Data General

Software Documentation

Advanced Operating System (AOS) Binder User's Manual

Advanced Operating System (AOS) Binder User's Manual

093-000190-03

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. 093-000190
©Data General Corporation, 1976, 1977, 1978, 1984
All Rights Reserved
Printed in the United States of America
Revision 03, October 1984
Licensed Material - Property of Data General Corporation

NOTICE

DATA GENERAL CORPORATION (DGC) HAS PREPARED THIS DOCUMENT FOR USE BY DGC PERSONNEL, LICENSEES, AND CUSTOMERS. THE INFORMATION CONTAINED HEREIN IS THE PROPERTY OF DGC; AND THE CONTENTS OF THIS MANUAL SHALL NOT BE REPRODUCED IN WHOLE OR IN PART NOR USED OTHER THAN AS ALLOWED IN THE DGC LICENSE AGREEMENT.

DGC reserves the right to make changes in specifications and other information contained in this document without prior notice, and the reader should in all cases consult DGC to determine whether any such changes have been made.

THE TERMS AND CONDITIONS GOVERNING THE SALE OF DGC HARDWARE PRODUCTS AND THE LICENSING OF DGC SOFTWARE CONSIST SOLELY OF THOSE SET FORTH IN THE WRITTEN CONTRACTS BETWEEN DGC AND ITS CUSTOMERS. NO REPRESENTATION OR OTHER AFFIRMATION OF FACT CONTAINED IN THIS DOCUMENT INCLUDING BUT NOT LIMITED TO STATEMENTS REGARDING CAPACITY, RESPONSE-TIME PERFORMANCE, SUITABILITY FOR USE OR PERFORMANCE OF PRODUCTS DESCRIBED HEREIN SHALL BE DEEMED TO BE A WARRANTY BY DGC FOR ANY PURPOSE, OR GIVE RISE TO ANY LIABILITY OF DGC WHATSOEVER.

This software is made available solely pursuant to the terms of a DGC license agreement which governs its use.

CEO, DASHER, DATAPREP, ECLIPSE, ENTERPRISE, INFOS, microNOVA, NOVA, PROXI, SUPERNOVA, PRESENT, ECLIPSE MV/4000, ECLIPSE MV/6000, ECLIPSE MV/8000, TRENDVIEW, SWAT, GENAP, and **MANAP** are U.S. registered trademarks of Data General Corporation, and **AZ-TEXT, DG/L, DG/GATE, DG/XAP, ECLIPSE MV/10000, GW/4000, GDC/1000, REV-UP, XODIAC, DEFINE, SLATE, microECLIPSE, DESKTOP GENERATION, BusiPEN, BusiGEN** and **BusiTEXT** are U.S. trademarks of Data General Corporation.

Advanced Operating System (AOS)
Binder User's Manual
093-000190

Revision History:

Original Release - April 1976

First Revision - April 1977

Second Revision - June 1978

Third Revision - October 1984

CONTENT UNCHANGED
The content and change indicators in this revision are unchanged from 093-000190-02. This revision changes only printing and binding details.

Contents

Chapter 1 - Introduction to the Binder

Terms and Conventions 1-1
Partitions 1-1
Overlays 1-2
Shared Libraries 1-4
Non-Shared Libraries 1-4
Limit Symbols 1-4
Undefined Symbols 1-4
Program References to NMAX and ZMAX 1-4
Size and Location of Unlabeled Common 1-4
Stacks 1-5
Accumulating Symbol 1-5
Debugger Symbol File 1-5
Debugger Lines File 1-6

Chapter 2 - How to Operate the Binder

Operating Procedures 2-1
Command Files 2-3
Error Messages 2-3
Example of Binder Operation 2-5

Appendix A - Object File Formats

General Block Format A-1
Data Block A-2
Title Block A-4
End Block A-5
Unlabeled Common Block A-5
External Symbols Block A-5
Entry Symbols Block A-6
Local Symbols Block A-6
Library Start and End Blocks A-7
Address Information Block A-7
Shared Library Block Header A-7
Task Block A-7
Limit Block A-8
Named Common Block A-8
Accumulating Symbol Block A-8
Debugger Symbols Block A-9
Debugger Lines Block and Lines Title Block A-9
Object File Illustrations A-9

Illustrations

Figure	Caption	
1-1	Page Zero Map	1-1
1-2	Context Partition Map	1-2
1-3	Overlay Area Built for a Single Partition	1-3
1-4	Overlay Areas with Two Partition Directives	1-3
1-5	Overlay Areas with One Partition Directive	1-3
2-1	Sample Source Listings	2-5
2-2	Sample Assembler Listing	2-5
2-3	Sample Binder Output Listing	2-6
A-1	General Block Format	A-1
A-2	Data Block Structure	A-2
A-3	Title Block Structure	A-4
A-4	End Block Structure	A-5
A-5	Unlabeled Common Structure	A-5
A-6	External Symbols Block Structure	A-5
A-7	Symbol Entry in External Symbols Block	A-5
A-8	Entry Symbols Block Structure	A-6
A-9	Symbol Entry in Entry Symbols Block	A-6
A-10	Local Symbols Block Structure	A-6
A-11	Address Information Block Structure	A-7
A-12	Task Block Structure	A-7
A-13	Limit Block Structure	A-8
A-14	Named Common Block Structure	A-8
A-15	Accumulating Symbol Block	A-8
A-16	Debugger Symbols Block	A-9
A-17	Debugger Lines Block	A-9
A-18	EXMPL Assembler Listing	A-10
A-19	Contents of EXMPL.OB	A-10

Chapter 1
Introduction to the Binder

Terms and Conventions

The Binder utility links together object modules, produced by a language processor, to build a program file. The Binder also optionally produces an overlay file. A *program file* is an executable core-image; it is built and resides on disk until it is brought into a main memory context for execution. In the course of constructing a program file, the Binder builds a *symbol table file*. This file contains all names defined as global symbols within modules used in building the program and overlay files, and lists their values. Its name is the same as the name of the program file, with an extension ".ST".

Unless noted otherwise, all numbers used in this publication are decimal except core memory addresses, which are octal values.

The operating system allocates memory in pages of 2048 bytes each; the first page is page zero, the second page is page one, etc. Memory locations used by each program file can be considered in two categories: lower page zero, and all other memory. Lower page zero extends from location 0 through 377 inclusive. Locations 0-47 are reserved for absolute references such as "JSR @17", the call to the system call processor. Code or data which can be relocated anywhere from locations 50-377 is "lower page zero relocatable" or ZREL; the highest used ZREL address is "?ZMAX-1". Information that can be relocated above 377 is "normal relocatable" or *NREL*. The actual start of NREL program code or data will follow the last of a series of system tables such as the UST and task control block(s).

The area labeled "system tables" in Figure 1-1 will always contain a *User Status Table* (UST), one or more task control blocks (TCBs), and--if an overlay file has been defined--an *overlay directory*. You don't need to understand these tables to understand the Binder. However, since these tables are of considerable programming interest and appear in user space, they are described in the *AOS Programmer's Manual*.

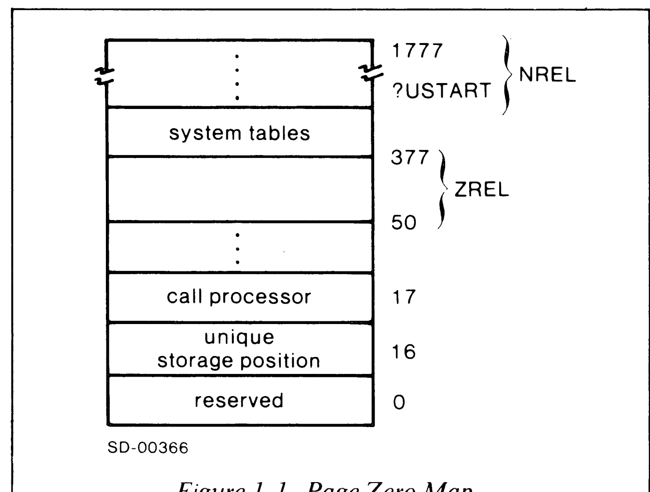


Figure 1-1. Page Zero Map

Partitions

A partition is an area of memory where a specific kind of code resides. The Binder groups code and data from object files (OBs) into separate memory partitions according to information contained either within the objects or in the Binder command line. By default, the Binder defines six partitions, one of each of the following six types:

- absolute
- ZREL
- NREL non-shared code (overlays permitted)
- NREL shared code (overlays permitted)
- NREL non-shared data
- NREL shared data

All partitions except the absolute partition consist of a sequence of contiguous memory locations. The absolute partition, by contrast, is merely a set of locations reserved by specific numeric addresses, which need not be in sequence. One or more OBs can specify additional NREL partitions to the Binder. Any OB may contain data destined for any one or for all of the partitions.

Strictly speaking, each of the six partition types can contain either code or data. The essential difference between code and data partitions is that code partitions may contain overlay areas, while data partitions may not. Shared and unshared partitions, on the other hand, are distinguished by their size and location within a context. Shared partitions always start at the beginning of a page boundary and by default are allocated in multiples of 2048 bytes at the top of the context (they can be forced elswhere). Unshared partitions have no such size or page boundary constraint, and immediately follow the system tables area.

Starting at the beginning of user NREL, the Binder allocates partitions in the following sequence: *Non-shared partitions* are first; they follow immediately the last system table or unlabeled common. Default non-shared partitions will reside at lower memory locations than any additional non-shared partitions you create. Specifically, from low to high addresses, the Binder creates non-shared partitions in the following order: Default non-shared code, default non-shared data, and non-shared code and data partitions in the order you create them. The location following the last address used in the highest non-shared partition is called *?NMAX*.

Between ?NMAX-1 and the next partition there may exist an area of memory which you can use for scratchpad or other use; you can determine and allocate the size and location of this area with system calls ?MEM and ?MEMI respectively.

At the upper end of the context is the shared area, consisting of both *shared code* and *shared data partitions*. During the first pass of its execution, the Binder determines the necessary size of this area, so it can begin each shared partition of a page boundary and allocate each as a multiple of 2048 bytes.

The order of shared partitions, from low to high addresses, is as follows: default shared data partition, default shared code partition, and other shared data and code partitions in the order you create them.

Figure 1-2 shows the relative positioning of partitions within a context.

