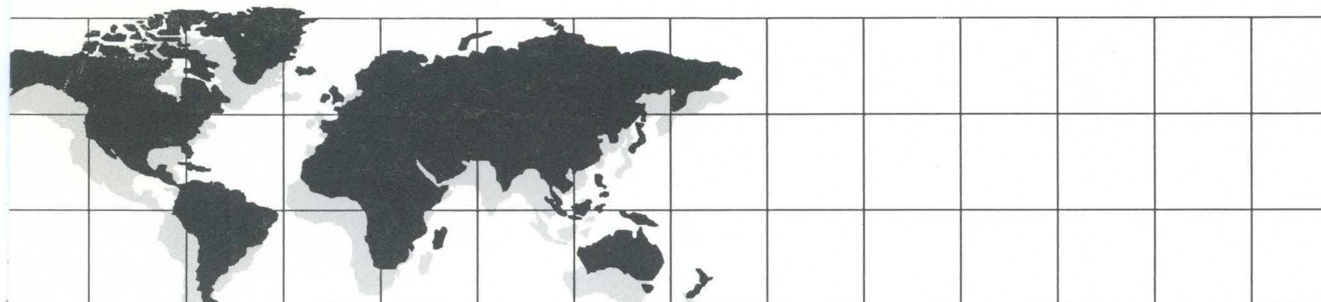


CTOS

Programming Utilities

Reference Manual
Building Applications

UNISYS

UNISYS

CTOS®
Programming Utilities
Reference Manual
Building Applications

Copyright © 1992, 1993 Unisys Corporation
All Rights Reserved
Unisys is a registered trademark of Unisys Corporation

CTOS Development Utilities 12.2, 12.3

Priced Item

November 1993

Printed in USA
4359 4969-100

The names, places, and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living, or otherwise. Any similarity or likeness of the names, places, and/or events with the names of any individual, living or otherwise, or that of any group or association is purely coincidental and unintentional.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THIS DOCUMENT. Any product or related information described herein is only furnished pursuant and subject to the terms and conditions of duly executed agreement to purchase or lease equipment or to lease software. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, special or consequential damages.

You should be careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

RESTRICTED RIGHTS LEGEND. Use, reproduction, or disclosure is subject to the restrictions set forth in DFARS 252.227-7013 and FAR 52.227-14 for commercial computer software.

Copyright © 1992, 1993 Unisys Corporation
All Rights Reserved
Unisys is a registered trademark of Unisys Corporation

Convergent, Convergent Technologies, CTOS, and SuperGen are registered trademarks of Convergent Technologies, Inc.

Context Manager, X-Bus, and X-Bus+ are trademarks of Convergent Technologies, Inc.

BTOS is a trademark of Unisys Corporation.

IBM, IBM PC, and OS/2 are registered trademarks of International Business Machines Corporation. Intel is a registered trademark of Intel Corporation. MS-DOS, and Microsoft are registered trademarks of Microsoft Corporation. Presentation Manager and Windows are trademarks of Microsoft Corporation. UNIX is a registered trademark of AT&T. XVT is a trademark of XVT, Inc.

Contents

About This Manual

Introduction xv
What This Manual Covers xv
Who Should Use This Manual xv
How This Manual Is Organized xvi
What Is New in This Manual xix
Terminology Used in This Manual xix
Conventions Used in This Manual xx
    Capitalization xx
    CTOS Naming Conventions xx
    CTOS File Naming xx
        Roots and Suffixes xxi
        File Suffixes xxi
Where to Find More Information xxii
Recommended Reading xxiv

Section 1. Introduction to Building Applications

Introduction 1-1
Building Applications Utilities 1-1
Relationship Among the Development Utilities 1-2

Section 2. What Is the Linker?

Introduction 2-1
What the Linker Does 2-1
User Interfaces 2-3
    Command Forms 2-3
    Configuration File 2-3
    Input File Types 2-4
Files the Linker Creates 2-4
Understanding the Loader 2-5

Section 3. Using the Linker Command Forms

Introduction 3-1
Using the Link V8 and Link V6 Command Forms 3-4
Link V6 and Link V8 Commands Parameter Fields 3-6
Examples 3-17
Example: Listing Object Modules 3-17
Example: Searching Libraries 3-20
Example: Using Overlays 3-20
Sorting Procedure Names in Overlays 3-23
Configuring the Linker 3-24
Search Path 3-27
Library References 3-28
Library Search List 3-29
Linker Configuration File Parameters 3-31
Customizing Virtual Memory Sizes 3-40

Section 4. Reading the Linker Map File

Introduction 4-1
A Simple Map File 4-1
Version 6 Map File 4-1
Addresses 4-2
Protected Mode Selectors 4-2
Names 4-3
Classes 4-3
Version 8 Map File 4-3
Map Files With Public Symbols, Line Numbers, and Details 4-5
Version 6 Map File 4-5
Library References 4-5
Public Symbols 4-9
Line Numbers 4-9
Command Form Parameter Details 4-10
Configurable Linker Work Areas 4-10
Version 8 Map File 4-11

Section 5. How the Linker Works

Introduction 5-1
Linking Overview: A Two-Pass Process 5-1
Pass One 5-1
Library Search Algorithm 5-2
Dynamic and Static Linking 5-2
Pass Two 5-3
From Source Modules to Run File on Disk 5-3
Arranging Object Module Components 5-4
Segment Element Names and Classes 5-4
Creating Linker Segments 5-8
Specifying Linker Segment Order 5-9
Combination Rules 5-9
Addressing Linker Segments 5-11
Placing Uninitialized (Communal) Variables in DGroup . 5-11
Alignment Attributes 5-12
Summary of Segment Ordering 5-15
Segment Limits 5-15

Section 6. Advanced Linker Features

Introduction 6-1
Program Memory Requirements 6-1
Run-Time Library Code 6-3
Simple Programs 6-3
Overlay Programs 6-3
Programs That Allocate Memory 6-4
Adjusting Stack Size 6-4
Reducing the Stack 6-4
Correcting Stack Overflow 6-5
Allocating Memory Space 6-5
DS Allocation 6-6
The Memory Array 6-7
Linking a Program With Overlays 6-9
Customizing Segment Ordering 6-10
First.asm File 6-10
First.asm File Example 6-11
Example of Correcting a Segment Ordering Error 6-14
Configuration File 6-16

Section 7. What Is the Librarian?

What the Librarian Does 7-1
How the Librarian Works 7-2
Uses for the Librarian 7-3

Section 8. Using the Librarian Command Form

Introduction 8-1
Command Form 8-1
Parameter Fields 8-2
Library Block Size 8-6
Conserving Library Space 8-7
Accommodating Large Modules 8-8
Library Index Procedures 8-9
Duplicate Public Symbol Names 8-9
Uninitialized Variables 8-9

Section 9. What Is the Module Definition Utility?

What the Module Definition Utility Does 9-1
Porting Presentation Manager Programs 9-1
How the Module Definition Command Works 9-2
A Closer Look at Module Definition Command Output . . . 9-5
Object Module 9-6
Import Library 9-6

Section 10. Using the Module Definition Command Form

Introduction 10-1
Command Form 10-1
Parameter Fields 10-2

Section 11. Writing a Module Definition File

Introduction 11-1
The Need for a Module Definition File 11-1
General Syntax Rules 11-2
Defining the Client Interface to a DLL 11-3
Using an Import Library 11-4
Using an Imports Statement 11-4

Porting Programs to CTOS 11-5
Statements Included for Compatibility 11-5
CTOS Extensions 11-6
Parameters Not Recognized 11-6
Segment Attribute Recommendations 11-6

Section 12. Module Definition Statements

Statements 12-1
Code 12-4
Data 12-6
Description 12-8
ExeType 12-9
Exports 12-10
HeapSize 12-12
Imports 12-13
Library 12-15
LoadType 12-17
Name 12-18
Old 12-20
ProtMode 12-21
RealMode 12-22
RunType 12-23
Segments 12-25
StackSize 12-27
Stub 12-28
Segment Attributes 12-29
Instance and Shared Attribute Field Effects 12-32
Using Only the Shared Field 12-32
Using Both the Shared and Instance Fields 12-32

Section 13. What Is the Resource Librarian?

Introduction 13-1
What Are Resources? 13-3
What the Resource Librarian Does 13-3
How Resources Are Stored 13-5
How the Resource Librarian Identifies Resources 13-7

Section 14. Using the Resource Librarian Command Form

Introduction 14-1
Command Form 14-1
Parameter Fields 14-2
Examples of Adding Resources 14-7
Example 1: Adding a Data File 14-7
Example 2: Adding a Single Resource From a Run File 14-8
Example 3: Adding Multiple Resources From a Run File 14-8
Examples of Deleting Resources 14-9
Example 1: Deleting a Single Resource From a Run File 14-9
Example 2: Deleting Multiple Resources From a Run File 14-9
Example of Extracting a Resource 14-10
Example: Extracting a Resource From a Run File 14-10

Section 15. Using the Resource Librarian Configuration File

Introduction 15-1
Resource Librarian Configuration File Format 15-1
Example Resource Librarian Configuration File 15-2

Appendix A. Status Codes A-1

Appendix B. Run File Reference B-1

Appendix C. Object Module Formats (OMF) C-1

Appendix D. Calling Medium Model Procedures from a DLL . . . D-1

Appendix E. Version 4 Link Command E-1

Glossary Glossary-1

Index Index-1

Figures

1-1. The CTOS Application and DLL Development Process 1-3

2-1. Linking Object Modules Into a Run File 2-2

5-1. How the Linker Builds a Run File 5-6
5-2. Combination of Stack and Common Segment Elements 5-10
5-3. How the Linker Builds a Run File (Not all Data in DGroup) 5-13

6-1. Real Mode Program With DS Allocation 6-7
6-2. A Program With the Memory Array 6-8

7-1. Using the Librarian to Manage Application Object Modules 7-2

8-1. Library Blocks 8-7

9-1. Creating a DLL for Use by an Application 9-3
9-2. Linking an Application to Access a DLL During Execution 9-4
9-3. Module Definition Command Output 9-5

13-1. Adding Resources to a CTOS Run File 13-2
13-2. Tasks Performed by the Resource Librarian 13-4

Tables

ATM-1. File Suffixes xxi

3-1. Recommendations for Selecting a Command Form 3-3
3-2. Overview of Linker Parameter Fields 3-5
3-3. Run File Mode Options 3-11

12-1. Module Definition Statements 12-2
12-2. Attribute Definitions 12-30
12-3. Segment Attribute Default Values 12-31
12-4. Shared Field Effects 12-32
12-5. Instance and Shared Field Effects 12-33

13-1. Resource Librarian File Definitions 13-6

Examples

3-1.	Linker Configuration File	3-25
3-2.	Library Reference Examples	3-29
4-1.	Sample Version 6 Map File	4-2
4-2.	Sample Version 8 Map File Showing a Nonshared Segment Entry	4-4
4-3.	Sample Map for a Version 6 Run File Showing Lists of Public Symbols, Line Numbers, and Details	4-6
4-4.	Sample Map for a Version 8 Run File Showing Public Symbols, Line Numbers, and Details	4-12
6-1.	First.asm File	6-12
6-2.	The Map File Produced by the First.asm File	6-13
6-3.	Map File Showing Segment Ordering Error	6-14
6-4.	First.asm File Showing Corrected Segment Order	6-15
8-1.	Sample Cross-Reference Listing	8-5
11-1.	Module Definition File Example	11-2

About This Manual

Introduction

This manual describes how to use the following utilities to build applications to run on CTOS operating systems:

- Linker
- Librarian
- Module Definition utility
- Resource Librarian

This manual should be used in conjunction with other manuals that describe the Unisys family of information processing systems. (For a list of these manuals, see "Related Documentation" later in his section.)

What This Manual Covers

The CTOS Development Utilities provide libraries and tools that the programmer uses to develop CTOS applications. This manual specifically focuses on the tools that are used to build executable files and product libraries.

Who Should Use This Manual

This manual is designed for developers who need to build applications that run on the CTOS operating system. This manual makes the following assumptions:

- You understand the CTOS operating system, its file specifications, and Executive commands. If you do not, see the *Executive User's Guide* and the *Executive Reference Manual*.

- You have a basic familiarity of programming on the CTOS operating system. If you do not, see the introductory section of the *CTOS / Open Programming Practices and Standards, Application Design* and the *CTOS Concepts Manual*.
- You are familiar with the Intel architecture of 80X86 processors.
- If you are developing dynamic link libraries (DLLs) or porting these libraries to CTOS, you should understand dynamic linking on CTOS systems.

How This Manual Is Organized

This manual is organized as follows:

Section 1. Introduction to Building Applications

This section provides a general overview of the tools used to build applications.

Part I Linker

Section 2. What Is the Linker?

This section introduces the Linker and the various Linker commands. It also discusses the configuration file, the input to the Linker, and the files the Linker creates.

Section 3. Using the Linker Command Forms

This section describes the Linker command forms and parameter fields. Also included is a description of the configuration file.

Section 4. Reading the Linker Map File

This section explains how to read the map file produced by the Linker.

Section 5. How the Linker Works

This section provides theory on dynamic and static linking and explains additional options available for linking more complex programs.

Section 6. Advanced Linker Features

This section describes the program memory requirements and includes a description of how to customize segment ordering.

Part II Librarian

Section 7. What Is the Librarian?

This section describes the Librarian, a utility used to organize object modules into groups.

Section 8. Using the Librarian Command Form

This section describes the Librarian command form and parameter fields. The last part of the section describes how to specify library block size.

Part III Module Definition Utility

Section 9. What Is the Module Definition Utility?

This section provides a conceptual overview of the Module Definition utility and introduces new terms pertaining to dynamic linking. It also describes how to use the Module Definition utility to define the interface between a client and a dynamic link library.

Section 10. Using the Module Definition Command Form

This section describes how to use the Module Definition command form and explains command output.

Section 11. Writing a Module Definition File

This section describes general syntax rules for module definition file statements and porting considerations, including compatible statements and CTOS extensions to module definition syntax.

Section 12. Module Definition Statements

This section describes the module definition file statements. It also describes how to interpret the results of using segment attributes, which can be specified in Code, Data, and Segment statements.

Part IV Resource Librarian

Section 13. What Is the Resource Librarian?

This section describes the Resource Librarian, a utility used to place data resources into run files.

Section 14. Using the Resource Librarian Command Form

This section provides a detailed explanation of the Resource Librarian command form and parameter fields.

Section 15. Using the Resource Librarian Configuration File

This section describes the Resource Librarian configuration file.

Appendix A. Status Codes

This appendix describes error messages produced by all four utilities, their possible causes, and actions you can take to resolve them.

Appendix B. Run File Reference

This appendix describes all the fields in Version 8 and Version 6 run file headers.

Appendix C. Object Module Formats (OMF)

This appendix describes Intel object module formats (OMF) used by the Linker.

Appendix D. Using Medium Model Procedures as Dynamic Link Libraries

This appendix describes issues involved in calling medium model procedures from DLLs.

Appendix E. Version 4 Link Command

This appendix describes the **Link** command user interface, the Version 4 run file headers, run file format, and Version 4 map file format.

A glossary follows the appendixes.

What Is New in This Manual

This manual includes the following new information:

- A new introductory section to the manual provides an overall frame of reference to the utilities and shows the relationships among them.
- The introductory section for each utility includes an illustration that represents the development process for that particular utility.
- The Linker support for Microsoft C case sensitivity is described in Section 3, "Using the Linker Command Forms."
- It describes new Linker configuration file functionality, such as displaying the current usage state of the Linker internal tables as described in Section 3, "Using the Linker Command Forms."
- Several Linker error codes are described in Appendix A, "Status Codes."

Terminology Used in This Manual

There are three commands available for linking: Link V8, Link V6, and Link. This manual uses the phrase "the Linker command forms" when the text generally describes any of these commands. Otherwise, the specific command name is used.

Conventions Used in This Manual

This manual uses the following notations:

- Variable names are shown in *italics* for quick reference.
- File names are shown in *italics* for quick reference.
- Command names are shown in **boldface**.
- Configuration file entries are shown in *italics*.
- User-entered values are shown in **boldface** .
- Brackets [ ] enclose optional variables.
- The notation "..." means that the preceding variable(s) can be repeated.

Capitalization

By default, the utilities described in this manual are not case sensitive. For example, you are not required to enter parameter values, for example, in a particular case. Names can be entered in uppercase or lowercase letters with no change in meaning. However, for ease in reading, this manual follows the naming conventions for standardization as described in "CTOS Naming Conventions," below.

CTOS Naming Conventions

For information about CTOS naming conventions, see "The CTOS Programming Environment" in the *CTOS Programming Guide, Volume 1*. For a comprehensive description of how to assign relevant variable names, see the *CTOS Procedural Interface Reference Manual*.

CTOS File Naming

The CTOS file naming conventions used in this manual are described below.

Roots and Suffixes

Program file names have root names and suffixes. The root name describes your source code text file. The suffix describes further information about the file, for example, whether it is the source file, the compiled version, the list file, the executable file, and so forth.

For example, the CTOS naming convention for describing a C language source file is to append the suffix *.c* to the root name.

If *MyProgram* is the root name, the source file name is *MyProgram.c*. (See Table ATM-1 for a list of suffixes used in this manual. Additional file suffixes are given in the *CTOS Programming Guide, Volume 1.*)

File Suffixes

Table ATM-1 describes the file suffixes used in this manual.

Table ATM-1. File Suffixes

Suffix	Use
.asm	Assembler source file (Assembler or masm).
.c	C language source file.
def.lst	List file from the Module Definition utility (not assigned by default).
def.obj	Object module from the Module Definition utility.
.dll	Dynamic link library from the Linker (not assigned by default).
imp.lib	Import library from the Module Definition utility (not assigned by default).
.lib	Library file created by the Librarian (not assigned by default).
.map	Map file from the Linker.
.obj	Object module from a compiler or the assembler.
.res	Binary resource file.
.run	Run file from the Linker (not assigned by default).
.sym	Symbol file from the Linker.

Where to Find More Information

This manual is one of a related manual set that documents the CTOS family of information processing systems. For a description of each manual in the set and for order information, see your sales representative or the *Unisys Customer Product Information Catalog.*

Documents referenced in this manual are listed below.

CTOS Executive Reference Manual

This reference manual is organized alphabetically by command name. It includes comprehensive information about Executive features and the commands packaged with the CTOS operating system and Standard Software.

CTOS/Open Programming Practices and Standards, Application Design

This manual describes the programming practices that must be followed in order for an application to run on all CTOS platforms. In addition, the manual serves as an introduction to programming in CTOS. It provides information on several core areas such as basic input/output (I/O), error handling, parameter management, guidelines for protected mode programming, writing nationalizable programs, writing system services, stack format and calling conventions, mixed language programming, writing multiprocess programs, overlays, customized SAM, and communications programming. It includes programming examples.

CTOS Operating System Concepts Manual, Volume 1 and 2

This two-volume set introduces the CTOS III 1.1 workstation operating system with multiprocessing capability. In addition, the manual describes the CTOS II 3.4 and CTOS I 3.4 workstation operating systems, and the CTOS/XE 3.4 operating system for shared resource processors. The text provides an orientation to basic system concepts the programmer needs to understand to write programs to be run on CTOS. Topics include DMA buffer management, system bus management, memory management, demand paging, dynamic linking, and semaphores.

CTOS Procedural Interface Reference Manual

This alphabetically organized four-volume reference manual describes each of the programming operations for CTOS III, the real mode and protected mode versions of CTOS II, and CTOS/XE. It also includes the data structures.

CTOS Programming Guide, Volume 1 and 2

Volume I describes the programming interfaces to many internal features of the operating system. It focuses primarily on features which may not be common to all versions of the operating system, or to all hardware platforms. Volume II describes the programming interfaces to features that are not internal to the operating system, but which are packaged with it. The *CTOS Programming Guide* is meant to be used in conjunction with *CTOS / Open Programming Practices and Standards.*

CTOS Programming Utilities Reference: Assembler

This guide describes using the CTOS Assembler. It is a new and revised edition that includes the information on the Assembler previously documented in the *CTOS Development Utilities Programming Reference Manual.* It is meant to be used in conjunction with the Intel CPU Programming Reference manuals.

CTOS Programming Utilities Reference: Customization

This manual describes using the keyboard customization utilities to support use of localized or custom keyboards.

CTOS Programming Utilities Reference: Installation and Command Overview

This manual provides an introduction to the programming utility commands distributed with the CTOS Development Utilities software package. It also provides reference information on each command.

CTOS Debugger User's Guide

This manual describes the features and commands supported by the real-mode and protected-mode debuggers for CTOS I 3.4, CTOS II 3.4, CTOS/XE 3.4, CTOS III 1.0. Hands-on exercises in using the commands to debug programs are provided. This guide also describes how to use the Debug File Utility to debug crash dumps and describes the Debugger Application Programming Interface. For CTOS III 1.1, there is improved handling of multiprocessing debugging for multiprocessor EISA-bus workstations.

