

LINKER/LIBRARIAN MANUAL

Specifications Subject to Change.

Convergent Technologies is a registered trademark
of Convergent Technologies, Inc.

Convergent, CT-DBMS, CT-MAIL, CT-Net, CTIX, CTOS,
DISTRIX, AWS, IWS, MegaFrame, MiniFrame,
NGEN, and X-Bus are trademarks of
Convergent Technologies, Inc.

CP/M-86 is a trademark of Digital Research.
MS and GW are trademarks of Microsoft Corp.
UNIX is a trademark of Bell Laboratories.

Second Edition (March 1985) A-09-00941-01-A

Copyright © 1985 by Convergent Technologies, Inc.,
San Jose, CA. Printed in USA.

All rights reserved. Title to and ownership of
the documentation contained herein shall at all
times remain in Convergent Technologies, Inc.,
and/or its suppliers. The full copyright notice
may not be modified except with the express
written consent of Convergent Technologies, Inc.

CONTENTS

SUMMARY OF CHANGES	vii
RELATED DOCUMENTATION	ix
CONVENTIONS	xi
1 USING THE LINKER (BINDER)	1-1
INTRODUCTION.....	1-1
USING THIS MANUAL.....	1-2
Understanding the Linker.....	1-2
Errors.....	1-2
Naming Conventions.....	1-2
INVOKING THE LINKER.....	1-3
Run File Versions.....	1-3
Version 4: Link Command Form.....	1-3
Version 6: Bind Command Form.....	1-4
Parameter Fields.....	1-4
2 HOW THE LINKER WORKS	2-1
TWO-PASS LINKER.....	2-1
LIBRARY SEARCH ALGORITHM.....	2-1
ARRANGING MEMORY.....	2-2
Segment Element Names and Classes.....	2-2
Creating Linker Segments.....	2-6
Combination Rules.....	2-7
Addressing Linker Segments.....	2-8
Alignment Attributes.....	2-9
LIMITS.....	2-11
STRUCTURE OF RUN FILE HEADERS.....	2-11
3 SIZING PROGRAMS AND READING THE MAP FILE	3-1
PROGRAM MEMORY REQUIREMENTS.....	3-1
Run-Time Library Code.....	3-1
Simple Programs.....	3-1
Swapping Programs.....	3-2
Programs that Allocate Memory.....	3-2
READING THE MAP FILE.....	3-2
Version 4.....	3-2
Addresses.....	3-2
Names.....	3-3
Classes.....	3-3
Version 6.....	3-3
Public Symbols and Line Numbers.....	3-4

4	FURTHER INFORMATION ABOUT LINKER	
	OPTIONS.....	4-1
	ALLOCATING MEMORY SPACE.....	4-1
	DS Allocation.....	4-1
	The Memory Array.....	4-3
	LINKING A SWAPPING PROGRAM.....	4-4
	ADJUSTING STACK SIZE.....	4-6
	Reducing the Stack.....	4-6
	Correcting Stack Overflow.....	4-6
	FILE NAMING CONVENTIONS.....	4-7
5	LIBRARIAN	5-1
	INTRODUCTION.....	5-1
	INVOKING THE LIBRARIAN.....	5-2
	PARAMETER FIELDS.....	5-3
6	TROUBLESHOOTING	6-1
	LINKER WARNINGS AND ERRORS.....	6-2
	LIBRARIAN ERRORS.....	6-49
	Multiply Defined Symbols.....	6-49
	Excess Public Symbols.....	6-49
	Housekeeping Concerns.....	6-50
	GLOSSARY	Glossary-1
	INDEX	Index-1

LIST OF FIGURES

<u>Figure</u>		<u>Page</u>
2-1.	How the Linker Builds a Run File....	2-4
2-2.	Combination of Stack and Common Segment Elements.....	2-8
3-1.	Sample Version 4 Map File.....	3-2
3-2.	Sample Version 6 Map File.....	3-4
3-3.	Sample Version 4 Map File with Lists of Public Symbols and Line Numbers..	3-5
3-4.	Sample Version 6 Map File with Lists of Public Symbols and Line Numbers..	3-6
4-1.	A Program with DS Allocation.....	4-2
4-2.	A Program with the Memory Array.....	4-4
5-1.	Sample Cross-Reference Listing.....	5-7

LIST OF TABLES

<u>Table</u>		<u>Page</u>
2-1.	Version 4 and Version 6 Run File Header Format.....	2-11
2-2.	Address Structure.....	2-14
2-3.	Prototype Descriptor Structure.....	2-15

SUMMARY OF CHANGES

The 10.0 release of the Linker and the Librarian are described in this second edition of the Linker/Librarian Manual. The manual has been reorganized and substantially rewritten for this edition.

The 10.0 Linker generates either a version 4 or a version 6 run file, depending on which of two commands is given. Both commands are documented, and their uses and differences are noted.

In addition to describing how to use the Linker, this edition contains much new material that explains how the Linker works and how to use its output. New topics are as follows:

- o library search algorithm
- o structure of the run file headers
- o determining the size of a program
- o reading the map file
- o load-time and run-time memory allocation methods
- o swapping programs (virtual code segment management)
- o adjusting the size of the stack

This edition also contains extensive documentation of Linker warning and error messages, with suggested causes and remedies.

RELATED DOCUMENTATION

The following manuals, or related products, are referenced in this manual. It may be helpful to have copies of them on hand when you are using this manual.

The complete Guide to Technical Documentation is provided in the Executive Manual or similar command-line interpreter manual for your operating system.

Assembly Language Manual

Context Manager Manual

CTOS Operating System Manual

CTOS Programmer's Guide

Debugger Manual

Executive Manual

Forms Manual

FORTRAN-86 Release Notice

Standard Software Release Notice

Status Codes Manual

In addition, the compiled-language programmer should have available the appropriate language manual.

The Assembly Language Manual specifies the machine architecture, instruction set, and programming at the symbolic instruction level.

The Context Manager Manual describes and teaches the use of the Context Manager, which allows the user to run applications concurrently and interchange them on the screen almost instantly.

The CTOS Operating System Manual describes the CTOS operating system. It specifies services for managing processes, messages, memory, exchanges, tasks, video, disks, keyboard, printer, timer, communications, and files. In particular, it specifies the standard file access methods: SAM, the sequential access method; RSAM, the record sequential access method; and DAM, the direct access method.

The CTOS Programmer's Guide (formerly titled the System Programmer's Guide) addresses the needs of the system programmer or system manager for detailed information on the CTOS operating system structure and system operation.

The Debugger Manual describes the Debugger, which is designed for use at the symbolic instruction level. It can be used in debugging C, FORTRAN, Pascal, and assembly language programs. (COBOL and BASIC, in contrast, are more conveniently debugged using special facilities described in their respective manuals.)

The Executive Manual describes the command interpreter, the program that first interacts with the user when the system is turned on. It describes available commands and discusses command execution, file management, program invocation, and system management. It also addresses status inquiry, volume management, the printer spooler, and execution of batch jobs.

The Forms Manual describes the Forms facility that includes (1) the Forms Editor, which is used to interactively design and edit forms, and (2) the Forms run time, which is called from an application program to display forms and accept user input.

The Status Codes Manual contains complete listings of all status codes, bootstrap ROM error codes, and CTOS initialization codes. The codes are listed in numerical order along with any message and an explanation.

CONVENTIONS

FORTRAN LANGUAGE NAMES

Two kinds of FORTRAN are supported. They are referred to in this manual and all related documentation by the names FORTRAN and FORTRAN-86. Discussions that include one of these names often do not apply to the other.

NUMBERS

Numbers are decimal except when suffixed with "h" for hexadecimal.

MEMORY ADDRESS

Memory address refers to the logical memory address. (See the "Memory Management" section in the CTOS Operating System Manual.)

VARIABLE NAMES

Variables are named according to a formal convention. The name of a variable implies some of its characteristics.

A variable name is composed of up to three parts: a prefix, a root, and a suffix.

PREFIXES

Prefixes used in this manual are as follows:

b	byte (8-bit character or unsigned number)
c	count (unsigned number)
i	index (unsigned number)
n	number (unsigned number, same as "c")
p	logical memory address (pointer: 32 bits consisting of the offset and the segment base address)

q	quad (32-bit unsigned integer)
rb	relative byte (a 16-bit offset from an arbitrary base address)
rg	array of . . .
sb	array of bytes, where first byte is the size
w	word (16 bit)

Prefixes can be compounded: for example, in this manual, the compounded prefix rbrg indicates the position of an array relative to the beginning of the run file header.

ROOTS

The root of a variable name can be unique to that variable, a commonly used root, or a combination of the two. Common roots that appear in this manual are

lfa	logical file address
mp	map
par	paragraph
sa	segment address
ra	relative address

SUFFIXES

The suffix identifies the use of the variable. The only suffix used in this manual is

Max	the maximum length of an array or buffer (thus one greater than the largest allowable index)
-----	--

EXAMPLES

iProtoDescMax	the maximum SN index
rbrgrle	the offset of the array of relocation pointers from the beginning of the run file header

INTRODUCTION

The Linker (also called the Binder) is a program development utility that combines object modules (files produced by compilers and assemblers) into run files.

A run file is the image of a task (in relocatable form) linked into the standard format required by the operating system loader. The run file consists of a header and a memory image. The header describes the run file and provides certain initial values and an array of pointers that allow the operating system to relocate the run file to any appropriate location in memory.

A run file produced by the Linker can thus be used with various memory configurations or as one of the tasks of a multitask application system.

The Linker has these features:

- o It resolves external references (references from one object module to variables and entry points of other object modules).
- o It builds a run file that can be efficiently loaded. The run file is organized so that the operating system can load it with a minimal number of disk accesses and data transfers.
- o It does not require specification of the eventual memory address of the task. (The run file is relocatable at load time.)
- o It can search through libraries of object modules and select exactly those modules required by a particular application.
- o It can construct run files containing overlays for use with the operating system's Virtual Code Segment Management facility (also called the Swapper).

USING THIS MANUAL

UNDERSTANDING THE LINKER

If you are linking a program written in a high-level language, and if there is nothing unusual about the program, you probably do not need to know much about how the Linker works or about most of its special features and options. You can read the beginning of the next section, "Invoking the Linker," fill in the first three fields of the Linker command form, and ignore most of the rest of this manual.

If you want to use overlays in your program, search libraries in other than the default manner, or directly control the way in which the task image will be arranged in memory, you need to understand more about the Linker before you can correctly complete the command form. See Section 2, "How the Linker Works."

ERRORS

If the Linker displays the screen message

```
There were n errors detected
```

at the end of the link, examine the map file. The errors are listed in this file. (See "Reading the Map File" in Section 3, "Sizing Programs and Reading the Map File.") Section 6, "Troubleshooting," discusses each of these errors and suggests possible causes and solutions.

A given version of the Linker is compatible with only certain versions of CTOS.lib and of the supported compilers and assembler. Using a noncurrent assembler, compiler, or CTOS.lib can result in errors, especially with a noncurrent overlay manager.

NAMING CONVENTIONS

Conventions used in this manual are described in the "Conventions" section at the beginning of this manual and in Section 4.

INVOKING THE LINKER

RUN FILE VERSIONS

The Linker can produce either a version 4 or a version 6 run file. A version 4 run file is appropriate for all 8086- and 80186-based operating system versions and is required for MegaFrame CTOS 3.0. A version 6 run file can also run on workstation operating system version 9.1. A version 6 run file should be generated to run on workstation operating system versions later than 9.1.

Two separate commands are used to invoke the Linker, one for each run file version. To generate a version 6 run file, use the command "Bind." To generate a version 4 run file, use the command "Link." These commands are installed as part of the Standard Software installation. (See the Standard Software Release Notice for details.)

VERSION 4: LINK COMMAND FORM

To create a version 4 run file, invoke the Linker from the Executive by entering "Link" in the command field and pressing RETURN. The parameter fields for the "Link" command form are as follows:

Link

Object modules	_____
Run file	_____
[Map file]	_____
[Publics?]	_____
[Line numbers?]	_____
[Stack size]	_____
[Max memory array size]	_____
[Min memory array size]	_____
[System build?]	_____
[Version]	_____
[Libraries]	_____
[DS allocation?]	_____
[Symbol file]	_____

VERSION 6: BIND COMMAND FORM

To create a version 6 run file, invoke the Linker from the Executive by entering "Bind" in the command field and pressing RETURN. The parameter fields for the "Bind" command form are as follows:

Bind

Object modules	_____
Run file	_____
[Map file]	_____
[Publics?]	_____
[Line numbers?]	_____
[Stack size]	_____
[Max array, data, code]	_____
[Min array, data, code]	_____
[System build?]	_____
[Version]	_____
[Libraries]	_____
[DS allocation?]	_____
[Symbol file]	_____

PARAMETER FIELDS

In either form, you must fill in the first two fields. You can allow optional fields (those in square brackets) to assume their default values by leaving them blank.

Each field is described briefly in the following pages. Cross references to explanatory sections appear under some field descriptions. Normally, you need to specify only the first three fields.

Press GO to execute the command.

Object modules

Enter a list of names of one or more object module or library files. Separate the names with spaces. The Linker combines these object modules to form a run file. An example of listing object modules is shown on the facing page. (For a discussion of file suffixes, see "File Naming Conventions" in Section 4.)

You can list both ordinary object modules and specific ones to be extracted from libraries. To extract from a library, use this syntax:

```
LibraryName (Module1 Module2 . . .)
```

where Module1, Module2, and so on, are the names of the object modules to be extracted. Note that these module names are separated by spaces. You may use or omit the space after LibraryName.