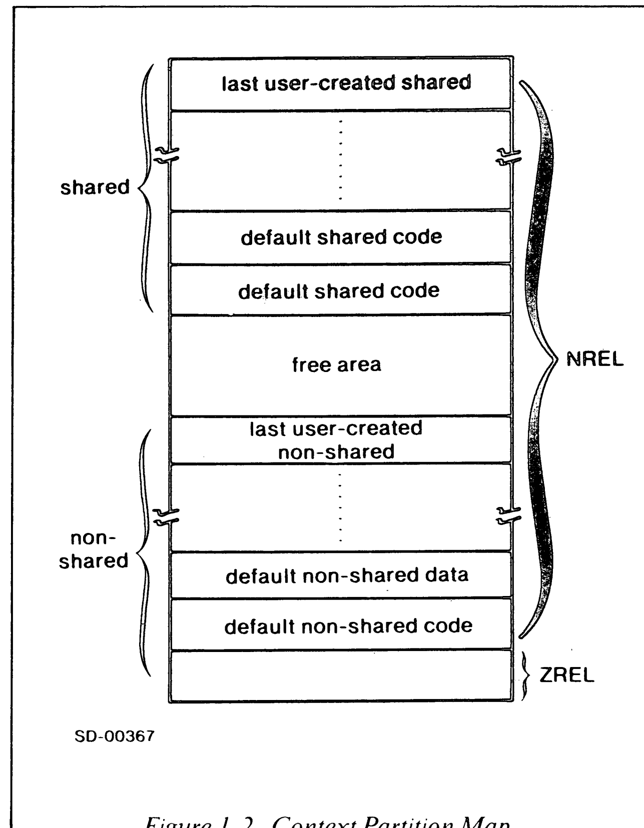


Figure 1-2. Context Partition Map

Overlays

Overlays allow you effectively to increase your address space. Your program can load one of many different code modules, called overlays, from disk into the same main memory area at different times.

Overlays reside in a disk overlay file and are read, upon command, into overlay areas within a root; the *root* is equivalent to the contents of the program file (identified by its ".PR" extension). The binder produces the overlay file and names it programname.OL. Associated with each overlay area are one or more overlays, where each overlay area is as large as the largest overlay associated with it. Overlay areas are allocated and built as multiples of 512 bytes in unshared areas, and as multiples of 2048 bytes in shared areas.

There may be up to 63 overlay areas within memory. Each overlay area can have up to 511 separate overlays associated with it. The largest overlay to be read into each overlay area determines the basic size of the overlay area. You can define a *total overlay area* whose size is some multiple of the basic overlay size. This permits several different overlays to be read side-by-side into *basic overlay areas* within the same total overlay area. Since the system may place these overlays into any of the basic areas within a total area, you must write these overlays relocatably (i.e., with code that is position-independent). Having stated that a total overlay area can be allocated as a multiple of its basic size, for simplicity this manual presents illustrations of overlays with total overlay areas which equal the basic overlay area size.

The Binder command line, described in the next chapter, defines overlays. Compilers can specify overlays in object files by including an address information block in one or more of the OB files which will be presented to the Binder. If a language supports this feature, its reference manual describes the way you can generate the required address information blocks.

The Binder command line reserves an overlay area for every series of objects within square brackets:

[a,b c,d]

A left bracket directs the Binder to build the beginning of an overlay area at the current base of the appropriate code partition. The *current base* of a partition is the address in that partition where the Binder will begin loading the current object. (Data partitions, the ZREL partition, and absolute partitions never receive overlays.) A comma in the command line terminates an overlay in the disk file. A right bracket terminates the area itself. Thus, in the above illustration, object *a* would form the first overlay to be read into the area, objects *b* and *c* would form the second overlay, and object *d* would form the third overlay. These overlays would then "belong" to a single overlay area.

If none of the objects for an overlay area contain code for any partition, that overlay is ignored in that partition; there can be no zero-length overlays. If one of the objects between square brackets contains a directive to the Binder to create a new partition, the Binder starts an overlay area immediately in that new partition. Each start of an overlay area in a non-shared partition begins at the current base; each overlay area is page-aligned to the next 2048-byte boundary in a shared partition.

Figure 1-3 shows how the Binder would construct an overlay file and an overlay area if all of the code in objects *a, b, c,* and *d* were destined for a single partition.

Figure 1-4 shows how it would build the overlay file if objects *a* and *d* contained code destined only for partition *x*, but *b* and *c* contained code destined only for partition *y*. Finally, Figure 1-5 indicates how the overlay file would look if objects *a, c,* and *d* were to form overlays for partition *x*, but overlay *b* were to go to partition *y*.

The examples in these figures show how you can use *command line specification* to define overlays and to place code destined for different partitions into multiple overlay areas. It is also possible for an OB to specify complete overlay areas by using an address information block. This facility is not currently available with either the Macroassembler or DGC high-level languages; however, you can use OB specification if you write your own language processor. OB specification of overlays is described in Appendix A.

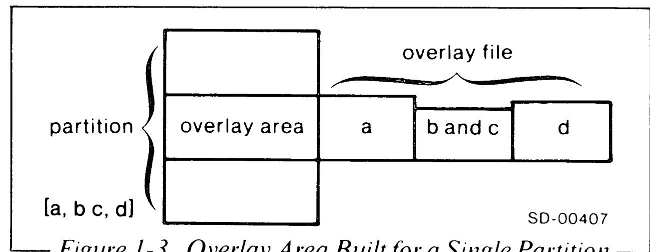


Figure 1-3. Overlay Area Built for a Single Partition

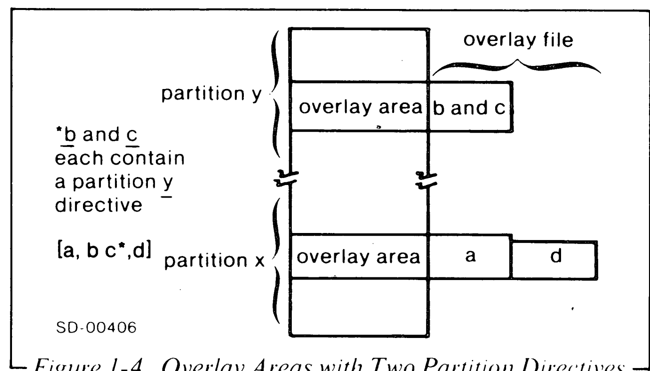


Figure 1-4. Overlay Areas with Two Partition Directives

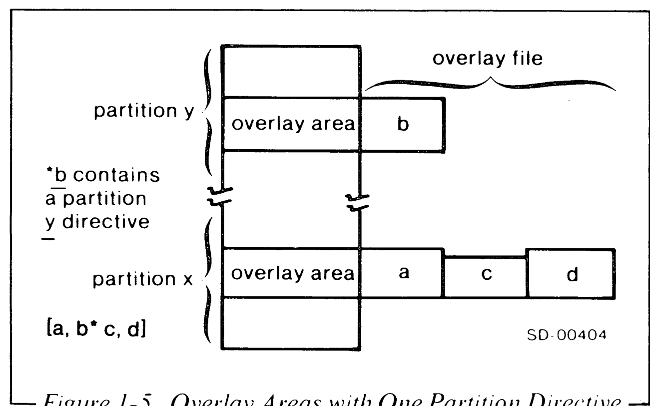


Figure 1-5. Overlay Areas with One Partition Directive

Shared Libraries

A shared library is a collection of program modules. When you request the system to load a module from a shared library, it first checks to see if the module is already in memory. If the module is in memory, the system does not read another copy into core but maps its area into your context. If the module is not in memory, the system reads it in from disk.

Shared routines are dynamically relocatable modules, built by the Shared Library Builder utility into a *shared library.* When a shared library is built it is assigned a numeric identifier from 0 to 63. Shared routines, by definition, are always read into shared memory areas. Shared routines are also always built in multiples of 2048 bytes.

Shared routines are always written to be reentrant, and usually they are position-independent. A shared routine can be written which is position-dependent only if it will be bound into a specific area in the program file (Binder switch /B) and if you take care to ensure that all absolute addresses are maintained consistently. Designing shared routines to be position-independent, and allowing the system to relocate them wherever required during the execution of the program, does away with these restrictions. Discussions of shared routines in the remainder of this manual presume position-independence. Binder function switch /M specifies the number of 2K-byte blocks which will be reserved as a single shared area for use by all shared routines.

Shared routines are loaded using one of three generalized procedure calls: ?RCALL, ?KCALL, and ?RCHAIN. Since generalized procedures can be written independently of the medium where they will ultimately reside, the Binder may change certain procedure calls into EJSR instructions. This would occur, for example, when a procedure call is made to a shared routine bound into the permanent root context. For more information on general procedure management and shared routines, consult Chapter 3 of the *AOS Programmer's Manual.*

Non-Shared Libraries

A non-shared library file (simply called a "library") is a collection of object files. You build libraries with the Library File Editor utility. Later, you can bind modules from a library into either the shared or unshared partition of the root.

Normally, the Binder will bind only those modules from a library that satisfy an unsatisfied external reference. If two modules with the same entry point exist in one or more libraries which you supplied in the Bind command line, only the first one will be bound into the program.

If you insert the .FORC pseudo-op in a module, the module will be unconditionally bound into the program, whether or not it satisfies an external reference from a previous module. You can also specify that a module should be force-bound when you build the library with the Library File Editor (See the *AOS Library File Editor* for details).

Limit Symbols

One binary can declare a symbol as a limit symbol. If a later binary declares the symbol as an entry, the Binder will not load code or data in the second binary that would be loaded at an address greater than or equal to the relocated value of the symbol. Only code and data in the same partition as the entry symbol will be affected. For more information about limit symbols, please see the discussion of .LMIT in the *Macroassembler Reference Manual.*

Undefined Symbols

By default, the Binder will resolve undefined symbols to be "-1". You can change this value by loading a user-defined symbol, "UNDEF", to which you have assigned a value other than -1.

Program References to NMAX and ZMAX

The Binder builds symbols named ?ZMAX and ?NMAX which contain the values of ZMAX and NMAX respectively. These symbols, containing the values of ZMAX and NMAX at bind time, will appear in the symbol map output by the Binder and you can reference them from within your code to get their initial values.

Size and Location of Unlabeled Common

The values of symbols ?CSZE and ?CLOC indicate the size and location respectively of unlabeled common. These symbols are displayed in the symbol map produced by the Binder, and you can reference them from within a program like any other symbols.

Stacks

If you do not specify a stack size when you bind your program (via BIND switch /Z), the Binder will allocate a stack for you. It does this by

- initializing the stack pointer (location 40_8) to an address in the scratch-pad area at the top of the unshared partition.
- adjusting ?NMAX to include the stack size. The default stack size is the fewest number of words that enable your program to execute all system calls.
- initializing the stack limit (location 42_8) to the size of the stack.

