL switch /DEBUG, it creates a .DS file. Typically, high-level language debuggers (such as SWAT) use .DS files, but you have the option of defining some other use for them. Figure B-18 shows the structure of the debugger symbols block; Figure D-1 illustrates the .DS file structure.

Beginning with Word 4, a debugger symbols block defines zero or more datawords. These datawords do not affect the program file or overlay file, and the choice of datawords is completely up to you.

Following the datawords, you can define zero or more relocation dictionary entries. Link uses them to relocate the datawords within the .DS file. You may use any relocation operation except 20_8.

If the relocation operation is 7 (*link relocation*), Link sets up a reverse chain of addresses in the .DS file, where the first dataword in the file is the address of the last dataword for which you invoked Link relocation.

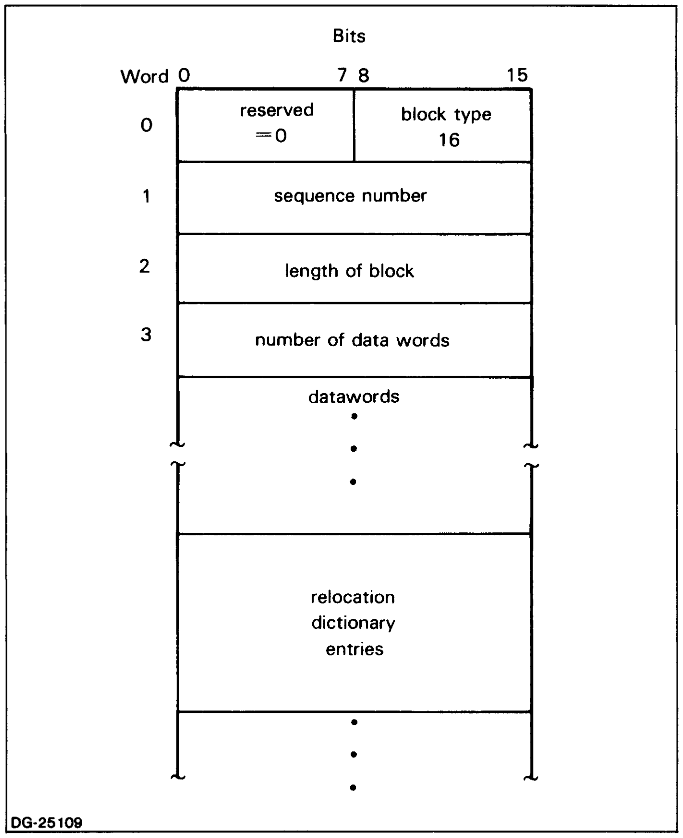


Figure B-18. Debugger Symbols Block

Debugger Lines Block/Lines Title Block

Given one or more debugger lines blocks in any input object module and the GLOBAL switch /DEBUG in the Link command line, Link creates a .DL (debugger lines) file. Link does not use the .DL file, but other utilities (typically high-level language debuggers) do. (Figure D-2 illustrates the .DL file.)

Link also creates a .DL file if it encounters both of the following:

- a lines title block in any input object module
- the /DEBUG switch on the Link command line

No object module may contain more than one lines title block.

You may include both debugger lines block(s) and a lines title block in the same object module. Assuming that /DEBUG is present, Link still creates only one .DL file per Link command line. All debugger lines block(s) and lines title block contribute to the same .DL file; however, only a lines title block generates a lines title directory entry. A *lines title directory entry* is a section of code within a .DL file that contains information on the relocation bases on the four predefined NREL partitions. Link generates one lines title directory entry for each object module that defines a lines title block.

The debugger symbols block, debugger lines block, and lines title block have nearly identical structures, the only difference being the block numbers — 16_8 for the debugger symbols block, 17_8 for the debugger symbols block, and 20_8 for the lines title block. (Refer to Figure B-18.)

Neither debugger lines blocks nor lines title blocks may use relocation operation 20.

Library End Block

The library end block follows the last object module in the library, and simply defines the end of the library. This block consists only of the three block header words, as Figure B-19 shows. LFE builds this object block.

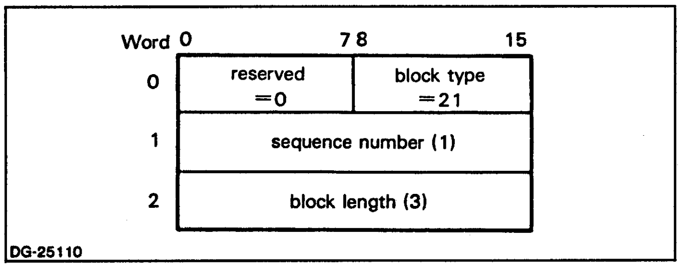


Figure B-19. Library End Block

Partition Definition Block

Partition definition blocks allow you to define partitions with almost any combination of attributes. Figure B-20 shows the structure of the partition definition block.

As we point out in Chapter 4, this is the recommended object block for generating user-defined partitions. Partition definition blocks have far more attribute versatility than either address information blocks or named common blocks.

You must generate one partition descriptor for each partition you're defining with this block.

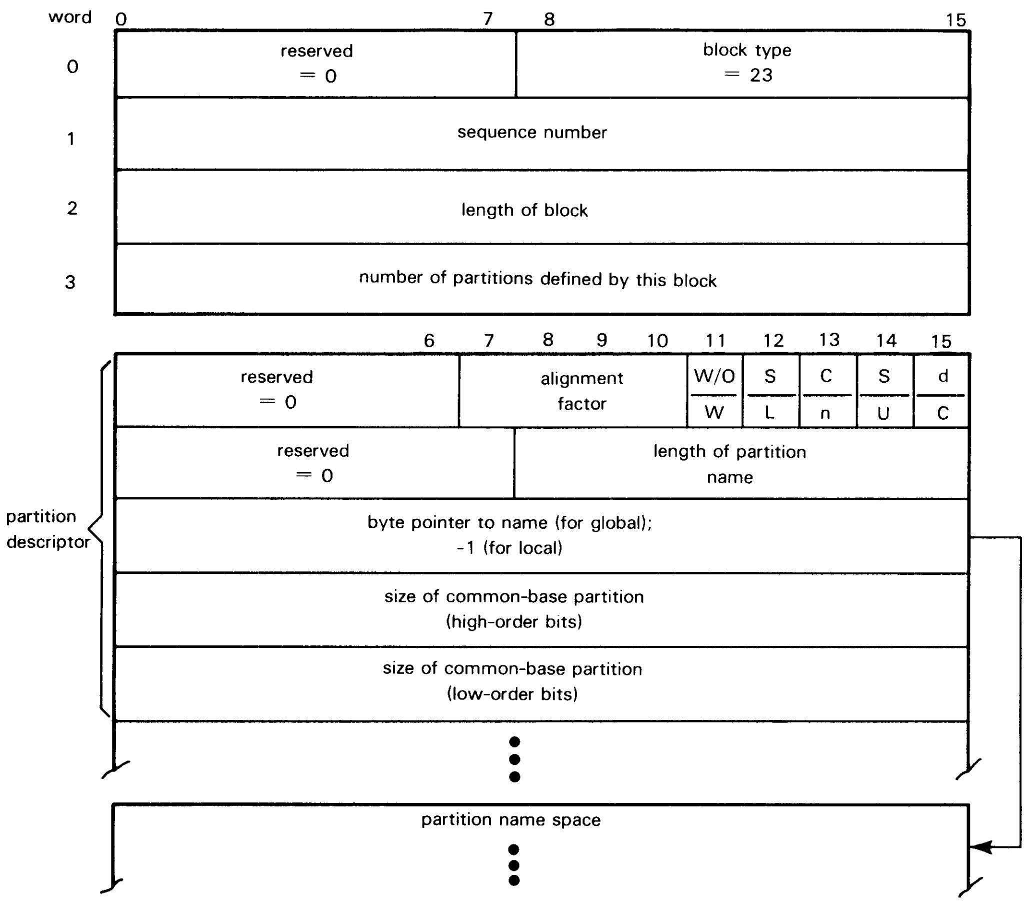
Word 0 of a partition descriptor defines all partition attributes. For instance, if bit 15 has a value of 1, Link gives the partition the data attribute.

All partitions generated by a partition definition block automatically have the NREL attribute.

The right byte of Word 1 in a partition descriptor tells Link how many characters are in the partition's name. Link truncates partition names longer than 40_8 characters to 40_8 characters.

Word 2 of a partition descriptor determines whether a partition will be global or local. If you want the partition to be global, put a byte pointer to the symbol name. If you want the partition to be local, put -1 (177777_8). (Chapter 4 describes global and local.)

Word 3 and Word 4 of each partition descriptor define the size (in 16-bit words) of a common base partition. If Link encounters two or more common base partitions with the same name, it sets the size of the partition to the largest. For instance, suppose a partition descriptor defines a common base partition named MATRIXB with a length of 10. Further assume that another partition descriptor (either in the same partition definition block or a different partition definition block) defines a common base partition named MATRIXB with a length of 20. In that case, Link builds a common base partition named MATRIXB with a length of 20.



w/o w	(bit 11)	= 1, if overwrite-without-message = 0, if overwrite-with-message
s 1	(bit 12)	ignored by AOS Link but used by AOS/VS Link. (Refer to *AOS/VS Link and Library File Editor User's Manual.)*
c n	(bit 13)	= 1, if common-base = 0, if normal-base
s u	(bit 14)	= 1, if shared = 0, if unshared
d c	(bit 15)	= 1, if data = 0, if code

DG-25111

Figure B-20. Partition Definition Block

Example

```
        0      1      2      3      4      5      6      7
0   000023 000012 000020 000002 000023 000004 000034 000000
10  000000 000505 000000 177777 000000 000052 041114 052505
```

The sample partition definition block shown above defines 2 partitions. Word 4 defines the attributes of the first partition: alignment factor = 0, overwrite-without-message, normal base, shared, data. Since Word 6 contains a positive number, this is a global partition. A byte pointer value of 34 corresponds to the left byte of Word 16. The name of this partition is therefore BLUE.