Do not use a space between the opening parenthesis and the first module name.

An example of specifying library modules is shown below.

EXAMPLE: LISTING OBJECT MODULES

To build a run file from the three object modules A.obj, B.obj, and C.obj, fill in the "Object modules" field of the Linker form this way:

```
Object modules A.obj B.obj C.obj
```

EXAMPLE: SPECIFYING LIBRARY MODULES

Assume that Z.lib contains the object modules V, W, and X. To build a run file consisting of object modules A, B, W, X, and C, fill in the "Object modules" field of the Linker form this way:

```
Object modules A.obj B.obj Z.lib(W X) C.obj
```

The same run file results if the original W.obj and X.obj are specified in the "Object modules" field as

```
Object modules A.obj B.obj W.obj X.obj C.obj
```

Using the Linker, you can construct a task containing code in overlays. (Swapping programs and the Virtual Code Segment Management facility are discussed in Section 4 under "Linking a Swapping Program.")

To use overlays, list first those modules whose code is to be permanently resident in memory. Then list the first module to be overlaid, followed by "/o". List the remaining modules in that overlay. Begin a new overlay by again appending "/o" to the first module in the new overlay. (See the example on the facing page.)

Run file Enter the name of the run file to be built. This field is not optional. It is conventional to use the suffix ".run" in creating the name of a run file.

[Map file] Default: RunFileName.map

Enter the name of the map file. The default map file name is derived as described in Section 4 under "File Naming Conventions."

If you do not complete the remaining 10 fields of the Linker command form, the Linker generates a short map file. This map contains an entry for each segment and shows the relative address and length of the segment in the memory image.

To interpret the map file, see "Reading the Map File" in Section 3.

EXAMPLE: USING OVERLAYS

To construct a run file using overlays, first list on the "Object modules" line those modules whose code you want to place entirely in the resident portion of the program.

```
Object modules A.obj
```

Then continue by listing the first module whose code you wish to place in an overlay. Append "/o" to the name of this module. (The "/o" is case insensitive.)

```
Object modules A.obj B.obj/o
```

Now list the names of all further object modules whose code is to appear in the same overlay with that of B.obj. Do not append "/o" to their names.

```
Object modules A.obj B.obj/o Z.lib(W)
```

When you wish to start a new overlay, once again list the name of the first module in that overlay and append "/o". Thus the module designated with "/o" and all modules thereafter are placed in one overlay, until the next "/o" designation is made.

Note that modules from one library can be placed in separate overlays: the "/o" syntax is simply used within the parentheses enclosing the library module list.

```
Object modules A.obj B.obj/o Z.lib(W X/o) D.obj/o
```

This run file consists of a resident portion with the code from A, and a nonresident portion made up of three overlays. One overlay contains the code from B and W. The second contains the code from X. The third contains the code from D.

Your module list cannot exceed one line. If it does, use the "at-file" convention discussed in the Executive Manual, Section 5, "Advanced Concepts."

[Publics?] Default: No public symbols listed

Enter "yes" or "y" if you want to include a list of the values (relative addresses) of all public symbols at the end of the map file. This list is sorted first alphabetically and then numerically.

If you enter "no" or do not respond, public symbols are not listed.

To interpret the map file, see "Reading the Map File" in Section 3.

[Line numbers?] Default: No line numbers listed

Enter "yes" or "y" to include a list of line numbers and addresses of all source statements in the map file. This list of line numbers comes after the list of public symbols, if any. Not all compilers produce object modules containing line number information.

If you enter "no" or do not respond, the map file does not include line numbers and addresses.

To interpret the map file, see "Reading the Map File" in Section 3.

[Stack size] Default: Compiler's estimate

To change the stack size from the compiler's estimate, enter the number of bytes in decimal. The stack is composed of words, so this number must be even.

The default value in this field is the compiler's estimate of the correct stack size.

For details on estimating stack size, see "Adjusting Stack Size" in Section 4.

The second parameters (Max data and Min data) specify the amount of short-lived memory that the application will use.

The third parameters (Max code and Min code) are reserved for future enhancement.

[System build?] Default: No system build

Enter "yes" or "y" to build special versions of the operating system. To do a system build, carefully follow the procedure described in the CTOS Programmer's Guide or the Standard Software Release Notice.

Normally, to link a swapping program, you use the "/o" notation in the "Object modules" field, as described above. The resulting program contains overlays and also several data structures for use by the Swapper.

If you use the "/o" notation and you also enter "yes" in the "[System build?]" field, the overlays are created, but the related data structures are not. This result is correct when you are building an operating system: hence the name of this field.

If you enter "no" or do not respond, normal data structures are created for swapping programs.

[Version] Default: No version specified

Fill in this field to add a version specification (in the form of an alphanumeric string) to the header of the run file. If this parameter has embedded spaces, the whole string must be enclosed by single quotation marks (').

The letters Ver and a space are prefixed to the string that you enter, and the result is placed in the run file header. Also, the public variable sbVerRun is automatically defined in the run file. It consists of the parameter string (not including "Ver ") preceded by a single byte containing the length of that string.

For example, if you enter

2.0

the string

Ver 2.0

appears in the run file header. The value of sbVerRun is

3	2	.	0
---	---	---	---

You can examine the version number in the run file header by using the Dump command in the Executive. (See the Executive Manual.)

If you do not respond, no version specification is produced. (Allowing this field to default may result in an unresolved external error for sbVerRun if you are linking the operating system or certain other supported software.)

[Libraries]

Default: Search CTOS.lib

Enter the list of library files that you want searched. Separate the names with spaces. The library files must have been created by the Librarian utility.

The default is to search only the library [Sys]<Sys>CTOS.lib, plus those libraries indicated by object modules such as those in Pascal and FORTRAN. CTOS.lib is always the last library searched.

The Linker treats the object modules that it selects from these libraries as if they had been specified in the "Object modules" field: that is, they are linked with the resident portion of a task that uses overlays. To link object modules obtained from libraries into overlays, you must name the library and the overlaid modules in the "Object modules" field, as discussed above under "Object modules."

To suppress all default library searching, enter "None" in this field. (It follows that you cannot have a library named "None.")

To suppress default searching of libraries other than the ones you want, including that of CTOS.lib, name the libraries you want searched, followed by "None." See the example below.

EXAMPLE: SEARCHING LIBRARIES

In this example, the Linker searches the libraries A.lib and B.lib, but no others.

[Libraries] A.lib B.lib None

[DS allocation?] Default: Language dependent

Enter "yes" or "y" to locate DGroup at the end of a 64K-byte segment addressed by the DS register. Under this arrangement, the last byte of DGroup is located at DS:0FFFF.

For modules in most languages, if you enter "no" or do not respond, there is no DS allocation, and DGroup begins at DS:0. However, Pascal and FORTRAN modules cause the default value in this field to be yes.

The purpose of DS allocation is to enable allocation of memory at run time that is addressable using DS. This field has meaning only with tasks that use a single value in DS during their entire execution and that include the group DGroup with DS equal to DGroup. This category includes all small- and medium-model tasks.

Object module procedures and tasks produced by the Pascal, FORTRAN, and BASIC compilers use a single value in DS during their entire execution and include the group DGroup with DS equal to DGroup. This feature must be used for linking Pascal tasks that make use of the Pascal heap.

For a discussion of groups, see the Assembly Language Manual. Models of segmentation are described in the "Languages, Stack Formats, and Calling Conventions" section of the CTOS Programmer's Guide.

[Symbol file]

Default: RunFileName.sym

Enter the name of a file to which the Linker writes a symbol table of the run file. The symbol table notes the locations of all public symbols within the program. (See the Debugger Manual for an explanation of the use of this file.)

The default is to derive the name of the symbol file from the name of the run file, as discussed under "File Naming Conventions" in Section 4.

If you do not want a symbol file, enter "[NUL]".

2 HOW THE LINKER WORKS

TWO-PASS LINKER

The Linker makes two passes through the modules being linked. On the first pass, it reads all object modules, extracting external and public symbol information, and builds a symbol table. It examines this symbol table for unresolved external references. If it finds such references, it searches the libraries that you specify for object modules whose public symbols resolve the external references.

On the second pass, the Linker assigns relative addresses, relocating as necessary, to all data in all object modules and links the object modules, constructing a run file ready for the operating system loader.

LIBRARY SEARCH ALGORITHM

Having built a symbol table during its first pass, the Linker runs through all the symbols, checking to see whether any of them occurs in the first library listed for searching. If it finds a symbol declared in a module in the library, it extracts that module from the library and links it into the program. The extracted module may contain further undefined symbols.

The Linker cycles over the entire list of symbols, old and new, comparing them to the first library until no further library modules are extracted. It then steps to the second library and repeats this process, and so on for the third and subsequent libraries.

When the Linker has completed the search of the last library, it may have extracted further undefined symbols in later libraries that were defined in earlier libraries. The Linker thus goes back to the first library and searches again for any undefined symbols. In this way, it cycles through all the libraries repeatedly until it has made one complete cycle without extracting any new module. At this point it stops and reports any symbols that remain undefined.

NOTE

If the same public symbol is defined in more than one library, and if that symbol is declared external in an extracted library module, you cannot assume that the definition used is in the first library listed for searching. From the point at which it extracted the module, the Linker steps on to the next library and extracts the first definition it encounters.

ARRANGING MEMORY

The Linker is basically a concatenation device that accepts object modules, takes them apart into their component pieces, collects pieces of the same types for efficiency, and uses a set of rules to put these collections back together in a certain order to form a memory image of the task in the run file.

The rules that the Linker uses in ordering the pieces are easiest to understand if we give names to the components and collections of components involved and build a model like that in Figure 2-1 to show what is happening. This model does not exactly represent what the Linker does, but it is accurate enough for our purposes. (This example is not intended to correspond to the examples of maps shown in Section 3 under "Reading the Map File.")

SEGMENT ELEMENT NAMES AND CLASSES

In the example in Figure 2-1, three object modules are to be linked. They have been listed in the "Object modules" field of the Linker command form as follows:

Object modules Mod1.obj Mod2.obj Mod3.obj

To provide an illustration of some Linker ordering rules, let us say that Mod1.obj was written in one language, and Mod2.obj and Mod3.obj in another.

Each of these object modules is made up of several segment elements, each of which has been declared PUBLIC. It happens that all these object modules have segment elements that contain code, data, constants, and stack, although this is not true of all object modules.

Each segment element in each module has both a name and a class. In high level languages, the compiler assigns name and class. In Figure 2-1, the name and class of each segment are separated by a slash:

```
Data/Data  
Modl_code/Code
```

This arrangement can become confusing because many compilers assign names to segment elements that are the same as the classes of those segment elements. Data/Data is an example. Usually the code segment element carries the name of the module: in Mod1.obj, the Mod1 segment element is of class Code. Most compilers append the class name as part of the code segment element name, which in this case results in Mod1_code.

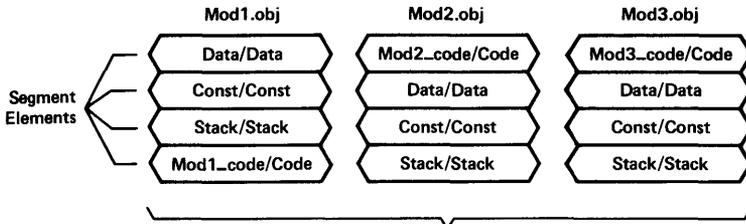
The most common classes are Code, Data, Const, and Stack. A given compiler always arranges the segment elements by class in a certain order.

In assembly language, you have more control over what the Linker ultimately does. You can assign any name you want to any segment element, and you can define more than one of a class and place them in any order within the module. You can make up whatever class names you want.

The segment elements in Mod1.obj are not in the same order as in the other two object modules. Apparently, the two different compilers that produced them do not order segment elements in the same way--or else, perhaps, an assembly language programmer wrote Mod2.obj and Mod3.obj and chose the order of segment elements within them.

Step 1.

Input Object Modules



Step 2.

Look at Mod1 for Order Sort

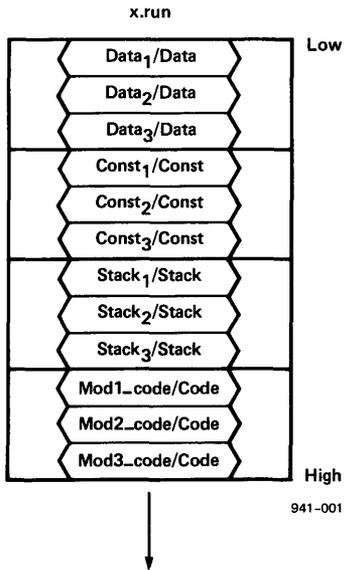
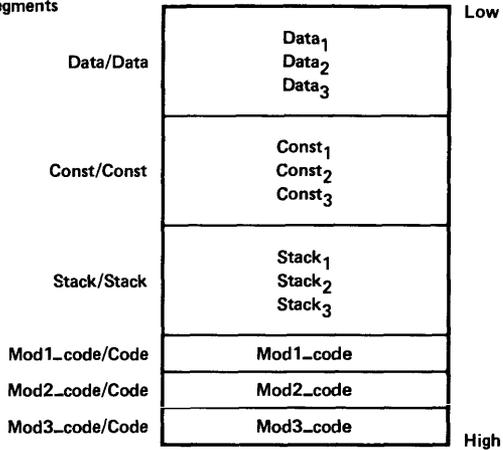


Figure 2-1. How the Linker Builds a Run File.
(Page 1 of 2)

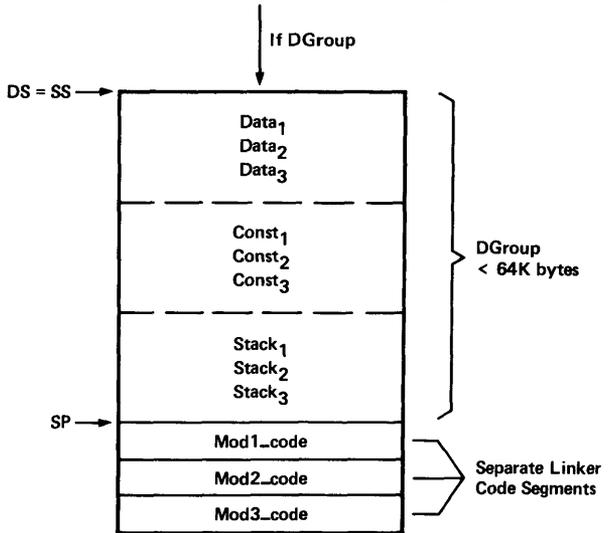
Step 3.