If you place any non-zero quantity in the stack pointer or the stack limit, the Binder assumes this value is an address and does not alter it.

In addition to allocating the stack, the Binder will provide you with a default stack-overflow handler (entry SFALT) from URT.LB. You can optionally write your own stack-overflow handler named SFALT and bind it into the program.

Accumulating Symbol

The accumulating symbol is used to maintain a count. Consult the description of ".ASYM" in the *AOS Macroassembler Reference Manual* or the appropriate language manual for information regarding its use. In essence, this symbol is given an initial value, either in the Binder command line or in a module. The next time that the symbol is referenced (in an accumulating symbol block), the Binder updates it, using the relocation base and operation specified in that block. The following steps are performed to use this symbol as a counter.

First, the symbol is given an initial value (e.g., zero). Subsequent modules declare it first as an external, and then as an accumulating symbol. In the accumulating symbol block, the relocation base indicates the external declaration of the symbol itself, specifying word relocation. The Binder will add the previous value of the symbol to the current value, and this sum becomes the new symbol value.

The symbol's value can be manipulated in other ways using other relocation bases and other operations. The Macroassembler automatically generates the external reference and sets the relocation of the accumulating symbol to reference the symbol itself using word relocation. Thus Macroassembler users can utilize an accumulating symbol only as a counter. The essential difference between an accumulating symbol and an entry symbol is that an accumulator symbol does not receive multiple definition errors (the Binder gives these to other symbols).

Debugger Symbol File

The Binder can create two optional output files to help implement high-level language debuggers: the *debugger symbols file,* and the *debugger lines file.* Neither the Macroassembler nor Debug supports or uses this facility. The debugger symbols file has the name "programfile.DB", and the Binder creates it on the first occurrence of a debugger symbols block in an object file, if the function switch /D is used in the Binder command line. The format of the debugger symbols block is described in Appendix A.

The Binder copies data from the debugger symbols block(s) to the debugger symbols file. The symbols block contains a provision for performing relocation operations on the data. One of these relocation operations, "link", uses this data for scoping variables. Scoping of variables is a technique which permits distinct, modular procedures in the same program to use the same variable name. High-level language debuggers need this feature to find each of the variables bearing the same name.

In link relocation, the following operations occur. On the first use of the link operation, the Binder copies a 0 to the debugger symbols file. On subsequent uses of the link operation, it copies to the debugger symbols file the file position (in words) where the link operation was last used in the debugger symbols block. After it has processed the last object file it writes the file position of the last use of the link operation in word 0 of the debugger symbols file. (The data it has copied from the debugger symbols block commenced originally at word 1.)

Debugger Lines File

The second optional debugger file which the Binder can create is the debugger lines file. This file is also copied from object-file input; the Binder creates it upon the first occurrence of a debugger lines block, if the /D function switch is used. The format of this block is specified in Appendix A. (A second block type, the lines title block, must appear immediately before the end block in any OB file using the lines file feature.) The debugger lines file will have the name OB "programfile.DL" and the following format:

word 0	Address n+1
word 1	Copied data.
.	
.	
n	Copied data.
n+1	Number of objects.
n+2	Directory.

Each object file copied to the debugger lines file has an entry in the lines file directory, with the following format:

word 0	Word offset into the lines file where this object starts.
word 1	Word offset where this object ends.
words 2-3	Current base in the default NREL non-shared code partition. Current base plus the number of words of data in this object, i.e., the current base of the next object.
words 4-5	Current base in the default NREL shared code partition. Current base of the next object.
words 6-7	Current base in the default NREL non-shared data partition. Current base of the next object.
words 8-9	Current base in the default NREL shared data partition. Current base of the next object.
word 10	Count of words of lines title data.
words 11-n	Lines title data.

End of Chapter

Chapter 2
How to Operate the Binder

Operating Procedures

Use the CLI to issue Binder commands. Each Binder command names input object modules and directs the Binder to build an executable program file, with an optional overlay file. By default the Binder selects named input files with an extension ".OB", and produces a program file whose filename ends with the extension ".PR". Also by default, the Binder scans the user runtime library, URT.LB. This library contains code modules that are bound into the user context to execute certain types of system calls (such as task management calls).

You can modify the operation of the Binder by applying one or more switches in the command line. A *switch* is a right slash character followed immediately by a switch character or character sequence. Switch characters may be alphabetic or numeric; switch character sequences consist of one or more alphabetic characters followed by "=", then by either a numeric or alpabetic value. Switches applied directly to the "BIND" command name are *function switches;* other switches are *argument switches,* and their effect extends only to the arguments they modify.

The format of the Binder command is:

```
         { object module                            }
XEQ BIND {                                          }
         { [object modules...] command file/C       }
```

You must use a command file to bind overlays. (See Command Files, below). Function switches that you can use in the Binder command line are as follows; all numeric switch values and arguments are decimal.

Switch	Action
/B	Produce a listing of the symbol file with symbols ordered both alphabetically and numerically.
/D	Produce a Debugger Symbol File, used in implementing high-level language debuggers. See the Debugger Symbol File description in Chapter 1 of this manual.
/E	Output the load map to the output file, even if a listing file has been specified.
/H	List all numbers in hexadecimal.
/I	Build a non-executable program file, lacking a UST, TCBs, and all other system databases; URT.LB is not scanned. This switch allows you to build a file larger than 32K.
/K=n	Allocate n TCBs for multitask use, regardless how many (if any) are specified in a .TSK statement.
/L	Produce a listing file, using the currently-specified CLI @LIST file.
/L=name	Produce a listing file, using the file name.
/M=n	Reserve n 2K-byte blocks of memory for shared library routines.
/N	Do not scan the user runtime library, URT.LB.
/O	Suppress error flags whenever bind-overwrites occur. A bind-overwrite occurs when one module places code in one or more locations and a succeeding module overwrites these locations.

Switch	Action
/P=name	Assign name to a program file. If you don't use this switch, the program file will be given the name of the first module in the Binder command line (with the extension ".PR").
/S	Produce a shared routine for a shared library.
/T=n	Specify the highest address in the shared partition. If n is not a multiple of 2048 bytes, the Binder rounds it down to the next lower 2048-byte multiple. If you don't use this switch, your shared code partition will be placed at the top of the 64K-byte context.
/Z=n	Specify the size of the stack for the default task. If you omit this switch, a 30-word stack is allocated.

Argument switches you can use are:

Switch	Action
/AM=n	Set a total overlay area equal to n basic areas. Apply this switch only to a right bracket in an overlay specification.
name/B	Bind the externally referenced routines from name shared library into the root context.
/C	Specify the name of a command file (required when defining overlays using square brackets).
/D	Load non-shared code in this module as non-shared data. This is currently the only way that you can create a non-shared data partition, if you are using the Macroassembler.
/F	Use the starting execution address contained in the last block of this object file. (See Appendix A.) If you use this switch, the BINDER will ignore all valid starting addresses in later object files.
/H	Load non-shared code in this module as shared code. If you apply this switch to a non-shared library, modules extracted from the library will be bound into the shared code partition. Note that the standard way you place code into the shared code partition when using the Macroassembler, is to use the ".NREL 1" pseudo-op in your code.
/O	Allows overwrites in this module (see /O function switch).
/R	Issue a warning if any code in this module is not position-independent.
/S	Convert shared code modules to unshared code modules. For example, FORTRAN currently produces shared code only; this switch lets a FORTRAN program be unshared. *
/U	Load local symbols from this module into the symbol file. This switch will work only if you applied /U to this same module in the earlier Macroassembler command.
name/V=number	Create an accumulating symbol, name, with absolute relocation, and initialize it to the value number. You can create more than one accumulating symbol by using this switch repetitively. If you name an accumulating symbol that is also defined, within a module, as an accumulating symbol, there is no conflict; any value the .ASYM pseudo-op specifies will simply be added to the current sum of the accumulating symbol.
name/X	Exclude the shared library routine, name, from being bound into the root context. This switch must immediately follow the /B switch. For example, this command ...LIB/B A/X B/X... is valid. This expression would exclude routines A and B from being bound into the root. This command ...LIB/B A/X C/R B/X... does not use the /X switch correctly; routine B would not be excluded in this case.
n/Z	Set the current ZREL base to n. If the current ZREL base exceeds n, then the current base remains as is and n is ignored.

Command Files

If you want to produce one or more overlay files, you must supply a command filename as the last argument in the command line. If you are binding only overlay files and no program files, the command filename may be the only argument in the Bind command line.

The command file must contain all command line information including object filenames, switches, and brackets, starting at or before the first left bracket. No command line information can follow commandfile/C.

You can use the CLI command CREATE to build command files; for example:

```
)CREATE/I CFILE)
))[OVLY1, OVLY2, OVLY3] [OVLY4, OVLY5])
))))
```

This creates command file CFILE, which contains five object file names; the two sets of brackets will define two overlay areas for these overlays. (Brackets reserve overlay areas, as described in Chapter 1.) Now, the Bind command

```
XEQ BIND MYPROG CFILE/C)
```

creates MYPROG.PR and overlay file MYPROG.OL. When MYPROG executes, it will have two overlay areas in memory -- area 0, which will receive overlays OVLY1, OVLY2, or OVLY3, and area 1, which will receive OVLY4 or OVLY5. MYPROG will load and use the overlays one-by-one into each area as it needs them.

Error Messages

If the Binder encounters any error conditions during its execution, it will output one or more error messages to the output file (by default, you will receive error messages on the console printer or screen). The following alphabetical list names these messages and explains their meanings. Error messages that refer to a symbol will display the name of that symbol.

ATTEMPT TO MAP OVERLAY TO DATA PARTITION

One or more overlays were declared within an OB, and you attempted to place one or more of them into a partition declared as a data partition. This error can occur only when an Address Information Block is used (see Appendix A).

ATTEMPT TO OVERWRITE DATA

You attempted to place two data words in the same location. No error is flagged if the two data words are equal, or if the first word is zero.

ATTEMPT TO RELOCATE DATA FROM UNDEFINED SYMBOL

The Binder attempted to relocate data from an undefined symbol. It needs the value of the named symbol to relocate the data word at the named location, but the symbol is still undefined after the Binder's first pass.

COMMAND FILE SYNTAX ERROR

You have an illegal sequence in a command file.

DISPLACEMENT OVERFLOW

Overflow occurred into the left byte while the Binder performed a displacement location operation (see Appendix A, Data Blocks). This occurs typically when the code references an NREL value as though it were ZREL. Alternatively, the code may have attempted to do a PC-relative access in a single-word memory reference instruction, but the effective address was too far away.