Word 11 defines the attributes of the second partition: alignment factor = 12_8, overwrite-with-message, common base, unshared, data. Since it has the common base attribute, the length of the partition, 52_8 words, is defined by Word 15. Since Word 13 contains a -1, this is a local partition.

Revision Block

A revision block specifies the earliest acceptable revision of Link and, optionally, it specifies how you want Link to interpret specific object blocks. You may use only one revision block per object module. Figure B-21 shows the structure of the revision block.

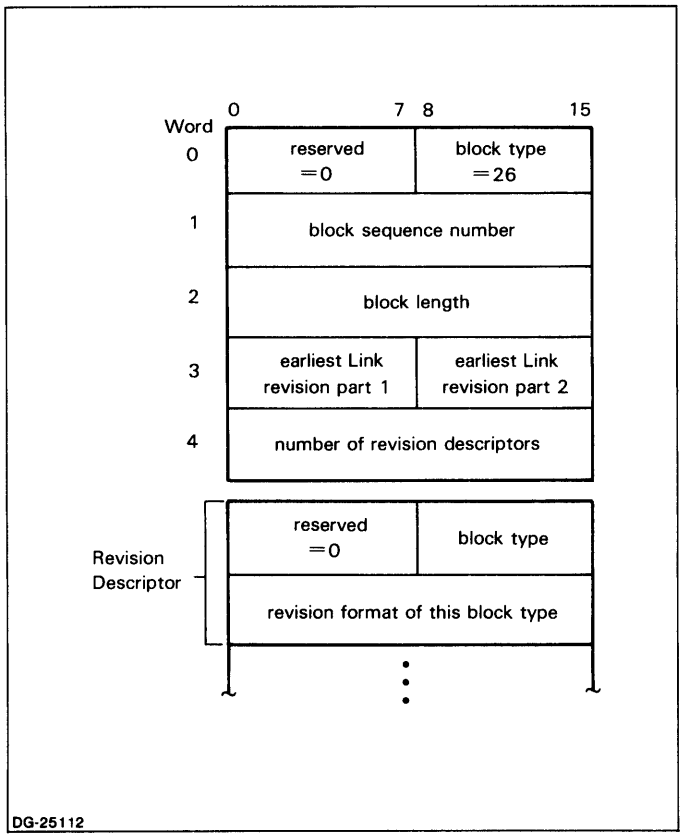
Word 3 of a revision block contains a major and minor revision number. Link itself (that is, the program file LINK.PR) also contains a major and minor revision number. You will cause a fatal Link error (see Appendix A) if the number in Word 3 is greater than Link's own revision number.

Word 4 contains the number of object block revision descriptors that the revision block defines. Each revision descriptor is two words long. The right byte of the first word of a revision descriptor defines the object block that you want Link to interpret differently. The second word defines the revision format that you want Link to use on this block type. Unlike the two-part revision numbers in title blocks and module revision blocks, the revision format in a revision block is only a one-part revision number. The revision format tells Link how it should interpret the block type.

The default revision number for all object block types is 0.

NOTE: A revision descriptor applies only to a particular block type and only within the defining object module. You can, for instance, define data block revision format 0 in an object module, and define data block revision format 1 in a different object module.

Currently, the data block is the only block type with more than one format. (See the "Data Block" section.) However, future revisions of Link may define multiple formats for other object block types.



Word
0
7
8
15
0
reserved =0
block type =26
1
block sequence number
2
block length
3
earliest Link revision part 1
earliest Link revision part 2
4
number of revision descriptors
Revision Descriptor
reserved =0
block type
revision format of this block type
DG-25112

Figure B-21. Revision Block

Example

```
0      1      2      3      4      5      6
000026 000002 000007 002002 000001 000000 000001
```

The sample revision block shown above defines a major revision of 4 (left byte of Word 3) and a minor revision of 2 (right byte of Word 3). Word 4 contains a 1 which means that this revision block defines one revision descriptor. The right byte of Word 5 contains a 0. Table B-1 says that block type 0 is the data block. Word 6 contains a 1. Therefore, Link will interpret all data blocks in the object module as revision format 1.

Filler Block

Aside from the standard object block header, Link does not read or interpret any information in a filler block. For this reason, a filler block is a good place to store information like copyrights or documentation. Figure B-22 shows the structure of the filler block.

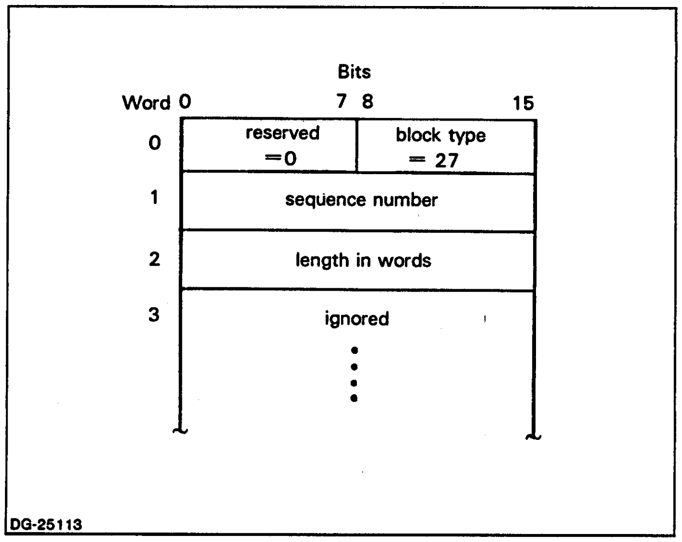


Figure B-22. Filler Block

Module Revision Block

The module revision block defines a revision number and directs Link to store it in offset USTRV of the User Status Table (UST). Since the title block can also define a revision number, Link uses the module revision block's revision number only if the revision number in the title block is set to -1. Figure B-23 shows the structure of the module revision block.

NOTE: Under AOS, offset USTRV holds a two-part revision number, but under AOS/VS, offset USTRV holds a four-part revision number. The title block can define only a two-part revision number; however, the module revision block can define a four-part revision number. When building an AOS program file, Link uses all four parts for listing files (i.e., the files generated by /L or /L=filename), but uses only the first two parts for offset USTRV. When building an AOS/VS program file, AOS/VS Link uses all four parts for listing files and for offset USTRV.

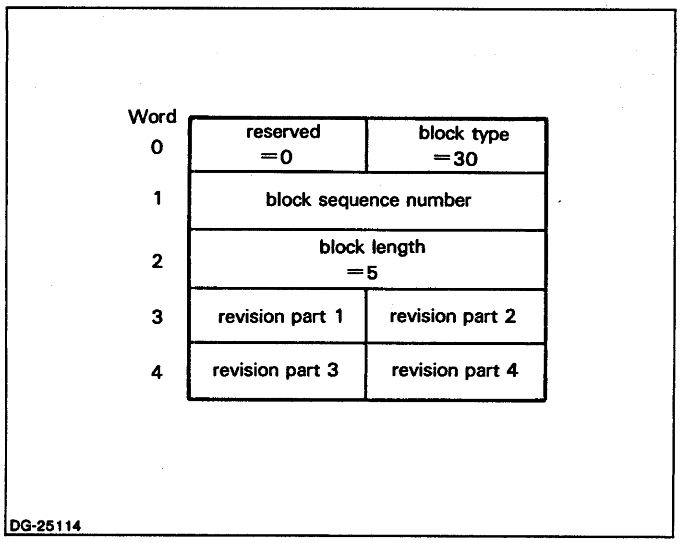


Figure B-23. Module Revision Block

Alignment Block

Functionally similar to the PARTSYM switch partition/ALIGN=n, an alignment block aligns and pads partitions with the normal base attribute, and aligns partitions with the common base attribute. An alignment block allows you to align any predefined partitions or user defined partitions. Figure B-24 shows the structure of the alignment block.

The external number (Word 3) tells Link which partition you want aligned. For instance, if Word 3 contains external number 4, Link attempts to align the predefined Unshared Code partition. (Chapter 4 explains the external numbering scheme.)

An *alignment factor* is the number x (between 0 and 12_8 inclusive) in the following equation:

relocation base = $(I)2^x$

where I is any integer. Thus, an alignment factor of x tells Link to set the relocation base of this partition at the next free integral multiple of 2^x. For instance, suppose an alignment block contains an external number of 6 (the predefined Unshared Data partition) and a alignment factor of 12_8. Because 2^{12} equals 2000_8, Link must set Unshared Data's relocation base to some integral multiple of 2000_8.

If the external number in an alignment block refers to a partition with the normal-base attribute, Link pads and aligns every object module's contributions to that partition. For example, consider an alignment block containing external number 13 (a user-defined normal base partition entitled RED) and alignment factor 10_8. Since $2^{10} = 400_8$, Link must set RED's relocation base to an integral multiple of 400_8, say 5400. If any other object modules contribute to RED, they too will be aligned on an integral multiple of 400_8. Link pads the gaps created by alignment with 000000s.

If the external number in an alignment block refers to a partition with the common base attribute, then Link aligns the partition's relocation base, but it does not pad it.