Establish Linker Segments



Step 4.

Establish Segment Addressing



941-001

Figure 2-1. How the Linker Builds a Run File.
(Page 2 of 2)

CREATING LINKER SEGMENTS

After the Linker has resolved all external references in the modules, it builds the run file. Starting with the first module listed (Mod1.obj), it takes the first segment element in that module, looks at its class, creates a category for that class, and places the segment element in that category. It then makes a second category for the second class of segment element that it encounters, and so on through the first module.

In the example, the result is the creation of four categories arranged in the same order as the segment element classes in Mod1.obj: data, constants, stack, and code.

These categories will eventually become linker segments.

Having pulled apart Mod1.obj in this way, the Linker goes on to Mod2.obj. It takes each segment element in Mod2.obj, examines its class, and places it in the linker segment already created for that class. If there is no linker segment for that class, the Linker creates a new one for it at the end of the list of linker segments.

When the Linker has sorted the parts of all three modules, the result is as shown in Step 2 of Figure 2-1.

NOTE

Linker segments are ordered by class in the same order that appears in the first module listed. Thus, you can impose an ordering template on the Linker by writing an assembly language module that does nothing except declare segment elements in the desired class order. Place this module first in the list of modules to be linked.

This template object module often is called First.obj. (See the FORTRAN-86 Release Notice for an example of a ready-made First.obj and its use.)

COMBINATION RULES

Our model is incomplete without an indication of how segment elements are combined or superimposed to form linker segments.

In most cases, the Linker appends one segment element to another as it goes through the modules and does not distinguish boundaries between a segment element from one module and that from the next. This is true of data and constant segment elements.

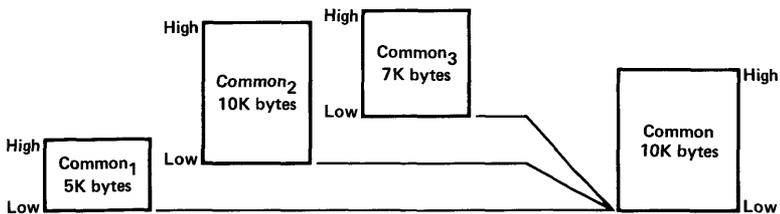
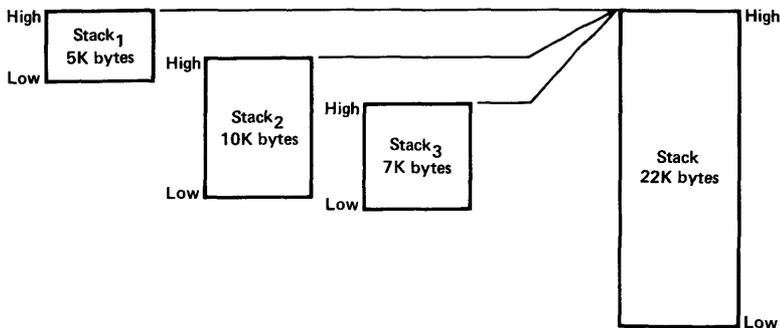
In the case of stack segment elements, the Linker combines them by overlaying them with their high addresses superimposed but with their lengths added together. It then forces the total length of this aggregate stack segment to a multiple of 16 bytes. This arrangement is shown in Figure 2-2. The fact that high addresses are superimposed is unimportant unless you have created a label at the high end of one of the stack segment elements. In such a case, the label floats to the high end of the aggregate stack.

Compilers construct stack segments automatically. However, if your entire program is written in assembly language, you must define an explicit stack segment. (See the Assembly Language Manual for details.)

Segment elements that have been assigned the combination attribute COMMON in assembly language (not shown in this example) are special also. When COMMON segment elements are combined, they are overlaid with low addresses superimposed. The length is that of the largest element. This arrangement is also shown in Figure 2-2.

Code segment elements are placed together, but they are not combined unless they have identical names as well as the same class. (This rule actually applies to all segment elements, but it is most obvious with code segment elements.)

Step 3 in Figure 2-1 shows the linker segments created in this example.



941-002

Figure 2-2. Combination of Stack and Common Segment Elements.

ADDRESSING LINKER SEGMENTS

Finally, the Linker establishes the way in which the hardware segment registers address these linker segments when the program is run. In most cases, a group has been defined in the program.

A group is a named collection of linker segments addressed at run time with a common hardware segment register: that is, 16-bit offset addressing can be used throughout the group. All the locations within the group must be within 64K bytes of each other.

It is typical for a program to contain a group called DGroup, which contains data, constants, and stack. (The medium-model compiled languages use DGroup. In assembly language, you can define whatever groups you want, or none.) For DGroup, the hardware segment register is DS. SS has the same value.

In a version 4 run file, other portions of the program may fall between the beginning and the end of a group, as long as the distance from the beginning to the end of the group does not exceed 64K bytes.

In a version 6 run file, all the linker segments in a group must be contiguous. The Linker combines all the segments of a group into one segment, which is addressed with one selector.

The example in Figure 2-1 contains DGroup, which is shown in Step 4. In this type of run file, information is retained about where within DGroup the data, constant, and stack linker segments begin and end. The value of the SS register is set equal to that of DS. SP is set to equal the highest address in the group, as shown in the figure.

Groups are discussed in the Assembly Language Manual. Models of segmentation are covered in the "Language, Stack Formats, and Calling Conventions" section of the CTOS Programmer's Guide.

ALIGNMENT ATTRIBUTES

Segment elements have alignment attributes. Most compiled languages assign these attributes automatically, but in assembly language, you assign them explicitly. (See the Assembly Language Manual, Section 2, "Programs and Segments," for details.)

A segment can have one of several alignment attributes. These are

- o byte (a segment that can be located at any address)
- o word (a segment that can be located only at an address that is a multiple of two)
- o paragraph (a segment that can be located only at an address that is a multiple of 16)

The Linker packs segments containing data and code end to end. Alignment characteristics can cause a gap between the segments. The Linker adjusts the relative addresses in the segments accordingly.

SUMMARY OF SEGMENT ORDERING

All public segment elements having the same segment name and class name are combined in the order seen by the Linker.

All segment elements having the same class name are placed together in the order seen by the Linker.

All segment elements of the first class seen by the Linker are placed first in the run file, followed by all the segment elements of the second class, and so on.

A group definition does not affect segment ordering. Rather, a group definition asserts that all segments in a group are contained within a 64K-byte region in the run file. This is required if all the segments of the group are to be addressed using a single value in a segment register. In version 6 run files, all segments in a group must be contiguous.

LIMITS

Exact limits on the size of a program that can be linked are difficult to compute. In general, the maximum size of a linkable program and the speed at which the link takes place are directly related to the amount of memory available and the number of public symbols in the program.

If a run file is too large for the Linker, it is probably too large to run on an 8086 processor.

STRUCTURE OF RUN FILE HEADERS

The header of the run file produced by the Linker contains a variety of information describing the file. The format of the version 6 run file is shown in Table 2-1. The version 4 run file is identical through the cIdiv field at offset 36. The remainder of the format exists only in version 6 run files. The version 4 parameters saStack, saStart, and saMainDs are called snStack, snStart, and snMainDs in version 6.

Table 2-1. Version 4 and Version 6 Run File Header Format. (Page 1 of 3)

<u>Offset</u>	<u>Field</u>	<u>Size (bytes)</u>	<u>Description</u>
0	wSignature	2	Run file signature
2	ver	2	Run file format version
4	cpnRes	2	Run file size
6	irleMax	2	Maximum relocation entry index
8	cparDirectory	2	Relocation directory size
10	cparMinAlloc	2	Minimum memory array size
12	cparMaxAlloc	2	Maximum memory array size

Table 2-1. Version 6 Run File Header Format.
(Page 2 of 3)

<u>Offset</u>	<u>Field</u>	<u>Size</u> (bytes)	<u>Description</u>
14	saStack (snStack)	2	Initial stack segment
16	raStackInit	2	Initial stack offset
18	wchksum	2	Run file checksum
20	raStart	2	Initial code offset
22	saStart (snStart)	2	Initial code segment
24	rbrgrle	2	Relocation directory offset
26	iovMax	2	Maximum overlay index
28	saMainDs (snMainDs)	2	Initial data segment (large model)
30	allFs	2	Constant 0FFFFh
32	verAlt	2	Alternate run file format version
34	rbIdiv	2	Idiv table offset
36	cIdiv	2	Size of idiv table
<u>Version 6 Only:</u>			
38	qbMinData	4	Minimum virtual data partition size
42	qbMaxData	4	Maximum virtual data partition size
46	rbRgProtoDesc	2	Prototype descriptor table offset

Table 2-1. Version 6 Run File Header Format.
(Page 3 of 3)

<u>Offset</u>	<u>Field</u>	<u>Size (bytes)</u>	<u>Description</u>
48	iProtoDescMax	2	Maximum prototype descriptor index
50	rbRgRqLable	2	Resident request fixup table offset
52	iRqLableMax	2	Maximum resident request fixup index
54	rbMpSnSa	2	SN to SA translation table
56	iSnMax	2	Maximum SN index
58	snFirst	2	SN of first prototype descriptor
60	slCode	2	First code segment selector
62	cSlCode	2	Count of code segments
64	slData	2	First data segment selector
66	cSlData	2	Count of data segments
68	slStack	2	Stack segment selector
70	cSlStack	2	Constant 1
72	lfaSbVerRun	4	File address of sbVerRun
76	dateTime	4	Time stamp
80	cModify	2	Modify count
82	qbMinCode	4	VM hint information
86	qbMaxCode	4	VM hint information

The wSignature and ver fields (offsets 0 and 2) identify the run file and its version. The cpnRes field gives the size of the run file, excluding overlays.

The next four fields (offsets 6 through 12) provide information about relocation data in the file. The relocation directory is an array of locators used by the operating system in relocating the file. These locators (rle) have the structure shown in Table 2-2. The memory array is described under "Allocating Memory Space" in Section 4, "Further Information About Linker Options."

Table 2-2. Address Structure.

<u>Offset</u>	<u>Field</u>	<u>Size</u> <u>(bytes)</u>
0	ra	2
2	sa	2

At offsets 14 through 22, initial values are given for the stack and code segments.

At offsets 24 through 28, the location of the relocation directory and the number of overlays are identified. The saMainDs field gives the initial value of the DS register for a large model program (for example, a FORTRAN-86 program).

The information at offsets 30 through 36 pertains to correction by the Linker and the operating system code of a known hardware problem with the IDIV instruction on early versions of the 80186 processor.

The version 4 run file header ends at this point. The remaining fields occur only in the version 6 header.

The fields qbMinData and qbMaxData (offsets 38 and 42) are used in sizing partitions on the 8086/80186 processors and for determining an upper bound on how much data space a process can control on the 80286 processor.

The rbRgProtoDesc and iProtoDescMax fields at offsets 46 and 48 contain the offset and maximum index of the prototype local descriptor table (LDT). The 80286 loader refers to this prototype data structure in building an LDT.

The prototype LDT structure is shown in Table 2-3. The first field, limit, is the limit of a segment. The second, lfaLow, is the logical file address (lfa) of the segment on the file. This lfa allows the segment to be found when it is to be faulted in. Since the lfa is a 24-bit quantity, the next field, lfaHi, is needed to supply the high eight bits of this address. The field at offset 5, bAccess, identifies the type of the segment as code, data, and so on.

Table 2-3. Prototype Descriptor Structure.

<u>Offset</u>	<u>Field</u>	<u>Size (bytes)</u>
0	limit	2
2	lfaLow	2
4	lfaHi	1
5	bAccess	1
6	reserved	1
7	reserved	1

The fields at offsets 50 through 58 resolve issues involved in creating a version 6 run file that can run both in real mode (8086/80186 and 80286 processors) and in protected mode (80286 processor). Different types of addressing are used in the two cases.

A version 6 run file uses call gates and global pointers to address certain operating system structures in protected mode on the 80286 processor. The two fields at offsets 50 and 52 allow the version 6 file to be converted to the flexible additive address mechanism that must be used for such addressing if the task is to be run in real mode.

The fields at offsets 54 through 58 describe a table that maps each of the 80286 protected-mode selectors to a real-mode segment address (SA).

The next six fields (offsets 60 through 68) separately identify and describe the code, data, and stack portions of a version 6 run file that is to be run in real mode on a variable-partition operating system.