DUPLICATE SEARCH OF URT.LB

You explicitly listed URT.LB in your command line and did not use the /N command switch to suppress the automatic search of URT.LB. This condition is a warning, and does not affect the program file produced.

ENTO RESOLVED TO SYMBOL

The named symbol is an .ENTO symbol, but the code defined it in terms of another symbol.

EXTERNAL SYMBOL REFERENCE OUT OF RANGE

A relocation dictionary entry in a *DATA BLOCK* is too large. Re-assemble or re-compile the source to produce a new object.

INSUFFICIENT CONTIGUOUS BLOCKS

The Binder cannot create the overlay file because there is an insufficient amount of available contiguous disk space.

INSUFFICIENT MEMORY FOR BINDER

You did not allocate enough memory to run the Binder.

LIBRARY START BLOCK ERROR

There are two or more START BLOCKS in the library. You must rebuild it with LFE.

LIMIT FOLLOWS LIMITED SYMBOL

The OB declaring a symbol as a limit came after the OB defining the symbol as an entry symbol.

MISUSE OF ACCUMULATING SYMBOL

You're using an accumulating symbol in an invalid way.

MULTIPLY DEFINED SYMBOL

The named symbol was defined with different values in two or more OBs.

NO CURRENT LIST

You used the /L command switch and no LIST file is currently assigned. See the CLI manual for a description of LIST file assignment.

NON-POSITION INDEPENDENT CODE

You used an argument /R switch, and the data for the named location is not position-independent.

NO OBJECT FILE SPECIFIED

No object files were specified in the Bind command line. There must be at least one object filename in the command line or command file.

OB ERROR IN LIBRARY - BLOCK NUMBER INCORRECT

A block sequence number error occurred in a library. You will get this error if you attempt to treat as a library a file which is not in the prescribed library format.

OB ERROR - BLOCK NUMBER INCORRECT

A sequence number in an OB block is incorrect. This error is usually triggered if you attempt to bind a non-OB file.

OB FILE ERROR - DATA RELOCATION OUT OF RANGE

An erroneous entry appears in the relocation dictionary of a data block.

OVERLAY COMMON REF IN ROOT OR DIFF. OVERLAY

A common area that resides in an overlay was referenced from the root or from a different overlay.

PAGE ZERO OVERFLOW

The Binder has placed more than 330_8 data words into the ZREL partition.

PARTITION TYPE MISMATCH

Two or more OBs defined the same new memory partition, but the types differed.

PROGRAM FILE LARGER THAN 32K

Your program file will not fit in memory. Put part of it in overlays.

PROGRAM START ADDRESS UNSPECIFIED

None of the object files given to the Binder specified the program start address. See the .END pseudo-op in the Macroassembler manual.

STACK FAULT HANDLER MISSING

You haven't provided a stack fault handler and you used the /N switch which inhibits the Binder from searching URT.LB, hence preventing it from binding SFALT from URT.LB.

SYMBOL FILE OVERFLOW

The maximum size of the symbol file has been exceeded.

SYMBOL UNDEFINED AT END OF PASS 1

The named symbol was not defined.

TOO MANY TASKS

You've exceeded the maximum number of tasks (32).

UNDEFINED COMMON

You defined a common symbol in terms of an undefined symbol.

UNDEFINED EXTERNAL DATA LOCATION

You attempted to locate one or more data words relative to a symbol, but the symbol was still undefined after pass 1.

WARNING: FILE EXCEEDS 32K

You are using the /I switch and your file is bigger than 32K words. This is just a warning since you cannot execute a file built with the /I switch; the bind will continue.

Example of Binder Operation

Figures 2-1 and 2-2 illustrate sample assembler and source listings of modules input to the Binder; Figure 2-3 shows the listing output by the Binder. There are three modules in these illustrations: ROOT, OVR1, and EXMPL. OVR1 and EXMPL will build an overlay file. A detailed discussion of the internal structure of EXMPL is given at the end of Appendix A.

Meanings attached to certain symbols in the assembler listing below are as follows:

Symbol	Meaning
-	ZREL partition, word relocation.
'	unshared code, word relocation.
!	shared code, word relocation.
=	ZREL partition, byte relocation.

The Binder command line employs a command file since an overlay definition is made:

XEQ BIND/L=@LPT CMD/C)

The command file, CMD, contains the following string:

ROOT [OVR1 EXMPL])

The listing output by the Binder on the line printer is shown in Figure 2-3.

ROOT.PR contains 100_8 words of unshared code. Entry point RT1 is at relative offset 0. OVR1 contains 20 words of unshared code; entry OVR1 is at relative offset 0. EXMPL contains 2 words of ZREL, 4 words of unshared code and 2 words of shared code. Entry EXMP1 is at offset 0 in the shared partition and EXMP2 is at relative offset 0 in the shared partition. Note that ESYM, declared external in EXMPL, is undefined.

?CSZE is the size of unlabeled common (there isn't any), and ?CLOC is its starting address (none).

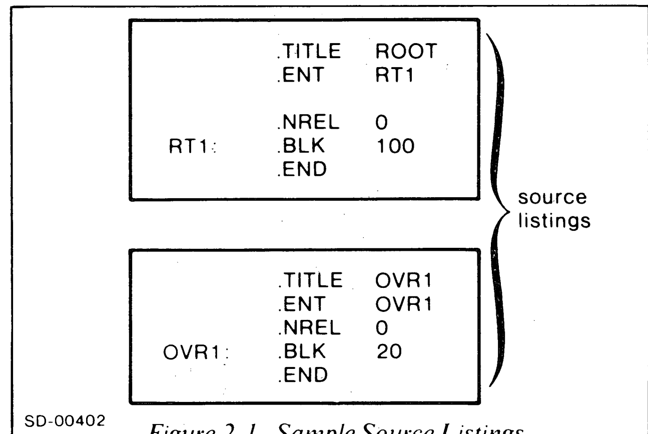


Figure 2-1. Sample Source Listings

```
0001 EXMPL AOS ASSEMBLER REV 00.05                     09:43:32 02/01/77
                              .TITLE   EXMPL
02                            .ENT     EXMP1,EXMP2
03                            .EXTN    ESYM
04
05                            .ZREL
06
07 00000-000000'              EXMP1
08 00001-000000!              EXMP2
09
10        000000              .NREL    0
11 00000'000123 EXMP1:        123
12 00001'000000!              EXMP2
13 00002'000000'              EXMP1
14 00003'000000$              ESYM
15
16        000001              .NREL    1
17 00000!000000'EXMP2:        EXMP1
18 00001!000000!              EXMP2
19                            .END     EXMP1

**00000 TOTAL ERRORS, 00000 PASS 1 ERRORS
```

Figure 2-2. Sample Assembler Listing

```
ROOT.PR CREATED BY AOS BINDER REV 00.03 ON 1/28/77 AT 16:25:16
SYMBOL UNDEFINED AT END OF PASS 1   ESYM
ROOT
OVR1
EXMPL
ATTEMPT TO RELOCATE DATA FROM UNDEFINED SYMBOL  000615 ESYM
SCALL


000001 WORDS OF ABSOLUTE DATA
ZMAX: 000053
NMAX: 001223
START OF SHARED: 074000
LENGTH OF SHARED: 004000


PARTITION    TYPE    START    END   #OF OVERLAY AREAS
000004    NSHR CD   000472 001171   000001
000007     SHR CD   074000 076130   000001


  AREA  START  LENGTH  PARTITION  #OF OVERLAYS
000000 000572  000400     000004    000001
000001 074000  002000     000007    000001


    ?CSZE    000000
    ?SLSZ    000000
    ?URTB    000052
    ?ZMAX    000053
    RT1      000472
    ?USTA    000472
    OVR1     000572
    EXMP1    000612
    ?NMAX    001223
    EXMP2    074000
    ?XSAV    076060
    ?XLD     076111
    ?XSV     076111
    SFALT    076121
    ?CLOC    177777
U   ESYM     177777
```

Figure 2-3. Sample Binder Output Listing

End of Chapter

Appendix A
Object File Formats

Each object file processed by the Binder must consist of a series of blocks of information in a well-specified form. Ordinarily, as a user of Data General software you need not be aware of the various object file formats; the Macroassembler and all compilers produce objects with the required formats. Programmers writing language processors that will use the Binder, however, must adhere to the formats the Binder expects.

Each language processor outputs an object (OB) as a series of OB blocks. The first and last of these are invariably a title block and an end block respectively. In between these two blocks there may appear any combination of the available block types, there may be any number of each type of block, and they may appear in nearly any order. OB files have the following characteristics:

- no limit to the size of any file block;
- any block type may be present or absent (except the title and end blocks);
- block order is not generally fixed, with exceptions described below.

The Binder sets the following five block order rules. First, external symbol blocks must precede the data blocks or entry blocks that contain data or values relocated from the symbols. The title block must be the first block in the OB file, and if there is an address information block, it must be the second block in the file. If there is a lines title block, it must be the next-to-last block in the file; the last block must always be the end block.

General Block Format

Each of the different types of blocks conforms to the general structure shown in Figure A-1.

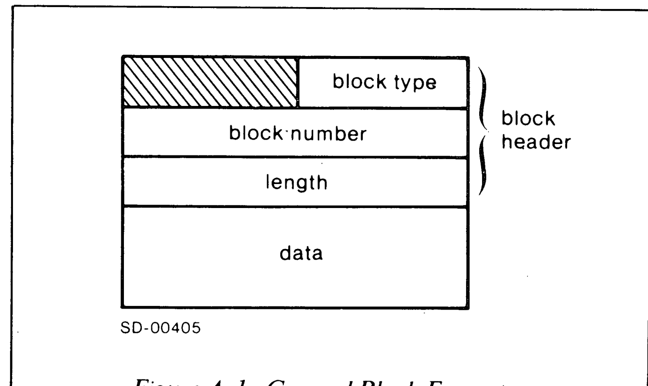


Figure A-1. General Block Format

Block number is a sequence number. These numbers must be sequential, starting at 1 (this is a validity check). *Length* is the length in words of the entire block, including the header. *Block type* is an integer

value denoting the specific kind of block. There are 17 types of blocks, and the octal values associated with each type are as follows:

0	data	11	shared library
1	title	12	task statement
2	end	13	limit
3	unlabeled common	14	named common
4	external symbols	15	accumulating symbol
5	entry symbols	16	debugger symbols
6	local symbols	17	debugger lines
7	library start	20	lines title
10	address information	21	library end

The remainder of this appendix defines and describes the block formats of each of the block types. All byte pointers described in block formats are pointers relative to the beginning of the block (i.e., the first byte in the block has byte pointer "0"). An example abstract from an object file is shown at the end of this appendix to illustrate some of the more common block types.