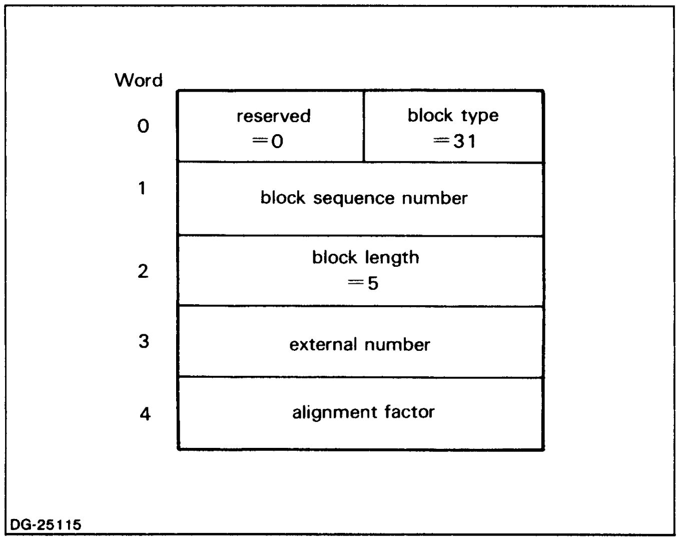


Figure B-24. Alignment Block

End of Appendix

Appendix C
Link-Generated Symbols and Symbol Types

Table C-1 explains all Link-emitted external symbols and Link-defined entry symbols and their values. These symbols fall into one of four categories:

A	Link emits this external symbol and a system library (either URT.LB or SYS.LB) defines the corresponding entry symbol. You should not define this symbol. If you do define this symbol, Link has no way of detecting the error.
B	Link emits this external symbol and a system library (either URT.LB or SYS.LB) defines the corresponding entry symbol; however, you can override the system library routine by defining this symbol in an input object module.
C	Link defines this entry symbol. You will cause a Link error if you try to define this symbol.
D	Link defines this entry symbol, but you may override Link's value for this symbol by defining this symbol in one of your input object modules or by using the appropriate Link switch.

Table C-1. Link-Generated Symbols

Symbol	Category	Value
??RCA	A	an entry symbol into the resource manager — a resource handler routine. Some indirect user control through resource call optimization switches. (See the "Resource Call Optimization Switches" section in Chapter 4.)
?CHAN	D	maximum number of I/O channels this process can open. By convention, some AOS language processors use this symbol for similar purposes. You may control the value of ?CHAN with the /CHANNELS=n switch.
?CLOC	C	relocation base of the unlabeled common area, default value is -1. External number 3 is equivalent to ?CLOC.
?CSZE	C	size of the unlabeled common area, default value is 0. Indirect user control through unlabeled common blocks.
?LBOT	D	base value of EXTC links, default value is ?UNDF.
?LODO	A	entry symbol into an URT.LB or SYS.LB object module that contains some of the code necessary to execute system calls.
?NBOT	D	relocation base of NREL, default value is 400. Although rarely used, external number 2 is equivalent to ?NBOT.
?NMAX	C	highest unshared NREL address that Link used plus 1.
?NTOP	D	highest address in the .PR file, default value is 77777_8, but you may override default with either /KTOP=n or /NTOP=n.
?SBOT	C	lowest shared address in the .PR file.
?SRES	D	size (in pages) of the shared reserve area, default value is 0, but you may override the default by using the /SRES=n switch.

(continues)

Table C-1. Link-Generated Symbols

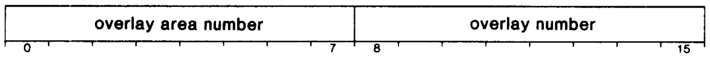
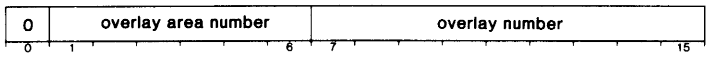
Symbol	Category	Value
?TBOT	C	lowest address in the PRSYM table minus 1. Link defines ?TBOT only if you set the /PRSYM switch.
?TTOP	C	highest address in the PRSYM table (RDOS-style symbol table). Link defines ?TTOP only if you set the /PRSYM switch.
?UNDF	D	the default symbol value. Link gives undefined symbols the value of ?UNDF. ?UNDF has a default value of -1, but you may override the default.
?URTB	A	an entry symbol in an URT.LB object module that handles many system calls.
?USTA	C	highest system table address used plus 1.
?ZBOT	D	lowest used address in the predefined ZREL partition; default value is 50, but you may override the default with the /ZBOT=n switch.
?ZMAX	C	highest address used in ZREL plus 1
CFALT	B	starting address of the commercial fault handler routine.
DEBUG	B	an entry symbol within a SYS.LB routine that handles debugging.
FFALT	B	starting address of the floating-point fault handler routine.
SFALT	B	starting address of the stack fault handler routine.
TMAX	A	entry symbol into a SYS.LB routine that must be in your .SV file if you intend to do multitasking.
TMIN	A	relocation base of a SYS.LB routine called the single-task scheduler.
USTAD	C	relocation base of the RDOS or RTOS User Status Table (UST).

(concluded)

Symbol Types

Table C-2 lists all currently supported symbol types and type numbers. Except for PENT (which applies only to AOS) and SOENT (which applies only to the AOS .SY file), all symbol types apply to RDOS, RTOS, and AOS.

Table C-2. Symbol Types

Symbol Type	Type Number	Meaning
ENT	0	standard entry symbol; defined by an entry symbols block.
EXT	1	standard external symbol; defined by an external symbols block.
COMM	2	common symbol; Link gives this symbol type to any partition with the common base attribute.
ASYM	3	accumulating symbol; defined by an accumulating symbols block or by the /VAL=n switch.
ENTO	4	overlay entry symbol; defined by an entry block. Under RDOS or RTOS, ENTO's value is equal to the value of the following word:  Under AOS, ENTO's value is equal to the value of the following word:  0 overlay area number overlay number 0 1 6 7 15
TITLE	5	title symbol; defined by a title block
PENT	6	procedure entry symbol; defined by an entry symbols block.
EXTS	7	suppressed external symbol; defined by an external symbols block.
LOCAL	10	local symbol; defined by a local symbols block.
EXTC	11	chain external; defined by an external symbols block.
LIMIT	12	not supported by AOS Link.
BOUND SLPENT	13	not supported by AOS Link.
SLPENT	14	not supported by AOS Link.
PART	15	partition; Link gives this symbol type to any partition generated by an address information block.
SOENT	16	operating system ENTO

End of Appendix

Appendix D
Link-Generated Output Files

This appendix details the structures of the following Link-generated files:

- AOS .PR file
- AOS, RDOS, and RTOS .OL file
- AOS .ST file
- AOS .DS file
- AOS .DL file
- AOS .SY file
- RDOS .SV file
- RTOS .SV file

In these figures, statements that appear in parentheses are conditionally built. For instance, Figure D-1 includes the following:

(overlay directory)

Since Link builds the overlay directory only if the program contains overlays, overlay directory is enclosed by parentheses.

We listed symbol names where appropriate. For details on these symbols, see Table C-1.

AOS .PR File

Figure D-1 shows the structure of the .PR file. A .PR file becomes executable when AOS maps it into logical address space. Refer to the *AOS Programmer's Manual* for information on the system tables (UST, TCB, overlay directory, and resource-handler table). Also, see Chapter 2 for a summary of logical address space. In addition, refer to the appropriate manual in the Programmer's Reference Series for ECLIPSE-Line Computers for more information on the reserved storage locations (addresses 00000 through 00047).

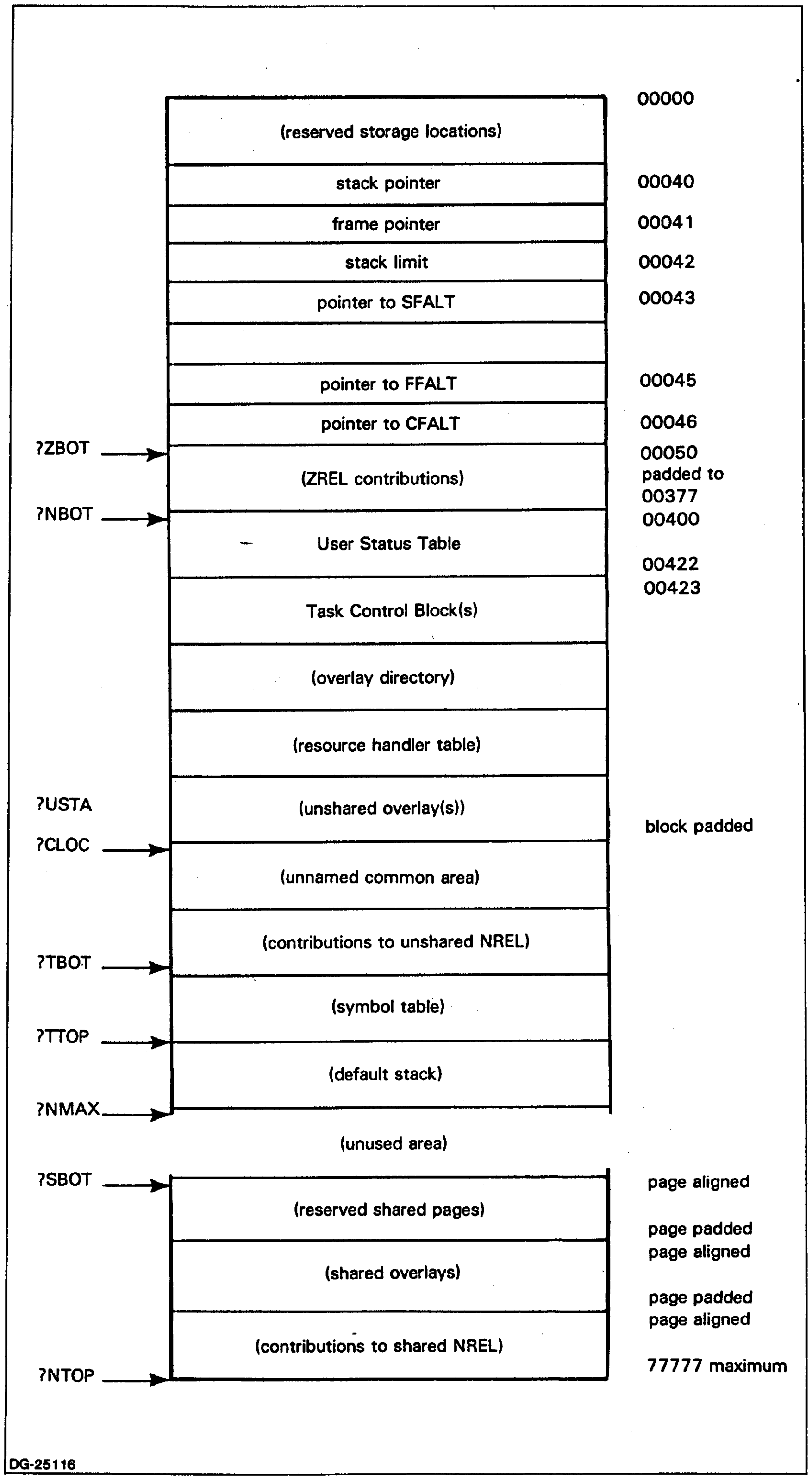


Figure D-1. .PR File Structure

.OL File

Figure D-2 shows the structure of the AOS, RDOS, and RTOS .OL (overlay) file.