Recommended Reading

For additional reference information on microprocessor architecture, see the following books:

- Intel Corporation. *iAPX 286 Programmer's Reference Manual.*
- Intel Corporation. *80386 Programmer's Reference Manual.*
- Intel Corporation. *80486 Programmer's Reference Manual.*

For a discussion of OMF formats, see the following books:

- Intel Corporation. *8086 Relocatable Object Module Formats.* Intel Corporation, 1981. Order no. 121748-001.
- MS-DOS Encyclopedia. Microsoft Press, 1991.

For background information on OS/2 dynamic linking, see the following book:

- Michael J.Young. *Software Tools for OS/2: Creating Dynamic Link Libraries.* Addison-Wesley, 1989.

Section 1
Introduction to Building Applications

Introduction

This section introduces the concept of building applications. It includes the following information:

- A description of the tools that can be used to create an executable file. Details of each utility are described in the following sections of this manual.
- A process flow diagram that shows the development process for building CTOS applications and dynamic link libraries (DLLs).

The libraries that are shipped with the building applications utilities are listed in the *Software Release Announcement*.

Building Applications Utilities

The CTOS Development Utilities provide libraries and tools for developing CTOS applications. This manual describes the core set of utilities used most often in the development process:

- Linker
- Librarian
- Module Definition utility
- Resource Librarian

The other development utilities (not described in this manual) are

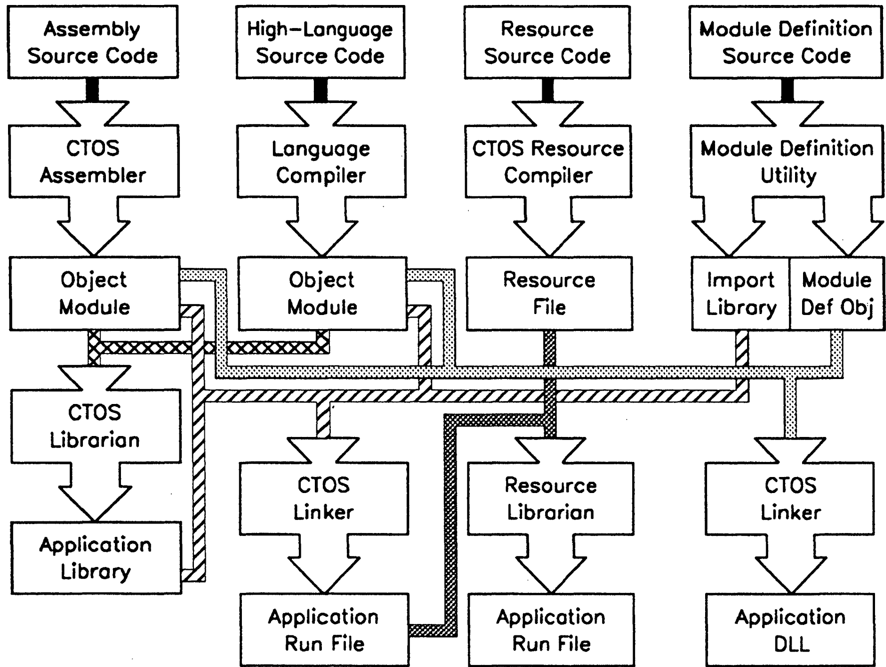
- Assemble
- Convert Nls.sys
- Convert Public Case
- Convert Sys.keys
- Create Keyboard Data Block
- Histogram
- PMake
- Wrap

For more information about these, see the *CTOS Programming Utilities Reference Manual: Installation and Command Overview.*

Relationship Among the Development Utilities

Figure 1-1 below shows the relationship among the various utilities described in this manual when all are used. The part of this diagram that applies to an individual utility is extracted and shown in the introductory section to that utility.

Figure 1-1. The CTOS Application and DLL Development Process



Briefly, here is an overview of how to use each utility:

Linker	You use the Linker to build executable files to run on the CTOS operating system. It produces standard CTOS run files as well as dynamic link libraries. The Linker is capable of both static and dynamic linking.
Librarian	You use the Librarian to create libraries, add object modules to static libraries, and delete and extract object modules from libraries. The Librarian can also produce a sorted cross-reference of object modules and public symbols within a library.
Module Definition utility	You use the Module Definition utility to create object modules that contain special data required by the Linker to build dynamic link libraries and set up import library information that defines client interfaces to those libraries.
Resource Librarian	You use the Resource Librarian to maintain sets of data resources within run files. For example, you can use the Resource Librarian to include resources such as dialog boxes for graphical user interfaces, or even symbol files in the run file. You can add, delete, or extract resources from a run file or a binary resource file. The Resource Librarian can also produce a list of resources within a run file or a binary resource file.

Section 2
What Is the Linker?

Introduction

This section contains an overview of the Linker. It is a high-level introduction to what the Linker does, how it accepts user input, and the files it creates.

Details on how the Linker creates a run file are described in Section 5, "How the Linker Works." If you are curious about the theory of linking, read that discussion now. If you use optional Linker features, you will need to know about classes, segments, and other subjects described there. However, if you have an immediate need to link a program, proceed to "Linker User Interfaces" in Section 3, "Using the Linker Command Forms." It explains how to link a simple program without requiring knowledge of how the Linker works internally.

To find out how to read the map file, a file the Linker creates that contains information on how the run file was organized, see Section 4, "Reading the Linker Map File."

If you want to learn about program memory requirements, linking with overlays, and customizing segment ordering, read Section 6, "Advanced Linker Features."

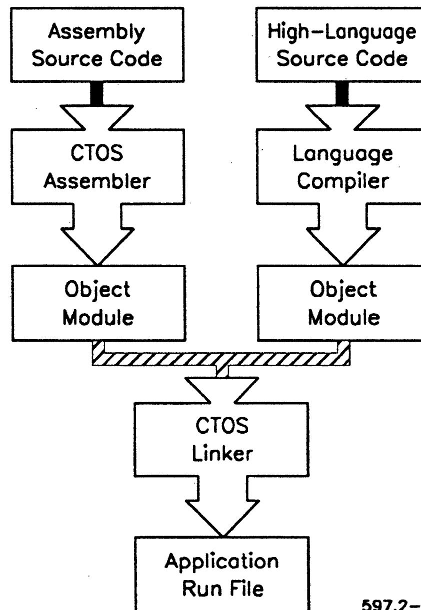
What the Linker Does

The Linker accepts object modules (files produced by compilers and assemblers), separates them into their component pieces, and groups pieces of the same type. Then it recombines the components in a prescribed order to form an executable image called a run file, adjusting memory references accordingly. A run file is the image of a task (in relocatable form) linked into the standard format required by the operating system loader.

Figure 2-1 below depicts how object modules are linked into a run file. The Linker also accepts object files and libraries from the Module Definition utility in support of DLLs.

Linking is one part of the application and DLL development process. To see how the Linker works in conjunction with the other building applications utilities, see Figure 1-1 in Section 1, "Introduction to Building Applications."

Figure 2-1. Linking Object Modules Into a Run File



User Interfaces

The Linker accepts user input through two main vehicles: a Linker command form and a configuration file. Your input to the Linker is an *object module*. The object module can be an assembly language module, the output of any CTOS compiler, or the output from the Module Definition utility.

Command Forms

This section describes two different Linker command forms: one for the **Link V8** command and one for the **Link V6** command. (An earlier version of the command, **Link**, is discussed in Appendix E, "Version 4 Link Command." **Link** is no longer recommended but is still supported.) Recommendations on which command to use are included in Section 3, "Using the Linker Command Forms."

Version 8 run files are supported only on virtual memory operating systems. If you need to run your application on another CTOS platform, use a Version 6 run file. For more information, see Section 5, "How the Linker Works."

Input from the command form takes precedence over input defined in other sources. For example, a module definition file (used when you are creating a DLL) can define some of the same parameters you can specify in a Linker command form (such as stack and heap size). If, however, you specify different values for the same parameters in both places, the Linker uses the command form values. (For details on module definition files, see Section 11, "Writing a Module Definition File.")

Configuration File

Input can also be specified in the Linker configuration file (default name: *LinkerConfig.sys*). In this file you can specify the library or library group you want the Linker to search. You can also define more complex parameter templates for performing procedures such as packing code or translating far to near calls.

Input File Types

The Linker uses object modules as input.

Input Type	Description
Object Module	Object modules can be created by supported compilers or assemblers (including the object module created by the Assembler from the *First.asm* file. For information on the *First.asm* file, see Section 6, "Advanced Linker Features").

Files the Linker Creates

The Linker creates three types of files: a run file, a symbol file, and a map file, as described below.

File Type	Meaning
Run File	The Linker combines object modules (files produced by compilers and assemblers) into run files. A run file is linked into the standard format required by the operating system loader. The run file consists of at least a header and a memory image. The header describes the run file and provides certain initial values. It also contains an array of pointers to intersegment references that allow the operating system to relocate the run file to any appropriate memory location and access dynamically linked modules. A run file produced by the Linker can thus be used with various memory configurations or as one of several run files in a multitasking program. Run files are relocatable at run time.

Symbol File	The Linker writes the names and locations of all the public symbols in a program to the symbol file. This file is useful for debugging. (See Section 5, "How the Linker Works," for details on how the Linker creates the symbol file. See the *CTOS Debugger User's Guide* for details on how to use it to debug programs.)
Map File	The map file contains an entry for each linker segment and shows the relative address and length of the segment in the run file memory image. If there are link errors, they are displayed in this file. (See Section 4, "Reading the Linker Map File," for details on how to use the map file.)

Understanding the Loader

The loader sets up the environment for the run file.

Using a run file for input, the loader produces an executable image in memory. It reads the run file and interprets it to create memory, selectors, and a process in an environment in which that run file can execute.

On a virtual memory operating system, the loader fetches the DLLs a program needs, loads them, and binds the client to them.

Three CTOS operations invoke the loader:

- Exit, where the current application program is terminated and the exit run file is loaded.
- Chain, where a run file replaces the current application.
- LoadPrimaryTask (or LoadInteractiveTask), where a run file is loaded into a vacant application partition.

Section 3
Using the Linker Command Forms

Introduction

This section describes the Linker command forms and includes the following information:

- Recommendations for selecting a particular Linker command form.
- A description of the **Link V8** and **Link V6** command forms.
- Descriptions of all parameter fields.
- A description of run file mode options.
- A description of the Linker configuration file (the default name is *LinkerConfig.sys*) and the configuration file parameters.

Linker User Interfaces

The Linker can read user-specified input from two sources:

- A command form
- An optional configuration file

The configuration file and command form work together to specify the parameters for the link. You can specify all the information you need from the command form. The configuration file offers you a chance to set up the Linker specifically for your environment and the options you use most often.

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.

If you are linking a program that has special requirements, you should know the command line parameters and configuration file options that can help you. For example, if your program allocates additional memory at run time, you must provide an estimate of that memory. You can do this with a couple of fields in the Linker command form. See "Link V6 and Link V8 Commands Parameter Fields" for detailed descriptions of these and all the parameter fields.

This section first describes how to use the command form options, then goes on to describe the configuration file.

Selecting the Command Form to Use

The Linker offers two command forms: **Link V6** and **Link V8.**

Though the **Link V6** and **Link V8** command forms are exactly alike, the commands produce different run files. **Link V6** generates a run file that runs on all operating systems. You use **Link V8,** however, when you need to use DLLs or if you want your run file to run on a virtual memory operating system only.

You can select the command form you need based on the recommendations given in Table 3-1.

Table 3-1. Recommendations for Selecting a Command Form

Linking Command*	Link V8	Link V6
Type of Run File	V8 Run File	V6 Run File
Executes On	CTOS III	CTOS I CTOS II CTOS III CTOS/XE
Execution Mode	Protected	Protected and Real
DLL Support	Yes	No

* For information on the **Link** command, see Appendix E.

Note: *The **Link V8** command generates a run file that only runs on virtual mode operating systems. This is also the only command that supports DLLs. In all other cases, use **Link V6**.*

Using the Link V8 and Link V6 Command Forms

To use the Linker for a simple link, follow these steps:

1. On the Executive command line, type either **Link V6** or **Link V8**, and press **RETURN**.

2. Fill in the command form according to your needs. An example command form is shown below. The parameter fields are described in detail in Table 3-2.

   Typically, you enter information in very few of the fields. Note that the command form shows *YourCCompiler.lib* in the *[Libraries]* field.

```
Link V8
  Object modules                 Prog.obj
  Run file                       Prog.run
  [Map file]
  [Publics?]
  [Line numbers?]
  [Stack, Dgroup heap size]
  [Max array, data]
  [Min array, data]
  [Run file mode]                Protected
  [Version]
  [Libraries]                    YourCCompiler.lib
  [DS allocation?]
  [Symbol file]
  [Copyright notice?]
  [File to append]
  [Linker config file]
```

3. Press **GO**. The Linker generates a run file and writes error information to the screen.

The Linker searches the system libraries by default for the public names of system procedure calls. However, it does not resolve references to procedure calls in the programming language you use unless you are either using a default language library, or you enter the specific name of your language library either in the *[Libraries]* field of the command form or in the Linker configuration file.

If you need the map file to include more information about your run file, or if you need to specify certain memory constraints, you can fill out the other parameter fields. This is described in detail later in this section.

Table 3-2. Overview of Linker Parameter Fields

Field Name	Description
Object modules	Name(s) of the object module file(s) or a library file with a list of object modules to be drawn from it.
Run file	Name of the run file you want to create.
[Map file]	Name of the file that shows the relative address and length of each segment in the memory image.
[Publics?]	If Yes, the Linker lists the relative addresses of all public symbols at the end of the map file.
[Line numbers?]	If Yes, the Linker adds a list containing the address and line number of each source statement.
[Stack, Dgroup heap size]	Stack size and the DGroup heap size.
[Max array, data]	Maximum amount of short-lived memory that will be reserved within a partition.
[Min array, data]	Minimum amount of short-lived memory that must be available within a partition.
[Run file mode]	Run file mode. See Table 3-3 for a description of the options. The keyword you enter determines the type of run file.
[Version]	Version number in the header of the run file and for the public variable *sbVerRun*.
[Libraries]	Name(s) of library files to search for unresolved externals. The library files must have been created by the Librarian utility.
[DS allocation?]	If Yes, the program can allocate memory using the DS segment address and the ExpandAreaSL call.
[Symbol file]	File to which the Linker writes a symbol table of the run file.
[Copyright notice?]	If Yes, the Linker includes a copyright notice in your run file.
[File to append]	If Yes, the Linker appends a file of your choosing to the run file.
[Linker config file]	Linker configuration file that you want to use for this link.

Link V6 and Link V8 Commands Parameter Fields

The parameter fields for all the linking commands are described below.

Object modules

Enter the name of one or more object module files.

You can also include the name of a library file with a list of the object modules to be drawn from it. To do so, enclose the list of object modules in parentheses after the name of the library file. Separate the names with spaces. For example,

A.obj B.obj Ctos.lib (C D)

The Linker combines these object modules to form a run file.

See "Examples," later in this section, for examples of how to fill in this field to list object modules in overlays.

Run file

Enter the name of the run file to be created; for example, *FileName.run*. (If you're using the **Link V8** command, this file name can be that of a dynamic link library (DLL); for example, *FileName.dll*.)

[Map file]

Default: *FileName.map*

Enter the name of the map file.

If you do not want a map file, enter **[Nul]**, or enter **[Vid]** to see any Linker output.

If you leave this field blank, the suffix to the run file name (if any) is dropped and *.map* is added. For example, if the file is *Prog.run*, the default map file is *Prog.map*.

(To interpret the map file, see Section 4, "Reading the Linker Map File.")

[Publics?]

Default: No

Enter **Yes** to direct the Linker to list the relative addresses of all public symbols at the end of the map file. The Linker sorts the publics alphabetically by name and numerically by address.

If you enter **No** or leave the field blank, no public symbols are listed.

(For samples of map files listing public symbols, see Section 4, "Reading the Linker Map File.")

[Line numbers?]

Default: No

If your object modules contain line numbers, enter **Yes** to direct the Linker to add a list containing the address and line number of each source statement following the list of public symbols in the map file.

Not all compilers produce object modules containing line number information.

If you enter **No** or leave this field blank, the map file does not include line numbers and addresses.

[Stack, Dgroup heap size]

Default: See below

You can enter two values in this field separated by a space.

The first value directs the Linker to change the stack size. Enter the number of bytes in decimal format. The stack is composed of words, so this number must be even.

Optionally, you can enter a second value (also in decimal number of bytes) to direct the Linker to adjust the DGroup heap size for the Microsoft C runtime. Separate the first and second values with a space. Entering a second value causes the Linker to create a data segment in DGroup with name, Heap, and class, Heap and assign it the value specified. To force the heap to be as large as possible, specify the keyword MaxVal instead of a byte value. Specifying MaxVal adds enough heap to make DGroup = FFEFh. If you use a heap value and do not enter a stack value, enter it as a pair of single quotation marks.

If you leave this field blank, the default is the compiler's estimate of the correct stack size for each module summed and the Microsoft C compiler's estimate of the DGroup heap segment size, also for each module summed.

You can alternatively adjust the heap size through a parameter in the Linker configuration file or a module definition (*.def*) file.

If the Linker encounters more than one heap segment within the object modules, it creates a heap of the maximum size specified.

For details on estimating stack size, see Section 6, "Advanced Linker Features." For details on adjusting the DGroup heap size using the Linker configuration file, see "Configuring the Linker," later in this section. See Section 11, "Writing a Module Definition File" for details on creating module definition files.

[Max array, data]

Default: 0 0

See the description of *[Min array, data]* below. *[Max data]* is used by **Link V8**. *[Max array]* is not used by **Link V8**.

[Min array, data]

Default: 0 0

Use these fields to specify the amount of short-lived memory the application must allocate for real mode and the amount of short-lived and long-lived memory the application must allocate for protected mode. *[Max array, data]* and *[Min array, data]* are useful for specifying the maximum amount of SL memory within a partition. For virtual memory operating systems, the *[Max array, data]* value specified can exceed the size of the partition.

Each field can contain two values (decimal number of bytes), separated by a space.

A value of 0 for *[Max array, data]* means to use all available memory. If you do not fill in a value for *[Max array, data]*, a warning appears in the map file reminding you that your program, if loaded in a large partition, may force others to be swapped out.

For *[Max array, data]* and *[Min array, data]*, fill in the first parameter in each field to leave data space (the memory array) above the highest memory address of a program.

[Min data] is used by **Link V8**. *[Min array]* is not used by **Link V8**.

For details on the memory array and program memory requirements, see Section 6, "Advanced Linker Features."

[Run file mode]

Default: Protected mode for **Link V8**; real mode for **Link V6** or **Link**.

Enter one of the options shown in Table 3-3. The keyword that you enter directs the Linker to make an entry in the run file header that specifies the type of run file to the operating system loader.

If you enter the option PMOS, Protected, GDTProtected, or LowDataGDTProtected when linking a run file, any run file executes in real mode on a real mode operating system.

The run file mode options are described in Table 3-3. Additionally, there is a special option not presented in the table. This option allows you to conditionally run a program in real or protected mode. For any keyword associated with the protected mode operation, you can also specify one more value, the operating system version number, for example:

```
Link V8
 Object modules                  ______________________
 Run file                        ______________________
  [Map file]                     ______________________
  [Publics?]                     ______________________
  [Line numbers?]                ______________________
  [Stack, Dgroup heap size]      ______________________
  [Max array, data]              ______________________
  [Min array, data]              ______________________
  [Run file mode]                Protected 13.0
  [Version]                      ______________________
  [Libraries]                    ______________________
  [DS allocation?]               ______________________
  [Symbol file]                  ______________________
  [Copyright notice?]            ______________________
  [File to append]               ______________________
  [Linker config file]           ______________________
```

This means the run file is executed in protected mode if the operating system internal version number is 13.0 or higher. Otherwise, it runs in real mode.

Note: *All run file mode options apply to the* ***Link V6*** *command. As shown in Table 3-3, not all the options apply to the* ***Link V8*** *command.*

Table 3-3. Run File Mode Options

Keyword	Applies to Link V8 Command Form?	Description
<blank>	No	Causes the run file to execute in protected mode when using the **Link V8** command and in real mode when using the **Link V6** command.
V4	No	Generates a Version 4 run file when using **Link V6**.
No	No	Causes the run file to execute in real mode when using **Link V6**.
Yes	No	Is equivalent to the SuppressStubs option. See the description of SuppressStubs.
Real	No	Causes the run file to be executed in real mode when using **Link V6**.
Protected	Yes	Indicates that the run file can run in protected mode on a protected mode operating system or in real mode on a real mode operating system. In protected mode, the run file uses a local descriptor table (LDT). When using **Link V8**, this option creates a Version 8 run file. When using **Link V6**, this option creates a Version 6 run file.
GDTProtected	Yes	Indicates the run file uses only the GDT and can run in protected mode on protected mode operating systems.

continued

Table 3-3. Run File Mode Options (cont.)