At offsets 70 through 86, several items are declared that simplify routine operations. The `lfaSbVerRun` field allows the operating system to find the location in a run file where the version number is declared, so that that number can be changed by a utility program without relinking.

The `dateTime` stamp allows the Debugger to compare a symbol file to a run file and report an error if there is a difference.

The `cModify` field allows a count to be kept of the number of times a run file has been modified.

The `qbMinCode` and `qbMaxCode` fields pertain to the use of virtual memory. They indicate to the operating system the approximate size of the working set in bytes.

3 SIZING PROGRAMS AND READING THE MAP FILE

PROGRAM MEMORY REQUIREMENTS

Determining the actual amount of memory that a run file needs is important in all applications. For example, it allows the user to minimize the size of the partition that the program will require when executed under the Context Manager. If the program is too large, it will not execute on systems with small memory.

The memory requirement depends on these considerations:

- o size of the data segment (for example, stack plus constants plus variables)
- o size of the resident code (both the code written by the programmer and the code extracted from libraries)
- o size of the overlay area, if swapping is used
- o extra memory allocated at load time (the memory array) or later (by calls to AllocMemorySL or AllocMemoryLL)

RUN-TIME LIBRARY CODE

For compiled languages like Pascal, even a minimal program requires the language run-time library, as well as associated support code from CTOS.lib. Almost all programs require 20K to 40K bytes of space for run-time library code for this reason. The largest component usually is Sequential Access Method code.

Code from the run-time library is included in the map file.

SIMPLE PROGRAMS

For simple programs, you can read the memory required directly from the map. The size is the "stop" address of the last segment (usually MEMORY) listed in the map. This number is the hexadecimal count in bytes from the first byte of the first segment.

SWAPPING PROGRAMS

Swapping programs should usually be sized on the "stop" address of the last segment of the resident portion, with the size of the required swap buffer added in.

PROGRAMS THAT ALLOCATE MEMORY

To size a program that allocates memory, add the maximum amount of memory that will be allocated. For programs that do DS allocation (for example, Pascal programs that use the "New" function), add the extra amount of DS that will be required. Use of memory array is subject to the availability of a minimal amount. (See "Allocating Memory Space" in Section 4, "Further Information about Linker Options.")

READING THE MAP FILE

VERSION 4

Figure 3-1 shows a sample map file for a version 4 run file.

Addresses

The first three columns in the map show the beginning and ending addresses and the length of each segment. The starting addresses under "Start" are offsets, not absolute addresses. The offsets are relative to the base memory address at which the operating system loads the run file. This base address is determined at run time.

Linker (Version)				
Start	Stop	Length	Name	Class
00000h	00020h	0021h	EXAMPLE_CODE	CODE
00022h	00022h	0000h	CONST	CONST
00022h	00087h	0066h	DATA	DATA
00090h	0009Bh	000Ch	STACK	STACK
0009Ch	0009Ch	0000h	MEMORY	MEMORY

Program entry point at 0000:0000

941-003

Figure 3-1. Sample Version 4 Map File.

Names

The fourth column gives the name of each segment. In the case of a code segment, this name is not the file name of the module.

In most high-level language programs, you assign this module name at the beginning of the module. The compiler creates the code segment name by appending an underscore and a suffix to this assigned module name, and the Linker reports the resulting name here.

In assembly language, you can directly name each segment as you wish. The Linker does not append a suffix to the segment name.

Many programmers choose to assign the same name as both the file name of a module and the module name within the program, for easy reference. This convention is particularly helpful when you are using the map to decide what segments to place in overlays, since file names, and not internal module names, are entered in the "Object modules" field of the Linker command form. You are not required to use this convention, however.

Classes

The fifth column in the map gives the class of each segment. The Linker groups segments by class and uses class to assign order in the program. (See "Ordering Linker Segments" under "How the Linker Works," above.)

VERSION 6

Figure 3-2 shows a map file for a version 6 run file. It is similar in format to the version 4 map file, but it includes another column of numbers in parentheses between "Length" and "Name."

These numbers are 80286 selectors. For each code segment, this selector is the value of the CS register while it is executing, if you are running in 80286 protected mode. For a data segment, this number is the selector that would be used to access data within it.

Linker (Version)				
Start	Stop	Length	Name	Class
00000h	00020h	0021h	(0084h) EXAMPLE_CODE	CODE
00030h	00030h	0000h	(008Ch) CONST	CONST
00030h	00095h	0066h	(008Ch) DATA	DATA
000A0h	000ABh	000Ch	(008Ch) STACK	STACK
000B0h	000B0h	0000h	(008Ch) MEMORY	MEMORY
Program entry point at 0000:0000 (0084:0000)				

941-004

Figure 3-2. Sample Version 6 Map File.

For all segments within a given group, the selector number is the same. (See the Assembly Language Manual for a discussion of groups.)

PUBLIC SYMBOLS AND LINE NUMBERS

Figure 3-3 is the same version 4 map file with the values of all public symbols and their addresses listed. The symbols are sorted first alphabetically and then numerically. The public symbol lists are followed by a list of line numbers.

You request a list of public symbols by entering "y" in the "[Publics?]" field of the Linker command form. You request a list of line numbers separately by entering "y" in the "[Line numbers?]" field.

The Address column in Figure 3-3 contains the notation XXXX:YYYYh, the hexadecimal address of the public symbol.

The Overlay column contains "Res" if the symbol is in the resident portion of your task, an integer (n) if it is in the nth overlay, and "Abs" if it is absolute. An absolute symbol is one with a specified place in memory (for example, an address within the operating system).

Line numbers are intended for use during debugging. They allow you to examine a known part of your program at a known address, even though there is no public symbol at that address. The addresses, however, are relative to the beginning of the run file, so you must do some arithmetic in order to use them.

Linker (Version)				
Start	Stop	Length	Name	Class
00000h	00020h	0021h	EXAMPLE_CODE	CODE
00022h	00022h	0000h	CONST	CONST
00022h	00087h	0066h	DATA	DATA
00090h	0009Bh	000Ch	STACK	STACK
0009Ch	0009Ch	0000h	MEMORY	MEMORY

Publics by name	Address	Overlay
ANOTHERSAMPLEPROCEDURE	0000:000Dh	Res
MAIN	0000:0012h	Res
SAMPLEDATA	0002:0002h	Res
SAMPLETABLE	0002:0004h	Res
SAMPLEPROCEDURE	0000:0008h	Res

Publics by value	Address	Overlay
SAMPLEPROCEDURE	0000:0008h	Res
ANOTHERSAMPLEPROCEDURE	0000:000Dh	Res
MAIN	0000:0012h	Res
SAMPLEDATA	0002:0002h	Res
SAMPLETABLE	0002:0004h	Res

Line numbers for EXAMPLE_CODE							
4	0000:0008H	5	0000:000BH	6	0000:000DH	7	0000:000DH
8	0000:0010H						
9	0000:0012H	10	0000:0012H	11	0000:0015H	12	0000:001AH
13	0000:001FH						
14	0000:0000H	15	0000:0008H				

Program entry point at 0000:0000

941-005

Figure 3-3. Sample Version 4 Map File with Lists of Public Symbols and Line Numbers.

Figure 3-4 shows a list of public symbols and a list of line numbers and addresses in a version 6 map file.

In the list of public symbols in the version 6 map, the name of the public symbol is followed by two addresses. The first is the address in real mode, and the second is that in protected mode.

In a version 6 run file, operating system absolute addresses are converted to an 80286-compatible form (call gate and global descriptor table, or GDT), but they are still denoted as absolute in this listing. Application-defined absolute addresses are not permitted in version 6 run files.

Linker (Version)					
Start	Stop	Length		Name	Class
00000h	00020h	0021h	(0084h)	EXAMPLE_CODE	CODE
00030h	00030h	0000h	(008Ch)	CONST	CONST
00030h	00095h	0066h	(008Ch)	DATA	DATA
000A0h	000ABh	000Ch	(008Ch)	STACK	STACK
000B0h	000B0h	0000h	(008Ch)	MEMORY	MEMORY
Publics by name		Address	Overlay		
ANOTHERSAMPLEPROCEDURE		0000:000Dh	(0084:000Dh)	Res	
MAIN		0000:0012h	(0084:0012h)	Res	
SAMPLEDATA		0003:0000h	(008C:0000h)	Res	
SAMPLETABLE		0003:0002h	(008C:0002h)	Res	
SAMPLEPROCEDURE		0000:0008h	(0084:0008h)	Res	
Publics by value		Address	Overlay		
SAMPLEPROCEDURE		0000:0008h	(0084:0008h)	Res	
ANOTHERSAMPLEPROCEDURE		0000:000Dh	(0084:000Dh)	Res	
MAIN		0000:0012h	(0084:0012h)	Res	
SAMPLEDATA		0003:0000h	(008C:0000h)	Res	
SAMPLETABLE		0003:0002h	(008C:0002h)	Res	
Line numbers for EXAMPLE_CODE					
4	0000:0008H	5	0000:000BH	6	0000:000DH
8	0000:0010H				
9	0000:0012H	10	0000:0012H	11	0000:0015H
13	0000:001FH				
14	0000:0000H	15	0000:0008H		
Program entry point at 0000:0000 (0084:0000)					

941-006

Figure 3-4. Sample Version 6 Map File with Lists of Public Symbols and Line Numbers.

4 FURTHER INFORMATION ABOUT LINKER OPTIONS

ALLOCATING MEMORY SPACE

Normally, when a task is loaded in a partition, its high end is placed at the high-address end of memory. (See Section 4, "Memory Management," and Section 5, "Task and Partition Management," in the CTOS Operating System Manual, Volume 1.)

During compilation or assembly, a program can allocate memory that it will need during execution. This extra memory takes up space in the program's disk file.

Sometimes it is more efficient for a program to allocate a portion of memory only at load time or during execution. Usually, if a program needs to allocate short-lived memory during execution, it does so via calls to `AllocMemorySL` or `ExpandAreaSL`, and the memory is allocated toward lower addresses. This memory is addressed with 32-bit segment-and-offset addresses.

The Linker allows you to choose either or both of two unrelated options for allocation of memory space at load or run time. These options are DS allocation and the memory array.

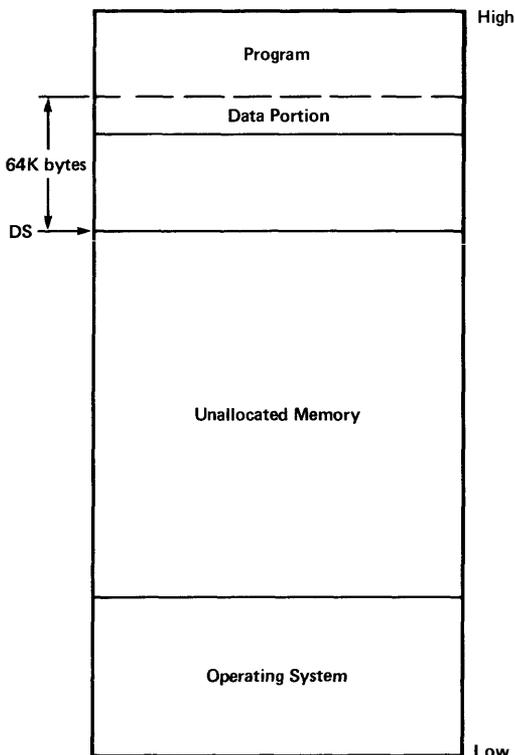
DS allocation allows your program to allocate short-lived memory toward lower addresses as usual, but to address it efficiently with only 16-bit offset addresses.

The memory array option allows the program to allocate memory at high addresses above the program.

DS ALLOCATION

The data segment (addressed by DS) has a maximum size of 64K bytes, and your program takes up a certain amount of that. DS allocation allows you to define a maximum-size data segment, even though your program's data segment would normally be smaller. The excess space in this maximum data segment extends beyond your program toward lower memory addresses. You allocate memory in this space with `AllocMemorySL` or `ExpandAreaSL`, and you can address within this space with 16-bit offset addresses from DS.

To achieve this arrangement, you specify "yes" in the "[DS allocation?]" field of the Linker command form. The Linker then gives DS the lowest possible value that still allows the data segment to encompass your program's data (or DGroup). This arrangement is shown in Figure 4-1.



941-007

Figure 4-1. A Program with DS Allocation.

Note that the program must be arranged with the data segment as its first or lowest-address segment. If your compiler does not order the classes in this way, or if you are writing in assembly language, you must specify the segment ordering in the first object module listed for linking. (See "Creating Linker Segments" in Section 2, "How the Linker Works.")

DS allocation has several advantages. It allows the 16-bit DS-relative addressing discussed above. Also, memory allocated within this space adjoins the common pool of available memory below the program and can be deallocated and reallocated flexibly by the program. However, the program must make procedure calls for memory allocation, and the 16-bit addressable space is less than 64K bytes.

THE MEMORY ARRAY

The memory array is allocated at the high-address end of your program at load time, not through procedure calls. To use the memory array, you specify values in the "[Max memory array]" and "[Min memory array]" fields of the Link command form or in the first parameters of the "[Max array, data, code]" and "[Min array, data, code]" fields of the Bind command form. The memory array is shown in Figure 4-2.

You do not have to know the size of your program or how much memory is available in the partition to specify a memory array. The cParMemArray field of the Application System Control Block structure contains the number of paragraphs of memory array actually available. If the partition cannot accommodate the minimum memory array you requested, the program is not loaded, and the operating system returns a status code and error message.