In the figure, "Overlay Area n" refers to the overlay area within a .PR or .SV file that an overlay can contribute to at runtime. Within an .OL file, Link always builds overlays sequentially; however, it does not necessarily build overlay areas sequentially.

Virtual overlays (RDOS and RTOS) and shared overlays (AOS) are page-aligned and page-padded. Unshared AOS overlays and conventional RDOS overlays are block-aligned and block-padded.

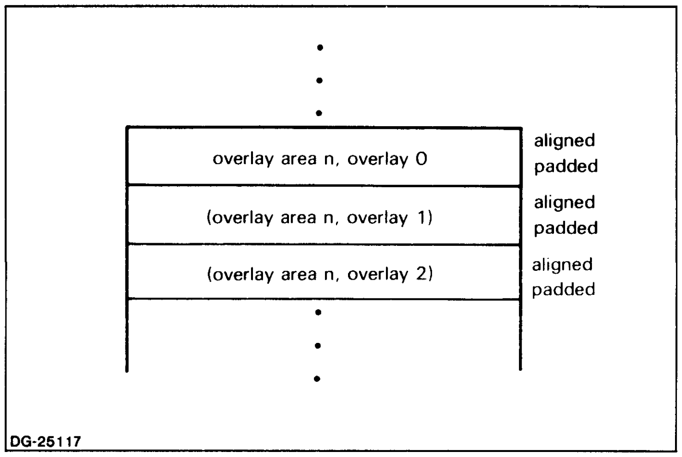

overlay area n, overlay 0
aligned
padded
(overlay area n, overlay 1)
aligned
padded
(overlay area n, overlay 2)
aligned
padded
DG-25117

Figure D-2. AOS, RDOS, and RTOS .OL File Structure

.ST File

Figure D-3 shows the structure of the AOS .ST (symbol table) file. By default, Link builds the .ST file; however, if you include either /N or /SUPST on the Link command line, Link does not build it. The primary purpose of the .ST file is to store symbol names, symbol types, and symbol values for possible use by a debugger. The .ST file is nonexecutable, and it does not affect program file execution.

Link uses the following hash function:

	ASCII value of first character
+	ASCII value of last character
+	ASCII value of third character (if present)

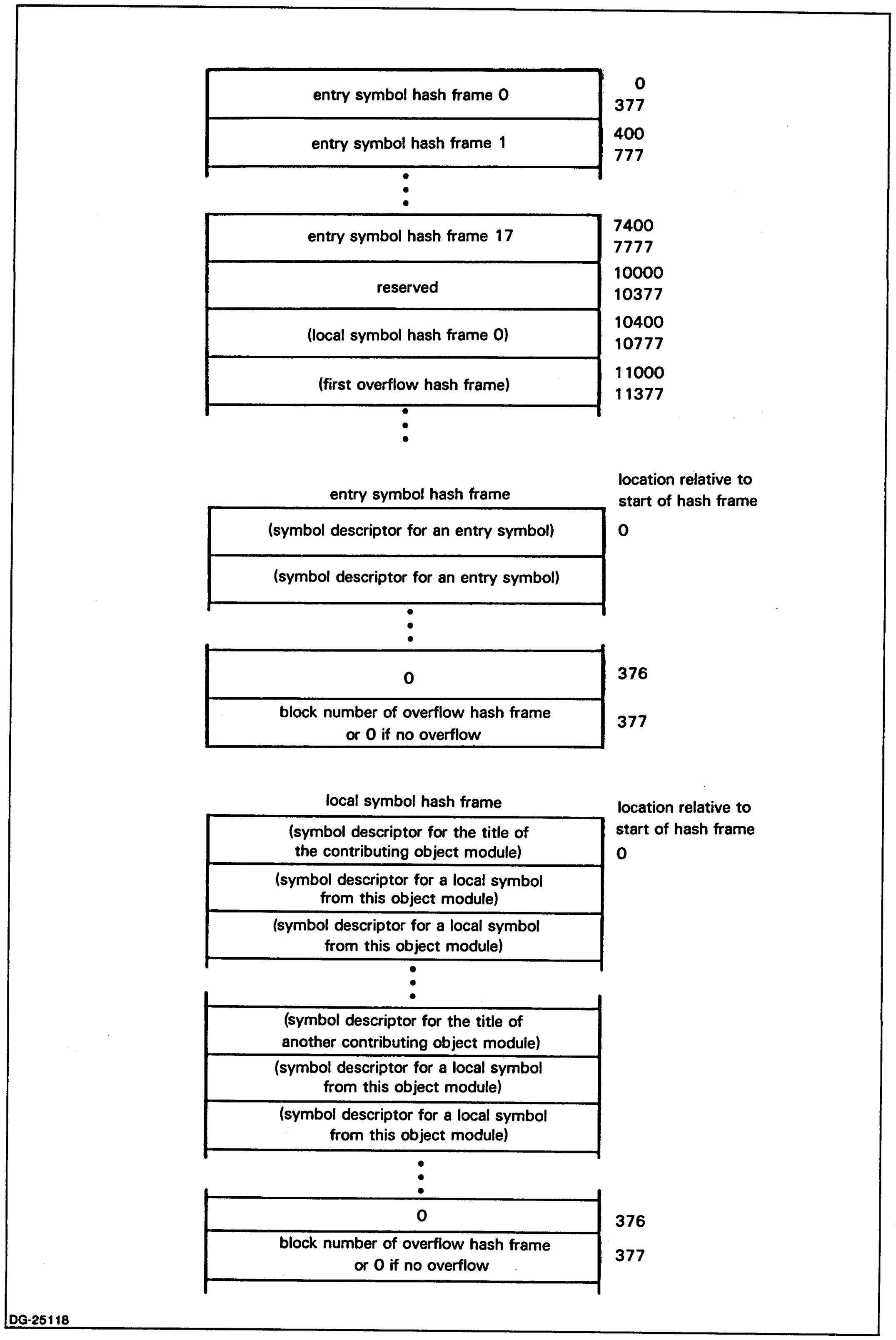
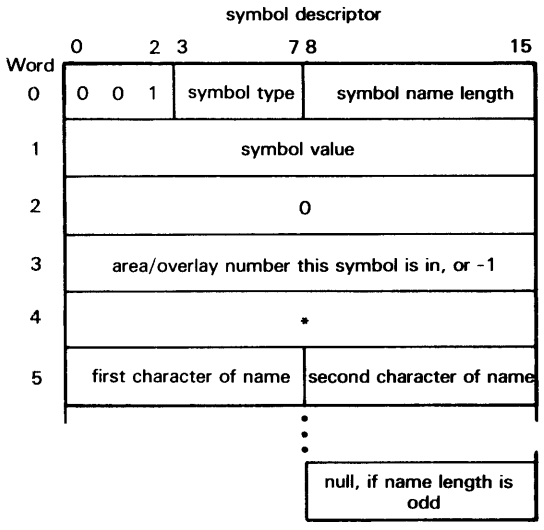

entry symbol hash frame 0
0
377
entry symbol hash frame 1
400
777
entry symbol hash frame 17
7400
7777
reserved
10000
10377
(local symbol hash frame 0)
10400
10777
(first overflow hash frame)
11000
11377
entry symbol hash frame
location relative to start of hash frame
(symbol descriptor for an entry symbol)
0
(symbol descriptor for an entry symbol)
0
376
block number of overflow hash frame or 0 if no overflow
377
local symbol hash frame
location relative to start of hash frame
(symbol descriptor for the title of the contributing object module)
0
(symbol descriptor for a local symbol from this object module)
(symbol descriptor for a local symbol from this object module)
(symbol descriptor for the title of another contributing object module)
(symbol descriptor for a local symbol from this object module)
(symbol descriptor for a local symbol from this object module)
0
376
block number of overflow hash frame or 0 if no overflow
377
DG-25118

Figure D-3. .ST File Structure (continues)



*

if a LOCAL symbol	contains a pointer to the start of the title entry.
if a COMM symbol	contains the length of the partition.
if a PART symbol	contains the length of the partition.
if a PENT symbol	contains address of the appropriate Resource Handler Table (RHT) entry.
any other symbol type	undefined

DG-25118

Figure D-3. .ST File Structure (concluded)

.DS and .DL Files

Figures D-4 and D-5 illustrate the structures of the .DS and .DL files. If any object module on the Link command line contains a debugger symbols block, and if you include the GLOBAL switch /DEBUG, then Link generates the .DS file. If any object module on the Link command line contains a debugger lines block or a lines title block, and if you include the GLOBAL switch /DEBUG, then Link generates the .DL file.

Except for the lines title directory in the .DL file, the composition and organization of both files is user-defined (through relocation dictionary entries in debugger symbols blocks or debugger lines blocks.) Aside from lines title directories, the structure of both files is very similar.

If you set up a backwards-linked list (with relocation operation 7 in the relocation dictionary entries), then the first word in the .DS file will point to the terminal point in the list. If you did not set up a backwards-linked list, the first word in the .DS file will contain a 0.