Data Block

The data block groups program code or data. Figure A-2 illustrates a map of the data block.

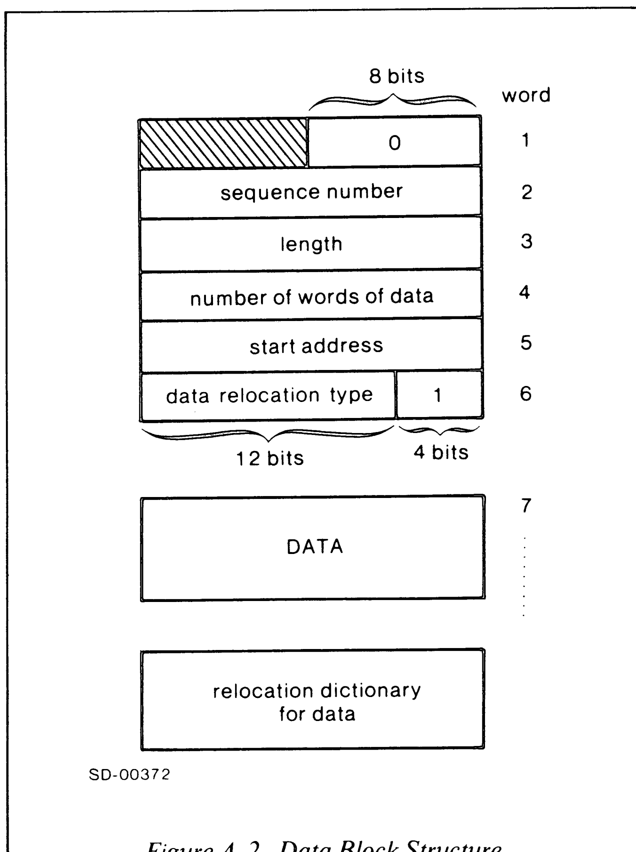


Figure A-2. Data Block Structure

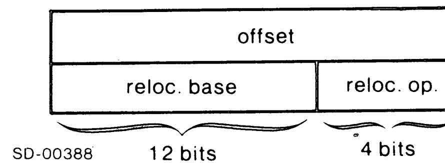
The type, 0, is in the right byte of the first word in this structure. Following the length word is a 16-bit value describing the number of data words in the block; these start at word 7. The *start address* indicates where the beginning of this block of data will be placed; the start address value is relative to the current base for this partition. The address relocation base can indicate either one of eight default data partitions, an external symbol relocation value, or a partition created by an Address Information Block (described below). Default partition numbers and their associated types are as follows:

Number	Type
0	absolute
1	ZREL
2,3	reserved for system use
4	non-shared NREL code
5	shared NREL data
6	non-shared NREL data
7	shared NREL code
8 to 2^{12}	external symbol relocation value

A single data block can specify only one partition.

Following the last data word is a series of two-word entries in the data relocation dictionary. Entries in this dictionary describe the relocation operation, if any, to be applied to the data words. Each word of data may have any number of entries in the relocation dictionary, since more than one relocation operation may be applied to the data. Absolute data has no entry in the relocation dictionary.

Each entry in the dictionary is a two word group with the following structure:



The *offset* indicates the data word (in the current data type within this module) to which this dictionary entry applies. Offsets are expressed in terms of the current base of the data type, not in terms of the relative order of the data words in the block. For example, if the block *start address* is 132, the offset for the first data word is 132, not 0 or 1.

The *relocation base* indicates either a partition or an external symbol used as a base. Relocation bases 0-7 always indicate memory partitions (and are defined in the same manner as the address relocation base shown above). Relocation bases greater than the highest

partition number referenced within this OB refer to external symbols. Relocation bases higher than 7 may be declared to be memory partitions by an address information block (described later in this appendix). External symbols are numbered in the order that they appear in the OB. Thus if an OB uses nine partitions, the first external symbol used as a relocation base would have relocation base 10.

The maximum number of externally defined symbols that a given object file can reference is 4050 minus the number of partitions. The default number of partitions is seven.

Finally, the relocation operation field describes the type of relocation the Binder will perform on the data word. The following lists the numeric values and types of these operations ("base" is the value of the symbol or the start of the OB in the specified partition).

Value	Type
0	absolute (no operation).
1	word, base + data.
2	byte, 2 * base + data.
3	displacement, base + right byte of data, error on overflow into left byte.
4	PC relative, base - data.
5	overlay, data replaced by area number/overlay number (.ENTO).
6	multiply, data * base.
7	link (see *Debugger Symbols Block)*.
10	call relocation.
11	GREF, same as word relocation, but a carry from the low-order 15 bits never alters bit 0.
12	word PC relative relocation (base + data)- the real address where data will be placed.
13	target relocation.

You can understand *call and target relocation* only after familiarizing yourself with general resource calls, described in Chapter 3 of the *AOS Programmer's Reference Manual.* Call (10) and target (13) relocation refer to the operations performed upon general resource calls ?RCALL, ?RCHAIN and ?KCALL and their procedure entry arguments. These relocation types are employed by the system to implement the resource calls.

Language processors (Macroassembler, etc.) apply call relocation to the system call itself to change it to either an EJSR instruction or to a JSR to the appropriate resource call manager. They apply target relocation to each in-line argument of each resource call, to transform it into an appropriate procedure entry descriptor. Table A-1 illustrates the results of call and target relocation applied to every possible resource call and its argument, when the argument is passed in-line (i.e., not at the top of the stack).

As described in the *AOS Programmer's Reference Manual,* the resolution of each general resource call depends upon the the type of call, the type of medium from which the call is made, and the type of medium to which the call is made.

Entry descriptors referred to in Table A-1 are identical to those described in Chapter 3 of the *AOS Programmer's Reference Manual.* Thus if "?RCALL FRED" is issued from the root, with procedure entry "FRED" also found in the root, the Binder would create two data words: EJSR (with absolute indexing) followed by the absolute address of entry FRED.

If instead of passing the resource call argument in-line, it is passed on the top of the stack, then ?RCALL, ?RCHAIN, and ?KCALL are changed to JSR @14, JSR @12, and JSR @13 respectively, with no relocation operation. Following this data word is a zero word, indicating to the resource manager at run time that the resource call argument is found on the top of the stack.

Word PC relative relocation is not currently supported by the Macroassembler. This relocation adds together the base and the data, and then subtracts the relocated value of the offset, i.e., the actual address at which the relocated data will be placed.

Note that absolute relocation need not generally be used. If you do not wish to relocate a piece of data, don't put an entry for the data in the relocation dictionary.

Table A-1. Call and Target Relocation

From \ To		Root or Bound Shared Routine	Overlay	Shared Routine
Root	?KCALL	EJSR 0 α (?KCALL and ?RCALL)	JSR @ 13 ω	JSR @ 13 σ
	?RCALL		JSR @ 14 ω	JSR @ 14 σ
	?RCHAIN	JSR @ 15 α	JSR @ 15 ω	JSR @ 15 σ
Over-lay	?KCALL	JSR @ 13 α	JSR @ 13 ω	JSR @ 13 σ
	?RCALL	JSR @ 14 α	JSR @ 14 ω	JSR @ 14 σ
	?RCHAIN	JSR @ 15 α	JSR @ 15 ω	JSR @ 15 σ
Shared Routine	?KCALL	JSR @ 13 ϵ (all targets)		
	?RCALL	JSR @ 14 ϵ (all targets)		
	?RCHAIN	JSR @ 15 ϵ (all targets)		

entry descriptors

- α absolute address
- ω address of overlay descriptor
- σ 1B0 + address of shared library descriptor
- ϵ 1B0 + 1B1 + library # + external #

(1B0 + 1B1 + library #: left byte; external #: right byte)

Title Block

The .TITLE pseudo-op in your Macroassembler code generates the title block. (See the appropriate higher level language manual for the corresponding feature.) This must be the first block in OB file, and has the structure shown in Figure A-3. Bit 4 of word one is the force-bind bit. If it is set, the module is unconditionally bound.

The third word, *length*, describes the total number of words in the block. The left byte of word 4 contains the major revision number, and the right byte contains the minor revision number of the OB. Word 5 counts the number of bytes in the title; word 6 is a byte pointer to the start of the title string (in word 7).

Contents	word
(shaded) | 1	1
sequence number (1)	2
length	3
rev no. (major/minor)	4
byte length	5
title byte pointer	6
title	7

SD-00373

Figure A-3. Title Block Structure

End Block

This block is the last block in the OB file, and has the structure shown in Figure A-4.

The *start address* specifies the starting execution address of the program. The Binder computes the actual execution address from the start address and the relocation base, generally using word relocation. In general, only one OB file in a Binder command sequence will carry a real start address; other OBs place a -1 in word 4. However, if more than one OB contains a valid start address, the Binder will use the last address specified, unless you used the /F switch in the command sequence (see Chapter 4).

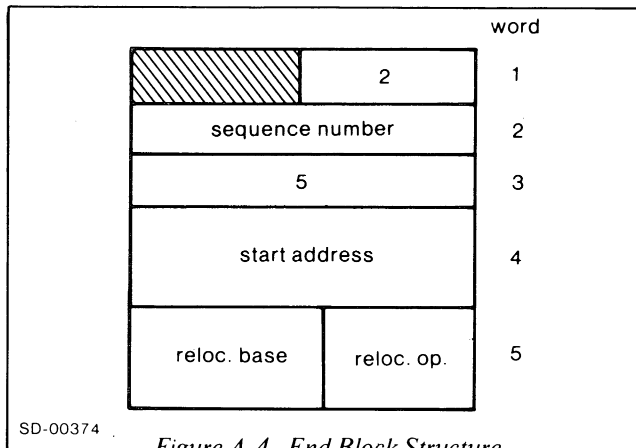


Figure A-4. End Block Structure

Unlabeled Common Block

This block has the structure shown in Figure A-5.

Word 4 contains the size of unlabeled common. The relocation base (word 5) and relocation operation (also word 5) are the same as for the relocation dictionary entries in the data block.