Keyword	Applies to Link V8 Command Form?	Description
HighMemProtected	Yes	On X-Bus hardware, specify Yes if your application does not communicate with Mode 3 DMA devices or system services that communicate with such devices. Specify No to prevent your application from loading above the 16Mb point on any X-Bus hardware. Not specifying this keyword causes the system to load only the code portion of a run file (not loaded remotely) above 16M bytes.
HighMemGDTProtected	Yes	Provides the same function as HighMemProtected but applies to GDT-based rather than LDT-based programs. See the *Intel x86 Microprocessor Programmer's Reference Manual* for information about LDT and GDT.
LowDataGDTProtected	Yes	Indicates that the run file data should be made accessible to real mode programs. The run file uses the GDT and can run in protected mode. It is used by special operating system services that return pointers to their data (for example, the bit-mapped video service).
NRelProtected	No	Indicates that, unlike most Version 6 run files, this file is to be run only in protected mode. Specifying NRelProtected reduces the size of the run file header by the size of the Relocation Entry Table because this table is not created. Note that if you specify this option, the run file is not backward compatible on real mode operating systems.

continued

Table 3-3. Run File Mode Options (cont.)

Keyword	Applies to Link V8 Command Form?	Description
CodeSharingServer	Yes	Indicates that one copy of system service code is to be shared by multiple instances of the service. This option prevents initialization code from being deallocated for reuse as part of short-lived memory.
HighMemCodeSharingServer	Yes	On X-Bus hardware, specify Yes if your application does not communicate with Mode 3 DMA devices or system services that communicate with such devices. Specify No to prevent your application from loading above the 16Mb point on any X-Bus hardware. If you do not specify this keyword, the system loads only the code portion of a run file (not loaded remotely) above 16M bytes.
SuppressStubs	No	Instructs the Linker not to generate virtual memory management data structures (such as RgStubs) if the object modules list contains overlays. Specifying SuppressStubs reduces the size of the Version 6 run file header. Such a program can only run in protected mode. If the object modules list does not contain overlays or if the operating system on which the program is executed uses paging, this option has no effect.
PMOS	No	Indicates the run file is to be run under the PMOS (protected mode operating system) Service. This option is documented for historic reasons only. If PMOS is installed on your real mode operating system, see your PMOS documentation for details.

[Version]

Default: No

To specify a version in the header of the run file, enter an alphanumeric string. If the version has embedded spaces, enclose the entry in single quotation marks ('). The Linker performs the following two procedures to the version string:

- It adds the prefix *Ver* to your entry and places the version number in the strings table in a Version 8 run file header or in the Version 4 or Version 6 run file header.
- It defines the Public variable *sbVerRun* in the statics segment in DGroup as your string. The first byte of *sbVerRun* contains the string length.

For example if you enter 1.0, the run file header contains 'Ver 1.0', and the statics segment in DGroup contains *sbVerRun* as the number 3 followed by the ASCII characters 1, ., and 0.

You can use the Executive **Version** command to display the version number from the run file header. (See the *CTOS Executive Reference Manual* for details on how to use the **Version** command.)

Note: *If you are linking an operating system, you should specify a version to avoid getting an unresolved external error for the Public variable sbVerRun in the statics segment in DGroup.*

[Libraries]

Default: *Ctos.lib* and *CtosToolKit.lib*

To direct the Linker to search library files in addition to the default libraries, enter the file name. Separate each name with a space. The library files must have been created by the Librarian utility.

The default is to search the standard operating system libraries *Ctos.lib* and *CtosToolKit.lib* in *[Sys]<Sys>* plus the libraries indicated by object modules, such as those in Pascal and FORTRAN. The standard operating system libraries are always the last libraries 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 program 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. (See "Examples" later in this section to see how this is done.)

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 the standard operating system libraries, name the libraries you want searched, followed by the word **None**. (See the example in "Examples" later in this section.) You can also configure the libraries you want searched through a configuration file entry. (See "Configuring the Linker" later in this section.)

If duplicate definitions appear, the Linker defaults to the first definition and emits a "multiple definition of symbol" message in the map file.

[DS allocation?]

Default: Compiler dependent

Enter **Yes** to locate DGroup at the high address end of the 64K-byte segment addressed by the DS register. Under this arrangement, the last byte of DGroup is located at DS:0FFFF. By doing so, memory can be allocated from DGroup using the ExpandAreaSL operation. (This is the main reason for an application to choose this option specifically, so it should be mentioned.)

Enter **No** to locate DGroup at the low address end of the 64K-byte segment addressed by the DS register. When you enter **No**, there is no DS allocation, and DGroup begins at DS:0.

If you leave this field blank, the default is dependent on which compiler generated the object modules. For most compilers, including Microsoft C, the default would be No. For programs linked with any Pascal or Metaware High C™ small or compact model object modules, the default is Yes.

Object module procedures and tasks produced by the Pascal 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.

[Symbol file]

Default: *FileName.sym*

Enter the name of a file to which the Linker writes a symbol table of the run file. The symbol table notes the names and locations of all public symbols within the program.

The suffix to the run file name (if any) is dropped and *.sym* is added. For example, if the file is *Prog.run*, the default symbol file is *Prog.sym*.

If you do not want a symbol file, enter **[Nul]**.

(See the *CTOS Debugger User's Guide* for details on how to use this file when debugging.)

[Copyright notice?]

Default: None

Enter **Yes** to include the copyright string in your run file. The default copyright string is:

```
'Copyright 1993 Unisys Corporation. All Rights
Reserved.'
```

To include your own copyright in the run file, enter a string enclosed in single quotation marks (') in the field.

You can also use the *:Copyright:* entry in the Linker configuration file. Enter a string enclosed in single quotation marks.

[File to append]

Default: None

Enter the name of a file that you would like to append to your run file. This field is typically used to add legal information identifying a run file.

[Linker config file]

Default: *LinkerConfig.sys* in current directory or *[Sys]<Sys>LinkerConfig.sys*

Enter the name of a Linker configuration file to use as the configuration file for this link. If you leave this blank, the default Linker configuration file is used.

See "Configuring the Linker" later in this section for more information.

Examples

The following discussion shows several examples of how you can list object modules and libraries in a Linker command form.

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 command form this way:

```
Link V8
 Object modules                    A.obj B.obj C.obj
 Run file                          Prog.run
 [Map file]                        ______________________
 [Publics?]                        ______________________
 [Line numbers?]                   ______________________
 [Stack, Dgroup heap size]         ______________________
 [Max array, data]                 ______________________
 [Min array, data]                 ______________________
 [Run file mode]                   ______________________
 [Version]                         ______________________
 [Libraries]                       ______________________
 [DS allocation?]                  ______________________
 [Symbol file]                     ______________________
 [Copyright notice?]               ______________________
 [File to append]                  ______________________
 [Linker config file]              ______________________
```

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 (minus the *.obj* suffix) to be extracted. Note that these module names are separated by spaces.

For example, assume that the *Z.lib* library 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:

```
Link V8
 Object modules                  A.obj B.obj Z.lib(W X) C.obj
 Run file                        Prog.run
  [Map file]                     ____________________________
  [Publics?]                     ____________________________
  [Line numbers?]                ____________________________
  [Stack, Dgroup heap size]      ____________________________
  [Max array, data]              ____________________________
  [Min array, data]              ____________________________
  [Run file mode]                ____________________________
  [Version]                      ____________________________
  [Libraries]                    ____________________________
  [DS allocation?]               ____________________________
  [Symbol file]                  ____________________________
  [Copyright notice?]            ____________________________
  [File to append]               ____________________________
  [Linker config file]           ____________________________
```

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

```
Link V8
 Object modules                      A.obj B.obj W.obj X.obj C.obj
 Run file                            Prog.run
 [Map file]                          ____________________________
 [Publics?]                          ____________________________
 [Line numbers?]                     ____________________________
 [Stack, Dgroup heap size]           ____________________________
 [Max array, data]                   ____________________________
 [Min array, data]                   ____________________________
 [Run file mode]                     ____________________________
 [Version]                           ____________________________
 [Libraries]                         ____________________________
 [DS allocation?]                    ____________________________
 [Symbol file]                       ____________________________
 [Copyright notice?]                 ____________________________
 [File to append]                    ____________________________
 [Linker config file]                ____________________________
```

Example: Searching Libraries

This example shows how the Linker searches the libraries specified in the Linker command form. In this example, the Linker searches the libraries *A.lib* and *B.lib*, but no others.

```
Link V8
 Object modules                A.obj
 Run file                      Prog.run
 [Map file]
 [Publics?]
 [Line numbers?]
 [Stack, Dgroup heap size]
 [Max array, data]
 [Min array, data]
 [Run file mode]
 [Version]
 [Libraries]                   A.lib B.lib None
 [DS allocation?]
 [Symbol file]
 [Copyright notice?]
 [File to append]
 [Linker config file]
```

See also "Configuring the Linker" later in this section for a description of how the Linker searches libraries when there are multiple library references in the Linker configuration file.

Example: Using Overlays

The Linker supports virtual code management (overlays) in Version 4 and Version 6 run files. On a virtual memory operating system, all run files are paged into memory on demand and the virtual code (overlay) segments of Version 4 and Version 6 run files are treated the same as the resident segments; they are paged in on demand. The Version 8 run file format does not support virtual code management. For more information on the paging service, see the *CTOS Operating System Concepts Manual.*

Using the *Object modules* field of the Linker command form, you can construct a program containing code in overlays.

To use overlays, list first those modules with code that 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 below.)

1. To construct a run file using overlays, first list in the *Object modules* field those modules that have code you want to place entirely in the resident portion of the program. The command would look like this:

```
Link V8
 Object modules                A.obj
 Run file                      Prog.run
 [Map file]
 [Publics?]
 [Line numbers?]
 [Stack, Dgroup heap size]
 [Max array, data]
 [Min array, data]
 [Run file mode]
 [Version]
 [Libraries]
 [DS allocation?]
 [Symbol file]
 [Copyright notice?]
 [File to append]
 [Linker config file]
```

2. Then continue by listing the first module that has code you want to place in an overlay. Append **/o** to the name of this module. (The **/o** is not case sensitive.) Now the command form would look like this:

```
Link V8
 Object modules                     A.obj B.obj/o
 Run file                           Prog.run
 [Map file]
 [Publics?]
 [Line numbers?]
 [Stack, Dgroup heap size]
 [Max array, data]
 [Min array, data]
 [Run file mode]
 [Version]
 [Libraries]
 [DS allocation?]
 [Symbol file]
 [Copyright notice?]
 [File to append]
 [Linker config file]
```

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

4. 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.

```
Link V8
 Object modules          A.obj B.obj/o C.obj D.obj/o E.obj
 Run file                Prog.run
 [Map file]
  .
  .
  .
 [Linker config file]
```

This run file consists of a resident portion with the code from *A* and a nonresident portion made up of two overlays. One overlay contains the code from *B* and *C*. The second contains the code from *D* and *E*.

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.

Your module list cannot exceed one line. If it does, see the discussion of at-files in the *CTOS Executive Reference Manual.*

Sorting Procedure Names in Overlays

If an overlay contains unsorted procedure names, swapping will probably not work in real mode. Any module whose procedure names are emitted in an order different from the order expected by the virtual code management may cause the program to hang or crash.

The **Sort Public Procedure Names** command was developed as a solution to this situation. This utility reads in an object module or library and determines whether the public names are listed in the order in which they occur in the module. If the public names are not ordered in this way, the utility reorders them. For more information, see the *CTOS Programming Utilities Reference Manual: Installation and Command* Overview.

Configuring the Linker

You can make linking easier by using the Linker configuration file to specify default values. You can also use the Linker configuration file to specify additional parameter data as you would in the Linker command form and to make it easier to enter the names of a group of libraries to search. Example 3-1 below shows the contents of the default Linker configuration file.

The Linker configuration file uses the standard CTOS configuration file format. Each entry has the format:

```
:Keyword:Value
```

The valid keywords are described later in the subsection "Linker Configuration File Parameters." Three special keywords are used to define library references. They are described later in the subsection "Library References."

The default Linker configuration file is *LinkerConfig.sys* and is shown below. The file shown specifies only the standard operating system libraries *Ctos.lib* and *CtosToolKit.lib* and the nationalization library *ENLS.lib*. You can add other libraries of your choice.

Example 3-1. Linker Configuration File

```
***  Parameters with Yes/No values.
:Details:No
:FarCallTranslation:No
:LineNumbers:No
:NonContiguousGroupOK:No
:MultipleDefSymbolsOK:No
:Publics:No
:SuppressWarnings:No

***  Parameters that specify area sizes within the Linker's
virtual memory

***  Used by the Linker to create V4, V6, or V8 run files
MaxRgPdhCG:32
MaxRgRle:127
MaxWorkingData:1024

***  Used by the Linker to create CodeView output file only
MaxCodeViewLines:511
MaxCodeViewCode:32

***  Used by the Linker to create V4 or V6 run files only
MaxRgRqlable:127
MaxRgIdct:32
MaxRgRlePStub:32

***  Used by the Linker to create V8 run files only
MaxV8Strings:64
MaxStringsTOC:16
MaxImportData:16
MaxExportData:16

***  Parameters that specify sizes up to 64Kb
MaxArray:0
MinArray:0

***  Parameters that specify sizes up to double 64Kb
```

Example 3-1. Linker Configuration File (cont.)

```
*** MaxData and MinData specify the amount of short-lived
memory the application will allocate for real mode, the
amount of short-lived and long-lived memory the application
will allocate for protected mode and the amount of virtual
short-lived and long-lived memory the application will
allocate for virtual mode OS.
MaxData:0
MinData:0

***  Miscellaneous parameters
:PackCode:No
:CaseSensitive:Yes/No/MSC

***  Parameters with no Linker defaults;
      parameter values are examples (there is no preceding ":")
DGroupHeapSize:default specified by compilers that created
object files
StackSize:default specified by compilers that create object
files
RunFileMode: Default for V4 or V6 is Real; Default for V8 is
Protected
ClassOrder:default specified in object files
ClassOrder:(MEMORY STACK DATA CONST CODE)
Undefined:Symbol of an external (Void Undefined is default)
Copyright:"Test CopyRight Example" (Void CopyRight is
default)

***  LibraryReference and CharacterCode parameters

Under each "LibraryReference" heading is a list of library
files.  The "Default" reference contains the list of
libraries to be searched when nothing is specified.  The
other lists can be accessed by specifying the name of the
reference on the [Libraries] field of the "Link V6" form as
follows ...

Link V6
Object modules           MyFile.obj
Run file                 MyFile.Run
...
Libraries                (Services)
...
```

Example 3-1. Linker Configuration File (cont.)

```
Note that none of the libraries in the Default list are
searched when another list is specified.

The LibraryReference list also contains the
:CharacterCodeSet: parameter.  Values for CharacterCodeSet
are SingleByte (default), Japan, China, and Korea.

:LibraryReference:Default
:LibraryFile:[sys]<sys>Ctos.lib
:LibraryFile:[sys]<sys>CtosToolKit.lib
:LibraryFile:[sys]<sys>ENLS.lib
:CharacterCodeSet:SingleByte

:LibraryReference:Japanese
:LibraryFile:[sys]<sys>Ctos.lib
:LibraryFile:[sys]<sys>CtosToolKit.lib
:LibraryFile:[sys]<sys>ENLS-J.lib
:CharacterCodeSet:Japan

:LibraryReference:XVT
:LibraryFile:[sys]<sys>Ctos.lib
:LibraryFile:[sys]<sys>CtosToolKit.lib
:LibraryFile:[sys]<sys>ENLS.lib
:CharacterCodeSet:[sys]<sys>XVT.lib

:LibraryReference:Service
:LibraryFile:[sys]<sys>Ctos.lib
:LibraryFile:[sys]<sys>CtosToolKit.lib
:LibraryFile:[sys]<sys>ENLS.lib
:LibraryFile:[sys]<sys>Mouse.lib
:LibraryFile:[sys]<sys>Forms.lib
:LibraryFile:[sys]<sys>Exec.lib
```

Search Path

When you invoke the Linker through the Executive, the Linker uses the following search path to locate the Linker configuration file:

1. The file name specified in the *[Linker config file]* field of the Linker command form.
2. The current user's directory for the file named *LinkerConfig.sys.*

3. A *.user* file *:LinkerConfigFile:* entry. (The entry for *:LinkerConfigFile:* contains the name of a user configuration file.)
4. The directory *[Sys]<Sys>* for the file named *LinkerConfig.sys*.
5. If it cannot find the file, or if it cannot find an entry in *LinkerConfig.sys* for a parameter, it uses the default values in the Linker code. See "Link V6 and Link V8 Commands Parameter Fields" earlier for these values.

Library References

You can use the configuration file to make it easier to complete the *[Libraries]* field of the Linker command form. In the configuration file you can specify a name, called a *library reference*, that identifies a set of libraries to search.

You can also specify the value of the character set bits in the header of the run file. For more information about the character set, see the discussion on the *:CharacterCodeSet:* configuration file entry.

A library search list is called a reference. A reference is defined in the configuration file by a library reference keyword string followed by the name of the reference list:

```
:LibraryReference:ReferenceName
```

Libraries to search are defined in the configuration file by listing them. They appear after the Library Reference definition in the format:

```
:LibraryFile:[Vol]<Dir>LibraryName
```

Character code sets to search can also be defined. To do this, use the format:

```
:CharacterCodeSet:
```

You can then specify the name of the character code set to be searched. (See the example "Japan" in Example 3-2.)

Example 3-2 shows three examples of different library references, excerpted from the Linker configuration file, shown earlier.

Example 3-2. Library Reference Examples

```
:LibraryReference:Default
 :LibraryFile:[Sys]<Sys>Ctos.lib
 :LibraryFile:[Sys]<Sys>CtosToolKit.lib
 :LibraryFile:[Sys]<Sys>ENLS.lib

:LibraryReference:Japanese
 :LibraryFile:[Sys]<Sys>Ctos.lib
 :LibraryFile:[Sys]<Sys>CtosToolKit.lib
 :LibraryFile:[Sys]<Sys>ENLS_J.lib
 :CharacterCodeSet:Japan

:LibraryReference:Services
 :LibraryFile:[Sys]<Sys>Ctos.lib
 :LibraryFile:[Sys]<Sys>CtosToolKit.lib
 :LibraryFile:[Sys]<Sys>ENLS.lib
 :LibraryFile:[Sys]<Sys>XVT.lib
 :LibraryFile:[!Sys]<Sys>Mouse.lib
 :LibraryFile:[!Sys]<Sys>Forms.lib
```

The first library reference shown in Example 3-2 is the default library reference. It consists of the following three libraries: *Ctos.lib, CtosToolKit.lib*, and *ENLS.lib* in *[Sys]<Sys>*. The Linker searches the default library list if you do not specify either an explicit library name or a different reference in the *[Libraries]* field of the Linker command form.

Library Search List

You must include the entry *:LibraryReference:Default* in the Linker configuration file. The Linker automatically searches this list without your having to specify it in the *[Libraries]* field of the Linker command form.

If you want to specify a library reference other than the default in the Linker command form, you must enter the name of the reference, enclosed in parentheses in the *[Libraries]* field. For example, for the Linker to search the libraries listed under the Services library reference shown in Example 3-2, you enter **(Services)** in the *[Libraries]* field as shown below:

```
Link V8
 Object modules            ______________________________
 Run file                  ______________________________
 [Map file]                ______________________________
 [Publics?]                ______________________________
 [Line numbers?]           ______________________________
 [Stack, Dgroup heap size] ______________________________
 [Max array, data]         ______________________________
 [Min array, data]         ______________________________
 [Run file mode]           ______________________________
 [Version]                 ______________________________
 [Libraries]               (Services)____________________
 [DS allocation?]          ______________________________
 [Symbol file]             ______________________________
 [Copyright notice?]       ______________________________
 [File to append]          ______________________________
 [Linker config file]      ______________________________
```

You can specify only one library reference to be searched in the command line at a time. Note that none of the libraries in the default reference list is searched if you specify a different library reference on the command line.

The Linker searches each library list from left to right in the following order:

1. Libraries specified in *Object modules* in the Linker command form.
2. Libraries specified in the *[Libraries]* in the Linker command form.
3. Any language-specific libraries it determines must be read.
4. Libraries in the library reference list in the configuration file.

If no configuration file exists, the Linker reads the *Ctos.lib* and *CtosToolKit.lib* files from the *[Sys]<Sys>* directory and uses the built-in default values for the language libraries.

If the Linker does not find the reference list you specified in a configuration file, it places a warning in the map file, and the link operation continues until completion. The warning message indicates a failure to resolve the external references in the libraries.

Linker Configuration File Parameters

Linker configuration file parameters can appear in any order within the Linker configuration file. These are optional fields. If they are omitted from the configuration file, or, if no values are specified, the Linker uses default values. Some of these values can also be specified in the Linker command form. If a parameter value is specified in both the command form and the configuration file, the command form input takes precedence.

The parameters available for use in the Linker configuration file are as follows:

:CaseSensitive:

Default: No

Enter **Yes** for a case-sensitive link. If you use Microsoft C and want to link with a non-case-sensitive library (such as *Ctos.lib*), enter **MSC.**

:CharacterCodeSet:

Default: None

This parameter controls how a 2-bit flag is set in Version 6 run file headers and how a Word flag is set in Version 8 run file headers. This flag notifies VAM that the run file is expecting a particular video environment.