Within the .DL file, Link generates one lines title directory for each lines title block it encounters. This directory contains the value of the relocation base of this module's contributions to the predefined NREL partitions. It also defines the relocation base of the next module's contributions to those partitions. Link uses the datawords and relocation dictionary entries in lines title blocks to generate "lines title block data".

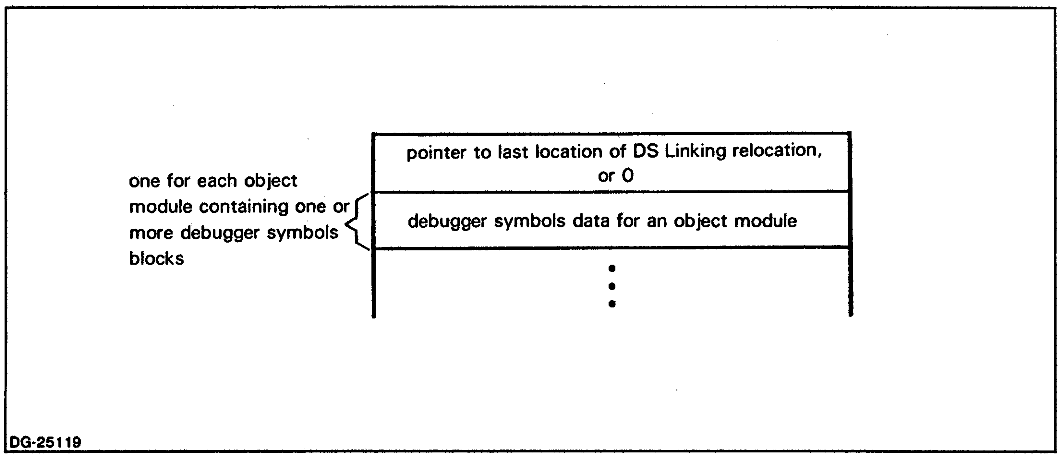

pointer to last location of DS Linking relocation, or 0
debugger symbols data for an object module
one for each object module containing one or more debugger symbols blocks
DG-25119

Figure D-4. .DS File Structure

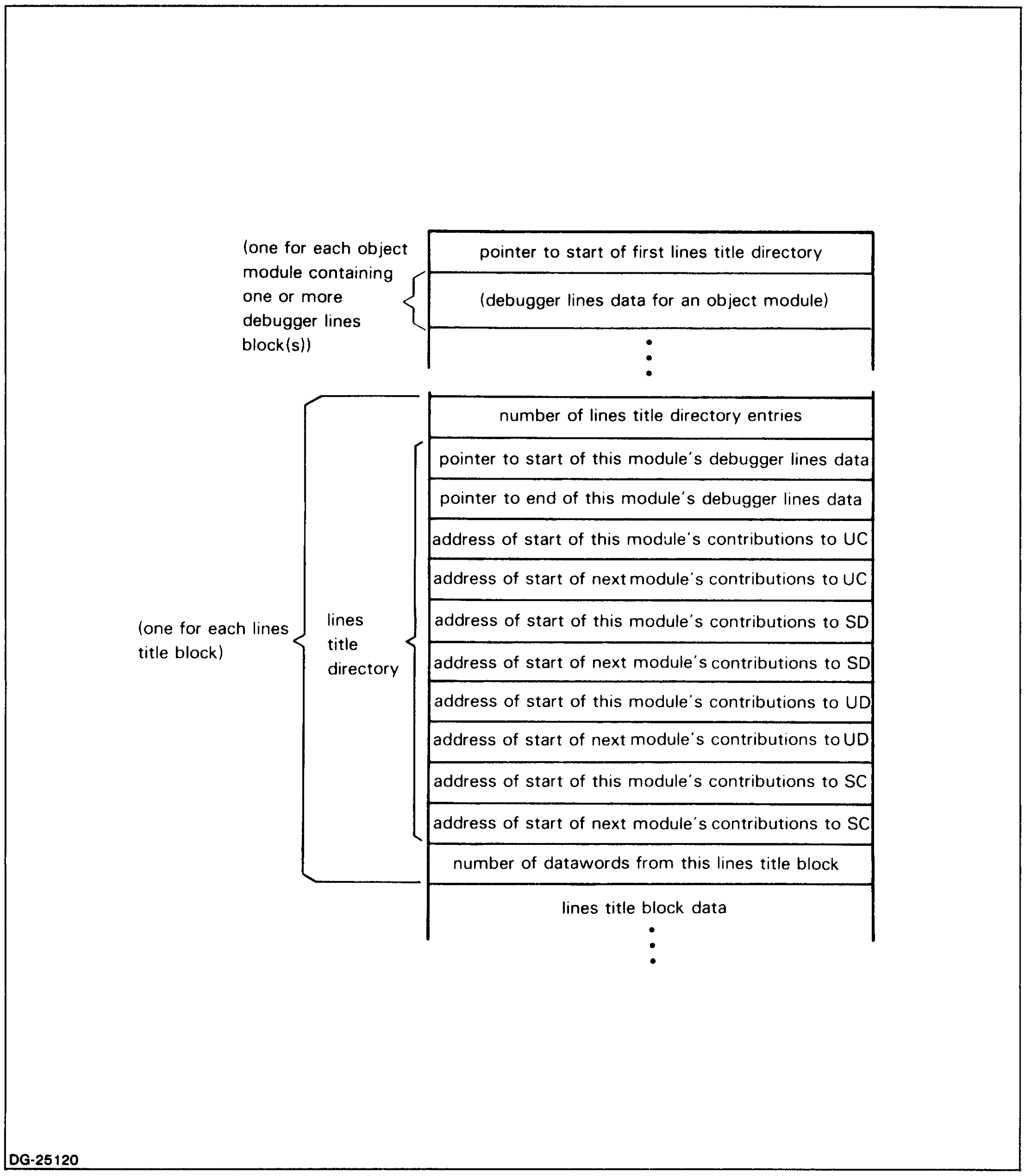

(one for each object module containing one or more debugger lines block(s))
pointer to start of first lines title directory
(debugger lines data for an object module)
(one for each lines title block)
lines title directory
number of lines title directory entries
pointer to start of this module's debugger lines data
pointer to end of this module's debugger lines data
address of start of this module's contributions to UC
address of start of next module's contributions to UC
address of start of this module's contributions to SD
address of start of next module's contributions to SD
address of start of this module's contributions to UD
address of start of next module's contributions to UD
address of start of this module's contributions to SC
address of start of next module's contributions to SC
number of datawords from this lines title block
lines title block data
DG-25120

Figure D-5. .DL File Structure

AOS .SY File

Figure D-6 shows the structure of the AOS .SY (operating system) file.

In an .SY file, the UST and the TCBs have a nearly identical structure to the UST and TCBs in a .PR file. The only current difference concerns offset USTOD of the UST. In a .PR file, USTOD contains a pointer to the overlay directory (or 0 if there is no overlay directory); while in a .SY file, USTOD contains the value of ?NMAX rounded to the next highest page boundary. (Refer to the *AOS Programmer's Manual* for details on the UST and TCB.)

Although, Link never builds .PR files larger than 32_{10} pages (100000_8 words), Link usually builds .SY files many times larger than 32 pages. Instead of storing overlays in an .OL file as it does for a .PR file, Link stores an .SY file's overlays within the .SY file itself. Of course, at execution time, AOS does not map in the entire .SY file. Instead, an .SY file maintains one overlay area (not marked in Figure D-6) and maps in overlays from itself as needed.