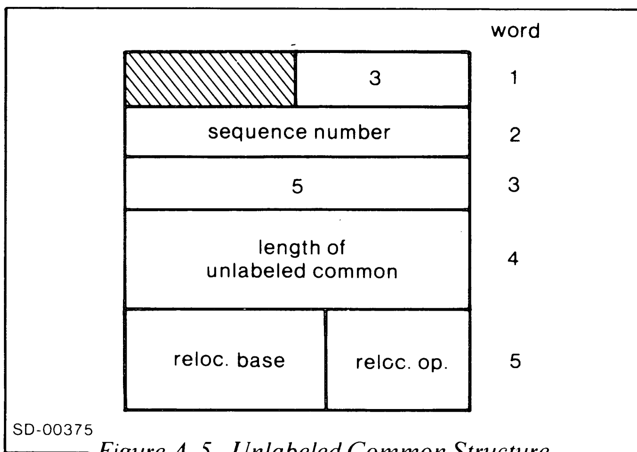


Figure A-5. Unlabeled Common Structure

External Symbols Block

The external symbols block groups those symbols which are declared external. The structure of this block appears in Figure A-6.

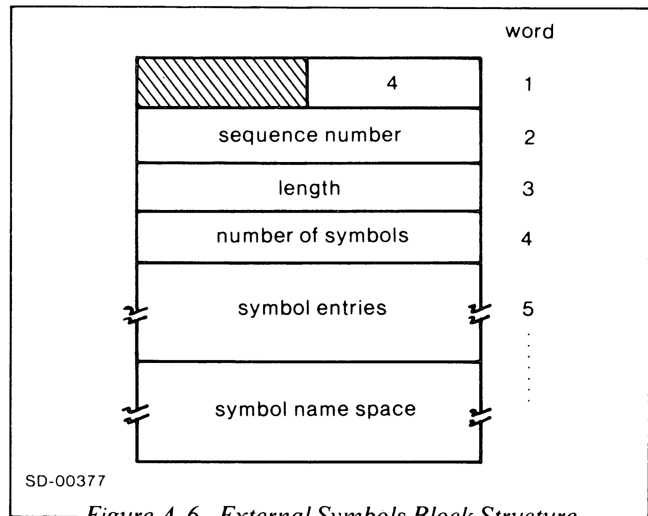


Figure A-6. External Symbols Block Structure

Word 3 contains the number of words in this block. Word 4 contains the number of symbols grouped in this block. The maximum number of external symbols is limited to 4096, because the size of the relocation base field in each entry of the relocation dictionary in data blocks is only 12 bits. For each symbol there is a corresponding entry in the symbol entries section of this block, commencing at word 5. Each symbol entry is two words long, as shown in Figure A-7.

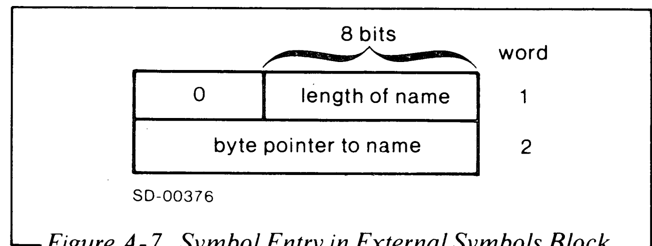


Figure A-7. Symbol Entry in External Symbols Block

The name length (8 bits) permits symbols to be up to 255 characters long (even though the Macroassembler currently truncates symbols to five characters). The byte pointer points to the position of the name string in symbol name space (the last section of the external symbols block), relative to the start of the block. The length of the name is given in characters. Thus a single string, such as "ABCD", can be pointed to by distinct symbol entries for symbols "ABC" and "CD".

The order of symbols in this block must correspond to the order that they are described in the relocation

dictionary entries in the data block. The Binder assigns a unique numeric ID (starting at a number one greater than the highest relocation base assigned to partitions or overlays) to each external symbol based on its position in the external symbols block (or blocks). It is this ID which appears in the relocation base of each dictionary entry for external symbols in the data block.

Entry Symbols Block

The entry symbols block groups those symbols which are declared as entries. The structure of this block appears in Figure A-8.

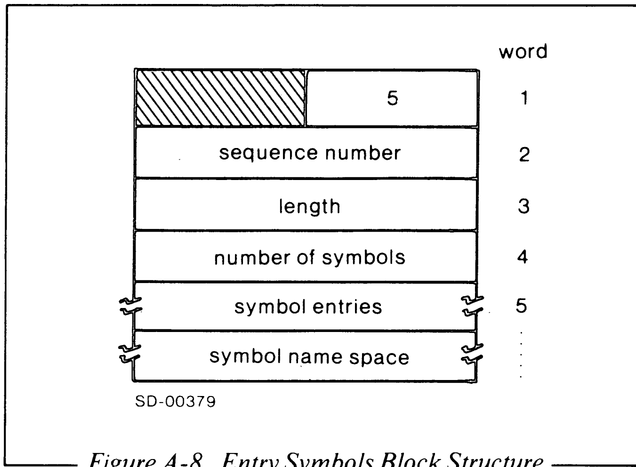


Figure A-8. Entry Symbols Block Structure

For each symbol there is a corresponding entry in the symbol entries section of this block, beginning at word 5. Each symbol entry is four words long, as shown in Figure A-9.

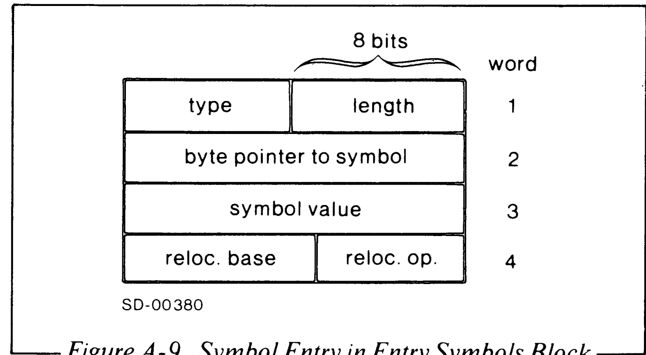


Figure A-9. Symbol Entry in Entry Symbols Block

Word 1 indicates the number of characters in the symbol, and its type: .ENT(0), .ENTO(4), or .PENT(6). Word 2 contains a byte pointer to the symbol name string in symbol name space, which follows the last symbol entry. Word 3 contains the value of the symbol, i.e., its relative position in the partition specified by the relocation base (word 4) and relocation operation.

Local Symbols Block

These structures are exactly like the entry symbols structures (Figures A-8 and A-9). The Binder ignores local symbol blocks unless you apply argument switch /U to the module.

The local symbols block groups those symbols which will not be referenced by other modules. The structure of this block, and the symbol entry which is used in the block, are shown in Figure A-10. See /U argument switch.

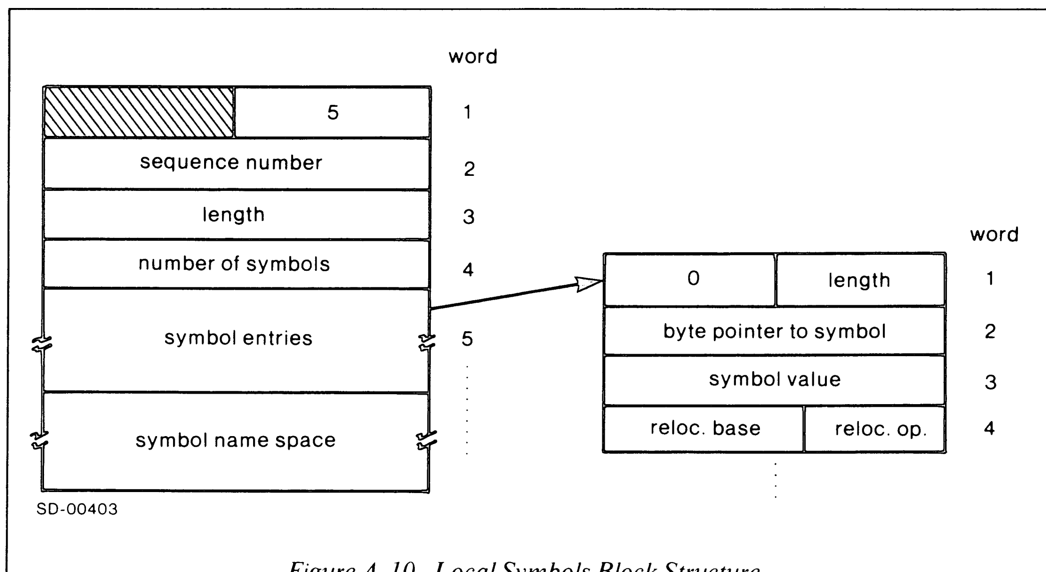


Figure A-10. Local Symbols Block Structure

Library Start and End Blocks

Each non-shared library the Binder uses to produce a program file is essentially a series of object files grouped together into a single library file. Each of the formerly separate object files becomes a separate module in the new library file. At the beginning of this new file there is a library start block and at the end there is a library end block. Each block consists of two major sections: an initial 4-word header, and a series of module descriptors (one per module) followed by a record name space area. A detailed description of library start and end block structure is provided in the *Library File Editor Reference Manual.*

Address Information Block

Current versions of the Macroassembler and DGC high-level languages supported by the AOS system, except for COBOL, do not generate address information blocks (AIBs) even though the Binder can recognize these blocks. Thus this block is of interest to you only if you write a language processor capable of generating AIBs.

There can be only one AIB per OB. Code for each partition appears in the data blocks that follow the AIB. As shown in Figure A-11, there are up to three separate parts in each AIB: the basic portion (words 1-4), new partition descriptors, and partition-sectioning descriptors.

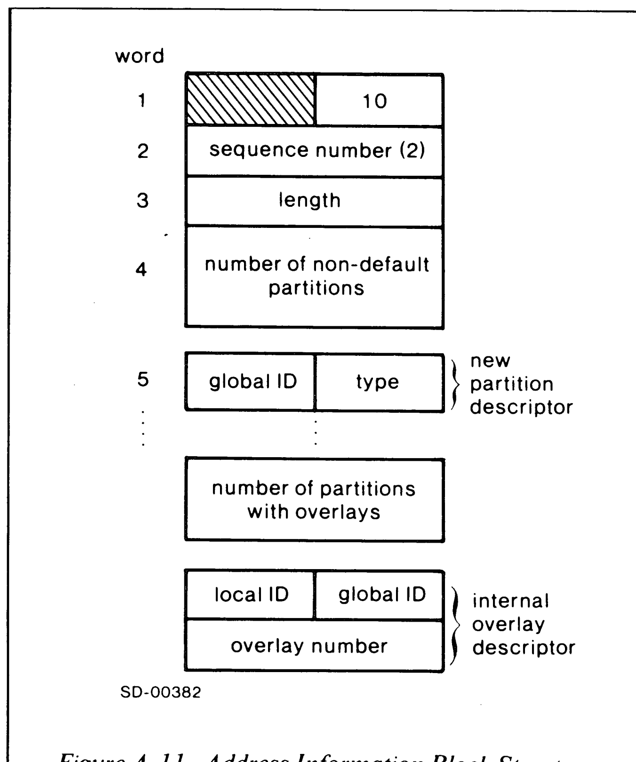


Figure A-11. Address Information Block Structure

AIBs perform two distinct functions: they can define partitions, and they can define overlays. Word 4 specifies the number of new partitions being created. Each new partition descriptor has two eight-bit fields; the left field is the partition number that you are assigning to a new partition, and the right field is the type of this partition. In order to use the partition in a subsequent OB, you must recreate it. Be sure to give it the same partition type; otherwise you will get an error.