If you want the task always to load at the lowest possible address, that is, with maximum memory array at the end of the task, set the minimum to 0 and the maximum to 1000000.

The memory array has several advantages. It is not limited to less than 64K bytes but can occupy all available memory in a partition. The program need make no procedure calls to allocate memory during execution. The task is at lower addresses than the memory array. The memory array can be referenced from DS if DGroup is placed at the end of the program.

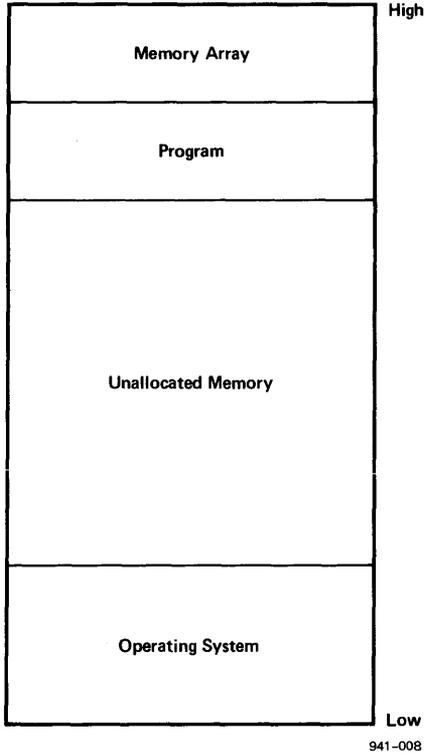


Figure 4-2. A Program with the Memory Array.

The memory array is static, however. You cannot reclaim any of it for other uses, and it persists throughout execution. Also, in the form described here, it cannot, in general, be referenced from DS. Usually, the ES register is loaded with the lowest address of the memory array.

LINKING A SWAPPING PROGRAM

The Virtual Code Segment Management facility, informally referred to as the Swapper, allows an application program that is larger than the memory in its partition to run, but with a performance trade-off. For this purpose, the program's code is divided into variable-length code segments. One, the resident code segment, is permanently in memory. The remaining segments, or overlays,

reside on disk until they are needed. When a procedure in a nonresident overlay is called, the Overlay Manager of the Swapper brings it into memory.

The term "code segment," as used here, is not the same as a linker segment. A Swapper code segment, whether resident or in an overlay, may contain several linker code segments. An overlay, for example, can include differently named code segments originating from several different modules.

Only code, and not data, is placed in overlays.

Module code segments produced by high-level language compilers are pure, so a particular Swapper code segment in memory that is no longer needed can be overlaid by another Swapper code segment. When the first code segment is needed again, it is reread from the run file. Under this system, only code segments, and not data segments, are swapped. Nothing is written back to disk, so there is no need for a disk swap file.

The Swapper can be used with programs written in all of the system's high-level languages, and also by assembly language programs that follow certain rules. Little or no modification is needed to make an existing program swap. You must write a small amount of initialization code, and you must specify in the Linker command form which modules are to contribute code to which overlays. (See "Invoking the Linker" in Section 1 for details on what to enter in the "Object modules" field.)

The Swapper model is more fully described in Section 6, "Virtual Code Segment Management," in the CTOS Operating System Manual. To write a swapping program or to adapt an existing one, see "Virtual Code Segment Management" in the CTOS Programmer's Guide for detailed directions. See also the appropriate language manual. In some languages, you cannot place certain modules from the run-time library in overlays. In assembly language, you must follow call/return conventions and certain other rules if your swapping program is to work. Small-model programs in the C language are not compatible with the Swapper.

ADJUSTING STACK SIZE

All compilers produce information in object modules from which the Linker can compute the size of the required stack segment. For safety, this information usually specifies a stack that is larger than the actual requirements.

REDUCING THE STACK

If your program has a data segment that is close to the 64K-byte size limit, in many cases you can reclaim space by reducing the stack size. For example, if you link a program that uses Forms, ISAM, and Graphics, the Linker supplies extra stack space for each of these products. Examine the size of the default stack by looking at the map file. It is often possible to reduce the amount of stack space by as much as one third without problems.

To estimate the needed stack size more closely, run the program under the Debugger and set a breakpoint at the end of execution or at another convenient point after which the stack has just reached its largest requirement. Because the stack is initialized to zeros, you can now check to see how much of the low part of the stack is still zeros in order to find the maximum requirement. Allow another 128 bytes (64 bytes for interrupt handlers and 64 bytes for making requests) and reduce the stack size accordingly.

CORRECTING STACK OVERFLOW

In rare cases, the compiler can supply information that causes the Linker to undercompute the required stack size. An example is a task with many recursive procedures.

The stack grows down from higher to lower addresses. If a program's needs exceed the stack size, the stack can overwrite whatever precedes it in the link map, causing abnormal program behavior. In this case, relink the program, specifying a larger stack in the Linker command form. The amount of stack needed is highly program dependent and cannot be estimated neatly. Increase the stack to the maximum allowed within

the limitations of your data segment. If the program now runs, reduce the stack size according to the guidelines under "Reducing the Stack," above.

FILE NAMING CONVENTIONS

In the "[Map file]" and "[Symbol file]" fields of the Linker command forms, you can specify file names if you wish. If you do not, a default file name is assigned as follows: the run file name is treated as a character string, and any suffix beginning with a period is removed. For a map file, the new suffix ".map" is added. For a symbol file, the new suffix is ".sym".

For example, if the run file is Prog.run, the default map file is Prog.map and the symbol file is Prog.sym. If the run file is [Dev]<Jones>Main, the map file is [Dev]<Jones>Main.map and the symbol file is [Dev]<Jones>Main.sym.

Supported compilers and the assembler automatically append ".obj" to object module file names.

5 LIBRARIAN

INTRODUCTION

The Librarian is a program development utility that creates and maintains libraries of object modules. A library has three uses:

1. It can be a parameter in the "[Libraries]" field of the Linker command form, to specify that the Linker should search the library for object modules that satisfy unresolved external references. (See Section 1, "Using the Linker (Binder)," for details.)
2. It is a convenient unit in which to collect several object modules and distribute them as a single file. The Librarian extraction facility, which is described below and is also available in the Linker, can be used to extract specific modules from the unit.
3. It is a convenient unit in which to collect several forms created with the Forms Editor. (See the Forms Manual.)

In the first use, you do not have to know the names of the object modules composing a library. The Linker's library search algorithm automatically selects from the library exactly the required modules. Linking object modules from a library specified in the "[Libraries]" field of either Linker command form is faster than linking the same modules specified individually in the "Object modules" field, because in the former case only one file is opened.

In the second and third uses, you must specify the desired object module or form name to extract it from the library.

The Librarian builds or manipulates libraries in these ways:

- o It builds a new library when given the name of a new library file and the object modules that are to compose it.

- o It modifies an existing library when given the names of object modules to be added to or deleted from it. (This includes the case in which a module in a library is to be replaced by a new module with the same name.)
- o It extracts from a library one or more object modules when given the names of the desired object module files.
- o It produces a sorted cross-reference listing of the object modules and public symbols in the library.

INVOKING THE LIBRARIAN

To invoke the Librarian from the Executive, type the command "Librarian", or a unique abbreviation, in the command field and press RETURN. The parameter fields of the Librarian command are as follows:

Librarian

Library file	_____
[Files to add]	_____
[Modules to delete]	_____
[Modules to extract]	_____
[Cross-reference file]	_____
[Suppress confirmation?]	_____

You must fill in the "Library file" field. You can allow optional fields (those in square brackets) to assume their default values by leaving them blank.

Press GO to execute the Librarian command.

You can request multiple operations in one invocation of the Librarian. Modules are deleted, added, and extracted, in effect, in that order. A cross-reference listing reflects the state of the library after all operations are completed.

If you are revising a library module and wish to reinsert it, it is most efficient to use the "[Files to add]" field and allow the Librarian to overwrite the preexisting module of the same name. Deleting the old module and adding the new one takes approximately twice as long.

If you do not wish to receive a message asking for confirmation in this kind of situation (for example, when the operation occurs in a Submit file), respond "y" to the "[Suppress confirmation?]" field.

At the end of a library is a text string that identifies its version. Whenever you modify a library, the text string is lost. For suggestions on housekeeping to avoid confusion resulting from use of altered libraries, see "Librarian Errors" in Section 6, "Troubleshooting."

PARAMETER FIELDS

Library file Enter the file name of the object module library. Typically, it has the form

LibraryName.lib

If the specified file already exists, it is the starting point for any library to be built. Before changes are made, the contents of the file are preserved intact in a file whose name is the original name plus the suffix "-old". However, if no files are added and no modules are deleted (for example, if you only request a listing), then the input library is not modified and no "-old" file is generated. If modifications are requested, then the updated library is named as specified by "Library file."

If the specified file does not exist, you are prompted to confirm the creation of a new library file. You can suppress this request for confirmation by specifying "yes" in the "[Suppress confirmation?]" field.

[Files to add]

Default: No files added

Enter the list of files containing object modules that you want to add to the library. Separate the names with spaces.

If you do not respond, no files are added.

The name of the added module within the library is derived from the name of the added object file. All leading volume, directory, and subdirectory specifications are dropped. Any final extension beginning with a period is dropped. For example, if the file name is

```
[Sys]<Jones>Sort.obj
```

then the module name is Sort. If the file name is

```
<Jones>Working>Sort
```

then the file name is also Sort.

You are prompted for confirmation if an object module that you want to add has the same name as an object module already in the library. If you confirm the replacement, the file containing the module with the same name replaces the existing object module.

You are also prompted for confirmation if a public symbol declared in a module that is to be added conflicts with a public symbol already in the library. If you confirm the duplication, the module containing the duplicate definition is added, but the public symbols (both old and new) are removed from the index of symbols searched by the Linker.


```

Librarian (Version)
ANOTHERSAMPLEPROC.....example  ANOTHERSAMPLEPROC2.....example2
MAIN.....example  MAIN2.....example2
SAMPLEDATA.....example  SAMPLEDATA2.....example2
SAMPLEPROC.....example  SAMPLEPROC*.....example2
SAMPLETABLE.....example  SAMPLETABLE2.....example2

```

example (Length 0093H bytes)

```

      ANOTHERSAMPLEPROC      MAIN      SAMPLEDATA
      SAMPLEPROC      SAMPLETABLE

```

example2 (Length 0093H bytes)

```

      ANOTHERSAMPLEPROC2      MAIN2      SAMPLEDATA2
      SAMPLEPROC*      SAMPLETABLE2

```

941-009

Figure 5-1. Sample Cross-Reference Listing.

[Suppress confirmation?] Default: Prompts for confirmation

Respond to this field with "yes" or "y" if you do not want prompts for confirmation when creating new library files (with the "Library file" field) or replacing existing object modules (with the "[Files to add]" field).

If you enter "no" or do not respond, the Librarian issues prompts for confirmation.

6 TROUBLESHOOTING

LINKER WARNINGS AND ERRORS.....	6-2
Bad max parameter.....	6-3
Bad numeric parameter.....	6-4
Bad yes/no parameter.....	6-5
IDIV instruction in overlay.....	6-6
Illegal segment address reference type n.....	6-7
Input file read error, bad object module....	6-11
Invalid Abs#1 Fixup.....	6-12
Invalid Abs#2 Fixup.....	6-12
Module compiled with public is not resident.....	6-13
Multiply defined symbol.....	6-14
Non "CODE" class loaded into overlay.....	6-15
Noncontiguous GROUPS not 286 compatible....	6-16
No 'OverlayFault' procedure loaded.....	6-17
No run file.....	6-18
No STACK segment.....	6-19
Odd length STACK.....	6-20
Odd size stack requested; rounded up.....	6-21
Proc near nnnn in FileName doesn't follow CALL/RET conventions.....	6-22
Program size exceeds Linker capacity.....	6-23
Relocation offset is too large.....	6-24
Relocation offset from group is too large.....	6-24
Relocation offset of near reference is too large.....	6-26
Requested stack size exceeds 64K.....	6-28
Segment of absolute or unknown type.....	6-29
Symbol file hash table overflow.....	6-30
Symbol table capacity exceeded.....	6-31
Too many PUBLIC symbols.....	6-32
Unresolved externals.....	6-33
NUMERIC STATUS CODES.....	6-34
LIBRARIAN ERRORS.....	6-49

LINKER WARNINGS AND ERRORS

If an error occurs during linking, the Linker displays the message

There were n errors detected

Descriptions of the errors are written to the map file.

The Linker reports three kinds of status messages:

Warning	Informative. Often, the link is completed. The program may or may not run.
Error	Definitely indicates that there is a problem that would prevent the linked program from running.
Fatal error	Causes the link to be aborted.

Linker status messages are listed on the following pages. Unnumbered status messages appear first, in alphabetic order. Numbered messages follow in numeric order.

Numeric status codes for the Linker have the range 4400 through 4423. Status codes from this range that do not appear in the following list are part of normal internal Linker error checking, and you should never see one displayed. Should you observe such a status code, it results from a Linker or compiler error and should be reported.

Many errors result from conditions that are under the control of the assembly language programmer, but that are under the control of the compiler, not the programmer, in a high-level language.

Bad max parameter

EXPLANATION

In the "Link" command form fields "[Max memory array size]" and "[Min memory array size]" or in the "Bind" command form fields "[Max array, data, code]" and "[Min array, data, code]," the minimum specified is greater than the maximum.

SYSTEM ACTION

The link is not initiated.

OPERATOR RESPONSE

Invoke the Linker again, and verify that the value of "[Max memory array size]" is greater than that of "[Min memory array size]."