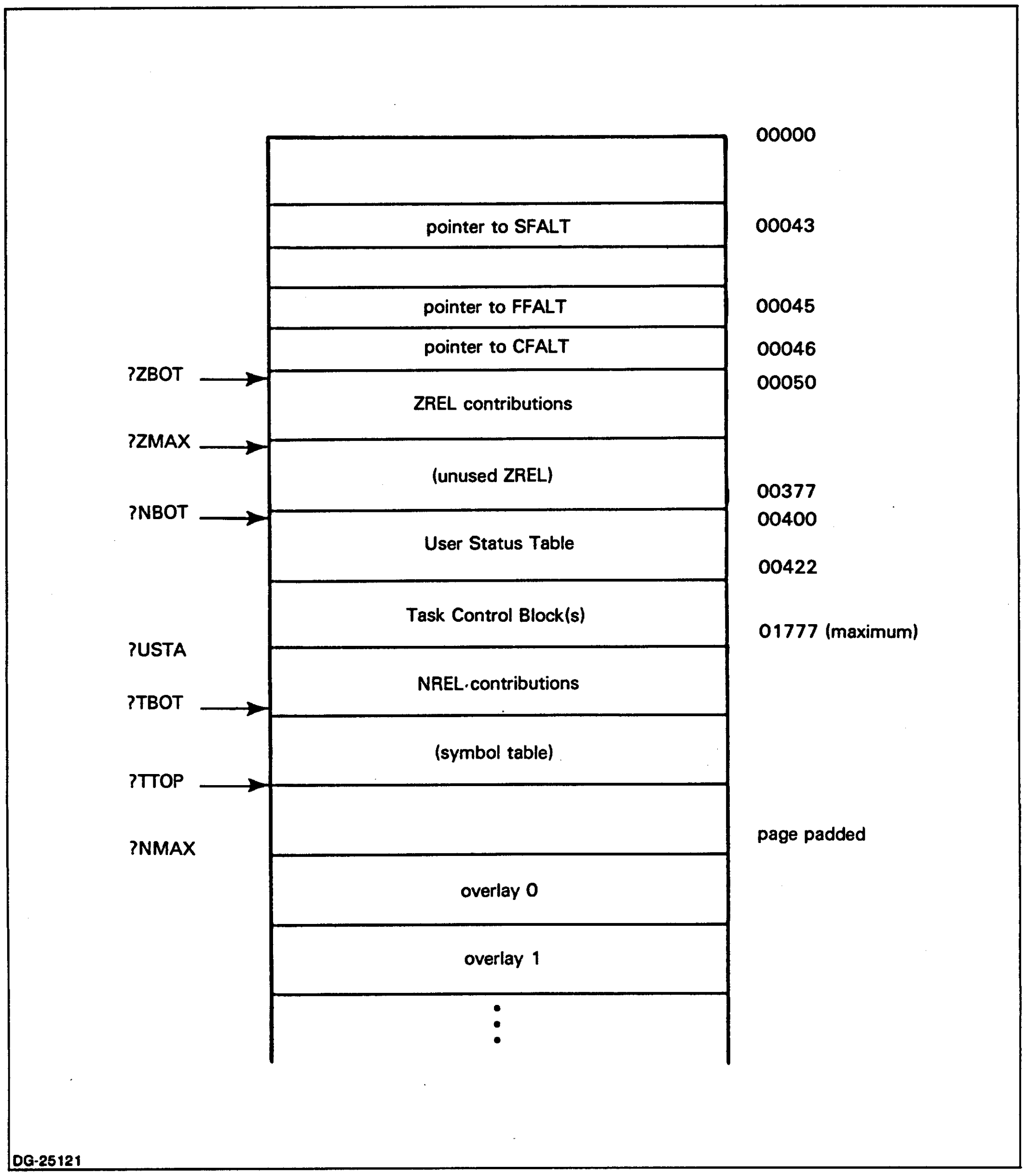


Figure D-6. AOS .SY File Structure

RDOS .SV File

Figure D-7 shows the structure of the RDOS .SV file. Refer to the *Real-Time Disk Operating System Reference Manual* for more details on the RDOS .SV file structure. Refer to Chapter 7 for more information on cross-linking for RDOS.

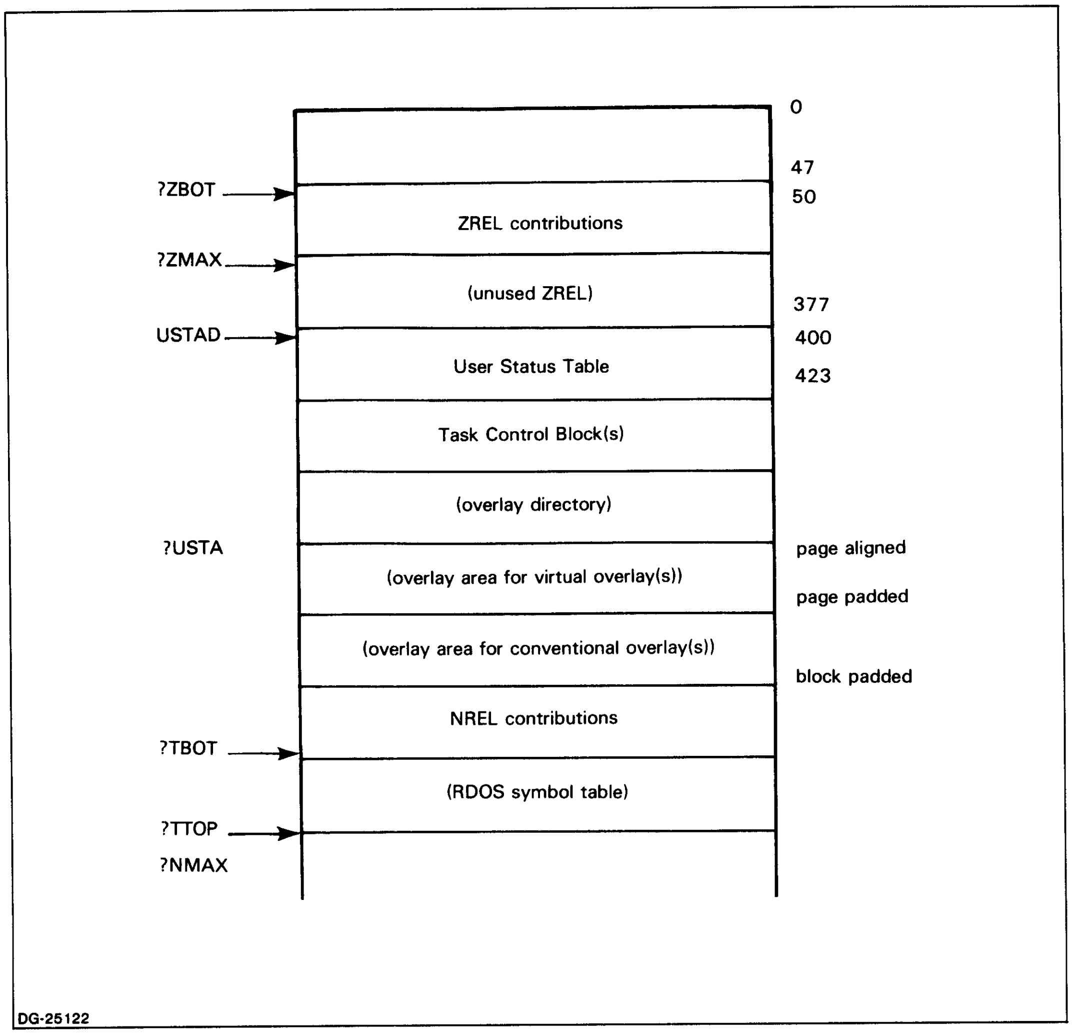


Figure D-7. RDOS .SV File

RTOS .SV File

Figure D-8 shows the structure of the RTOS .SV file. Refer to the *Real-time Operating System Reference Manual* for more details on the .SV file structure. Refer to Chapter 7 for information on cross-linking for RTOS.

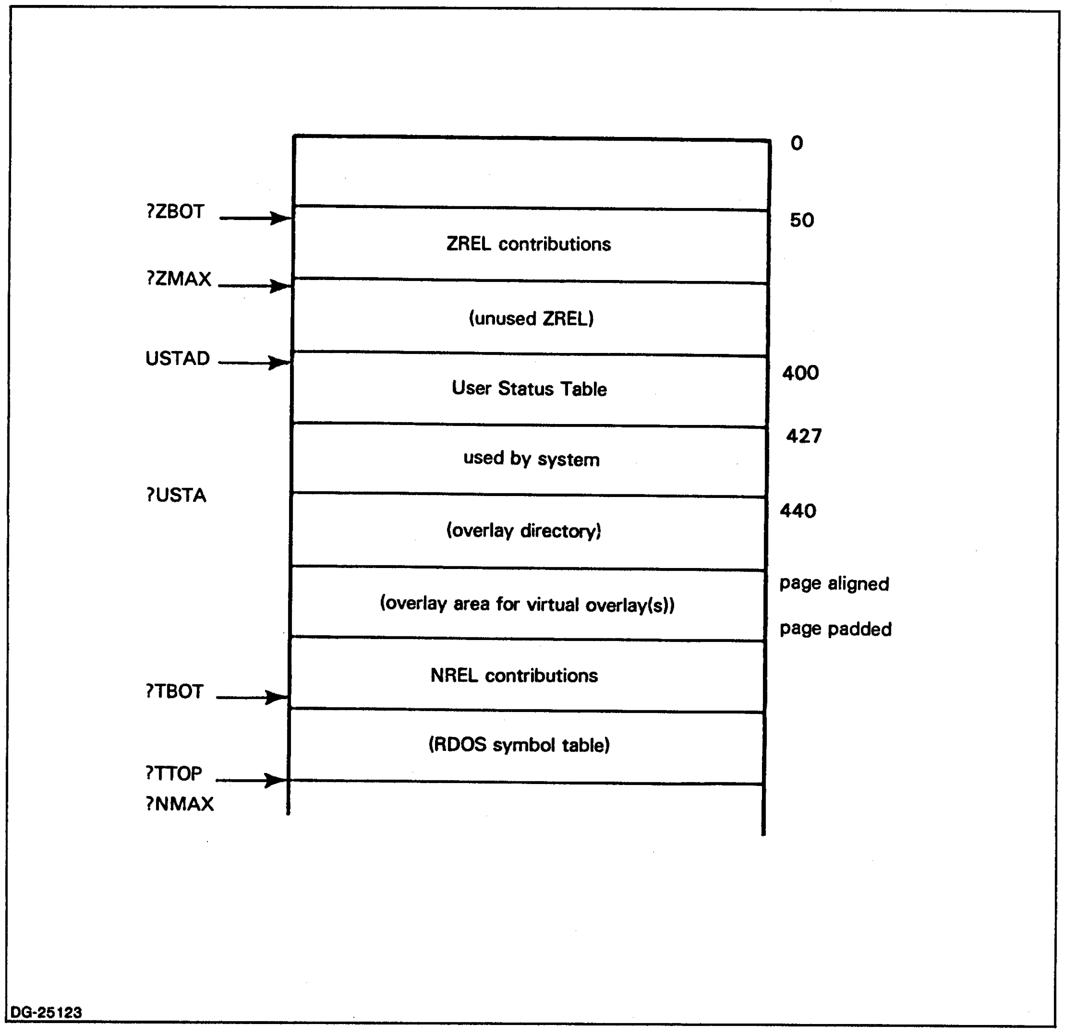


Figure D-8. RTOS .SV File Structure

End of Appendix

Appendix E
LFE Errors

BLOCK NUMBER OUT OF SEQUENCE

An object block in one of the object files specified on the LFE command line does not have the right sequence number. Refer to Appendix B for information on sequence numbers. There are two possible causes for the error. First, LFE may have tried to interpret a nonobject module as an object module. The second possibility, though rare, is that the language processor generated an object block with the wrong sequence number. If you can verify this, please contact the compiler or assembler maintenance personnel.

DUPLICATE INPUT FILE SPECIFICATION

You specified an input file (with /I) more than once on the LFE command line.