If you use the AIB to reserve an overlay area in any partitions (default or non-default types), then following the last new partition descriptor (if any) you must have a single word describing the number of partitions with overlays. Remember: you can use the AIB to define overlay areas only if you do not use the square brackets convention in a Binder command line. Using the AIB to define overlay areas permits you to allocate only one such area per partition.

Partition-sectioning descriptors let you define overlays within an OB. You can direct data from this OB into many different overlays in any code partition. All of the overlays destined for any partition will go to the same overlay area in that partition.

The size of any overlay area will equal the size of the largest overlay that you define. Recall that unshared overlays will always be multiples of 512 bytes, and shared overlays will be multiples of 2048 bytes.

Shared Library Block Header

Each shared library constructed by the Shared Library Builder will consist of a series of reentrant shared routines, preceded by a shared library block header. The structure of this header is described in the *Shared Library Builder Reference Manual.*

Task Block

All AOS assemblers and compilers generate a task block for each program to indicate how many task control blocks the program requested. If there is no task block and no /K switch in the command line, the Binder presumes the program needs only one task (the default task). This block has the structure shown in Figure A-12. The number of tasks must be less than 256.

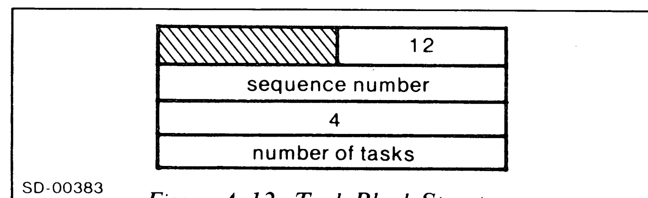


Figure A-12. Task Block Structure

Limit Block

Each limit symbol will form a limit block. This block has the structure shown in Figure A-13.

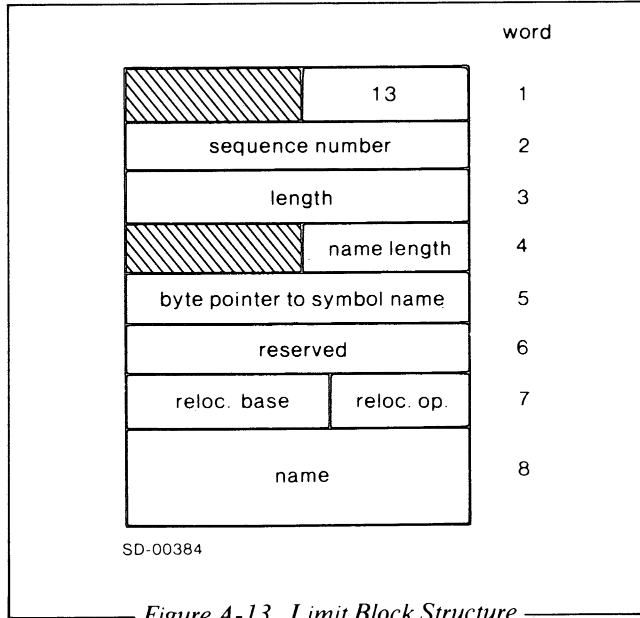


Figure A-13. Limit Block Structure

The right byte of word 4 contains the size of the limit symbol in bytes. See the *Macroassembler Reference Manual* for a discussion of limit symbols.

Named Common Block

The named common block is generated when the program references a named common symbol. This block has the structure shown in Figure A-14.

Words 5 and 9 have the same meaning as relocation dictionary entries in the data block. Words 4 and 5 apply to the size of named common, and word 9 describes the partition into which the common area will be placed. The relocation operation field of word 9 is always set to one, since word relocation is always used.

Accumulating Symbol Block

Each accumulating symbol used builds an accumulating symbol block. This block has the structure shown in Figure A-15.

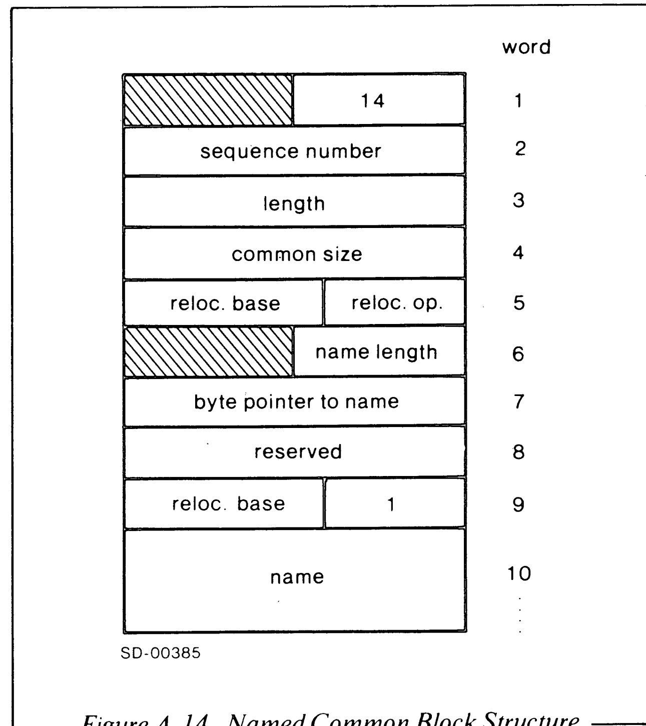


Figure A-14. Named Common Block Structure

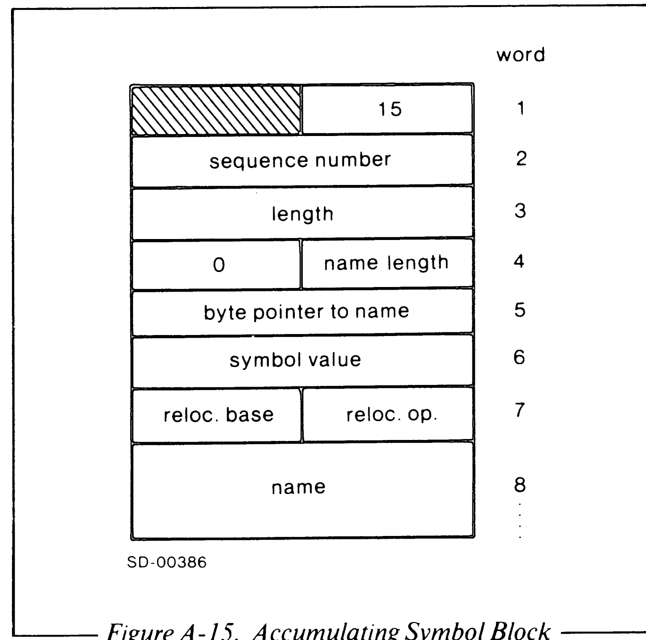


Figure A-15. Accumulating Symbol Block

Debugger Symbols Block

The Binder copies data from the debugger symbols block into the debugger symbols file program-file.DB when function Binder switch /D appears in the command line, and the Binder detects a debugger symbols block. The format of the debugger symbols block is shown in Figure A-16.

Following the last data word is a series of two-word entries in the data relocation dictionary. Each word of data may have any number of entries in the relocation dictionary, since more than one relocation operation may be applied to the data. Absolute data has no entry in the relocation dictionary.

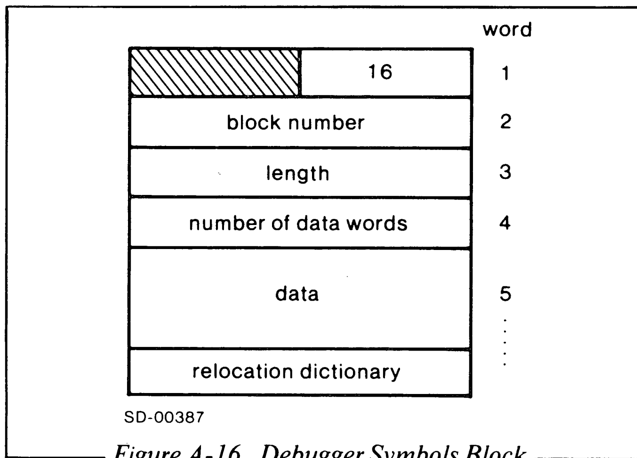
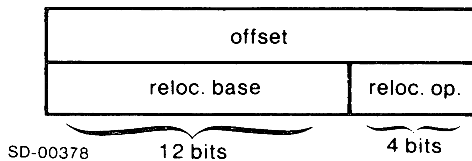


Figure A-16. Debugger Symbols Block

Each entry in the dictionary is a two-word group with the following structure:



The entry structure is defined and used exactly like dictionary entries for data in data blocks. On the first use of the link operation, the Binder copies a 0 to the debugger symbols file. On subsequent uses of the link operation, it copies to the debugger symbols file the file position (in words) where the link operation was last used in the debugger symbols block. After it has processed the last object file it writes the file position of the last use of the link operation in word 0 of the debugger symbols file. (The data it has copied from the debugger symbols block commenced originally at word 1.)

Debugger Lines Block and Lines Title Block

The Binder copies from object file input into the debugger lines file, named program-file.DL. It creates this file when it sees a debugger lines block in an OB. The next to last block in this OB must be a lines title block. The debugger lines block and lines title block are structured as shown in Figure A-17.