Bad numeric parameter

EXPLANATION

In either Linker command form, an unrecognized parameter has been entered in a field where a numeric parameter was expected.

SYSTEM ACTION

The link is not initiated.

OPERATOR RESPONSE

Invoke the Linker again, and verify numeric entries in the various fields of the command form.

Bad yes/no parameter

EXPLANATION

A yes/no response to a field in either Linker command form is expected and what is entered does not correspond to either answer.

SYSTEM ACTION

The link is not initiated.

OPERATOR RESPONSE

Invoke the Linker again, correcting the yes/no response parameter. Check the following fields:

[Publics?]
[Line numbers?]
[System build?]
[DS allocation?]

IDIV instruction in overlay

EXPLANATION

This message arises only when a Pascal or FORTRAN program contains code that results in an IDIV (integer division) instruction within an overlay. On an early version of the 80186 microprocessor, a problem existed with integer division on negative operands. System software automatically detects and corrects the problem on affected machines at program load time. However, if IDIV instructions appear in an overlay, the Linker can detect but cannot correct the problem.

The message is reported when you link the program on any system, but it indicates a real problem only if you plan to run the resulting run file on one of the affected systems.

SYSTEM ACTION

The link is completed successfully.

PROGRAMMER RESPONSE

Move the code containing IDIV into the resident, or ensure that all integer-division operands are positive. This is the safest course if you do not know on which machines your program will be run.

The alternative is to avoid using the DIV operator in Pascal or an I/J construction in FORTRAN (where I and J are integers) unless you are sure that all operands are positive.

Illegal segment address reference type n

EXPLANATION

This group of messages, where n can have the values 1 through 5, is related only to the use of overlays. These messages can result from either user or Linker errors. (For a discussion of the Swapper, see Section 6, "Virtual Code Segment Management," in the CTOS Operating System Manual, Volume 1. See also the section with the same title in the CTOS Programmer's Guide.)

Type 1

You are attempting to call a procedure in an overlay, but the Linker has not previously recognized and created a stub for that procedure while setting up the data structures necessary for swapping. This message is usually preceded by the message "Proc near nnnn in FileName doesn't follow CALL/RET conventions" earlier in the link. However, because the Linker cannot detect all cases of call/return violations, the Type 1 message may appear alone. Its meaning is the same in either case.

Type 2

This type occurs only in assembly language or PL/M programs. In a swapping program, it is illegal to use only one part of the two-part address of a procedure. This error indicates that a procedure address has been used partially or incorrectly. For detailed rules for writing a swapping program in assembly language, see "Virtual Code Segment Management and Assembly Code" in the Assembly Language Manual, Section 9, "Accessing Standard Services from Assembly Code."

Illegal segment address reference type n
(continued)

Type 3

As with the Type 2 error, this error indicates that the parts of a procedure address have been separated. In this case, however, the error results from a known problem in noncurrent versions of the Assembler. This error should not occur with a current Assembler.

Type 4

As with Types 2 and 3, a procedure address has been split. This error results from a compiler error in which code that is incompatible with the Swapper is produced. It should not occur.

Type 5

This error occurs only in assembly language programs when a segment and offset are used in other than the two allowed ways: a long CALL instruction or a DD instruction. For details, see "Virtual Code Segment Management and Assembly Code" in the Assembly Language Manual, Section 9, "Accessing Standard Services from Assembly Code."

SYSTEM ACTION

The link is aborted.

PROGRAMMER RESPONSE

Type 1

Examine your assembly language program for call/return violations. If a call/return violation warning was issued, it indicates the location at which to examine your program. (The location cited with the Type 1 message merely indicates where the call occurred. In the absence

Illegal segment address reference type n
(continued)

of a warning message, you can use this information if you have a compilation listing by looking at the assembly code generated by the compiler to see what was being called.)

Certain run-time library modules in noncurrent versions of high-level languages also generated code that violated call/return conventions. Either place such modules and the calls to them in the resident portion of your program's code or upgrade your language version to current level.

See the CTOS Operating System Manual, Section 6, "Virtual Code Segment Management," and the analogous section in the CTOS Programmer's Guide for a detailed discussion of writing a swapping program. See also the appropriate language manual and release notice for language-specific restrictions.

Type 2

Examine your assembly language code for incorrect use of partial procedure addresses. In PL/M, it is possible to generate the same error by using the construction

```
P = @ProcedureName
```

which generates the statement

```
MOV AX, SEG ProcedureName
```

This instruction is illegal in the context of overlays.

To find the address of a procedure name in PL/M in a swapping program, you must define it as a static constant in a DECLARE statement. For example:

```
DECLARE pProcedureName POINTER INITIAL  
    (@ProcedureName);
```

Illegal segment address reference type n
(continued)

Type 3

This error is not under the control of the programmer. If your assembler is noncurrent, upgrade to the current version.

Type 4

This error is not under the control of the programmer. Make sure your compiler is current.

Type 5

Examine your assembly language code for use of a procedure address used in instructions other than a long CALL or DD. This error usually results from using a far JMP, which is illegal in an overlaid program.

Input file read error, bad object module

EXPLANATION

This error can result when you have submitted an invalid object module to the Linker.

SYSTEM ACTION

The link is aborted.

PROGRAMMER RESPONSE

Check your compiler to see that it is current.

Check to see if you have submitted a source file or an object module that is a misnamed text file.

Invalid Abs#1 Fixup

Invalid Abs#2 Fixup

EXPLANATION

These messages pertain to FORTRAN-86 and indicate relocation errors. They may result from either a Linker or a compiler error. It should not appear.

SYSTEM ACTION

The link is completed, but the resulting run file is invalid.

PROGRAMMER RESPONSE

The conditions that cause these errors are not under the control of the programmer. Check to see that your Linker and compiler versions are current.

Module compiled with public symbols is not resident

EXPLANATION

This error occurs only in BASIC programs. The BASIC compiler includes an option for compilation with public symbols. In a swapping program, a BASIC module compiled with public symbols cannot be placed in an overlay.

SYSTEM ACTION

The link is completed. The run file is not executable.

PROGRAMMER RESPONSE

Do not place such a module in an overlay.

Multiply defined symbol

EXPLANATION

The same public symbol cannot be defined more than once in a program. (The same public symbol can be defined in more than one place in a library.) If a public symbol is defined more than once, the Linker uses the first definition that it encounters. (This definition is not necessarily the first one you would encounter in traversing all the library modules in the order in which they appear in the library. See "Library Search Algorithm" under "How the Linker Works" in Section 1, "Linker (Binder).") All subsequent references to that symbol go to the first-encountered definition. On encountering the second definition, the Linker ignores the definition and issues this error.

SYSTEM ACTION

The link is completed.

PROGRAMMER RESPONSE

When the Linker encounters the second definition of the symbol and reports this error, it does not report the location of the first definition in the map. If the second (or later) definition is the one you wish to preserve, use the Librarian to generate a listing that reports the location of each occurrence of the multiply defined symbol, and reorder the object module list to put the desired module first.

Non "CODE" class loaded into overlay

EXPLANATION

In a swapping program, only segments with the class name "code" can be placed in overlays. Placing a segment of any other class in an overlay produces this error.

SYSTEM ACTION

The link is completed. In most cases the run file is not executable.

PROGRAMMER RESPONSE

In general, place only code in overlays.

The program may run if the affected overlay is not to be used as an overlay: for example, if you have placed data in an overlay as a convenient way of getting it into a run file but not loading it into memory.

Noncontiguous GROUPS not 286 compatible

EXPLANATION

Splitting of grouped segments with insertion of nongroup material, as described under the "Relocation offset is too large" error, below, is not permitted in a program to run on the 80286 processor. Submitting such a module to the Linker to produce a version 6 run file results in this error.

SYSTEM ACTION

The link is completed; the run file is invalid.

PROGRAMMER RESPONSE

If your compiler (for example, certain versions of the FORTRAN-86 compiler) produces object modules in which splitting occurs, link your FORTRAN program with First.obj as the first object module in the list, as described in the FORTRAN-86 Release Notice; or write your own template assembly language module to reorder the segments, as described under the "Relocation offset is too large" error, below.

No 'OverlayFault' procedure loaded

EXPLANATION

In a swapping program, no call to InitOverlays or InitLargeOverlays was made, and thus the overlay handler is not loaded.

SYSTEM ACTION

The link is completed, but the program should not be run.

PROGRAMMER RESPONSE

Insert a call to InitOverlays or InitLargeOverlays in your program. For details on writing a swapping program, see "Virtual Code Segment Management" in the CTOS Programmer's Guide.

No run file

EXPLANATION

No run file name was specified in the Linker command form. The run file name field is not optional and does not default.

SYSTEM ACTION

The link is not initiated.

OPERATOR RESPONSE

Invoke the Linker again, and enter a name in the "Run file" field.

No STACK segment

EXPLANATION

A stack segment must exist in the program. This error usually occurs in an assembly language program when you have forgotten to include the stack. It may also occur when you are linking an unusual program that you do not intend actually to run.

SYSTEM ACTION

The link is completed, but the program will not run. Attempting to run such a program by pressing CODE-GO causes a system crash without entry into the Debugger.

PROGRAMMER RESPONSE

In an assembly language program, include a stack segment.

Odd length STACK

EXPLANATION

The length of stack segments must be an even number of bytes. The Linker verifies this length and reports this error if it finds an odd count.

The odd-length stack, if found, was generated by the compiler.

SYSTEM ACTION

The link is completed successfully. The stack size is rounded up to the next even number.

PROGRAMMER RESPONSE

This problem is not under the control of the programmer. Verify that your compiler is current.

The run file will probably execute correctly.

Odd size stack requested; rounded up

EXPLANATION

Because the stack is composed of words, the length of the stack must be an even number of bytes. This warning results when you request an odd-length stack in the "Stack size" parameter of either Linker command form or in an assembly language program.

SYSTEM ACTION

The stack size is automatically rounded up to the next even number. The link proceeds normally.

OPERATOR RESPONSE

No action is needed. To avoid such messages, specify an even-length stack for the ["Stack size"] parameter or in your assembly language program.

**Proc near nnnn in FileName doesn't follow
CALL/RET conventions**

EXPLANATION

In a swapping program, accepted call/return conventions are not followed. Because the Swapper must trace the stack during a return to a procedure in an overlay, an abnormal stack causes such a program to fail.

The Linker cannot detect all cases of call/return convention violations; and not all such reports are fatal. However, when such a warning is accompanied by a later message of the form

Illegal segment address reference of type n

a fatal error has occurred. (See the discussion of the "Illegal segment address reference" errors, in this section.)

SYSTEM ACTION

The link is completed.

PROGRAMMER RESPONSE

A true call/return convention violation usually occurs in an assembly language program, but it can result from the use of a noncurrent compiler or from placing an inappropriate run-time library module in an overlay.

This condition probably is not a problem in a nonswapping program. However, it is best to follow the call/return conventions because program maintenance may make a program swap at a later time.

For details on writing a swapping program, see "Virtual Code Segment Management" in the CTOS Programmer's Guide; see also the appropriate language manual.

Program size exceeds Linker capacity
Too many PUBLIC symbols

EXPLANATION

These errors may be reported because insufficient memory is available to the Linker. There is no fixed limit on the size of the program to be linked, but certain tables built by the Linker must be resident in memory. If these tables cannot be built, one of these errors results.

SYSTEM ACTION

The link is aborted.

OPERATOR RESPONSE

Increase the amount of memory available to the Linker.

Relocation offset is too large

Relocation offset from group is too large

EXPLANATION

Some high-level language compilers use a medium model in which the sum of data, constant, and stack segments is limited to 64K bytes. Usually, these error messages mean that your program contains too much data, which causes the sum of these segments to exceed 64K bytes. This situation can arise when a program with a large data declaration is being ported from another system.

In some cases, one of these errors arises when a compiler inserts another kind of area between two of the segments mentioned. For example, the FORTRAN compiler produces object modules in which data and constant segments are separated by a common area, thus:

data, common area, constants, stack

The Linker retains the same segment ordering that it encounters in the first object module listed for linking. Thus, the run file in this example would contain the linker segments data, common, constants, and stack, in that order. The total extent of these linker segments, from the beginning of data to the end of stack, must be less than 64K bytes. A large common area may cause the sum to exceed 64K bytes.

Another cause of this split-and-insertion problem is the placement of the memory segment (which contains nothing but is taken into account when the Linker checks for this error) at the end of the series of segments.

Any violation of this 64K-byte restriction produces one of these relocation offset errors. (A "Segment size exceeds 65520" error, status code 4405, is generated if any one segment exceeds 64K bytes.)

Relocation offset is too large

Relocation offset from group is too large

(continued)

SYSTEM ACTION

The link is completed, but the run file is invalid.

PROGRAMMER RESPONSE

When this error is caused simply by excessive length, dynamically allocate short-lived memory (using AllocMemorySL) in your program, rather than declaring large arrays, buffers, and so on. In FORTRAN, you can reduce data segment lengths by moving variables into common blocks.

When the error is caused by split and insertion, as described above, use an assembly language program that does nothing but declare the class names of the segments in a different order, placing this module first in the list of modules to be linked. This first module serves as a template, and the Linker then orders segments from the following modules in the same way. (For a discussion of ordering by class name, see the Assembly Language Manual. For a FORTRAN-86 program, see "Using Arrays Larger than 64K" in the FORTRAN-86 Release Notice, for details.)