Enter a character code. You can specify any of the following values:

SingleByte (default)
Japan
Korea
China

If you have *:CharacterCodeSet:China* in your *LinkerConfig.sys* file, the Linker will set the appropriate flags in the resulting run file header. At execution time, these flags will be read by the loader and passed to VAM so that the correct video environment will be established. The reason for this is that Chinese and Japanese run files have certain video and font requirements that VAM handles.

:ClassOrder:

Default: See below

You can use the *:ClassOrder:* parameter to specify preferred segment class ordering.

Here is an example of the values in the *:ClassOrder:* parameter:

```
:ClassOrder: (CODE ENDCODE FAR_DATA FAR_MSG FAR_BSS ??SEG
DATA1 RINGODATA INSTDATA MEDIATORTBL BEGDATA DATA CONST
MSG BSS ENDBSS STACK HEAP MEMORY MEM_DATA)
```

Classes are collected in the order in which they appear in between the parentheses. Segments in other classes discovered in later modules will be located after the classes listed here.

If *:NonContiguousGroupOK:* is No, segments that are part of DGroup are collected together following the order of class names (either from the *:ClassOrder:* parameter or the order within the *.obj* modules).

For Version 8 run files, the ordering is also determined by operating system requirements. The operating system requires the segments to be ordered as follows:

1. Data
2. Nonshared Data
3. Code
4. Nonshared Code

Within each of the four categories, the Linker orders the segments as specified in the *:ClassOrder:* parameter, or as they appear in the *Object modules* parameter in the Linker command form.

:CopyRight:

Default: None

You can place a copyright string in the run file. Here is an example of a copyright string:

```
:CopyRight: "Copyright 1993 All Rights Reserved."
```

If your string has spaces in it, enclose the string inside quotation marks.

In the Linker command form, you can specify whether or not you want the copyright notice to be included in the run file. Unless you enter a copyright string in the configuration file, a default copyright string is included.

:DefaultConfigFile:

Default: See below

You can use a local Linker configuration file to include certain link options for a specific language or program by specifying the file name on the Linker command form. Your local configuration file becomes the primary configuration file. The Linker can still make references to the default configuration file for parameters not configured in the local configuration file. For this to take place, however, you must specify the default configuration file name (secondary configuration file) for the *:DefaultConfigFile:* entry in your local configuration file. The secondary configuration file can specify another default configuration file, but if that configuration file specifies a third, it is ignored.

Note: *The default configuration file can bear any name of your choosing, as long as the Linker can locate it.*

:Details:

Default: No

Enter **Yes** to direct the Linker to enhance the map file to include library reference information and documentation of link parameters such as run file version, stack size, heap size, maximum array, maximum data, minimum array, minimum data, run file mode, version (as specified on the parameter line), whether DS Allocation is used, and so forth. See the discussion in Section 4, "Reading the Linker Map File."

:DGroupHeapSize:

Default: "No DGroup Heap"

To specify the decimal number of bytes in the DGroup heap, enter a decimal integer from 0 to 65,535 (64K). To create a DGroup of 64K bytes, enter the keyword **MaxVal** instead of a number.

DGroup heap size can also be specified on the Linker command form or, for DLLs, in the Module Definition file. See "Link V6 and Link V8 Commands Parameter Fields" earlier in this section and Section 11, "Writing a Module Definition File," respectively, for more information.

:DsAllocation:

Default: No

Enter **Yes** to locate DGroup at the end of the 64K-byte segment addressed by the DS register. If the data segment grows, it overwrites whatever comes before it. You'll get a warning message stating that some segments precede DGroup and that they will be overwritten.

You can also specify DS allocation in the *[DS allocation?]* field of the Linker command form. The command form entry takes precedence over the configuration file entry. See "Link V6 and Link V8 Commands Parameter Fields" earlier in this section.

:FarCallTranslation:

Default: No

Enter **No** to prevent translation of far to near instructions.

Enter **Yes** to direct the Linker to translate far call instructions to near calls and far jump instructions to near jumps if the location called or jumped to is in the same segment. This option is generally used if the *:PackCode:* option is set to Yes or to a decimal integer. The Linker does this by overwriting the following instruction sequence:

```
CallDirectFar to Label
```

with the sequence

```
NOP, PushCS, CallDirectNear to Label
```

In some cases (especially with low-level languages), where code and data are intermixed, the Linker (interpreting it as code) will process it as code. The Linker inadvertently overwrites what appears to be an instruction sequence that it assumes starts with CallDirectFar but in fact it does not.

:FileToAppend:

Default: See below

Enter the name of the file to append to the run file. If you do not include the name of a file, the Linker uses the file that is named in the *[File to append]* field of the Linker command form. The command form entry takes precedence over the configuration file entry. If you leave this field blank, and you leave *[File to append]* in the Linker command form blank, no file is appended.

See "Link V6 and Link V8 Commands Parameter Fields" earlier in this section.

:LibraryReference:

Default: See below

Enter the library reference name you want to use to refer to a group of libraries that you are likely to want to search for a particular link. The library reference name is specified in the *[Libraries]* field of the Linker command form.

See the subsection "Configuring the Linker" for details.

:LibraryFile:

Default: None

Enter the library file name you want to specify in a group of libraries to be searched when a particular library reference name is specified in the *[Libraries]* field of the Linker command form.

See the subsection "Configuring the Linker" for details.

:LineNumbers:

Default: No

Enter **Yes** to direct the Linker to print line numbers from the object file in the map file.

Note that including the entry ***:Details:Yes*** also causes line numbers to be included in the map file.

You can also use the *[Line numbers?]* field in the Linker command form to specify whether or not you want line numbers displayed in the map file. The command form entry takes precedence over the configuration file entry.

:MultipleDefSymbolsOK:

Default: No

Enter **Yes** to flag multiple definitions of symbols as a warning rather than an error.

:NonContiguousGroupOK:

Default: No

Enter **Yes** to prevent the Linker from collecting segments in DGroup. (For details on how the Linker arranges segments into classes, see Section 5, "How the Linker Works.")

:PackCode:

Default: No

Enter **No** to direct the Linker not to combine adjacent code segments.

Enter **Yes** to direct the Linker to combine adjacent code segments into the same physical segment (up to a maximum size of 64K bytes) using the same LDT selector. If adding a code segment would cause a segment to exceed 64K bytes, the Linker starts another code segment and adds subsequent segments to this next segment until either it runs out of code segments to add or it fills the physical segment. Filling the segment repeats the process.

Enter a decimal integer of bytes from 0 to 65535 to combine code segments up to the number of bytes specified.

:Publics:

Default: No

Enter **Yes** to list public symbols on the map file.

You can also specify publics in the *[Publics?]* field of the Linker command form. The command form entry takes precedence over the configuration file entry.

:RunFileMode:

Default: Protected for **Link V8**; real for **Link V6**

Enter the run file mode to the operating system loader. Possible values are Protected and Real.

You can also specify run file mode in the *[Run file mode?]* field of the Linker command form. The command form entry takes precedence over the configuration file entry.

:StackSize:

Default: *Compiler estimate*

This parameter specifies the size of the stack segment. Enter a decimal integer from 0 to 65535.

You can also specify stack size in the *[Stack, DGroup heap size]* field of the Linker command form. The command form entry takes precedence over the configuration file entry.

:SuppressWarnings:

Default: No

Enter **Yes** to suppress all warning information.

:Undefined:

Default: None

Enter a symbol name as an "external."

The *:Undefined:* parameter directs the Linker to enter the specified symbol name as an "unresolved external" before any object modules are opened. An unresolved external name forces the Linker to search for the public reference to resolve an external name. If an unresolved external is uniquely resolved within a library, this feature can be used to force the Linker to extract an object module from a library.

:MaxArray:

Default: 0

This specifies the maximum memory array above the highest memory address of a task. Enter a decimal integer from 0 to 65535.

This parameter is ignored for Version 8 run files.

You can also specify maximum memory array in the *[Max array, data]* field of the Linker command form. The command form entry takes precedence over the configuration file entry.

:MaxData:

Default: 0

This specifies the maximum amount of short-lived memory the application will use. Enter a decimal integer from 0 to 4294967295. If you specify 0, all available memory is used.

For virtual memory operating systems, *:MaxData:* is the amount of virtual short-lived memory. As a result, on a virtual memory operating system, you can specify more memory than is contained in the application's partition.

You can also specify the maximum amount of short-lived memory in the *[Max array, data]* field of the Linker command form. The command form entry takes precedence over the configuration file entry.

:MinArray:

Default: 0

This specifies the minimum memory array above the highest memory address of a task. Enter a decimal integer from 0 to 65535.

This parameter is ignored for Version 8 run files.

You can also specify minimum memory array in the *[Min array, data]* field of the Linker command form. The command form entry takes precedence over the configuration file entry.

:MinData:

Default: 0

This specifies the minimum amount of short-lived memory the application will use. Enter a decimal integer from 0 to 4294967295.

You can also specify minimum amount of short-lived memory in the *[Min array, data]* field of the Linker command form. The command form entry takes precedence over the configuration file entry.

:MaxExportData:

Default: 16

This specifies the number of sectors allocated for internal Linker work area (Version 8 run files only). Enter a decimal integer from 0 to 16383.

:MaxImportData:

Default: 16

This is the number of sectors allocated for an internal Linker work area (Version 8 run files only). Enter a decimal integer from 0 to 16383.

:MaxRgIdct:

Default: 32

This is the number of sectors allocated for an internal Linker work area (Version 4 and Version 6 run files only). Enter a decimal integer from 0 to 16383.

:MaxRgPdhCG:

Default: 32

This is the number of sectors allocated for an internal Linker work area. Enter a decimal integer from 0 to 16383.

:MaxRgRlePStub:

Default: 32

This is the number of sectors allocated for an internal Linker work area (Version 4 and Version 6 run files only). Enter a decimal integer from 0 to 16383.

:MaxRgRle:

Default: 127

This is the number of sectors allocated for an internal Linker work area. Enter a decimal integer from 0 to 16383.

:MaxRgRqlable:

Default: 127

This is the number of sectors allocated for an internal Linker work area (Version 4 and Version 6 run files only). Enter a decimal integer from 0 to 16383.

:MaxStringsTOC:

Default: 16

This is the number of sectors allocated for an internal Linker work area (Version 8 run files only). Enter a decimal integer from 0 to 16383.

:MaxV8Strings:

Default: 64

This is the number of sectors allocated for an internal Linker work area (Version 8 run files only). Enter a decimal integer from 0 to 16383.

:MaxWorkingData:

Default: 1024

This is the number of sectors allocated for an internal Linker work area (Version 8 run files only). Enter a decimal integer from 0 to 16383.

Customizing Virtual Memory Sizes

In order to avoid burdening most links with the disk activity associated with maximizing the size of Linker virtual memory, virtual memory blocks are sized for an average to large run file. If you have a very large run file, you can tailor required virtual memory sizes depending on the size of the link. This affects the performance of the Linker, not the resulting application.

The following Linker configuration file parameters are affected:

```
:MaxRgRle:
:MaxExportData:
:MaxImportData:
:MaxRgldct:
:MaxRgPdhCG:
:MaxRgRlePStub:
:MaxRgRle:
:MaxRgRlelable
:MaxStringsTOC:
:MaxV8Strings
:MaxWorkingData
```

The syntax is:

```
:Maxtablename:<number of 512-byte sectors>
```

For example, to specify 512 * 510 bytes for the RgRle table, the following entry should be made in the Linker configuration file:

```
:MaxRgRle:510
```

The current release default for the table values is reported in the map file if you specify **Yes** to *:Details:* in the Linker configuration file. If the map file reports an overflow (a fatal error), you must increase the maximum value as shown above.

The Linker places sizes of the areas, along with the size used, in the details portion of the map file output.

Section 4
Reading the Linker Map File

Introduction

This section describes how to read the Linker map file. A map file shows how the Linker builds a run file. It includes information such as where segments are located, what their sizes are, if there are any nonshared data segments not in DGroup, and a list of errors that occurred during the link process.

A Simple Map File

The Linker produces a map file containing information about how the Linker builds the run file.

The Linker produces a somewhat simpler map file for a Version 6 run file than it does for a Version 8 run file. However, for either run file you can specify whether or not the map file displays public symbols, line numbers, or details.

If you use the default values for *[Publics?]* and *[Line numbers?]* in the Linker command form and omit the Linker configuration file entry *:Details:*, the Linker produces a simple map file.

The discussion in the paragraphs that follow starts with the map file for a Version 6 run file because it is simpler.

Version 6 Map File

Example 4-1 shows a simple map for a Version 6 run file. It consists of three main components:

- Segment entries describing the size, location, and name of each segment
- The program entry point
- A breakdown of the warnings and errors detected in the link

Addresses

From left to right in Example 4-1, the first three columns show the beginning and ending offset of each segment from the start of the code/data and the length of each segment. The beginning addresses under the column heading "Start" are offsets. 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.

Example 4-1. Sample Version 6 Map File

```
Linker (version)

Run file         : V6>NoDetails.run
Link Start Time  : 01/30/92 13:51:20

Start      Stop       Length            Name          Class

00000000h  00000000h  0000h  (0084h)   ??SEG         ??SEG
00000000h  00000000h  0000h  (008Ch)   MEMORY        MEMORY
00000000h  000007FFh  0800h  (008Ch)   STACK         STACK
00000800h  00000810h  0011h  (008Ch)   DATA          DATA
00000812h  0000081Dh  000Ch  (008Ch)   CONST         CONST
00000820h  00000822h  0003h  (008Ch)   STATICS       CONST
00000830h  0000086Dh  003Eh  (0094h)   SAMPLE_CODE   CODE
00000870h  000008A4h  0035h  (009Ch)   FatalPro      CODE
000008B0h  0000DBE3h  D334h  (00A4h)   MEM_SEGS      MEM_SEGS

Program entry point at 0083:0000 (0094:0000)

No warnings detected
No errors detected
```

Protected Mode Selectors

The fourth column in parentheses after the "Length" column shows protected mode selectors. For each code segment, this selector is the value of the CS register while it is executing, if the program is running in protected mode. For a data segment, this number is the ES, DS, or SS register that would be used to access data within it.

Names

The fifth column in Example 4-1 lists the name of each segment. Note that in the example case of the name SAMPLE_CODE, class Code segment, the name shown is not the file name of the module.

In most high-level language programs, you assign the module name at the beginning of the module. The compiler creates the code segment name by appending the suffix **_CODE** to the assigned module name. The resulting name is what the Linker reports in the map file.

In assembly language, you can directly name each segment as you wish. The Assembler 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, because file names, and not internal module names, are entered in the *Object modules* field of a Linker command form. You are not required to use this convention, however. (See "Examples" in Section 3, "Using the Linker Command Forms" for information about overlays.)

Classes

The sixth (rightmost) column in the map lists the class of each segment. The Linker groups segments by class and uses class to assign order in the program.

Version 8 Map File

A Version 8 map file can show several additional pieces of information not included in the Version 6 map file. In Version 8 map files, if there are any nonshared data segments not in DGroup, these segments are listed as separate entries and can be identified by the string (Nonshared) enclosed in parentheses following the segment Class. (For a description of shared and nonshared segments, see the *CTOS Operating System Concepts Manual.*)

Example 4-2 shows the same segment entries that are shown in Example 4-1 plus one additional entry for a nonshared data segment. (See the last segment entry.)

Example 4-2. Sample Version 8 Map File Showing a Nonshared Segment Entry

```
Linker (version)

Run file        : V8>NoDetails.run
Link Start Time : 11/26/91 07:53:40


Config File     : Config.sys

Start      Stop       Length         Name          Class

00000000h  00000000h  0000h (0084h)  ??SEG         ??SEG
00000000h  00000000h  0000h (008Ch)  MEMORY        MEMORY
00000000h  000007FFh  0800h (008Ch)  STACK         STACK
00000800h  00000810h  0011h (008Ch)  DATA          DATA
00000812h  0000081Dh  000Ch (008Ch)  CONST         CONST
00000820h  00000822h  0003h (008Ch)  STATICS       CONST
00000830h  0000DB63h  D334h (0094h)  MEM_SEGS      MEM_SEGS(NonShared)
0000DB70h  0000DBADh  003Eh (009Ch)  SAMPLE_CODE   CODE
0000DBB0h  0000DBE4h  0035h (00A4h)  FatalPro      CODE


Program entry point at 0083:0000 (0094:0000)

No warnings detected
No errors detected
```

For all segments within a given group, the selector number is the same. (See the *CTOS Programming Utilities Reference Manual: Assembler* for a discussion of groups.) In the example, note that the selector number for the nonshared segment entry (0094h) is different from the selector number for the DGroup segments (008Ch).

Map Files With Public Symbols, Line Numbers, and Details

Examples 4-3 and 4-4 show map files displaying public symbols, line numbers, and other details about parameter information. You can generate this additional information by specifying **Yes** for the Linker fields *[Publics?]* and *[Line numbers?]* and for the configuration file entry *:Details:*.

Version 6 Map File

The map shown in Example 4-3 is similar to the Version 6 map depicted in Example 4-1. In addition it shows all public symbols and their addresses, line numbers, and the details of other command line parameters. Starting at the beginning of the map and proceeding to the end, the following paragraphs examine each of these extra information pieces.

Library References

In Example 4-3, we first see library reference information. This information appears in the map file when you specify **Yes** in the *:Details:* entry in the Linker configuration file. Library information can include

- The library name and block size (in bytes) for each library the Linker searches
- The library version, if available

Example 4-3. Sample Map for a Version 6 Run File Showing Lists of Public Symbols, Line Numbers, and Details (Part 1 of 3)

```
Linker x1/28

Run file        : V6>Sample.run
Link Start Time : 01/30/92 13:51:20


Config File     : TestConfig.sys

Library Reference:(Default) from file TestConfig.sys

  Library: [Sys]<Sys>ENLS.Lib
    Block size: 00512
    Version:    version (day month date, year, time)

  Library: [Sys]<Sys>Ctos.lib
    Block size: 00512
    Version:    version (day month date, year, time)

  Library: [Sys]<Sys>CtosToolKit.lib
    Block size: 00512
    Version:    version (day month date, year, time)
```

Example 4-3. Sample Map for a Version 6 Run File Showing Lists of Public Symbols, Line Numbers, and Details (Part 2 of 3)

```
Start       Stop        Length          Name           Class

00000000h   00000000h   0000h   (0084h) ??SEG          ??SEG
00000000h   00000000h   0000h   (008Ch) MEMORY         MEMORY
00000000h   000007FFh   0800h   (008Ch) STACK          STACK
00000800h   00000810h   0011h   (008Ch) DATA           DATA
00000812h   0000081Dh   000Ch   (008Ch) CONST          CONST
00000820h   00000822h   0003h   (008Ch) STATICS        CONST
00000830h   0000086Dh   003Eh   (0094h) SAMPLE_CODE    CODE
00000870h   000008A4h   0035h   (009Ch) FatalPro       CODE
000008B0h   0000DBE3h   D334h   (00A4h) MEM_SEGS       MEM_SEGS


Publics by name     Address                           Overlay

CHECKERC            00000087:0028h   (009C:0028h)     Res
DELAY               0000FFEF:033Ch   (00BC:800Eh)     CallGate
ErrorExit           0000FFEF:0334h   (00C4:800Ah)     CallGate
EXIT                0000FFEF:0336h   (00B4:800Bh)     CallGate
FatalError          00000087:0000h   (009C:0000h)     Res
fDevelopement       00000000:0810h   (008C:0810h)     Res
MAIN                00000083:000Dh   (0094:000Dh)     Res
PUTFRAMECHARS       0000FFDF:0436h   (00AC:8013h)     CallGate
sbVerRun            00000000:0820h   (008C:0820h)     Res


Publics by value    Address                           Overlay

ErrorExit           0000FFEF:0334h   (00C4:800Ah)     CallGate
PUTFRAMECHARS       0000FFDF:0436h   (00AC:8013h)     CallGate
EXIT                0000FFEF:0336h   (00B4:800Bh)     CallGate
DELAY               0000FFEF:033Ch   (00BC:800Eh)     CallGate
fDevelopement       00000000:0810h   (008C:0810h)     Res
sbVerRun            00000000:0820h   (008C:0820h)     Res
MAIN                00000083:000Dh   (0094:000Dh)     Res
FatalError          00000087:0000h   (009C:0000h)     Res
CHECKERC            00000087:0028h   (009C:0028h)     Res
```

Example 4-3. Sample Map for a Version 6 Run File Showing Lists of Public Symbols, Line Numbers, and Details (Part 3 of 3)

```
Line numbers for SAMPLE_CODE

00002  0083:0000H   00005  0083:0000H   00008  0083:0000H
00010  0083:0000H   00013  0083:0000H
00015  0083:000DH   00016  0083:0010H   00017  0083:002DH
00018  0083:003CH   00019  0083:0000H
00020  0083:0008H   00021  0083:000DH
Program entry point at 0083:0000 (0094:0000)

Linker Details

Linker Information:

Run file          : V6>Sample.run
Run file format : Version 6
Stack             : 02048
Run file mode     : 2004h; Protected
CharacterCodeSet: SingleByte

Version           : xC
Max array = 00000             Max SL Memory = 0000000000
Min array = 00000             Min SL Memory = 0000000000
DS Allocation not used(default)

Class ordering requested       :
          ??SEG  MEMORY  STACK  DATA  CONST  CODE

Configurable Linker work areas:
Area Name          Sectors Used    Max Sectors
RgIdct                00000           00032 (default)
RgPdhCG               00001           00032 (default)
RgRlePStub            00000           00032 (default)
RgRle                 00001           00127 (default)
WorkingData           00044           01024 (default)
RgRqLable             00001           00127 (default)

Link End Time    : 01/30/92 13:51:28

No warnings detected
No errors detected
```

Public Symbols

Following the segment entries the public symbols are first sorted alphabetically and then by location in the run file. To request that public symbols be displayed, you enter **Yes** in the *[Publics?]* field in the Linker command form.