DUPLICATE LISTING FILE SPECIFICATION

You specified a listing file (with /L) more than once on the LFE command line.

DUPLICATE OUTPUT FILE SPECIFICATION

You specified an output file (with /O) more than once on the LFE command line.

FILE ACCESS DENIED[1]

You tried to access a file, but you did not have permission to do so. Change the file's ACL.

FILE ALREADY EXISTS[1]

You specified an output file already contained in your working directory.

FILE IS NOT A LIBRARY <filename>

While using function-letter T, you used an .OB file as either an argument or as an input file. Both the argument and the input file must be libraries.

FUNCTION DOES NOT ACCEPT INPUT FILE

You specified an input file while using function-letter A, M, N, or T.

1 AOS system error. Refer to the *AOS Programmer's Manual* for more information.

FUNCTION DOES NOT ACCEPT OUTPUT FILE

You specified an output file while using function-letter D, I, M, N, or R.

/I SWITCH REQUIRES ARGUMENT

You failed to qualify an /I switch with a filename.

ILLEGAL FUNCTION

You did not specify a valid function-letter on the LFE command line. In most cases, this error occurs when you do not specify a function-letter and LFE interprets the first argument on the LFE command line as a function-letter.

INVALID BLOCK SIZE

One of the object blocks in an object module is more than 2000_8 words long. There are two possible causes for this error. The first is that LFE interpreted a title or some other nonobject module on the command line as an object module. The second possibility, though rare, is that the compiler or assembler produced an object block that was too large. If you can verify this, please contact the compiler or assembler maintenance personnel.

MODULE NOT FOUND <object module title>

This object module title is not present in the input library. First, for function-letters I or R, make sure that object module titles precede their corresponding object file arguments on the LFE command line. Second, make sure that you are using the name of an object module title and not the name of the .OB file that it was originally stored in. You can check title names with the function-letter T.

NO INPUT FILE SPECIFIED

While using function-letters D, I, R, or X you did not specify the input file. You can specify it either explicitly (with /I) or implicitly (by the file's position in the LFE command line.) Check the command line format for this function-letter.

NO MODULES IN LIBRARY

You have generated a library that contains only a library start block and a library end block; no object modules in-between. The probable cause of this error is that you deleted (with D) every object module in the library. Another possibility is that while creating a file (with M or N) LFE could not open any input files.

NO OUTPUT FILE SPECIFIED

You used function-letter A, T, or X in your LFE command line, but you did not specify an output file.

NOT ENOUGH ARGUMENTS

Either you did not specify any arguments (i.e., object files) on the LFE command line, or while using function-letters I or R, you did not specify an object file after a title.

/O SWITCH REQUIRES ARGUMENT

You failed to qualify an /O switch with a filename.

OBJECT FILE DOES NOT FOLLOW TITLE

While using either I or R, you did not qualify an object module title with an object file.

End of Appendix

Glossary

absolute code	datawords assigned to specific (i.e., absolute) addresses in a program file.
alignment factor	a number that Link uses as a partition attribute. Allows user to set the relocation base to a power-of-2 boundary.
attributes	the set of characteristics defining a partition; the attributes are: · absolute, ZREL, or NREL · shared or unshared · normal base or common base · alignment (0-12_8) · code or data · overwrite-with-message or overwrite-without-message · local or global (user-defined partitions only)
bit field relocation dictionary entry	an element within certain object blocks that Link uses to resolve the value of a contiguous set of bits in a particular dataword.
block	400_8 sequential words of memory.
byte pointer	a 16-bit value defining the start of a byte string.
cross-linking	logging on to an operating system, and using a relocatable linker to create a program file that can execute on a different operating system.
dataword	16 contiguous bits; code or data. An element that can fill one logical address in a program file.
displacement	in object blocks — a dataword's position relative to some other point in the object block or object module. In memory reference instructions — a value which tells the CPU which logical address to access.
.DL (debugger lines) file	an output file Link creates to store information for eventual high-level language debugging; built when any input object module contains one or more debugger lines blocks or one lines title block and the Link command line includes the /DEBUG switch.
.DS (debugger symbols) file	an output file Link creates to store information for eventual high-level language debugging; built when any input object module contains one or more debugger symbols blocks and the Link command line includes the /DEBUG switch.
entry symbol	a symbol defined in an entry symbols block. Entry symbols can be accessed by any object module.
error file	a Link-generated output file containing Link errors and standard Link output messages.

extended relocation dictionary entry	a relocation dictionary entry capable of defining relocation operations greater than 17.
external symbol	a symbol defined in an external symbols block; a symbol whose value is defined by a different object module.
forced load flag	a bit within a library file that, if set, forces Link to unconditionally load the object module it is associated with.
function-letters	Library File Editor (LFE) commands.
language processor	an assembler (e.g., MASM) or a high-level language compiler; a program that analyzes a file of source code and produces an .OB file.
library	a library start block, a library end block, and one or more object modules in-between; produced by LFE.
library file	a library; usually has the file name extension .LB.
Library File Editor (LFE)	the utility that creates, edits, and analyzes library files.
LFE command line	a set of directives beginning with X LFE that invokes the Library File Editor.
lines title directory	an optionally built section of the .DL file that describes the program file addresses of all four predefined NREL partitions.
Link	the name of the AOS relocatable linker (a program that can produce a program file, an overlay file, debugger files, and Link listing files from object files).
Link command line	a set of directives beginning with X LINK that invokes Link.
logical address space	the entire range of addresses (00000_8 to 77777_8) that a process may access.
GLOBAL switch	a switch flush with the word LINK in the Link command line. See Table 5-1.
NREL	addresses 00400 through 77777 in a .PR or .SV file.
object blocks	any of 31_8 sets of machine language that Link can interpret.
object block header	the first three words in every object block; states the block's type, length, and sequence number relative to the other object blocks in an object module.
object file	an .OB file or library.
.OB file	a file produced by a language processor that usually has the extension .OB. An .OB file contains one object module.
OB switch	a switch attached to an object file in the Link command line. See Table 5-2.
object module	a series of object blocks bounded by a title block and an end block; produced by assembling or compiling source code. Object modules are stored in either a library or an .OB file.
.OL (overlay) file	a Link-built file that retains a program's overlays.

OV switch	a switch attached to overlay area delimiter !* in the Link command line. See Table 5-4.
overlay	a shared or unshared routine residing in an .OL file that a process may call into logical address space as needed.
overlay area	a section of the .PR file reserved for overlays.
overlay delimiter	the character !; you include them on a Link command line to separate overlays.
overlay area delimiter	the series !* and *!; you use them on a Link command line to mark object files that can potentially contribute to the overlay file.
page	2000_8 sequential words of memory.
partition	a named contiguous section of a program file with the same set of attributes.
PARTSYM switch	a switch attached to the name of a partition or symbol in the Link command line. See Table 5-3.
.PR file	a program file that can run on AOS; also a program file that can run on AOS/VS.
program file	an executable file.
process	the executable physical memory copy of a .PR file.
.RB file (relocatable binary)	the RDOS equivalent of an .OB file; that is, a series of RLDR-compatible object blocks generated by an RDOS language processor and usually having the file name extension .RB.
relocatable code	datawords that Link can reposition (relocate) anywhere within a broad memory range in the .PR or .OL file. (Compare absolute code.)
relocation	the process in which Link assigns a value to a relocatable dataword or symbol.
relocation base	the lowest address of a partition; the value of a symbol.
relocation dictionary entry	an element within certain object blocks that Link uses to resolve the value of a particular dataword.
relocation entry	an element within certain object blocks that Link uses to resolve symbol values, or to allocate space for a set of datawords in a program file or debugger file.
relocation operation	a number within a relocation entry or relocation dictionary entry corresponding to a function. Link uses this function to calculate values for datawords and symbols and to calculate displacements from partitions.
Resource Calls	the system calls ?RCALL, ?KCALL, and ?RCHAIN; used to load and release overlays and to jump to the address of an entry symbol.
resource manager	a routine stored in URT.LB which Link loads (conditionally) into the .PR file to handle Resource Calls in some situations.
root	the part of a process that is stored in a program file; the part of a process that is not stored in an .OL file.
shared pages	physical memory pages accessible to more than one process.
.ST file	a nonexecutable file generated by Link and used for debugging. The file contains the names and values of all global symbols (and, optionally, local symbols) in the program.
stack	a special sequential section of logical address space initalized by Link and used by the CPU for certain machine language instructions.

.SV file	a program file that RDOS can execute; or, a program file that RTOS can execute.
switch	an optional element of a Link or LFE command line; a slash followed by an alphanumeric character or character string (e.g., /ALPHA).
SYS.LB	the RDOS system library.
system library	a library that Link scans by default; it contains vital routines that allow your process to communicate with the operating system at runtime.
system tables	Link-generated tables stored at the beginning of NREL; used at runtime by the operating system and by the process itself.
unshared pages	physical memory pages that only one process may access.
URT.LB	the AOS system library.
virtual overlays	overlays that reside outside logical address space but within physical memory; used in RDOS and RTOS only. By contrast, conventional overlays reside on disk until a process loads them into physical memory.
ZREL	addresses 00000 through 00377_8. Addresses that can be accessed from any location in memory by an instruction with an 8-bit displacement.

End of Glossary

Index

Within this index, "f" or "ff" after a page number means "and the following page" (or "pages"). In addition, primary page references for each topic are listed first. Commands, calls, and acronyms are in uppercase letters (e.g., CREATE); all others are lowercase.