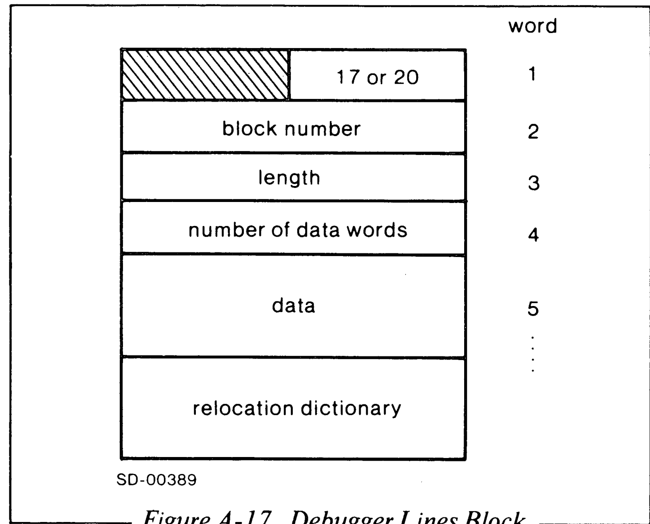


Figure A-17. Debugger Lines Block

Entries in the relocation dictionary are two words each, defined and used as entries in relocation dictionaries as described previously. Link relocation may not be used in the debugger lines file.

The lines title block is block type 20_8. Except for this difference, the lines title block is structured exactly like the debugger lines block, which has block type 17_8.

Object File Illustrations

Having discussed each of the possible object block structures separately, we can now examine an entire object file and see the structures of some actual blocks. For this illustration, let's use the source program entitled EXMPL. This program was shown in Figure 2-2; but we repeat it here, in Figure A-18.

The object file constructed by the Macroassembler from source file EXMPL is shown in Figure A-19. Words in this illustration are numbered from zero through 116 octal; we've drawn square brackets to delimit each block in the OB.

```
0001 EXMPL AOS ASSEMBLER REV 00.05                    09:43:32 02/01/77
                               .TITLE  EXMPL
02                             .ENT    EXMP1,EXMP2
03                             .EXTN   ESYM
04
05                             .ZREL
06
07 00000-000000'               EXMP1
08 00001-000000!               EXMP2
09
10        000000               .NREL   0
11 00000'000123 EXMP1:         123
12 00001'000000!               EXMP2
13 00002'000000'               EXMP1
14 00003'000000$               ESYM
15
16        000001               .NREL   1
17 00000!000000'EXMP2:         EXMP1
18 00001!000000!               EXMP2
19                             .END    EXMP1

**00000 TOTAL ERRORS, 00000 PASS 1 ERRORS
```

Figure A-18. EXMPL Assembler Listing

Words 0 through 11 comprise the *title block:* sequence number 1, length 9 words, no major or minor revision number, and title string "EXMPL" is preceded by a byte length of 5 and a byte pointer to the start of the string. Words 12 through 31 comprise the *entry block.* There are two symbols, EXMP1 and EXMP2, and thus there are two symbol entries (preceding symbol name space). The first of these four-word entries specifies that the symbol is 5 characters long, symbol value is "0" (first entry defined for this partition), relocation base is 7 (shared code), and relocation operation is 1 (word). This is EXMP2. The second symbol entry describes EXMP1: # 5 characters long, symbol value 0, relocation base 7, and relocation operation 1.

Words 32 through 41 comprise the *external block.* There is one symbol, it is 4 characters in length, and its name is ESYM.

Words 42 through 55 comprise a *data block.* There are two words of data, they start at relative offset 0 in ZREL; word-relocation is specified. Both data words are zero. The first dictionary entry specifies offset 0, relocation base is unshared code, and relocation operation is word. The second dictionary entry specifies offset 1, relocation base is shared code, and relocation operation is word.

Words 56-75 comprise a *data block* for the unshared code partition shown in Figure A-18. There are four words of data starting at offset 0, the type is unshared code with word relocation. The first data word is 000123; others are zero. There is no dictionary entry for offset 0 since it is absolute. At offset 3, the relocation base is 10 (first external symbol in the module), and the relocation operation is displacement.

Words 76-111 comprise a data block for the shared code partition. There are two words of data, and the first is placed at offset zero in this partition. The data type (word 103) is shared NREL code. There are two entries in the relocation dictionary; the first specifies that word zero uses word relocation applied using the non-shared NREL code base, and the second specifies word relocation and the shared NREL code base.

Finally, words 112-116 comprise the *end block.* The starting address (word 115) is zero, the relocation base (word 116, bits 12-15) is unshared code, and the relocation operation (bits 0-11) is word.

```
        WORD # DATA

  00   [000001 000001 000011 177777 000005 000014 042530 046520
  10    046000][000005 000002 000021 000002 000005 000030 000000
  20    000161 000005 000035 000000 000101 042530 046520 031105
  30    054115 050061][000004 000003 000010 000001 000004 000014
  40    042523 054515][000000 000004 000014 000002 000000 000021
  50    000000 000000 000000 000101 000001 000161][000000 000005
  60    000020 000004 000000 000101 000123 000000 000000 000000
  70    000001 000161 000002 000101 000003 000203][000000 000006
 100    000014 000002 000000 000161 000000 000000 000000 000101
 110    000001 000161][000002 000007 000005 000000 000101]
```

Figure A-19. Contents of EXMPL.OB

End of Appendix

Index

Within this index an "f" following a page number means "and the following page", "ff" means "and the following pages".

accumulating symbol 1-5, 2-2, A-10
accumulating symbol block 1-5, A-8
address information block A-1, A-7
argument switches 2-2
.ASYM 1-5, 2-2

call relocation A-3f
?CLOC 1-4
command file 2-1, 2-3
?CSZE 1-4

data block A-2, A-10
debugger lines block A-1, A-11
debugger lines file 1-6
debugger symbols block A-1, A-9
debugger symbols file 1-5, A-9

end block A-1, A-5, A-10
entry symbols block A-1, A-5f, A-12
error messages 2-3f
external symbols block A-1, A-6, A-10

.FORC 1-4
function switches 2-1

?KCALL 1-4, A-3f

library
  shared 1-4, 2-2
  start and end block A-7
  unshared 1-4, 2-2
limit block A-2, A-8
limit symbols 1-4, 2-3
.LMIT 1-4
lines title block A-1, A-9
link relocation 1-6
local symbols block A-2, A-6

?MEM 1-2
?MEMI 1-2

named common block A-2, A-8
NMAX 1-4, 1-5
?NMAX 1-2, 1-5
.NREL 2-2
NREL 1-1f, 2-3, A-2, A-10

object files 1-1, 2-1, A-1, A-9f
operating procedures 2-1ff
overlay
  area 1-2ff, 2-2, A-3, A-10
  command file 2-1, 2-3
  directory 1-1
  file 1-2f

partition 1-1f, 2-1, 2-3
  types 1-1f
program file 1-1

?RCALL 1-4, A-3f
?RCHAIN 1-4, A-3f
relocation dictionary A-2f, A-10
root 1-2ff, 2-2

SFALT 1-5
shared routines 1-4, A-4
shared library block A-1, A-8
Shared Library Builder 1-4
stacks 1-5
symbol table file 1-1, 2-1, 2-4
switches 2-1f

target relocation A-3f
task block A-1, A-7, A-9
.TITLE A-5
title block A-1, A-4, A-7
.TSK 2-1

UNDEF 1-4
undefined symbols 1-4, 2-3
unlabeled common A-5
unlabeled common block 1-5, A-5
URT.LB 1-5, 2-1
User Status Table (UST) 1-1, 2-1
?USTART 1-1

ZMAX 1-4
?ZMAX 1-2, 1-5
ZREL 1-1ff, 2-2ff, A-2

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

2. Hardware

	Qty. Installed	Qty. On Order
M/600		
MV/Series ECLIPSE®		
Commercial ECLIPSE		
Scientific ECLIPSE		
Array Processors		
CS Series		
NOVA®4 Family		
Other NOVAs		
microNOVA® Family		
MPT Family		

Other ____________
(Specify) ____________

3. Software

- ☐ AOS
- ☐ AOS/VS
- ☐ AOS/RT32
- ☐ MP/OS
- ☐ MP/AOS
- ☐ RDOS
- ☐ DOS
- ☐ RTOS
- ☐ Other

Specify ____________

4. Languages

- ☐ ALGOL
- ☐ DG/L
- ☐ COBOL
- ☐ Interactive COBOL
- ☐ PASCAL
- ☐ Business BASIC
- ☐ BASIC
- ☐ Assembler
- ☐ FORTRAN 77
- ☐ FORTRAN 5
- ☐ RPG II
- ☐ PL/1
- ☐ APL
- ☐ Other

Specify ____________

5. Mode of Operation

- ☐ Batch (Central)
- ☐ Batch (Via RJE)
- ☐ On-Line Interactive

6. Communication

- ☐ HASP
- ☐ HASP II
- ☐ RJE80
- ☐ RCX 70
- ☐ RSTCP
- ☐ 4025
- ☐ X.25
- ☐ SAM
- ☐ CAM
- ☐ XODIAC™
- ☐ DG/SNA
- ☐ 3270
- ☐ 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 Corporation, Westboro, Massachusetts 01580, (617) 366-8911

CUT ALONG DOTTED LINE

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)* ______________________

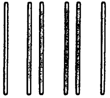
___ Introduction to the product ___ Tutorial Text ___ Other
___ Reference ___ Operating Guide ____________

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

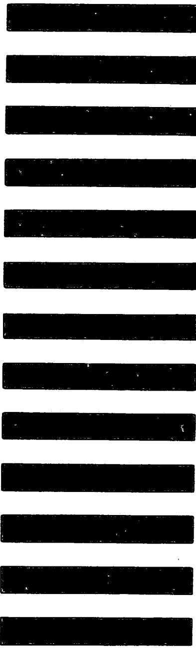


BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

POSTAGE WILL BE PAID BY ADDRESSEE

User Documentation, M.S. E-111
4400 Computer Drive
Westborough, Massachusetts 01581



134-664

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 __________

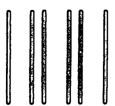
About the manual:		Yes	Somewhat	No
	Is it easy to read?	☐	☐	☐
	Is it easy to understand?	☐	☐	☐
	Are the topics logically organized?	☐	☐	☐
	Is the technical information accurate?	☐	☐	☐
	Can you easily find what you want?	☐	☐	☐
	Does it tell you everything you need to know	☐	☐	☐
	Do the illustrations help 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).

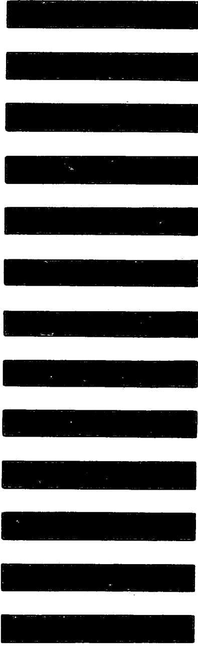
Remarks:

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

User Documentation, M.S. E-111
4400 Computer Drive
Westborough, Massachusetts 01581

Data General Corporation, Westboro, MA 01580

093-000190-03