In the list of public symbols in Example 4-3, the name of the public symbol is followed by two addresses. The first is the address in real mode, and the second is the address in protected mode.

The *Overlay* column contains *Res* if the symbol is in the resident portion of the program, an integer (*n*) if it is in the *n*th overlay, and *callgate* if it is a call gate to an operating system procedure.

Line Numbers

The public symbol lists are followed by a list of line numbers. To request line numbers separately, you enter **Yes** in the *[Line Numbers?]* field in the Linker command form.

Line numbers are intended for use during debugging. They allow you to examine a known part of a 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 to use them.

Line numbers are not always available from all modules.

Command Form Parameter Details

Near the end of the map file in Example 4-3, you see other details about Linker command form parameters. (See the portion of the map entitled "Linker Information." It is located after the program entry point line.) These details are displayed along with the library reference information when you specify **Yes** in the *:Details:* entry in the Linker configuration file.

Command form parameter details can include the following information:

Run file header format
Stack size
Heap size
Maximum array
Maximum data
Minimum array
Minimum data
Run file mode
Version (as specified on the command line)
Whether DS Allocation is used

Configurable Linker Work Areas

Just before the error and warning messages at the very end of the map file, is the portion of the map called "Configurable Linker work areas". This portion contains information about the Max Table parameters that you can define in the Linker configuration file.

Version 8 Map File

Example 4-4 shows a map for a Version 8 run file. The map includes all public symbols and addresses, line numbers, and other link details. The library references and Linker command form references are the same as those previously described in Example 4-3. (See the text describing library references and parameter details shown in Example 4-3. Those descriptions also apply to Example 4-4.)

In addition, when you specify **Yes** in the *:Details:* entry in the Linker configuration file, a Version 8 map file shows imported and exported publics by name. In Example 4-4 below, you can see the imported and exported publics displayed in the Linker Details portion of the map file. Note that under the imported publics section, both the name of the imported public and the dynamic link library (DLL) it is found in are displayed. (In this example, the DLL name is "user", and the procedure name is "MyImport".)

Example 4-4. Sample Map for a Version 8 Run File Showing Public Symbols, Line Numbers, and Details (Part 1 of 3)

```
Linker x1/28

Run file         : V8>Sample.run
Link Start Time  : 01/30/92 13:51:07


Config File      : TestConfig.sys

Library Reference:(Default) from file TestConfig.sys

  Library: [Sys]<Sys>ENLS.Lib
    Block size: 00512
    Version:    version (day month date, year, time)

  Library: [Sys]<Sys>Ctos.lib
    Block size: 00512
    Version:    version (day month date, year, time)

  Library: [Sys]<Sys>CtosToolKit.lib
    Block size: 00512
    Version:    version (day month date, year, time)

Start       Stop        Length         Name        Class

00000000h   00000000h   0000h  (0084h) ??SEG       ??SEG
00000000h   00000000h   0000h  (008Ch) MEMORY      MEMORY
00000000h   000007FFh   0800h  (008Ch) STACK       STACK
00000800h   00000810h   0011h  (008Ch) DATA        DATA
00000812h   0000081Dh   000Ch  (008Ch) CONST       CONST
00000820h   00000822h   0003h  (008Ch) STATICS     CONST
00000830h   0000DB63h   D334h  (0094h) MEM_SEGS    MEM_SEGS
                                                    (NonShared)
0000DB70h   0000DBADh   003Eh  (009Ch) SAMPLE_CODE CODE
0000DBB0h   0000DBE4h   0035h  (00A4h) FatalPro      CODE
```

Example 4-4. Sample Map for a Version 8 Run File Showing Public Symbols, Line Numbers, and Details (Part 2 of 3)

```
Publics by name      Address                          Overlay

CHECKERC             00000DBB:0028h    (00A4:0028h)   Res
DELAY                0000FFEF:033Ch    (00BC:800Eh)   CallGate
ErrorExit            0000FFEF:0334h    (00C4:800Ah)   CallGate
EXIT                 0000FFEF:0336h    (00B4:800Bh)   CallGate
FatalError           00000DBB:0000h    (00A4:0000h)   Res
fDevelopement        00000000:0810h    (008C:0810h)   Res
MAIN                 00000DB7:000Dh    (009C:000Dh)   Res
MyImport             00000000:0000h    (0000:0000h)   Imp
PUTFRAMECHARS        0000FFDF:0436h    (00AC:8013h)   CallGate
sbVerRun             00000000:0820h    (008C:0820h)   Res

Publics by value     Address                          Overlay

MyImport             00000000:0000h    (0000:0000h)   Imp
ErrorExit            0000FFEF:0334h    (00C4:800Ah)   CallGate
PUTFRAMECHARS        0000FFDF:0436h    (00AC:8013h)   CallGate
EXIT                 0000FFEF:0336h    (00B4:800Bh)   CallGate
DELAY                0000FFEF:033Ch    (00BC:800Eh)   CallGate
fDevelopement        00000000:0810h    (008C:0810h)   Res
sbVerRun             00000000:0820h    (008C:0820h)   Res
MAIN                 00000DB7:000Dh    (009C:000Dh)   Res
FatalError           00000DBB:0000h    (00A4:0000h)   Res
CHECKERC             00000DBB:0028h    (00A4:0028h)   Res

Line numbers for SAMPLE_CODE

00002  0DB7:0000H   00005  0DB7:0000H   00008  0DB7:0000H
00010  0DB7:0000H   00013  0DB7:0000H
00015  0DB7:000DH   00016  0DB7:0010H   00017  0DB7:002DH
00018  0DB7:003CH   00019  0DB7:0000H
00020  0DB7:0008H   00021  0DB7:000DH
Program entry point at 0DB7:0000 (009C:0000)

Linker Details

Exported publics by name        Exported name

MAIN                            MAIN
```

Example 4-4. Sample Map for a Version 8 Run File Showing Public Symbols, Line Numbers, and Details (Part 3 of 3)

```
Imported publics by name          Dll

MyImport                          User.MyImport

Linker Information:

Run file         : V8>Sample.run
Run file format  : Version 8
Stack            : 02048
Run file mode    : 2004h; Protected  (default)
CharacterCodeSet: SingleByte

Version          : xC
ax array = 00000            Max SL Memory = 0000000000
Min array = 00000            Min SL Memory = 0000000000
DS Allocation not used(default)

Class ordering requested        :
          ??SEG  MEMORY  STACK  DATA  CONST  CODE

Configurable Linker work areas:
Area Name        Sectors Used     Max Sectors
ExportData          00001            00016 (default)
rtData              00001            00016 (default)
RgPdhCG             00001            00032 (default)
RgRle               00001            00127 (default)
WorkingData         00044            01024 (default)
StringsTOC          00001            00016 (default)
V8Strings           00001            00064 (default)

Link End Time    : 01/30/92 13:51:16

No warnings detected
No errors detected
```

Section 5
How the Linker Works

Introduction

This section includes a description of how the Linker organizes a run file image and associates procedure references in the run file code to the actual procedures called.

Linking can be dynamic or static. In either case, the way the Linker searches for external references is the same. What actually happens to the definition of the reference when the Linker finds it determines whether the link is dynamic or static.

This section describes the Linker search algorithm. Then dynamic and static linking are examined. Following this description, the focus is on how the Linker organizes the contents of a run file.

A dynamic link library (DLL) can be a client program. The term "client program" here means any program that calls a DLL, even if it is a DLL itself.

Linking Overview: A Two-Pass Process

The Linker makes two passes through the modules being linked. What happens during these passes is described below.

Pass One

On the first pass, the Linker reads all object modules, extracting segment, group, external symbol, and public symbol information, and it builds an identifier information table in the Linker virtual memory. The Linker examines this 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.

Library Search Algorithm

Having built an identifier 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 can 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, steps to the third library, and so on for subsequent libraries.

When the Linker has completed the search of the last library, it may have extracted further undefined symbols from 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 there is a public symbol in more than one library, you have no way of knowing which library the object module containing it is drawn from. 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 proceeds to the next library and extracts the first definition it encounters.*

Dynamic and Static Linking

During its search, the Linker finds a public symbol defined either in the code and data of an object file, or as an import reference to a dynamic link library (DLL).

If the Linker finds a conventional module, it performs a static link. In a static link, the Linker copies the procedure code and data directly into the application run file.

If, on the other hand, the Linker finds a reference to an import entry, it sets up the run file so that it can perform a dynamic link at run time.

Dynamic linking is supported only by Version 8 run files executing on virtual memory operating systems.

A dynamic link import entry is not the actual procedure or data; it's simply a reference that indicates the name of the DLL where the procedure or data is stored and the location of the import in the DLL. The Linker writes import information in place of the address. Also, a pointer to this location in the code is written to the Relocation Entries table in the application run file header. (For details on run file format, see Appendix B, "Run File Reference.")

In a dynamic link, no procedure code or data is copied to the application run file. At load time the loader "fixes up" the memory address in the procedure call in the code being loaded to point to the referenced import so that execution can occur.

A static link is fairly straightforward. The run file already contains a copy of the procedure with addresses adjusted to point to the procedure.

For details on CTOS DLLs, see "Dynamic Link Libraries" in the *CTOS Operating System Concepts Manual.*

At the end of the first pass, the Linker reorganizes the segment order and allocates blocks of virtual memory for each segment.

Pass Two

On the second pass, the Linker reads data from the object modules. It assigns relative addresses, relocating as necessary, to all data in all the modules and then writes the fixed up data to the block of virtual memory for the segment in which the data is located. At the end of this step, the run file header is generated and the file is assembled by appending tables and segment data in the proper order.

From Source Modules to Run File on Disk

The Linker combines object modules into a run file in 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. It also contains an array of pointers that allow the operating system to relocate the run file to any appropriate memory location.

A run file produced by the Linker can thus be used with various memory configurations or as one of several run files in a multitasking program.

Arranging Object Module Components

The Linker accepts object modules, separates them 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 run file image in memory.

The rules that the Linker uses to order the contents of object modules are easiest to understand if we give names to the components and collections of components involved and build a model like that shown in Figure 5-1. For simplicity the model we use only approximates what the Linker actually does, but it is sufficient to convey how the Linker works. (Note that this example is not intended to correspond to the map examples in Section 4, "Reading the Linker Map File.")

Segment Element Names and Classes

The example in Figure 5-1 shows three object modules to be linked. They have been listed in the *Object modules* field of the Linker command form as follows:

```
Link V8
 Object modules                Mod1.obj Mod2.obj Mod3.obj
 Run file                      Prog.run
 [Map file]                    ____________________________
 [Publics?]                    ____________________________
 [Line numbers?]               ____________________________
 [Stack, Dgroup heap size]     ____________________________
 [Max array, data]             ____________________________
 [Min array, data]             ____________________________
 [Run file mode]               ____________________________
 [Version]                     ____________________________
 [Libraries]                   YourCCompiler.lib
 [DS allocation?]              ____________________________
 [Symbol file]                 ____________________________
 [Copyright notice?]           ____________________________
 [File to append]              ____________________________
 [Linker config file]          ____________________________
```

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

Each of these object modules includes several *segment elements*, each of which has been declared PUBLIC. It so happens that all these object modules have segment elements containing code, data, constants, and stack (although this is not always the case).

Each segment element in each module has both a name and a class. In high-level languages, the compiler assigns the name and class. In the example in Figure 5-1, the name and class of each segment are separated by a slash. For example, the following segment element has the name *Mod1_code* and is of class *Code*:

```
Mod1_code/Code
```

This nomenclature can be confusing because many compilers assign names to segment elements that are the same as the classes of the elements. For example, the following segment within DGroup has the name *Heap* and is of class *Heap*:

```
Heap/Heap
```

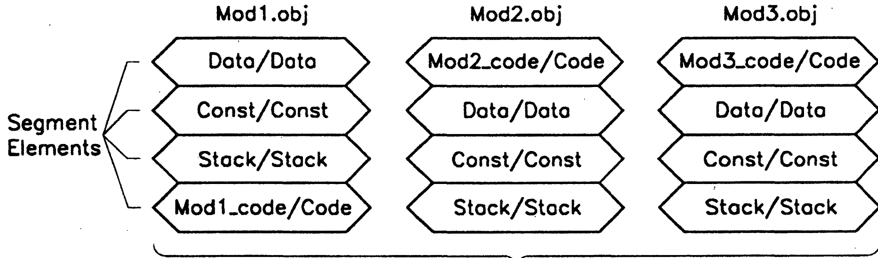
Usually the code segment element carries the name of the module: in *Mod1.obj,* the Mod1 segment element is of class Code. Some compilers append the class name as part of the code segment element name, which in this case results in Mod1_code.

Note: *The Linker treats any segment with a class name that ends in the string 'Code' as a code segment.*

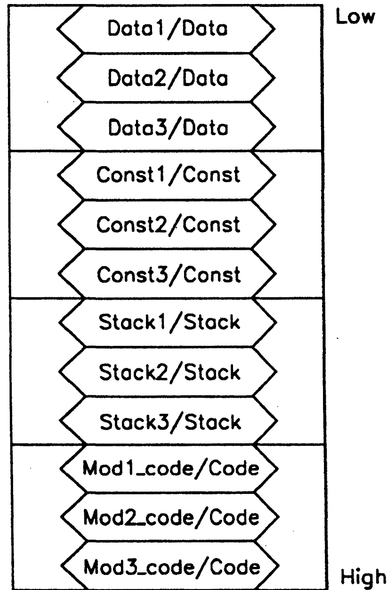
The most common classes are *Code*, *Data*, *Const*, and *Stack*. A given compiler always arranges the segment elements by class in a fixed order.

Figure 5-1. How the Linker Builds a Run File (Part 1 of 2)

Step 1.
Input Object Modules

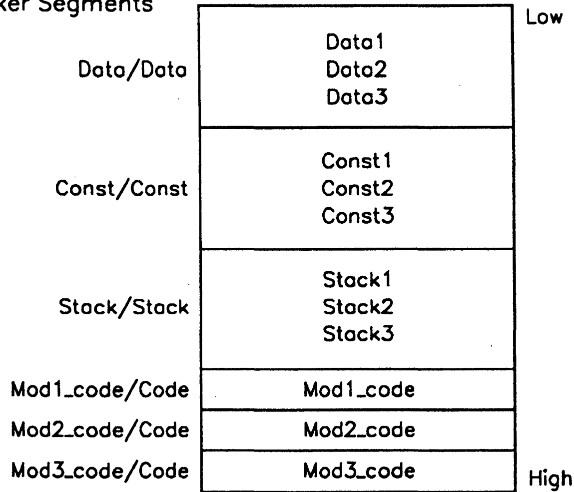


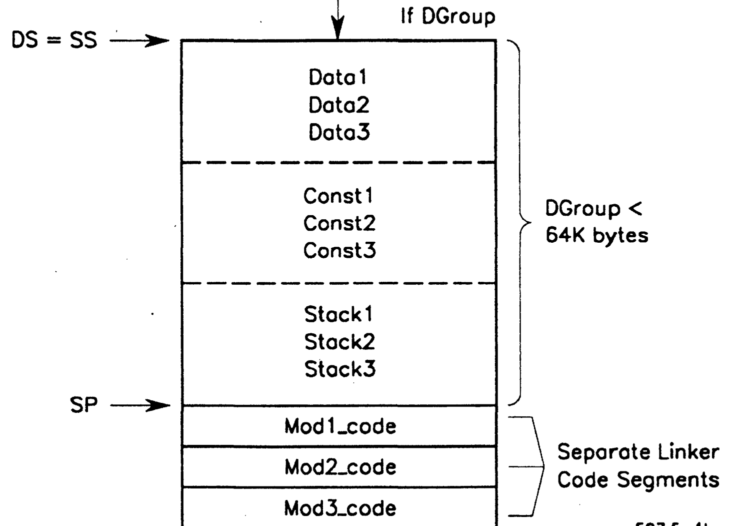
Step 2.
Look at Mod1 for Order Sort



597.5–1a

Figure 5-1. How the Linker Builds a Run File (Part 2 of 2)





Note: *The example in Figure 5-1 assumes that all data is in DGroup. However, this may not always be the case. Figure 5-3, later in this section, shows an example of how data falls outside of DGroup.*

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.

In the example, the segment elements in *Mod1.obj* are in a different order from those of the other two object modules. Apparently, the three object modules were produced by two different compilers that do not order segment elements in the same way, or an assembly language programmer wrote one or more of these modules and chose the order of segment elements within them. The order of classes in the first module is used as the order of classes in the run file.

Creating Linker Segments

In the first pass, at the same time the Linker is resolving all external references in the modules, it builds a table of classes and segment elements. Starting with the first module listed (in the example in Figure 5-1, *Mod1.obj*), it takes the first segment element in that module, examines 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.

As a result, a category is created for each class. In the example, four categories are created that are arranged in the same order as the four segment element classes in *Mod1.obj*: data, constants, stack, and code.

Having divided *Mod1.obj* in this way, the Linker proceeds to *Mod2.obj*. It takes each segment element in *Mod2.obj*, examines its class, and places it in the Linker category already created for that class. If there is no Linker category for the class, the Linker creates one and places it at the end of the list of Linker categories.

By sorting all the segment elements of the three modules into their respective categories, the Linker creates the four Linker segments shown in Step 2 of Figure 5-1. As can be seen, each category is a new Linker segment.

Specifying Linker Segment Order

Linker segment elements 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 and classes in the desired order. Place this module first in the list of modules to be linked. This template object module is often called *First.obj*.

The segment name and segment class are reported on the map file generated by the Linker. If segments appear out of order on the map file, you can correct the order with a *First.asm* module that uses the same segment name and segment class.

Combination Rules

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

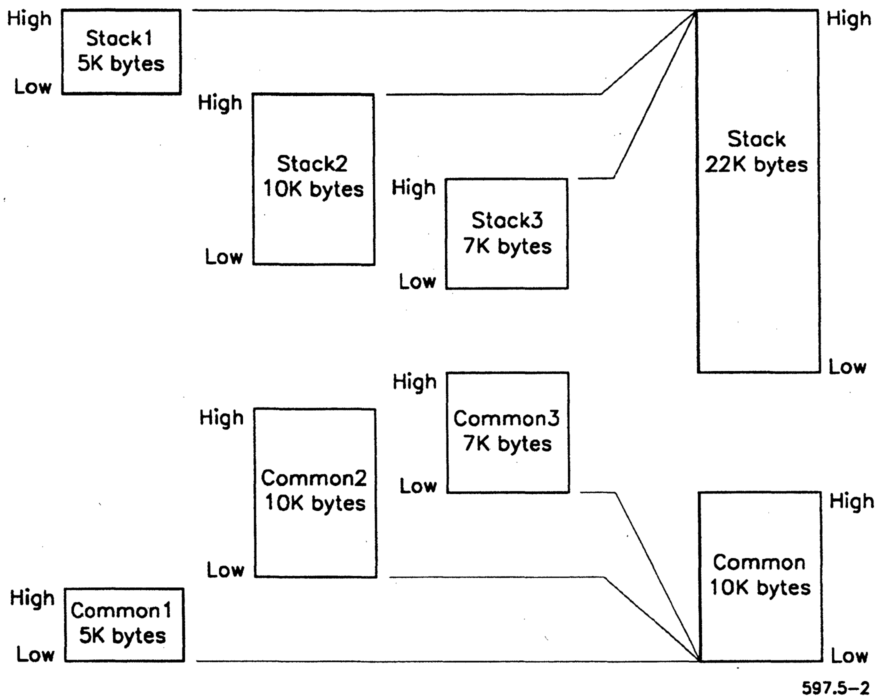
In most cases, the Linker appends one segment element to another as it goes through the modules and does not distinguish boundaries between a segment element from one module and that from the next module. This is true of data and constant segment elements.

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 the upper illustration in Figure 5-2. 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 *CTOS Programming Utilities Reference Manual: Assembler* for details.)

In the case of heap segment elements, 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 the lower illustration in Figure 5-2.

Figure 5-2. Combination of Stack and Common Segment Elements



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 5-1 shows the Linker segments created in this example.

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 memory model used by *Ctos.lib* uses DGroup. In assembly language you can define whatever groups you want, or none.) For DGroup, the hardware segment register is DS.

In protected mode, 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 5-1 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 be equal to the highest address in the group, as shown in the figure.