CUT ALONG DOTTED LINE

Data General

users group Installation Membership Form

Name ______ Position ______ Date ______

Company, Organization or School ______

Address ______ City ______ State ______ Zip ______

Telephone: Area Code ______ No. ______ Ext. ______

1. Account Category

☐ OEM
☐ End User
☐ System House
☐ Government
☐ Educational

2. Hardware

	Qty. Installed	Qty. On Order
M/600		
COMMERCIAL ECLIPSE		
SCIENTIFIC ECLIPSE		
AP/130		
CS Series		
Mapped NOVA		
Unmapped NOVA		
microNOVA		
Other (Specify) ______		
______		
______		

3. Software

☐ AOS ☐ RDOS
☐ DOS ☐ Other
☐ MP/OS

Specify ______

______

4. Languages

☐ Algol ☐ Assembler
☐ DG/L ☐ Fortran
☐ Cobol ☐ RPG II
☐ PASCAL ☐ PL/1
☐ Business BASIC ☐ Other
☐ BASIC

Specify ______

______

5. Mode of Operation

☐ Batch (Central)
☐ Batch (Via RJE)
☐ On-Line Interactive

6. Communications

☐ HASP ☐ CAM
☐ RJE80 ☐ XODIAC
☐ RCX 70 ☐ Other

Specify ______

______

7. Application Description

○ ______

______

______

______

______

______

8. Purchase

From whom was your machine(s) purchased?

☐ **Data General Corp.**
☐ Other
Specify ______

______

9. Users Group

Are you interested in joining a special interest or regional Data General Users Group?

○ ______

______

Data General

Data General Corporation, Westboro, Massachusetts 01580. (617) 366-8911

FOLD FOLD

TAPE TAPE

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

Postage will be paid by addressee:

DataGeneral

ATTN: Users Group Coordinator (C-228)
4400 Computer Drive
Westboro, MA 01581

Data General

TP____________

TIPS ORDER FORM
Technical Information & Publications Service

BILL TO:	SHIP TO: (if different)
COMPANY NAME____________	COMPANY NAME____________
ADDRESS ____________	ADDRESS ____________
CITY ____________	CITY ____________
STATE________ ZIP ________	STATE________ ZIP ________
ATTN: ____________	ATTN: ____________

QTY	MODEL #	DESCRIPTION	UNIT PRICE	LINE DISC	TOTAL PRICE

(Additional items can be included on second order form)

[Minimum order is $50.00]	TOTAL	
Tax Exempt #__________ or Sales Tax (if applicable)	Sales Tax	
	Shipping	
	TOTAL	

METHOD OF PAYMENT

☐ Check or money order enclosed
For orders less than $100.00

☐ Charge my ☐ Visa ☐ MasterCard
Acc't No.__________ Expiration Date__________

☐ Purchase Order Number:__________

SHIP VIA

☐ DGC will select best way (U.P.S or Postal)

☐ Other:
☐ U.P.S. Blue Label
☐ Air Freight
☐ Other ____________

NOTE: ORDERS LESS THAN $100, INCLUDE $5.00 FOR SHIPPING AND HANDLING.

Person to contact about this order ____________ Phone __________ Extension ______

Mail Orders to:

Data General Corporation
Attn: Educational Services/TIPS F019
4400 Computer Drive
Westboro, MA 01580
Tel. (617) 366-8911 ext. 4032

Buyer's Authorized Signature Date
(agrees to terms & conditions on reverse side)

Title

DGC Sales Representative (If Known) Badge #

DISCOUNTS APPLY TO MAIL ORDERS ONLY

012-1780

educational services

CU… ALONG DOTTED LINE

DATA GENERAL CORPORATION
TECHNICAL INFORMATION AND PUBLICATIONS SERVICE
TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form shown on the reverse hereof which is accepted by DGC.

1. **PRICES**
   Prices for DGC publications will be as stated in the Educational Services Literature Catalog in effect at the time DGC accepts Buyer's order or as specified on an authorized DGC quotation in force at the time of receipt by DGC of the Order Form shown on the reverse hereof. Prices are exclusive of all excise, sales, use or similar taxes and, therefore are subject to an increase equal in amount to any tax DGC may be required to collect or pay on the sale, license or delivery of the materials provided hereunder.

2. **PAYMENT**
   Terms are net cash on or prior to delivery except where satisfactory open account credit is established, in which case terms are net thirty (30) days from date of invoice.

3. **SHIPMENT**
   Shipment will be made F.O.B. Point of Origin. DGC normally ships either by UPS or U.S. Mail or other appropriate method depending upon weight, unless Customer designates a specific method and/or carrier on the Order Form. In any case, DGC assumes no liability with regard to loss, damage or delay during shipment.

4. **TERM**
   Upon execution by Buyer and acceptance by DGC, this agreement shall continue to remain in effect until terminated by either party upon thirty (30) days prior written notice. It is the intent of the parties to leave this Agreement in effect so that all subsequent orders for DGC publications will be governed by the terms and conditions of this Agreement.

5. **CUSTOMER CERTIFICATION**
   Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub-licensee of the software which is the subject matter of the publication(s) ordered hereunder.

6. **DATA AND PROPRIETARY RIGHTS**
   Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure.

7. **DISCLAIMER OF WARRANTY**
   DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS SUPPLIED HEREUNDER.

8. **LIMITATIONS OF LIABILITY**
   IN NO EVENT SHALL DGC BE LIABLE FOR (I) ANY COSTS, DAMAGES OR EXPENSES ARISING OUT OF OR IN CONNECTION WITH ANY CLAIM BY ANY PERSON THAT USE OF THE PUBLICATION OF INFORMATION CONTAINED THEREIN INFRINGES ANY COPYRIGHT OR TRADE SECRET RIGHT OR (II) ANY INCIDENTIAL, SPECIAL, DIRECT OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOSS OF DATA, PROGRAMS OR LOST PROFITS.

9. **GENERAL**
   A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer.

DISCOUNT SCHEDULES

DISCOUNTS APPLY TO MAIL ORDERS ONLY.

LINE ITEM DISCOUNT

5-14 manuals of the same part number - 20%
15 or more manuals of the same part number - 30%

DISCOUNTS APPLY TO PRICES SHOWN IN THE CURRENT TIPS CATALOG ONLY.

Data General

TIPS ORDERING PROCEDURE:

Technical literature may be ordered through the Customer Education Service's Technical Information and Publications Service (TIPS).

1. Turn to the TIPS Order Form.
2. Fill in the requested information. If you need more space to list the items you are ordering, use an additional form. Transfer the subtotal from any additional sheet to the space marked "subtotal" on the form.
3. Do not forget to include your MAIL ORDER ONLY discount. (See discount schedules on the back of the TIPS Order Form.)
4. Total your order. (MINIMUM ORDER/CHARGE after discounts of $50.00.)

   If your order totals less than 100.00, enclose a certified check or money order for the total (include sales tax, or your tax exempt number, if applicable) plus $5.00 for shipping and handling.
5. Please indicate on the Order Form if you have any special shipping requirements. Unless specified, orders are normally shipped U.P.S.
6. Read carefully the terms and conditions of the TIPS program on the reverse side of the Order Form.
7. Sign on the line provided on the form and enclose with payment. Mail to:

   TIPS
   Educational Services – M.S. F019
   Data General Corporation
   4400 Computer Drive
   Westboro, MA 01580
8. We'll take care of the rest!

educational services

User Documentation Remarks Form

Your Name ______________________ Your Title ______________________

Company ______________________

Street ______________________

City ______________________ State __________ Zip __________

We wrote this book for you, and we made certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve the manual. Please take a few minutes to respond. Thank you.

Manual Title ______________________ Manual No. __________

Who are you?	☐ EDP Manager	☐ Analyst/Programmer	☐ Other __________
	☐ Senior Systems Analyst	☐ Operator	__________

What programming language(s) do you use? ______________________

How do you use this manual? *(List in order: 1 = Primary Use)* ______________________

___ Introduction to the product	___ Tutorial Text	___ Other
___ Reference	___ Operating Guide	__________

About the manual:		Yes	Somewhat	No
	Is it easy to read?	☐	☐	☐
	Is it easy to understand?	☐	☐	☐
	Are the topics logically organized?	☐	☐	☐
	Is the technical information accurate?	☐	☐	☐
	Can you easily find what you want?	☐	☐	☐
	Does it tell you everything you need to know	☐	☐	☐
	Do the illustrations help you?	☐	☐	☐

If you have any comments on the software itself, please contact Data General Systems Engineering.
If you wish to order manuals, use the enclosed TIPS Order Form (USA only).

Remarks:

Date

134-664

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

POSTAGE WILL BE PAID BY ADDRESSEE

DataGeneral

User Documentation, M.S. E-111
4400 Computer Drive
Westborough, Massachusetts 01581

User Documentation Remarks Form

Your Name ______________________ Your Title ______________________

Company ______________________

Street ______________________

City ______________________ State __________ Zip __________

We wrote this book for you, and we made certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve the manual. Please take a few minutes to respond. Thank you.

Manual Title ______________________ Manual No. __________

Who are you? ☐EDP Manager ☐Analyst/Programmer ☐Other __________
☐Senior Systems Analyst ☐Operator __________

What programming language(s) do you use? ______________________

How do you use this manual? *(List in order: 1 = Primary Use)* ______________________

___ Introduction to the product ___ Tutorial Text ___ Other
___ Reference ___ Operating Guide __________

About the manual:		Yes	Somewhat	No
	Is it easy to read?	☐	☐	☐
	Is it easy to understand?	☐	☐	☐
	Are the topics logically organized?	☐	☐	☐
	Is the technical information accurate?	☐	☐	☐
	Can you easily find what you want?	☐	☐	☐
	Does it tell you everything you need to know	☐	☐	☐
	Do the illustrations help you?	☐	☐	☐

If you have any comments on the software itself, please contact Data General Systems Engineering.
If you wish to order manuals, use the enclosed TIPS Order Form (USA only).

Remarks:

Date

134-664

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

POSTAGE WILL BE PAID BY ADDRESSEE

DataGeneral

User Documentation, M.S. E-111
4400 Computer Drive
Westborough, Massachusetts 01581

Data General Corporation, Westboro, MA 01580

093-000254-02