Relocation offset of near reference is too large

EXPLANATION

A near call is one in which the called address is less than 64K bytes from the caller's address, and 16-bit addressing can be used. A far call is one outside this range, for which 32-bit addressing must be used.

In most high-level language compilers, the medium model is used, and any call between modules with a public symbol is a far call. However, in a small-model program such as that generated by the C compiler, or in certain assembly language programs, near calls are generated between modules. In this case, if the caller and the called address are more than 64K bytes apart, the above error is reported.

SYSTEM ACTION

The link is completed, but the run file is invalid.

PROGRAMMER RESPONSE

Make your program smaller, or reorder the object modules to bring references and addresses closer together.

If the caller and called addresses both come from a high-level language, this error usually results from an attempt to call a variable in a data segment.

If the caller or the called address comes from assembly language, make sure both addresses are in the same group. If possible, change the near call to a far call.

Relocation offset of near reference is too large
(continued)

In many cases, the error message reports a public symbol name, which you can use to identify the call that generated the error. If, however, the error message reports only a hexadecimal address, you should examine a compilation listing and determine from the assembly code what was being called when the error occurred.

Requested stack size exceeds 64K

EXPLANATION

The stack may not exceed 64K bytes in length.

SYSTEM ACTION

The link is not initiated.

OPERATOR RESPONSE

Invoke the Linker again, and enter a decimal number smaller than 64K in the "[Stack size]" field.

Segment of absolute or unknown type

EXPLANATION

Certain types of segments, such as absolute segments, are not generated by the Assembler or the supported compilers, but they are available from other related assemblers and compilers. The Linker cannot handle such segments, and it reports this message if you submit one to it.

SYSTEM ACTION

The link is completed, but the run file may be invalid.

PROGRAMMER RESPONSE

Verify that the object modules submitted were generated by the supported assembler or compilers.

Symbol file hash table overflow

EXPLANATION

The upper limit on the size of the symbol table in the symbol file (512 sectors, or 256K) has been exceeded. There may be too many public symbols in the program. This message may also appear if there are many public symbols with very long names.

SYSTEM ACTION

The link is aborted.

PROGRAMMER RESPONSE

Reduce the number of public symbols in the program. If many symbols have long names, reduce the length of the names.

Symbol table capacity exceeded

EXPLANATION

The capacity of the symbol table is a function of the number of symbols, the length of the symbol strings, and the use of overlays. Using overlays nearly doubles the amount of symbol table space required by a program. The symbol table capacity is 512K bytes, which must accommodate the symbol names and a certain overhead per symbol.

SYSTEM ACTION

The link is aborted.

PROGRAMMER RESPONSE

Reduce the number of public symbols in the program; or reduce symbol length by using a more terse naming convention.

Too many PUBLIC symbols

See "Program size exceeds Linker capacity," above.

Unresolved externals

EXPLANATION

The Linker matches external references with symbol definitions in other program or library modules. If such a match cannot be found for an external reference, this message results. The list of the undefined symbols and the modules in which they appear is written to the map file.

If you allow the "[Version]" field to default when you are linking the operating system or any product that uses a version number, an unresolved external error results and is recorded in the map file.

SYSTEM ACTION

The link is completed. In many cases the Linker can modify the code of a CALL instruction that calls an undefined symbol, changing it into a call to the Debugger. The Linker cannot, however, make this adjustment on indirect calls.

On this system, unlike some others, a program containing unresolved externals can be run. Doing so is risky, however. Although calls to the Debugger may happen in most cases, sometimes the result is a call to location 0:0 and a system crash.

PROGRAMMER RESPONSE

You can run a program containing undefined symbols if you know that they will not be called: for example, when development is still in progress and the routine referred to by the undefined name has not been written.

Verify the list of unresolved externals to make sure that the names reported are those that you expect. If an unresolved external is reported for the version number, relink the program, specifying a version number in the "[Version]" field of the Linker command form.

200	Write error on run file
to	Write error on symbol file
299	Write error on list file
	Cannot open temporary file
	Write error in temporary file
	VM read error

EXPLANATION

Any of these six messages indicates that a file system error has occurred. The numerical status code related to the file system is passed through by the Linker.

The first three messages often indicate that the disk is full.

The last three messages are related to the fact that where memory is limited, the Linker creates a temporary file called VM.tmp. If the directory for temporary files (the \$ directory) is damaged or missing, the Linker cannot create this file, and one of these errors is reported.

SYSTEM ACTION

The link is aborted.

PROGRAMMER RESPONSE

Determine the nature of the file system error and correct it. (For a list of file system errors, see the Status Codes Manual.)

200 Error during legalese
to
299

EXPLANATION

Using the development version of the Linker command form, it is possible to append to the run file a file named "legalese," which contains copyright or other information. One of these errors, typically with status code 203, results when the file to be appended is missing or unreadable.

SYSTEM ACTION

The link is completed normally, but the message appears to notify the user. The program is not abnormal and will run successfully.

OPERATOR RESPONSE

Supply the needed file if you wish to have its contents appended to the run file.

400 Not enough memory available

EXPLANATION

The Linker swaps and adapts itself to the amount of available memory on a system, but it must have a certain minimum memory in order to run. If this amount is not available, this message results.

SYSTEM ACTION

The link is aborted.

OPERATOR RESPONSE

If running the Linker under the Context Manager, reconfigure the partition size, adding more memory until the Linker can run.

If running the Linker under a single-partition operating system, do whatever housekeeping is necessary to increase the amount of available memory, or run the link on a system with more main memory.

The more memory is available to the Linker, the faster the link takes place.

1380 Heap errors
to
1390

EXPLANATION

Errors in this series are internal memory management errors. You should never see one of these errors reported.

SYSTEM ACTION

Such an error usually causes the system to exit to the Executive or to hang.

OPERATOR RESPONSE

If you ever observe such an error, you should report it.

**4400 Attempt to access data outside of segment
 bounds, possibly bad object module**

EXPLANATION

This error can result when an invalid object module produced by a faulty compiler is submitted to the Linker. However, it also results from a known problem in the assembler. If you do not use a segment directive in your assembly language program, or if you declare code or data outside any segment, the assembler supplies a segment called ??SEG. Such an object module is invalid and causes the Linker to produce this message.

SYSTEM ACTION

The link is aborted.

PROGRAMMER RESPONSE

In an assembly language program, make sure to include a segment directive. Other occurrences of this error, caused by compiler or Linker errors, are not within the programmer's control.

4403 **Too many segment or class names**
4404 **Too many segments**

EXPLANATION

No more than 255 segments can be declared in a single module. (The overall program may contain more than 255 segments.)

Further, no more than 255 different names of segments and classes can exist in one module.

SYSTEM ACTION

The link is aborted.

PROGRAMMER RESPONSE

Examine your program with this limitation in mind. You should not encounter this restriction in normal programming, but certain special kinds of modules may approach this limit. In such a case, divide the module.

4405 Segment size exceeds 65520

EXPLANATION

One segment cannot be larger than 65,520 bytes (not 65,536). This error pertains only to a single segment, not to a group or sum of segments (for example, DATA, CONST, and STACK).

SYSTEM ACTION

The link is aborted.

PROGRAMMER RESPONSE

If you are writing in assembly language or Pascal, reduce the size of the segment to less than 65,520. In FORTRAN-86 only, it is possible to use a special segment directive to create a 65,536-byte segment that is accepted.

In small-model C, reduce the total amount of data or code in your program, or move to a larger model of computation.

4406 Too many groups

EXPLANATION

A given program module may not contain more than 10 groups. The entire program may contain up to 256 groups. (For a discussion of groups, see the Assembly Language Manual, Section 2, "Programs and Segments.")

SYSTEM ACTION

The link is aborted.

PROGRAMMER RESPONSE

The limits should never be encountered under normal programming conditions. Check the number of groups defined in your program.

4407	Too many public symbols in one module
4408	Too many external symbols in one module

EXPLANATION

These errors occur only if the Linker is operating in insufficient memory.

SYSTEM ACTION

The link is aborted.

OPERATOR RESPONSE

Increase the amount of memory available to the Linker.

4409 Invalid object module

EXPLANATION

The Linker does not recognize the format of an object module. An invalid object module can result from a compiler error or from direct damage to the file.

SYSTEM ACTION

The link is aborted.

OPERATOR RESPONSE

Verify that you are indeed submitting an object module, and not a text file such as the source file, to the Linker.

4411 Too many common symbols in one module

EXPLANATION

This error occurs only if the Linker is operating in insufficient memory.

SYSTEM ACTION

The link is aborted.

OPERATOR RESPONSE

Increase the amount of memory available to the Linker.

**4413 Bad object module, segment or group index
 out of range**

EXPLANATION

An invalid object module, usually created by a faulty compiler, has been submitted to the Linker.

SYSTEM ACTION

The link is aborted.

PROGRAMMER RESPONSE

Upgrade to the current compiler.

**4414 Too many public procedures in
 resident/overlay**

EXPLANATION

The maximum number of procedures in the resident portion or any single overlay in a swapping program is 4,096.

SYSTEM ACTION

The link is aborted.

PROGRAMMER RESPONSE

This limit should seldom be reached in normal programming. If it is, divide the code into more overlays.

4418	Too many segments
4419	Too many segments
4420	Too many areas

EXPLANATION

The Linker requires a certain minimum memory to run successfully. If memory is insufficient, one of these errors may be reported.

SYSTEM ACTION

The link is aborted.

OPERATOR RESPONSE

Make more memory available to the Linker.

4422	Bad object module, external index out of range
4423	Bad object module, external index out of range

EXPLANATION

An invalid object module, usually created by a faulty compiler, has been submitted to the Linker.

SYSTEM ACTION

The link is aborted.

PROGRAMMER RESPONSE

Upgrade to the current compiler.

LIBRARIAN ERRORS

Many of the status messages reported by the Librarian are similar to those of the Linker because the two programs are related in their structure and function. The causes of Librarian status messages generally fall into categories:

- o command form entry errors
- o file system errors
- o invalid object modules
- o multiply defined symbols
- o symbol table full, excess public symbols

Most status messages are self explanatory and are described under the analogous Linker messages in the previous subsection. In many cases, the remedy is obvious.

MULTIPLY DEFINED SYMBOLS

Messages concerning multiply defined public symbols are exceptions. See "[Files to add]" and "[Cross-reference file]" under "Parameter Fields" in Section 5, "Librarian."

EXCESS PUBLIC SYMBOLS

Also, unlike the situation in the Linker, you cannot remedy errors involving excess public symbols by increasing the amount of available memory, since memory management in the Librarian is less sophisticated than that in the Linker. If one of these errors is reported, you must divide your library into two libraries.

In a library where there are many multiply defined symbols, the symbol table may be of adequate size if you choose to add, delete, or extract modules, but it may be exceeded if you request a listing. In order to list the symbols, the Librarian must expand the single statement of a multiply defined symbol, creating separate symbols with varying numbers of asterisks. In this process, the symbol table can be exceeded.

HOUSEKEEPING CONCERNS

Whenever you use the Librarian to add or delete a module, the identifying text string at the end of the library file is lost. This change is intentional, because a string that identifies a version should not remain if the library has been changed.

If this deletion causes problems for you, use the following housekeeping methods:

- o Create a <Lib> directory.
- o Copy the needed libraries into it from their distribution diskettes.
- o Rename each library with a descriptive name (for example, s9.01DBMS.lib). A common convention is to preserve the original version number in the new name for future reference as to the origin of the library.
- o If you use the Librarian to add or change a module, again rename the library (for example, 9.3CTOSSamGenAll.lib)
- o When you link, explicitly list the libraries in the "[Libraries]" field of either Linker command form (using the "at-file" convention if necessary), and be sure to use "none" as the last entry. (The "at-file" convention is discussed under "Substituting the Contents of a File for a List of Subparameters" in Section 5, "Advanced Concepts," in the Executive Manual.)

GLOSSARY

See also the Glossary in the CTOS Operating System Manual, Volume 2.

absolute symbol. An absolute symbol is a symbol that has a specified place in memory (as, for example, an address within the operating system).

alignment attribute. The alignment attribute of a segment element specifies whether it can be aligned on a byte, word, or paragraph boundary.

class name. A class name is an arbitrary symbol used to designate a class.

DGroup. DGroup is a group that usually includes data, constant, and stack linker segments. See **group**.

DS allocation. DS allocation, an option in the Linker, locates DGroup at the end of a 64K-byte segment addressed by the DS register.

external reference. An external reference is a reference from one object module to variables and entry points of other object modules.

extraction. Extraction is the copying of a module from a library into another file or into a program being linked. Extraction does not delete the extracted module from the library.

group. A group is a named collection of linker segments that is addressed at run time with a common hardware segment register. To make the addressing work, all the bytes within a group must be within 64K of each other.

Librarian. The Librarian is a program development utility that creates and maintains libraries of object modules. The Linker can search automatically in such libraries to select just those object modules referred to by a program.