Groups are described in the *CTOS Programming Utilities Reference Manual: Assembler*. Models of segmentation are covered in *CTOS / Open Programming Practices and Standards.*

Placing Uninitialized (Communal) Variables in DGroup

Usually the Linker places uninitialized variables in one segment, DGroup, that is addressed using the DS register. Uninitialized variables in Microsoft C (called communal variables), however, are an exception. They may or may not be placed in DGroup, depending on the memory model in which they are declared. Segments of class _BSS (small and medium model) are placed in DGroup, but segments of class Huge_BSS or FAR_BSS (huge, compact, or large memory model) are not. You can use the map file to identify segment classes.

For details on interpreting the map file, see Section 4, "Reading the Linker Map File." See the *MS-DOS Encyclopedia* for details on the structure of a Microsoft C program containing communal variables.

Alignment Attributes

Segment elements have alignment attributes. Most compiled languages assign these attributes automatically, but in assembly language, you must assign them explicitly. (See the *CTOS Programming Utilities Reference Manual: Assembler* for details.)

A segment can have one of several alignment attributes:

- Byte (a segment that can start at any address)
- Word (a segment that can start only at an address that is a multiple of 2)
- Paragraph (a segment that can start only at an address that is a multiple of 16)
- Page (a segment starting at an address that is a multiple of 512)

The Linker forces segments that have their own selectors to be paragraph aligned.

Note that segments in a group still follow alignment attributes.

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.

Figure 5-1, shown earlier in this section, assumes all data is located in DGroup. Since that may not always be the case, the example shown below depicts a slightly different arrangement, where all data is not in DGroup.

Figure 5-3. How the Linker Builds a Run File (Not all Data in DGroup) (Part 1 of 2)

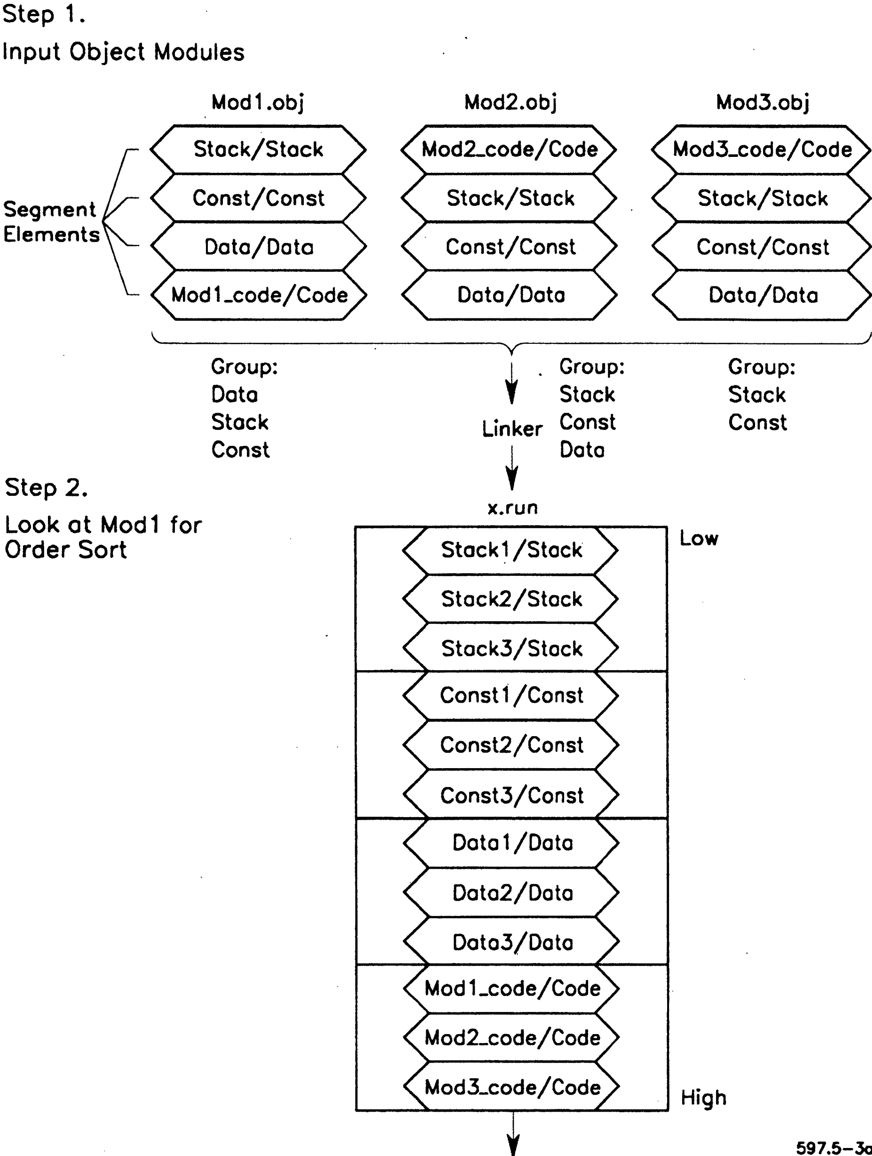
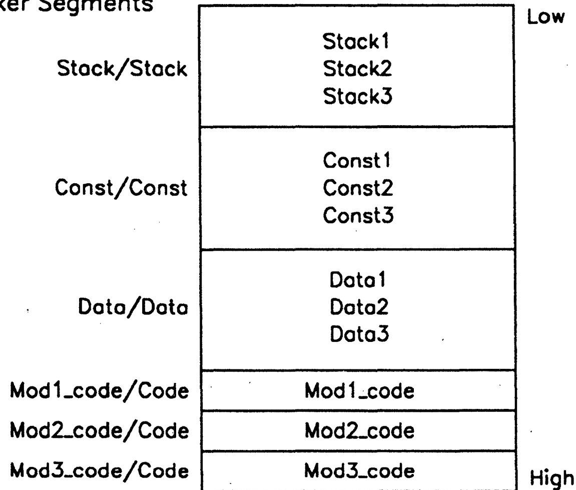
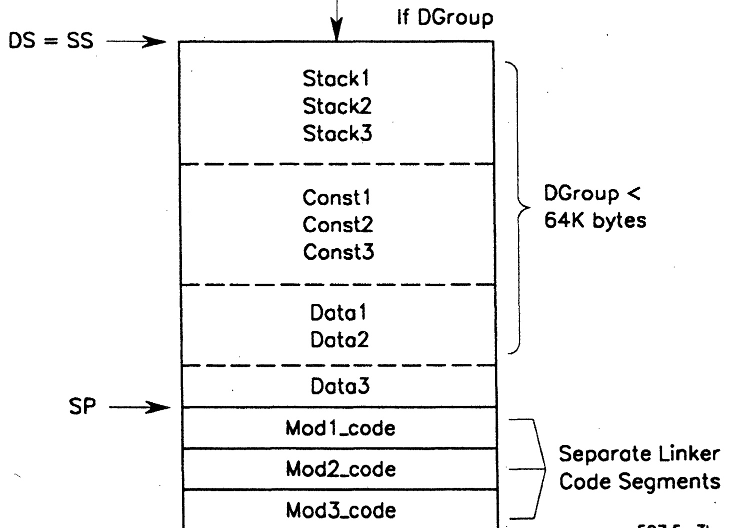


Figure 5-3. How the Linker Builds a Run File (Not all Data in DGroup) (Part 2 of 2)





Summary of Segment Ordering

All segment elements of the first class observed 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 an offset from a single value in a segment register. In Version 6 and Version 8 run files, all segments in a group are referenced by one selector and must be contiguous.

Segment 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 in the partition in which the link occurs and the number of public symbols in the program.

Section 6
Advanced Linker Features

Introduction

This section discusses estimating memory requirements, adjusting stack size, allocating memory space, using overlays, and customizing segment ordering.

Program Memory Requirements

The address space required by a program consists of the resident size of the program plus memory allocated by the program. The operating system creates the address space based on 1) the resident size of the program, 2) the data portion of the *[Min array, data]* and *[Max array, data]* fields in the Linker command form, and 3) the partition size and type.

The data portion of the *[Min array, data]* and *[Max array, data]* fields of each run file describe the minimum and maximum dynamic (allocated long-lived and short-lived memory) memory required by the program. These fields do not include the size of the resident portion of the program. Programs that contain minimum data and maximum data of 0 are said to be *unsized*.

The CTOS operating system supports two application partition types: *fixed* (type 2; see CreatePartition in the *CTOS Procedural Interface Reference Manual*) and *variable* (type 6). Fixed partitions are provided to offer a backward-compatible address space model with prior versions of CTOS. Variable partitions are partitions in which the address space can be modified to meet the needs of the currently loaded program.

For the purposes of this discussion, the components are named as follows:

R Resident size of program
m Minimum data from run file
M Maximum data from run file
P Partition size (from *CmConfig.sys* or the CreatePartition request)

For unsized programs loaded in a fixed partition:

The resulting address space = P

For unsized programs loaded in a variable partition:

The resulting address space = P

For sized programs loaded in a fixed partition:

The resulting address space = P with the constraint that R+m < P

For sized programs loaded in a variable partition in CTOS II:

The resulting address space = Min (R+M, P) with the constraint that R+m < P

For sized programs loaded in a variable partition in CTOS III:

The resulting address space = R+M with the constraint that R+m < P

For example, consider a program with a resident size of 200K bytes, minimum data of 50K bytes, and maximum data of 300K bytes:

R 200
m 50
M 300

The value for the resulting address space is calculated for a fixed partition size as follows:

- If P = 200, then program load is rejected, and status code 400 (insufficient memory) is returned
- If P = 400, the resulting address space is 400
- If P = 600, the resulting address space is 600

The value for the resulting address space is calculated for a variable partition size as follows:

- If P = 200, then program load is rejected, and status code 400 (insufficient memory) is returned
- If P = 400, for CTOS II the resulting address space is 400
- If P = 400, for CTOS III the resulting address space is 500
- If P = 600, the resulting address space is 500

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 the standard operating system libraries. 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 segments from the run-time library are listed 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 listed in the map. This number is the hexadecimal count in bytes from the first byte of the first segment.

Overlay Programs

Overlay 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.

The Linker supports virtual memory management (overlays) in Version 4 and Version 6 run files. On a virtual memory operating system, the virtual code (overlay) segments of Version 4 and Version 6 run files are treated the same as the resident segments; they are paged in on demand. The Version 8 run file format does not support virtual memory management because it is provided for all run files by CTOS III (and later) on which Version 8 run files execute. (For more information on the paging service, see the *CTOS Operating System Concepts Manual.*)

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 the memory array is subject to the availability of a minimal amount. (See "Allocating Memory Space" later in this section for details.)

Adjusting Stack Size

All compilers produce information in object modules about the amount of stack needed to execute the code in the module (assuming that there is no recursion). The Linker can compute the size of the required stack segment by adding these amounts together. 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. (For details, see Section 4, "Reading the Linker 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. See the *CTOS Debugger User's Guide* for more information.

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 program 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 successfully, reduce the stack size according to the guidelines in "Reducing the Stack."

Allocating Memory Space

A program can specify memory that it will need during execution, and the compiler will emit records that will include this amount of memory in the run file. This extra memory occupies 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 by calling AllocMemorySL or ExpandAreaSL, and the memory is allocated toward lower addresses. This memory is addressed with segment-and-offset addresses.

With the Linker, you can 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 (In the Linker configuration file, these are *:DSAllocation:*, *:MinArray:*, and *:MaxArray:*. In the Linker command form, these are *[DSAllocation?]*, *[Min array, data]*, and *[Max array, data]*).

Under the DS allocation option, your program can allocate some short-lived memory toward lower addresses as usual, but can address it efficiently as if it were part of DGroup with only 16-bit offset addresses.

Under the memory array option, the program can allocate memory at high addresses above the program.

Note: *The memory array is supported for backwards compatibility but is not recommended.*

DS Allocation

DS allocation allows your program to allocate some short-lived memory toward lower addresses as usual, but also allows it to address the memory efficiently with *near pointers* (16-bit offset addresses) relative to DGroup. This is accomplished by expanding the data segment to its maximum size as the means of providing run-time memory. The data segment (addressed by DS) has a maximum size of 64K bytes, and your program uses a certain amount of that, but the rest can be used for short lived memory.

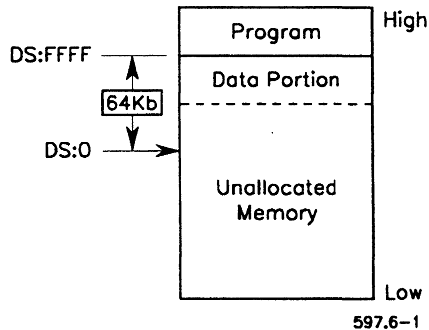
DS allocation allows you to define a maximum-size data segment, even though your program's data segment would normally be smaller. The excess space in this maximum data segment extends beyond your program toward lower memory addresses. You allocate memory in this space with AllocMemorySL or ExpandAreaSL. AllocMemorySL creates a new segment and returns the segment's selector and an offset of zero.

To achieve this, 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). The Linker arranges the data in DGroup so that the last byte of it is at offset 0FFFFh from DS (see Figure 6-1).

Under this arrangement, the first byte of program data starts at an offset x, where x is a value greater than 0. DGroup changes from the usual expand-up segment type to an expand-down type. The excess DGroup space between offset 0 and offset x-1 can be used by the program to allocate short-lived memory using ExpandAreaSL.

Note that the program usually is arranged with the data segment as its first or lowest-address segment. In real mode, many segments that precede DGroup are in danger of being overwritten as DGroup is expanded downwards. The Linker automatically places DGroup in its correct position when DS allocation is requested. 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.

Figure 6-1. Real Mode Program With DS Allocation

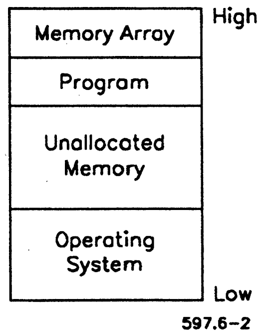


DS allocation has several advantages. It allows 16-bit DS-relative addressing. In addition, memory allocated within this space adjoins the common pool of available memory below the program, and it can be deallocated and reallocated flexibly by the program. However, the program must make procedure calls 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, in the Link V6 command form, you specify values in the first parameters of the *[Max array, data]* and *[Min array, data]* fields. (The *[Max array, data]* and *[Min array, data]* fields have no effect for the **Link V8** command.) Figure 6-2 shows the arrangement of your program with the memory array at the high-end address.

Figure 6-2. A Program With the Memory Array



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 program always to load at the lowest possible address, that is, with maximum memory array at the end of the program, set the minimum to 0 and the maximum to 0 to take all allowable memory.

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 program 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.

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 be referenced from DS. Usually, the ES register is loaded with the lowest address of the memory array.

Linking a Program With Overlays

Note: *On virtual memory operating systems, the paging service is used to swap portions of a Version 6 or Version 8 run file into or out of memory as needed. Calls to virtual memory management (overlay procedures) in these cases are ignored. Overlays, however, are supported in all Version 4 run files.*

The virtual memory management facility 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, it is brought into memory.

The term *code segment*, as used here, does not mean the module's code segment. An overlay, for example, can include differently named code segments originating from several different modules.

The overlay management scheme does not provide for saving changes made in overlays once they are in memory. Nothing is written back to disk, so there is no need for a disk swap file. Code segments produced by high-level language compilers are completely relocatable and read-only, so a particular overlay code segment in memory that is no longer needed can be overwritten by another code segment. When the first code segment is needed again, it is reread from the run file.

Virtual memory management 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 an overlay program. You must write a small amount of overlay initialization code, and you must specify in the Linker command form which modules are to contribute code to which overlays. (See "Examples" in Section 3, "Using the Linker Command Forms," for details on what to enter in the *Object modules* parameter field.)

The virtual memory management model is more fully described in the *CTOS Operating System Concepts Manual*. For details on how to write an overlay program or how to adapt an existing one, see *CTOS / Open Programming Practices and Standards Draft 1.0.* Also see your programming language manual.

The following cannot be put in overlays if the program is to be run in real mode:

- Modules in which publics are not defined in the order in which they occur in those modules
- Modules with zero length code segments
- In some languages, you cannot place certain modules from the run-time library in overlays. This is the case when publics are not defined in the order in which they occur in modules.
- Assembly language modules where the procedures do not follow call/return conventions and certain other rules

Small-model C language programs are not compatible with virtual memory management.

Customizing Segment Ordering

As stated previously, Linker segment elements are ordered by class in the same order in which the Linker encounters them. Therefore, the order of classes and segments in the first object module has a great influence on how the run file segments are arranged. You can customize segment ordering by

- Writing a *First.asm* file, generating a *First.obj* file from it and putting that file first in the object modules list

  or

- Using *:ClassOrder:* in the configuration file

First.asm File

You can influence the way the Linker arranges the run file segments by writing a special assembly language program and generating an object module from it. This object module is placed first in the list of modules to be linked. This template object module often is called *First.obj*.

The segment name and segment class are reported on the map file generated by the Linker. If segments appear out of order on the map file, you can correct the order with a *First.asm* module that uses the same segment name and segment class.

In order to do this, you need to know about how to specify SEGMENTS and GROUPS in the Assembler. Following is a brief description of the syntax; for more information, see the *CTOS Programming Utilities Reference Manual: Assembler*.

Here are some general guidelines for writing a *First.asm* file. (An example follows.)

- Use the SEGMENT directive to explicitly name a segment. The SEGMENT directive also controls the alignment, combination, and contiguity of segments. It has the following syntax:

  ```
  [segname] SEGMENT [align-type][combine-type]
  ['classname'][segname] ENDS
  ```

  The optional fields (in brackets) must be specified in the order given.

- Use the ENDS directive to indicate the end of a segment

- Use the GROUP directive to name the segments that are to be contained in a Group

  The Group directive specifies that certain segments lie within the same 64K bytes of memory. It has the following syntax:

  ```
  Groupname GROUP segname [,...]
  ```

  In this example, Groupname is a unique identifier used in referring to the group. Segname is the name field of a SEGMENT directive. A typical group name is DGroup, used to group data.

- Any text following a semicolon (;) on the line is a comment

First.asm File Example

In the example *First.asm* file shown below (Example 6-1), the sequence of segment ordering is for illustration only. You can order segments differently from that shown.

Example 6-1. *First.asm* File

```
; Segments of class 'CODE' appear first
; segname _TEXT, classname CODE
_TEXT          SEGMENT         WORD PUBLIC 'CODE'
_TEXT          ENDS

; Segments of class 'ENDCODE' appear second
; segname _ETEXT, classname ENDCODE
_ETEXT         SEGMENT         WORD PUBLIC 'ENDCODE'
_ETEXT         ENDS

; Segments of class 'FAR_DATA' appear next
; segname FAR_DATA, classname FAR_DATA
FAR_DATA SEGMENT          WORD PUBLIC 'FAR_DATA'
FAR_DATA ENDS

; Segments of class 'DATA' appear next
; segname NEAR_DATA, classname DATA
NEAR_DATA      SEGMENT         WORD  PUBLIC  'DATA'
NEAR_DATA      ENDS

; segname NEAR_DATA, classname DATA
DATA           SEGMENT         WORD  PUBLIC  'DATA'
DATA           ENDS

; Segments of class 'CONST' appear next
; segname CONST, classname CONST
CONST          SEGMENT         WORD  PUBLIC  'CONST'
CONST          ENDS

; Segments of class 'STACK' next
; segname STACK, classname STACK
;   Note that STACK is a special combine-type to be used
;   only with segname STACK, clasename STACK.
STACK          SEGMENT         WORD  STACK   'STACK'
STACK          ENDS

; Put the segments named NEAR_DATA,DATA,CONST,and STACK
; into DGroup
DGroup         GROUP           NEAR_DATA,DATA,CONST,STACK

END           ;  END directive identifies end of this module.
              ;  It is REQUIRED
```

The *First.asm* file shown in Example 6-1 produces the Linker segment order shown in the map file below.

Example 6-2. The Map File Produced by the *First.asm* File

```
Linker (version)

Run file        : first.run
Link Start Time : 01/07/92 08:59:46

Config File     : c.sys

Start      Stop       Length          Name          Class

00000000h  00000000h  0000h  (0084h)  ??SEG         ??SEG
00000000h  00000000h  0000h  (008Ch)  _TEXT         CODE
00000000h  00000000h  0000h  (0094h)  _ETEXT        ENDCODE
00000000h  00000000h  0000h  (009Ch)  FAR_DATA      FAR_DATA
00000000h  00000000h  0000h  (00A4h)  NEAR_DATA     DATA
00000000h  00000000h  0000h  (00A4h)  DATA          DATA
00000000h  00000000h  0000h  (00A4h)  CONST         CONST
00000000h  00000001h  0002h  (00A4h)  STATICS       CONST
00000010h  00000010h  0000h  (00A4h)  STACK         STACK

No warnings detected
No errors detected
```

The map file shown above was produced using only the *First.obj* file generated from the *First.asm* file shown in Example 6-1. In the Linker command form, specify **first.obj** as the first entry in the *Object modules* field, and specify **first.run** in the *Run file* field.

Example of Correcting a Segment Ordering Error

A *First.asm* file is useful in correcting segment ordering errors that appear in the map file. The following example shows an error that appears in the map file and how you can correct it by using a *First.asm* file.

The error message displayed in the map file shown in Example 6-3 mentions two segments (selectors 00A4h and 00B4h) that were specified to be in the same group, in this case, DGroup. The segment that occurs between them (selector 00ACh) is not specified to be a member of the group and is the cause of the error. (For protected mode, all the Linker segments in a group must be contiguous.

Example 6-3. Map File Showing Segment Ordering Error

```
Linker xl/6

Run file         : NonContiguousGroups.run
Link Start Time  : 01/07/92 09:07:12

Config File      : c.sys


Start      Stop       Length           Name        Class

00000000h  00000000h  0000h   (0084h)  ??SEG       ??SEG
00000000h  00000000h  0000h   (008Ch)  _TEXT       CODE
00000000h  00000000h  0000h   (0094h)  _ETEXT      ENDCODE
00000000h  00000000h  0000h   (009Ch)  FAR_DATA    FAR_DATA
00000000h  00000000h  0000h   (00A4h)  CONST       CONST
00000000h  00000001h  0002h   (00A4h)  STATICS     CONST
00000010h  00000010h  0000h   (00A4h)  STACK       STACK
00000000h  00000000h  0000h   (00A4h)  NEAR_DATA   DATA
00000000h  00000000h  0000h   (00ACh)  DATA        DATA

No warnings detected
No errors detected
```

The *First.asm* file shown in Figure 6-4 is the correct specification of the segments that should be in DGroup.

Example 6-4. *First.asm* File Showing Corrected Segment Order

```
; Segments of class 'CODE' appear first
; segname _TEXT, classname CODE
_TEXT          SEGMENT        WORD PUBLIC 'CODE'
_TEXT          ENDS

; Segments of class 'ENDCODE' appear second
; segname _ETEXT, classname ENDCODE
_ETEXT         SEGMENT        WORD PUBLIC 'ENDCODE'
_ETEXT         ENDS

; Segments of class 'FAR_DATA' appear next
; segname FAR_DATA, classname FAR_DATA
FAR_DATA SEGMENT        WORD PUBLIC 'FAR_DATA'
FAR_DATA ENDS

; Segments of class 'CONST' appear next
; segname CONST, classname CONST
CONST          SEGMENT        WORD  PUBLIC  'CONST'
CONST          ENDS

; Segments of class 'STACK' next
; segname STACK, classname STACK
;   Note that STACK is a special combine-type to be used only
;    with segname STACK, classname STACK.
STACK          SEGMENT        WORD  STACK   'STACK'
STACK          ENDS

; Segments of class 'DATA' appear next
; segname NEAR_DATA, classname DATA
NEAR_DATA      SEGMENT        WORD  PUBLIC  'DATA'
NEAR_DATA      ENDS

; segname DATA, classname DATA
DATA           SEGMENT        WORD  PUBLIC  'DATA'
DATA           ENDS

; Put the segments named NEAR_DATA,DATA,CONST,and STACK into DGroup
DGroup         GROUP          NEAR_DATA,DATA,CONST,STACK

END                  ;  END directive identifies end of this module.
                     ;  It is REQUIRED
```

Configuration File

The recommended way to specify class ordering is in the Linker configuration file. The *:ClassOrder:* parameter in the configuration file allows you to override the configuration specified in the *First.asm* file.

The example map file shown in Example 6-2 was produced by using a *First.asm* file. You can get the same Linker segment order shown in that map file by using *:ClassOrder:* as follows:

```
:ClassOrder:(CODE ENCODE FAR_DATA DATA CONST STACK)
```

Section 7
What Is the Librarian?

What the Librarian Does

The Librarian is a program development utility that creates and maintains libraries of object modules.

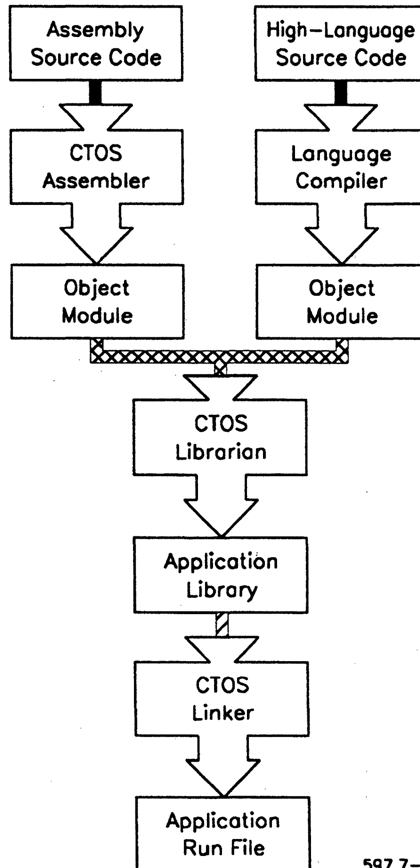
You can perform the following operations when you use the **Librarian** command from the Executive:

- Build a new library by specifying a new library file name and the object module(s) to be included in it
- Modify a library by specifying object modules to be added or deleted. (This includes the case in which a module in a library is to be replaced by a new module with the same name.)
- Extract one or more object modules from a library and place them in the current directory
- Produce a sorted cross-reference list of the object modules and public symbols in the library
- Set the library block size while a library is being created or change the block size in an existing library

How the Librarian Works

Figure 7-1 below shows how the Librarian manages application object modules.

Figure 7-1. Using the Librarian to Manage Application Object Modules



Maintaining libraries of object modules is one part of the application development process. To see how the Librarian works in conjunction with the other building applications utilities, see Figure 1-1 in Section 1, "Introduction to Building Applications."

Uses for the Librarian

You can use a library in the following ways:

- If you specify a library in the *[Libraries]* field of the Linker command form, the Linker searches the library for object modules that satisfy unresolved external references. 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.

  Placing object modules in a library and linking several object modules from a library specified in the *[Libraries]* field of the 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.

- You can use a library to collect several object modules and distribute them as a single file. The Librarian extraction facility can be used to extract specific modules from the unit. You must specify the desired object module to extract it from the library. For large portions with many object modules, this is a way to save system resources, such as free file headers. The Librarian extraction facility is described in Section 8, "Using the Librarian Command Form," and is also available in the Linker.

- You can use a library to collect several forms (which are actually object modules) created with the Forms Editor. (See the *BTOS II Forms Designer Programming Guide*.) You must specify the desired form name to extract it from the library.

Section 8
Using the Librarian Command Form

Introduction

This section shows how to use the **Librarian** command form. The parameters of the command are described also.

Command Form

To use the Librarian to build a new library, follow these steps:

1. On the Executive command line, type **Librarian**, and press **RETURN**.
2. Fill in the command form according to your needs. An example command form is shown below. The parameter fields are described in the next subsection.

```
Librarian
 Library file                 MyExample.lib
 [Files to add]               example.obj example2.obj
 [Modules to delete]          ________________________
 [Modules to extract]         ________________________
 [Cross-reference file]       ________________________
 [Suppress confirmation?]     ________________________
 [Library block size]         ________________________
 [Case sensitive?]            ________________________
```

3. Press **GO** to execute the **Librarian** command.

By specifying the appropriate fields, you can request multiple operations in one invocation of the Librarian. Modules are deleted, added, and extracted, in that order. See the cross-reference file for 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 want to receive a message asking for confirmation in this kind of situation (for example, when the operation occurs in a Submit file), respond **Yes** to the *[Suppress confirmation?]* field.

At the end of a library, a text string is often appended that identifies its version. Often version and copyright information are appended to the end of a library. Whenever you modify such a library, the data are lost; for example, if the data are in a *-old* library file but not copied into a more recent library file copied from the *-old* file.

Parameter Fields

The parameter fields in the Librarian command are described below.

Library file

Enter the file name of the object module library. Typically, it has the following 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 with the original name plus the suffix *-old*. However, if no files are added and no modules are deleted (for example, if you request a listing only), the input library is not modified and no *-old* file is generated. If modifications are requested, 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: None

Enter the object module files that you want to add to the library. Separate the names with spaces.

If you leave this field blank, 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 and directory specifications and file prefixes are dropped. Any final extension beginning with a period is dropped.

For example, if the file name is *[Sys]<Jones>Sort.obj*, the module name is *Sort*. If the file name is *<Jones>Working>Sort*, the file name is also *Sort*.

There is one exception to the naming rules just described. A subdirectory name is kept as part of the object file name in the library.

For example, if the file name is *Test\Sort.obj*, the module name in the library is *Test\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.

When the Linker searches a library for public symbols to match unresolved externals, it searches this index rather than the modules themselves. Therefore, it is unusual to create a library in which two object modules define the same public symbol. The symbol would be absent from the index and, hence, hidden from the Linker. (This kind of duplicate definition might reasonably occur in a library intended just as a convenient unit in which to collect object modules, and not for automatic search.)

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 at the end of the library.

You can suppress these requests for confirmation by specifying **Yes** in the *[Suppress confirmation?]* field.

[Modules to delete]

Default: None

Enter the list of modules that you want to delete from the library. Separate the names with spaces. A module name should not include the suffix *.obj*.

If you leave this field blank, no modules are deleted.

[Modules to extract]

Default: None

List the object modules in an existing library that you want to extract to form individual object module files. Separate the names with spaces. The specification is a list of entries in either of two forms:

ModuleName

or

FileName(ModuleName)

If the first form is used, files containing the specified object modules are created with names of the form *ModuleName.obj*. If the second form is used, the file name can be specified explicitly.

When the Librarian is used to modify a library (the most common use), the *[Modules to extract]* field is not used. Extraction does not modify a library.

If you leave this field blank, no modules are extracted.

[Cross-reference file]

Default: None

Enter the name of the file to which the Librarian is to write a cross-reference listing of public symbols and object module names. A cross-reference listing has two parts:

1. The first part lists public symbols in alphabetic order and, for each public symbol, the name of the module in which it occurs.

2. The second part lists module names in alphabetic order and, for each module, the names of the public symbols it defines.

If the same symbol is defined in different modules within a library, these duplicate symbol names are removed from the index of symbols to be searched by the Linker. However, they are listed in the cross-reference file. The first such symbol encountered is followed by no asterisk, the second by one asterisk, and so on. The modules in which they occur are listed (see Example 8-1).

Example 8-1 shows a sample listing produced if you use the *[Cross-reference file]* field.

Example 8-1. Sample Cross-Reference Listing

```
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
```

[Suppress confirmation?]

Default: No

Enter **Yes** 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) or when adding duplicate symbols.

If you enter **No** or leave this field blank, the Librarian issues prompts for confirmation.

[Library block size]

Default: 512

Enter a value that is an integral power of 2 between 16 (fourth power of 2) and 32768 to specify the size in bytes of a library block. For example, if you specify 32 (fifth power of 2), each library block contains 32 bytes and each object module will start on an even 32-byte boundary. If your object modules are very small, you can conserve space in your library by specifying a low value. For details, see "Library Block Size," below.

[Case sensitive?]

Default: No

Enter **Yes** to mark the library for case sensitive linking. This requires that the symbol within the object modules be case-sensitive.

Library Block Size

Blocks are the basic library units. Each object module is aligned with the beginning of a block. The default block size is 512 bytes. The last object module in a library must start within 64K minus 1 block from the start of the library.

This 512K-byte block size has been found to work well for most libraries. However, if most objects in a library are short (a DLL import library, for example), the library will have a lot of wasted space and will use up more disk space than necessary.

Conversely, if the library is to contain many object modules with an average size significantly greater than 512 bytes, it is possible that not all modules will fit in the library because they cannot be addressed.

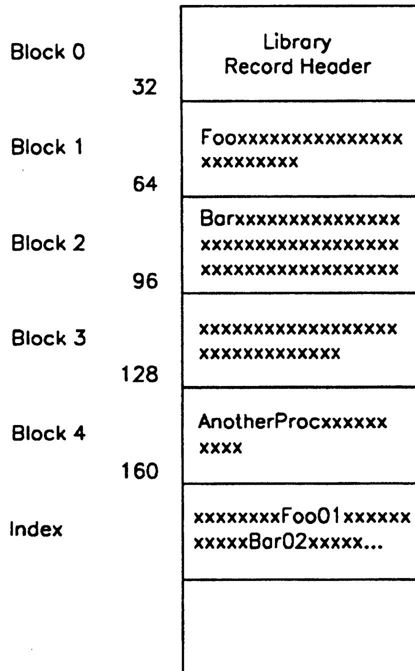
The way in which you adjust for each of the above situations is described in the paragraphs that follow.

Conserving Library Space

The *[Library block size]* parameter of the **Librarian** command allows you the flexibility of specifying block size. If, for example, the library you are writing contains many short modules, you can waste a significant amount of library space by using the default block size (512 bytes).

To illustrate this point, let's first look at the overall structure of a library. Figure 8-1 shows a simple object module library containing very small modules. The library is arranged in 32-byte blocks.

Figure 8-1. Library Blocks



Briefly, the sample library consists of three main components: a record header, the object modules, and an index.

The first 32-byte block contains the library record header. In this case, the header fits in the block. The object modules are contained in the blocks following the header, as described below:

- Object module *Foo* is 16 bytes long. It is aligned with beginning of block 1 and the code occupies the first half of the block.
- Object module *Bar* is 48 bytes long. It occupies all of block 2 and the first half of block 3.
- The object module *AnotherProc* starts at the beginning of Block 4.

To align the object modules on block boundaries, all unused bytes of partially filled blocks are padded with zeroes.

In this example, modules *Foo* and *Bar* require a total of 96 bytes (three 32-byte blocks). If block size had not been specified in the **Librarian** command form, these modules alone would require 1024 bytes (two 512-byte blocks).

Accommodating Large Modules

One consequence of not specifying a practical block size is that you may not be able to access all the library procedures. This can occur particularly if there are many modules with a size greater than 512 bytes.

Look again at Figure 8-1. The library index lists each object module procedure name followed by a word value; for example, Bar 02. The value 02 indicates that the object module in which the public *Bar* is defined starts in the second object module block.

Because of the one-word limit for specifying block position, the maximum number of blocks in the object module section of a library is 64K. If block size is small compared to object module size, and a library contains many such modules, the total number of blocks required by all the objects may exceed the 64K byte limit. To solve this problem, the block size should be made larger. To estimate the correct block size, do the following:

1. Issue the **Files** command on all object modules that you want to put in the library.
2. Multiply the total number of pages the object modules occupy by 512.

3. Divide by 64K.
4. Round the result up to the next number that is a power of 2, between 16 and 32,768, inclusive.

Library Index Procedures

Typically, object module names, public procedures, and variables are listed in the Library Index. This index is then searched to resolve external references. However, two cases in which public procedures and variables are not included in the Librarian index are described in the paragraphs that follow.

Duplicate Public Symbol Names

In most cases, the Librarian places public symbol names in the library index along with the block number in which the object module containing the procedure starts (see Example 8-1). The exception is when matching symbol names occur in two or more modules. In such a case, the Librarian does not list the name or starting block number in the index.

Uninitialized Variables

The Librarian does not list uninitialized (communal) variables in the library index. (For more information on uninitialized variables, see Section 5, “How the Linker Works.”)

Section 9
What Is the Module Definition Utility?

What the Module Definition Utility Does

You use the Module Definition utility when you plan to build applications that use dynamic link libraries (DLLs). The Module Definition utility creates object modules that contain special data required by the Linker to build DLLs. This utility also sets up the client interface to DLLs.

The Module Definition utility has a command form user interface that accepts a text file, called a module definition file, as input. Using this text file as input, the utility creates either an object module or an import library listing DLLs and DLL procedure names, or both. Using the information generated, the Linker and loader can associate DLL and DLL procedure names, which are declared externally in a client program, with the appropriate DLL procedure addresses.

Porting Presentation Manager Programs

The Module Definition utility is designed in part to make it easy to port DLLs from other platforms that use them, such as Presentation Manager. Presentation Manager is a collection of DLLs. Programs that call Presentation Manager (called clients) can run on CTOS if they have the proper interface to the DLL. You can use the Module Definition utility to define the interface, which is the list of imports a client program uses that allows it to make calls. This in turn provides the Linker with the information it requires to successfully link together object modules and import libraries.

The syntax of a CTOS module definition file is similar to that accepted by the Microsoft Linker. Any syntax rules not supported on CTOS are recognized and then ignored by the CTOS Linker. In addition, there are additional CTOS-unique features.

(For a list of supported features, see Section 11, "Writing a Module Definition File.")

How the Module Definition Command Works

The input to the **Module Definition** command is a text file that describes the name, attributes, exports, imports, and other characteristics of a DLL or an application run file. To define these characteristics, you include statements using the syntax described later in this section.

From the module definition file, the **Module Definition** command can create two different output files: an object module containing the list of exportable procedures formatted according to Intel Object Module Format (OMF) and an import library containing object modules in OMF format.

The object module is combined with the other object modules as input to the Linker for creating the DLL.

The import library is a standard library file that is combined with other libraries the Linker searches when resolving external references in the client. The client may be an application or another DLL. The import library provides information that is stored in the client header that allows the loader to locate the procedure in the DLL and determine its address.

Figure 9-1 shows how the Module Definition utility and Linker interact to produce a DLL and a client file (application run file). Figure 9-2 shows how an application accesses a DLL.

Figure 9-1. Creating a DLL for Use by an Application

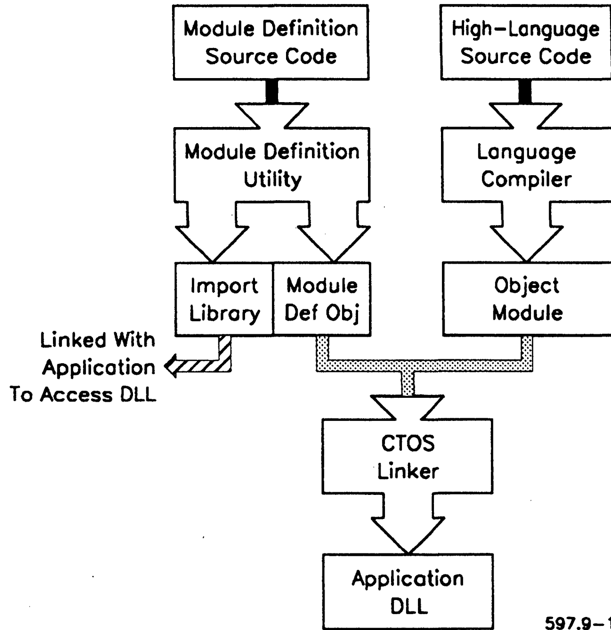
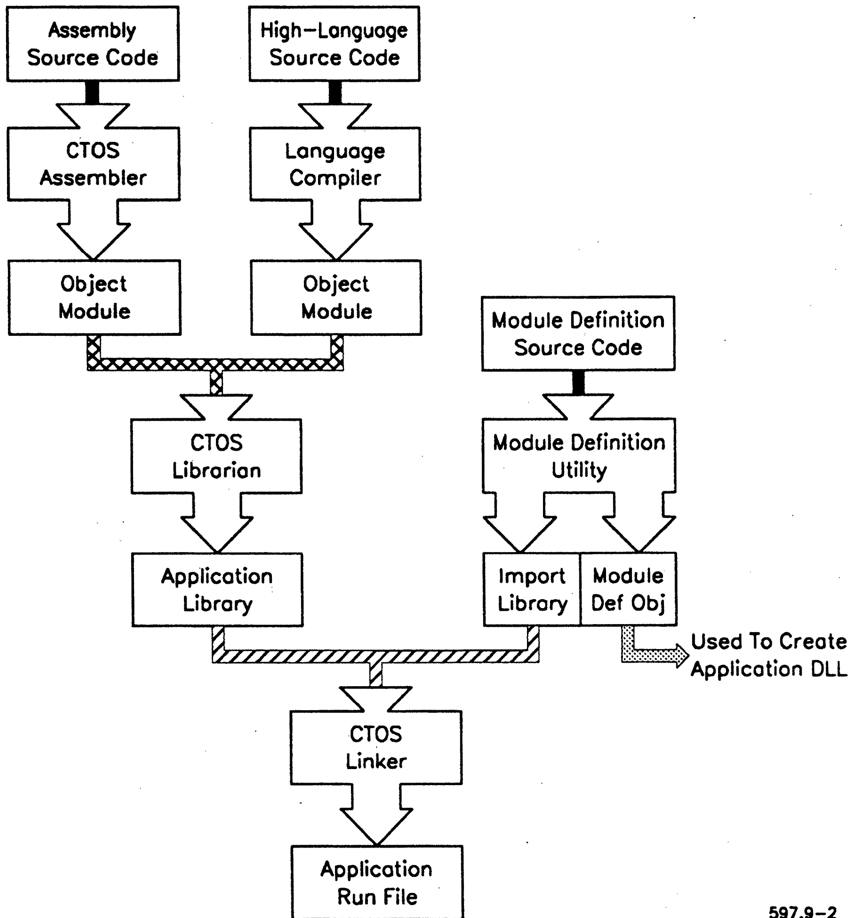


Figure 9-2. Linking an Application to Access a DLL During Execution

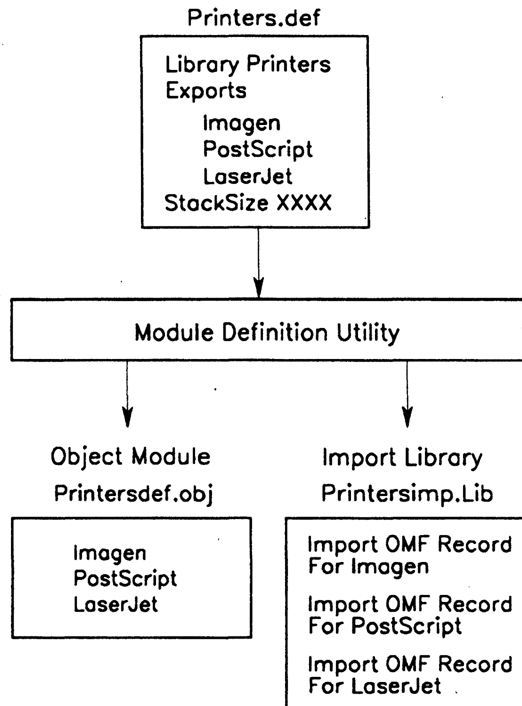


Building DLLs and application run files are one part of the application development process. To see how the Module Definition utility works in conjunction with the other building applications utilities, see Figure 1-1 in Section 1, "Introduction to Building Applications."