Linker. The Linker is a program development utility that combines object modules (files produced by compilers and assemblers) into run files.

linker segment. A linker segment is a single entity consisting of all segment elements with the same segment name.

map. The map file, which is created by the Linker, contains an entry for each linker segment and shows the relative address and length of the segment in the memory image. It can also list public symbols and line numbers with addresses.

memory array. The memory array is data space allocated at load time above the highest address of the task.

object module. An object module is the result of a single compilation or assembly. A single object module is contained in an object module file (.obj), while many object modules can be contained in a library file (.lib).

offset. The number of bytes by which a memory location is distant from the beginning of a segment.

overlay. An overlay is a code segment made up of the code from one or more object modules. An overlay is loaded into memory as a unit and is not permanently memory resident.

public symbol. An ASCII character string associated with a public variable, a public value, or a public procedure.

public variable. A public variable is a variable whose address can be referenced by a module other than the module in which the variable is defined.

relocation. The operating system relocates a task image in available memory by supplying physical addresses for the logical addresses in the run file at load time.

relocation directory. The relocation directory is an array of locators used by the operating system in relocating the task image.

resident. The resident portion of a program remains in memory throughout execution. See also **overlay** and **virtual code segment management**.

run file. A run file is the image of a task (in relocatable form) linked into the standard format required by the operating system loader. The run file consists of a header and a memory image.

segment. A segment is a contiguous (usually large) area of memory that consists of an integral number of paragraphs. Segments are usually classified into one of three types: code, static data, or dynamic data. Each kind of segment can be either shared or nonshared.

segment element. A segment element is a section of an object module. Each segment element has a segment name.

short-lived memory. Short-lived memory is an area of memory in an application partition. When a task is loaded, the operating system allocates short-lived memory to contain its code and data. Short-lived memory can also be allocated directly by a client process in its own partition. Common uses of short-lived memory are input/output buffers and the Pascal heap.

swapping. See **virtual code segment management.**

virtual code segment management. Virtual code segment management (also called swapping) is the method of virtual memory supported by the CTOS operating system. The code of each task is divided into variable-length segments that reside on disk in a run file. As the task executes, only those code segments that are required at a particular time actually reside in the main memory of the application partition; the other code segments remain on disk until they, in turn, are required. When a particular code segment is no longer required, it is simply overlaid by another code segment.

unresolved external reference. An unresolved external reference is a public symbol that is used by some module but not defined by any of the modules being linked.

INDEX

Page numbers in boldface indicate the principal discussion of the topic.

- 80286 processor
 - absolute address conversion for, 3-5
 - protected mode of, 3-3, 3-5
 - selectors on, 3-3
- Absolute addresses, 3-4, 3-5
- Address
 - of public symbol, 3-4
 - of task, 1-1, 3-2
 - relative, assigning, 2-1
- Addressing
 - allocated memory, 4-1 to 4-4
 - data segment, 4-1 to 4-4
 - linker segments, 2-8 to 2-9, 2-10
 - in protected mode, 2-15 to 2-16
 - in real mode, 2-15 to 2-16
 - segment, 1-13
- Algorithm, library
 - search; **see** searching
 - Libraries:
 - searching
- Alignment attributes, 2-9 to 2-10
- Allocation
 - data space, 1-9
 - memory, 1-13, 4-1 to 4-4
- Assembler, versions of, 1-2
- Assembly language, 2-3, 2-6, 2-9, 3-3
 - defining stack in, 2-7
 - segment order in, 4-2
 - swapping programs, 4-5
- Attributes
 - alignment, 2-9 to 2-10
 - combination, 2-7
- Bind command, 1-4, 1-9, 4-3
 - form, 1-4
 - parameter fields, 1-4 to 1-14
- C language, 4-5
- Class
 - name, 3-3
 - ordering by Linker, 2-3, 2-6, 2-10, 4-2
 - segment element, 2-3, 2-10
- Code segment
 - in map file, 3-3
 - name, 3-3
- Combination rules, segment element, 2-7
- Command form
 - Bind, 1-4
 - Librarian, 5-2
 - Link, 1-3
- Commands, 1-3, 1-4
- COMMON segment
 - elements, combination of, 2-7
 - length, 2-7
- Compatibility of software versions, 1-2
- Compiled language, 1-2, 2-3, 2-9, 3-3, 4-5, 4-6
- Compiler, 2-7
 - versions of, 1-2
- Context Manager, 3-1

Conventions, xi
 call/return, 4-5
 naming, 1-2, 1-6, 3-3

CTOS
 MegaFrame, 1-3
 versions, 1-3

CTOS.lib
 and program size, 3-1
 searching, 1-11 to 1-12
 versions of, 1-2

Data
 space, allocating, 1-9
 segment, 4-6
 addressing, 4-1 to 4-4
 in map file, 3-3

Debugger, 4-6

Default
 library searching, 1-12
 values, 1-4

Descriptor table, local, 2-15

DGroup, 1-13, 2-9, 4-2, 4-6

DS
 allocation, 1-13, 3-2, 4-1 to 4-3
 register, 2-9, 4-1 to 4-4
 value of, 1-13

Errors
 Librarian, 1-2, 6-49 to 6-50
 Linker, 1-2, 1-11, 6-2 to 6-48
 stack overflow, 4-6

ES register, 4-4

External references, 1-1, 1-11, 2-1 to 2-2, 3-4, 5-1

Extraction of library module, 1-5, 2-1 to 2-2, 5-1

Fatal error, 6-2

Fields, parameter; **see** Parameter fields

File names
 conventions, 4-7
 library, 5-3

Files, library; **see** Object modules

First.obj, 2-6

Forms, collection in library, 5-1

FORTRAN, DS allocation in, 1-13

FORTRAN-86, 2-6

Group, 2-8 to 2-9, 2-10; **see also** DGroup
 in version 4 run file, 2-9
 in version 6 run file, 2-9

Header, run file; **see** Run file header

High-level language, 1-2, 2-3, 2-7, 2-9, 3-3, 4-5

Image, task, 1-1

Invocation
 Librarian, 5-2
 Linker, 1-3

Language, compiled; **see** Compiled language

Language, high-level; **see** High-level language

LDT; **see** Local descriptor table

Length of segments, 2-7

- Librarian, 1-11, 5-1 to 5-7
 - command form, 5-2
 - cross-reference
 - listing, 5-2, 5-6
 - errors, 6-49 to 6-50
 - invoking, 5-2
- Libraries
 - adding modules to,
 - 5-2, 5-4
 - building, 5-1, 5-4
 - deleting modules from,
 - 5-2, 5-5
 - identifying text
 - string in, 5-3
 - modules in overlays,
 - 1-7
 - of object modules, 5-1 to 5-7
 - revising or modifying,
 - 5-2, 5-4, 5-5, 6-50
 - searching, 1-11, 2-1 to 2-2, 5-1, 5-5, 5-6
 - specifying for
 - linking, 5-1
- Limits on linkable program size, 2-11
- Line numbers
 - in map file, 3-4 to 3-5
 - listing in map file,
 - 1-8
- Link command, 1-3, 1-9, 4-3
 - form, 1-3
 - parameter fields, 1-4 to 1-14
- Linker
 - errors and warnings,
 - 1-11, 6-2 to 6-48
 - features of, 1-1, 1-2
 - invoking, 1-3 to 1-14
 - ordering rules, 2-2, 2-6, 2-10, 4-2
 - segment, 2-6, 2-7, 2-10
 - addressing, 2-8 to 2-9, 2-10
- Listing
 - Librarian; **see** Librarian
 - Linker; **see** Map file
- Loading, 1-1, 4-1, 4-3
- Local descriptor table, 2-15
- Long-lived memory, 1-9, 4-1
- Map file, 1-2
 - addresses in, 3-2
 - code segment in, 3-3
 - data segment in, 3-3
 - default name of, 1-6
 - error reporting in,
 - 6-2
 - line numbers in, 1-8, 3-4
 - name, 4-7
 - public symbols in,
 - 1-8, 3-4 to 3-5
 - reading, 3-1 to 3-6
 - specifying, 1-6
 - stack size in, 4-6
- Medium-model task, 2-9
 - DS allocation with,
 - 1-13
- Memory
 - address of task, 4-1, 4-3
 - allocation, 1-9, 1-13, 4-1 to 4-4
 - and program size,
 - 3-1, 3-2
 - arranging, 1-13, 2-2 to 2-10
 - task image in, 1-2
 - array, 1-9, 4-1, 4-3 to 4-4
 - and program size,
 - 3-2
 - image of task, 2-2
 - long-lived, 1-9, 4-1
 - requirements of program, 3-1
 - short-lived, 1-9 to 1-10, 4-1
- Model of computation or segmentation, 1-13, 2-9, 4-5
- Module name, 3-3
- Modules, object; **see** Object modules
- Multiply defined symbol; **see** Public symbol

Name
 library module, 5-4
 module, 2-3
 segment element, 2-3,
 2-10

Naming conventions, 1-2,
 1-6, 3-3
 for file names, 4-7

Object modules, 1-4,
 2-1, 2-2, 2-3
 adding to library,
 5-2, 5-4
 deleting from library,
 5-2, 5-5
 extraction from
 libraries, 1-5,
 5-1, 5-2, 5-5
 file names, 4-7
 Librarian listing of,
 5-2, 5-6
 libraries of, 5-1 to
 5-7
 listing as input, 1-4
 to 1-6
 names in library, 5-4
 specifying, 2-6

Operating system
 building, 1-10, 1-11
 versions, 1-3

Ordering
 rules, Linker, 2-2,
 2-6, 2-10, 4-2
 template, 2-6, 4-2

Order of classes, 2-3,
 2-6, 2-10, 4-2

Overflow, stack, 4-6

Overlays, 1-1, 1-2, 1-6,
 1-7, 1-10, 1-12,
 2-14, 3-3, 3-4, 4-4
 to 4-5
 library modules in,
 1-7

Parameter fields
 Bind command, 1-4
 Librarian, 5-2
 Link command, 1-3

Pascal
 DS allocation in, 1-13
 program size, 3-2

Program
 memory address, 3-2
 size, 3-1
 in Pascal, 3-2

Protected mode, 3-3, 3-5
 addressing, 2-15 to
 2-16

Public symbols, 1-1,
 1-14, 2-1 to 2-2
 Librarian listing of,
 5-2, 5-6
 in map file, 1-8, 3-4
 to 3-5
 multiply defined, 5-4,
 5-6
 number of, 2-11
 undefined, 2-1

Real mode addressing,
 2-15 to 2-16

References, external,
 1-1, 2-1 to 2-2, 5-1

Relocation, 1-1, 4-1,
 4-3
 directory, 2-14

Resident code segment,
 4-4 to 4-5

Rules
 ordering, 2-2, 2-6,
 2-10, 4-2
 segment element combi-
 nation, 2-7

Run file, 2-1
 building, 2-6 to 2-10
 finding size of, 3-1
 header, 1-1, 1-11
 structure of, 2-11
 to 2-16
 name, 1-6
 size limits, 2-11
 version 4, 2-11 to
 2-14
 version 6, 2-11 to
 2-16
 versions, 1-3, 1-9

Run-time library and
 program size, 3-1

sbVerRun, 1-11

Searching libraries,
 1-11, 2-1 to 2-2,
 5-1, 5-5, 5-6

Segment; **see also** Segment element
 addressing, 2-8 to 2-9, 2-10
 class, 2-2, 2-10, 3-3
 code, 2-7
 length, 3-2
 linker, 2-6, 2-7, 2-10
 meaning, 4-5
 name, 2-2, 2-10, 3-3
 ordering rules, 2-6, 2-10, 4-2
 registers, 2-8 to 2-9, 2-10
 selector value, 3-3
 Segmentation; **see** Model of computation or segmentation
 Segment element, 2-3, 2-10; **see also** Segment
 class, 2-3, 2-6, 2-10
 code, 2-3
 combination rules, 2-7
 COMMON, 2-7
 constants, 2-3
 data, 2-3
 name, 2-3, 2-10
 stack, 2-3, 2-7
 Selectors, 3-3
 Short-lived memory, 1-9 to 1-10, 4-1
 Size
 of group, 2-8, 2-10
 limits on linkable program, 2-11
 of program, 3-1 to 3-2
 Small-model tasks
 DS allocation with, 1-13
 swapping and, 4-5
 SP register, 2-9
 SS register, 2-9
 Stack
 default size, 1-8, 4-6
 initial values in, 4-6
 overflow, correcting, 4-6
 reducing size of, 4-6
 segment
 elements, combination of, 2-7
 length, 2-7
 specifying size of, 1-8, 4-6 to 4-7
 Status
 codes, 6-2 to 6-48
 messages, 6-1 to 6-50
 Swapper, 1-1, 1-2, 1-6, 1-7, 1-12, 2-14, 3-3, 3-4, 4-4 to 4-5; **see also** Overlays; Swapping programs
 Swapping programs; **see also** Swapper
 linking, 4-4 to 4-5
 size of, 3-2
 Symbol
 file, 1-14
 name, 4-7
 table, 1-14, 2-1
 public; **see** Public symbols
 System build, 1-10
 Task
 image, 1-1
 arranging in memory, 1-2
 memory address, 1-1, 1-2, 3-2, 4-1, 4-3
 memory image, 2-2
 Template, ordering, 2-6, 4-2
 Troubleshooting, 1-2, 6-1 to 6-50
 Undefined symbols, 2-1
 Unresolved external reference, 1-11, 2-1, 5-1
 Version 4 run file, 1-3, 1-9
 groups in, 2-9
 header format, 2-11 to 2-14
 map file for, 3-2

Version 6 run file, 1-3,
1-9
groups in, 2-9
header format, 2-11 to
2-16
map file for, 3-3 to
3-4, 3-5, 3-6
Version number, 2-16
examining, 1-11
Versions
of library, 5-3
operating system, 1-3
of run files, 1-3, 1-9
of software, compati-
bility, 1-2, 1-3
specifying, 1-10 to
1-11

Virtual Code Segment
Management, 1-1,
1-6, 1-7, 4-4 to
4-5; **see also**
Swapper; Swapping
programs; Overlays

Warnings, Linker, 6-2 to
6-48