A Closer Look at Module Definition Command Output

Upon closer examination of the OMF output files created by the **Module Definition** command, you will see how the contents of these files are used to create a DLL and a client user (see Figure 9-3). For information about how to build a DLL and a client that calls it, see "Creating a Dynamic Link Library" in the *CTOS Programming Guide*.

Figure 9-3. Module Definition Command Output



Object Module

The object module created contains the name of a DLL and the names of the procedures in the DLL that are exportable. The object module shown in Figure 9-3 identifies the following three procedures in the DLL called *Printers*:

- Imagen
- PostScript
- LaserJet

These procedure names are identical to public symbols in other object modules that will be linked with this object module to produce the DLL. Export procedures are just one type of DLL procedure. DLL code usually contains procedures that are not exported as well. A DLL, for example, can be a client of another DLL, and its code can be written such that it calls other "local" procedures that are not exported.

Import Library

The object modules in the import library are linked with other modules to create a DLL client. These objects provide the information the client uses to access the DLL. All the object modules in this import library are of the same type: a record in OMF format containing a DLL procedure name and the corresponding DLL name (see Figure 9-3). Optionally, the OMF record can contain an ordinal value indicating the position of the procedure in the DLL. The import library resolves external references to the DLL procedures called in a client program.

The **Module Definition** command creates an import library only if the module definition file contains both a Library statement and an Exports statement. The Library statement allows you to specify the DLL name. The Exports statement lists the names of the DLL procedures to be exported (see Figure 9-3). The *Printers.def* module definition file contains a library statement specifying the DLL called *Printers* and an Exports statement listing the three exportable procedures: Imagen, Postscript, and LaserJet.

The Import library (to the right in Figure 9-3) contains an import OMF record corresponding to each of the export procedures in the *Printers* DLL. Each import record contains one of the DLL procedure names and the name of the DLL (with the suffix *imp.lib* appended).

Note: *An import record can also contain an ordinal value that specifies a number unique to a DLL procedure. See the description of the Exports statement in Section 12, "Module Definition Statements." Ordinals are supported for compatibility but are not recommended.*

Section 10
Using the Module Definition Command Form

Introduction

This section shows how to use the **Module Definition** command form. The command fields are described later.

Command Form

To use the **Module Definition** command to create either an object module or an import library that lists dynamic link libraries (DLLs), follow these steps:

1. On the Executive command line, type **Module Definition**, and press **RETURN**.
2. Fill in the command form according to your needs. The parameter fields are described in the next subsection.

```
Module Definition
  Input file (.def)        PMWIN.def
  [Object module]          PMWINDef.obj
  [Import library]         PMWINImp.lib
  [List file]              PMWINDef.lst
  [Suppress warnings?]     Yes
  [Suppress ordinals?]     Yes
  [Upper case?]            
```

3. Press **GO**.

Note: *For ease in porting programs, it is recommended that you use the conventions for naming object, list, and library files described in the next subsection. Doing so will enable you to distinguish these files from the corresponding files produced by compilers or assemblers and the Librarian.*

Parameter Fields

The parameter fields of the **Module Definition** command are listed below. The default values are noted where appropriate.

Input file (.def)

Enter the name of module definition text file. (Guidelines for writing this text file are given in Section 12, "Module Definition Statements.") By convention, you add the suffix *.def* to the source file root name. If, for example, the source file root name is *X*, the input file name is *X.def*.

[Object module]

Default: *FileNamedef.obj*

Enter the name of the object module file to be created.

If you do not enter a name, the suffix (if any) is removed from the input file name and *def.obj* is appended. If, for example, the input file name is *X.def*, the suffix *.def* is removed, and *def.obj* is added to create the object module file *Xdef.obj*.

The object file created will contain the data subsequently supplied to the Linker to create a DLL or an application run file.

[Import library]

Default: None

To create an import library, enter the import library name. The convention is to replace the input file name suffix (if any) with *imp.lib*. If, for example, the input file name is *X.def*, the suffix *.def* is removed, and *imp.lib* is added to create the import library name *Ximp.lib*.

An import library is created only if the module definition text file contains a Library statement and an Exports statement. (For details, see "A Closer Look at Module Definition Command Output" in Section 9, "What Is the Module Definition Utility?")

[List file]

Default: *FileNamedef.1st*

If you do not enter a name, the suffix (if any) is removed from the input file name and *def.1st* is appended. If, for example, the input file name is *X.def*, the suffix *.def* is removed, and *def.1st* is added to create the list file *Xdef.1st*.

The **Module Definition** command writes any errors it detects to this file.

You can examine this file in the Executive by using the **Type** command and specifying the list file name in the command form. You can also print it out or edit it. (See the *CTOS Executive Reference Manual* for details on the **Type** command.)

[Suppress warnings?]

Default: No

Enter **Yes** to suppress warnings.

[Suppress ordinals?]

Default: No

Enter **Yes** to direct the Module Definition utility to ignore ordinals.

[Upper case?]

Default: No

Enter **Yes** to convert all export and import names to upper case. This is useful when porting a case-sensitive Microsoft environment.

Section 11
Writing a Module Definition File

Introduction

This section provides guidelines for writing a module definition file. It describes why a module definition file is necessary, general syntax rules for all module definition files, how to define the client interface to a dynamic link library (DLL), special considerations for porting programs to CTOS, and recommendations about segment attributes.

The Need for a Module Definition File

When you create a DLL, you need to decide on the DLL features you want to use. For example, you need to decide which procedures or data you want to make available to clients. You also need to decide whether or not you want them to share the data. You do that (and more) in the module definition file.

The Module Definition utility creates two output files, an import library and an object module. The most visible use of a module definition file is to declare those procedures and/or data that you want to make accessible to clients of the DLL. The import library contains the names of the procedures and data that were listed as exported. When an application is linked, the external references to procedures and data in the DLL are resolved by entries in the import library, exactly as if it were an object module library. The object module generated contains information that the loader uses when it loads the DLL. The object module is linked into the DLL.

Example 11-1 shows a typical module definition file. The statements that comprise this type of file are described in detail in Section 12.

Example 11-1. Module Definition File Example

```
LIBRARY QUEUE INITINSTANCE QueueInitialization

DESCRIPTION 'CTOS QUEUE EXAMPLE'

DATA Nonshared

SEGMENTS
    Global_Seg Class 'Far_data' Shared
    Queue_Seg Class 'Far_data' Shared

EXPORTS
    QueueInitialization
    QueueAdd
    QueueDelete
    QueueStatus
```

For more information, see “Creating a Dynamic Link Library” in the *CTOS Programming Guide*.

General Syntax Rules

A module definition file contains one or more statements. Each statement is described in detail in Section 12, “Module Definition Statements.” The following are general rules you must follow when writing statements:

1. To specify that an application be created, include a Name statement. To specify that a DLL be created, include a Library statement. If you use the Name or Library statement, it must precede all other statements in the module definition file. It is an error to have both a Name statement and a Library statement in the same module definition file.

   The recommended method is to use a Library statement and an Exports statement for your DLL. From this information, one import library is created that you can use to link with each client user. You use a Name statement only if you are creating a separate module definition file for each client. (For details, see “Using an Imports Statement” later in this section.)

Note: *In the absence of a Name or Library statement, the Linker identifies the resulting run file as an application. To be able to call DLL procedures, the application must be linked with the import library generated for the DLL.*

2. Source-level comments may be included by beginning a line with a semicolon (;).

3. The **Module Definition** command is not case sensitive. Enter keywords in uppercase or lowercase letters (for example, Name, Library, and Segments). For readability, however, you are encouraged to conform to the standard conventions for capitalization. The convention is to begin each new word in the statement name with a capital letter; for example:

   RunType
   HeapSize
   Exports

4. All numbers can be entered in decimal, hexadecimal, or octal by using "C" notation: 0x*NNN* in hexadecimal, 0*NNN* in octal, or *NNN* in decimal, where *NNN* is a valid number for the indicated base.

5. Place character string sequences inside single or double quotation marks. Strings have a maximum length of 255 characters.

6. Limit the size of names to 80 characters or less. Longer names are truncated, and a warning is issued. Names are converted to uppercase letters when placed in the output object module files.

7. Optionally place names such as SegmentName and EntryName in single or double quotation marks. Quotation marks are required if the name conflicts with a module definition keyword, such as Code or Data.

Defining the Client Interface to a DLL

The client interface to a DLL can be defined in either of two ways: by creating an import library or by creating a module definition file for each client that imports DLL procedures.

Using an Import Library

Using an import library is the recommended method of resolving references to DLL procedures in client programs. It guarantees that the contents of the client program will match what is in the DLL because the complete interface to the DLL is automatically generated. All you need to do is link the client with this import library and the references to DLL procedures are set up correctly. The same import library can be linked with each client that will use the DLL procedures.

To create an import library, you must specify an Exports statement and a Library statement in the Module Definition text file. See Section 12, "Module Definition Statements."

Using an Imports Statement

You can also define the client interface to a DLL by using an Imports statement in the module definition text file.

Note: *Although this method is supported for compatibility, it is not recommended for new programs.*

The Imports statement method requires that you create a separate module definition file for each client that will call the DLL. In the module definition file, you identify the DLL procedures with an Imports statement and (optionally) the client name with a Name statement.

You only use the Name statement if the module definition file is for a specific client application. (If you omit the Name statement, the name of the resulting run file produced by the Linker is the name provided in the *Run file* field of the **Linker** command.)

In the Imports statement you are required to list the names of all the DLL procedures the client will import. Errors are generated if the Linker input is not accurate. (For details on all statements you can define in a module definition file, see Section 12, "Module Definition Statements.") Also, each of these module definition files must be updated every time the DLL interface described in the file is changed. Otherwise, errors will occur.

The Imports library method, in contrast, leaves the responsibility of creating the client interface up to the computer. The interface is built directly from the contents of the Exports statement, the very same statement used to define the DLL in the first place. With this method, you avoid introducing human error into the linking process and duplication of error.

Porting Programs to CTOS

The **Module Definition** command facilitates porting programs from other platforms. Because the syntax of the module definition input file is very similar to that used for OS/2, you can run most module definition files through the CTOS Module Definition utility and the CTOS Linker without compatibility problems. The parser ignores syntax statements and parameters having no CTOS meaning.

Some module definition syntax rules are included for ease of porting but are not supported on CTOS. This means that the syntax rules are ignored in the CTOS environment (that is, they have no meaning on the CTOS operating system or are ignored by the CTOS Linker or the CTOS Module Definition utility). If such syntax rules are present in a module definition file, they will simply generate warnings. Other syntax rules are supported but are not necessarily recommended in new programs if there is a simpler or better solution. The following discussion provides caveats for each of these types of situations.

Statements Included for Compatibility

The following statements are included for compatibility with ported programs:

ExeType
Old
Stub

Including these statements in a module definition file is allowed but they have no effect. Consult your source documentation for details on these statements.

CTOS Extensions

The following are CTOS extensions to the Module Definition syntax:

LoadType
RunType

Parameters Not Recognized

Statement fields not recognized in the CTOS environment are ignored by the parser (for example, the *ResidentName* field in the Exports statement). These fields are allowed in ported programs, but they have no effect.

The "Statements" descriptions in Section 12 include caveats for all such fields.

Segment Attribute Recommendations

To describe segment attributes, it is recommended you use the Shared attribute (with the mnemonic values **Shared** or **Nonshared**) rather than the Instance attribute (with the values **None**, **Single**, or **Multiple**). Combining both attributes is allowed and can be done if you elect to use both attributes in a Data statement. Such combinations are supported for compatibility with ported programs. However, to avoid unnecessary complexity when creating new module definition files, observe the recommendations given in the Data statement description in Section 12.

Section 12
Module Definition Statements

Statements

Each of the syntax statements that you can use in a module definition file is described in this section. The statements are presented alphabetically by name. Table 12-1 defines each of the statements briefly. The following pages explain each statement in more detail.

Table 12-1. Module Definition Statements

Statement	Description
Code	Defines default attributes for segments of class Code within the run file produced by the Linker.
Data	Defines default attributes for data segments within the run file produced by the Linker.
Description	Inserts a specified string of text into the run file. Uses include copyright information or source control.
ExeType	Ignored by the **Module Definition** command. For much of the same functionality it provides, use the CTOS extension RunType. Included for compatibility.
Exports	Specifies the names and attributes of the procedures that are made available (that is, exported) to other run-time modules.
HeapSize	Specifies the size in bytes of the local heap for the run file produced by the Linker. The heap is placed in the automatic or default data segment (DGroup).
Imports	Provides information to the Linker for resolving external references to procedures in dynamic link libraries (DLLs). This statement is supported but not recommended.
Library	Specifies that the run file created by the Linker when it links in this object module will be a DLL rather than an application run file. It also specifies the DLL name and its initialization type.
LoadType	Specifies information used by the system loader. This information indicates where the resulting run file can be executed in memory.

continued

Table 12-1. Module Definition Statements (cont.)

Statement	Description
Name	Specifies that the resulting file created by the Linker when it links in this object module will be an application run file (rather than a DLL). It also specifies the application name and its type.
Old	Ignored by the Linker. It is provided for ease of portation.
ProtMode	Specifies that the run file produced by the Linker will be run in protected mode.
RealMode	Specifies that the run file produced by the Linker will be run in real mode.
RunType	Specifies additional information specific to the run file. The information includes minimum and maximum operating system version, minimum instruction set, minimum and maximum data sizes, and library search specification.
Segments	Defines attributes for specific, named code or data segments in the run file produced by the Linker.
StackSize	Specifies the stack size of the run file produced by the Linker.
Stub	Ignored by the **Module Definition** command. It is included for ease of portation.

The following pages contain detailed explanations of the statements listed above.

Code

Code [*attribute*]

Description

The Code statement defines default attributes for code segments within the run file or DLL produced by the Linker. (A segment is a code segment if its class name ends with "CODE".)

The attributes set in a Segment statement override any attribute setting within a Code statement. A warning is issued if such a conflict is encountered.

See "Segment Attributes," later in this section, for complete definitions of segment attributes and details on the effects of combining segment attributes.

Parameters

attribute

An attribute from the following list of mnemonic values (the default CTOS value is listed first in each case):

Attribute	Valid Mnemonic Values
Conforming	Nonconforming, Conforming
Discard*	Discardable, Nondiscardable
ExecuteOnly	ExecuteRead, ExecuteOnly
Iopl*	NoIopl, Iopl
Load*	LoadOnCall, Preload
Movable*	Moveable, Movable, Fixed
Shared*	Shared, Nonshared

*CTOS ignores this attribute for Code segments.

The attribute field must contain one of the values listed above. Each attribute may appear at most one time. Order is not important.

See Table 12-2, "Attribute Definitions," later in this section for a complete description of each attribute.

Example

Code ExecuteRead

Data

Data [*attribute*]

Description

The Data statement defines default attributes for data segments within the run file or DLL produced by the Linker.

The attributes set in a Segments statement override any attribute setting in a Data statement. However, they do not override any attribute setting in a Segments statement for data segments of any other class name. A warning will be issued if such a conflict is encountered.

Note: *The Instance attribute is supported for compatibility with ported programs.*

The Instance attribute fixes the attributes of the automatic data segment (DGroup). It is also used as a default value for other data segments. The default for non-DGroup segments can be overridden for a specific segment with a Segments statement. If there are conflicts between the instance attribute for a segment set with a Data statement and the Shared attribute of the Segments statement for a segment data of class 'Data', a warning is issued, and the conflicting data is ignored.

See "Segment Attributes" later in this section for complete definitions of segment attributes and details on the effects of combining segment attributes.

Parameters

Note: *It is recommended that you use the Shared attribute rather than the Instance attribute in new programs. Instance is supported for compatibility with ported programs.*

attribute

An attribute from the following list of mnemonic values (CTOS values by default are listed first):

Attribute	Valid Mnemonic Values
Discard*	Discardable, Nondiscardable
Instance	Single, Multiple, None
Iopl*	NoIopl, Iopl
Load*	LoadOnCall, Preload
Movable*	Moveable, Movable, Fixed
ReadOnly	ReadWrite, ReadOnly
Shared	Shared, Nonshared

*This attribute is ignored by virtual memory operating systems for Data segments.

The attribute field must contain one of the attributes from the above list. Each attribute may appear at most one time. Order is not important.

See Table 12-2, "Attribute Definitions," later in this section for a complete description of each attribute.

Example

Data Shared

Description

Description *'string'*

Description

The Description statement inserts a specified string of text into the run file. Uses include copyright information or source control.

Parameters

string

The string to be inserted.

Example

Description 'This is my copyright'

ExeType

ExeType [*OS2* | *Windows* | *DOS4*]

Description

The **Module Definition** command ignores this statement. For much of the same functionality it provides, use the CTOS extension RunType.

Exports

Exports [*exportDefinitions...*]

exportDefinitions

exportEntryName
[=*internalName*][@*ord*[ResidentName]][*pWords*][NoData]

Description

The Exports statement specifies the names and attributes of the procedures that are made available (that is, exported) to other run time modules.

The Exports statement is meaningful only for the following cases:

- Procedures within DLLs
- Procedures that are exported to another module (such as "exporting" a window procedure to Presentation Manager)
- Procedures that execute with I/O privilege (not supported on CTOS)

Packaging the Exports information in a library relieves client applications from having to use an Import statement in a module definition file to access the exported function. Instead, the name of the import library can be specified in the *[Import library]* parameter field of the **Module Definition** command form.

Parameters

exportEntryName

Is a string that defines the exported procedure name as it is known to other modules. If the exported procedure name is multiply defined, a warning is returned.

internalName

Is a string that defines the actual name of the export procedure as it appears within the module itself. If the actual name is not specified, *exportEntryName* is assumed to be the internal name as well. Internal names should not be multiply defined.

ord

Is an optional value supported for compatibility. If an ordinal value is specified, it should not be duplicated in another Export definition.

Note: *The ord value is supported for compatibility with ported programs but is not recommended in new programs.*

NoData

Directs the Linker to ignore any GW switch action in Microsoft languages. For an explanation of GW switch, refer to the Microsoft C documentation.

Note: *CTOS ignores the following fields. For ease of portation, they can be present in a program but have no effect.*

ResidentName

pWords

Example

```
Exports DisplayData
        PutNumGlobal
        PutNumNonshared
        InitDisplay
```

HeapSize

HeapSize *number* maxVal

Description

HeapSize specifies the size in bytes of the local heap for the run file produced by the Linker. The heap is placed in the automatic or default data segment (DGroup).

Parameters

number

The size (in bytes) of the local heap.

maxVal

Sets the heap size to (64K-DGroup) bytes. *maxVal* directs the loader to allocate a total of 64K bytes for DGroup.

Example

HeapSize MaxVal

Imports

Imports [*ImportDefinitions...*]

ImportDefinitions

[*internalName=*] *moduleName.entry*

Description

Note: *This statement is supported for compatibility with ported programs. For new programs, it is recommended that you use an import library, instead. For more information about setting up an import library, see "Defining the Client Interface to a DLL" in Section 11.*

The Imports statement provides information to the Linker for resolving external references to procedures in DLLs.

The Imports statement is supported. However, it is recommended that, with your DLL, you provide an import library containing the information for resolving externals. To create such a library, specify the name of the import library in the *[Import library]* field of the Module Definition command form.

Parameters

internalName

The name that the importing module actually uses to call the procedure. If *internalName* is not specified, it is assumed to be the same name as *entry*. If internalName is multiply defined, a warning will be issued.

moduleName

The name of the DLL that contains the function. If the *moduleName* is the same as either your application name specified in the Name statement, or your library name specified in your Library statement, a warning is issued for self-referencing.

entry

The name of the procedure to be imported from the DLL identified by *moduleName*. For compatibility, an ordinal value is also supported for this parameter. The ordinal value is set in the Exports statement when the DLL is created. (See the description of the ord parameter in the Exports statement.) If an ordinal value is given, *internalName* is required.

Example

```
Imports DisplayLib.DisplayData
      DisplayLib.PutNumGlobal
      DisplayLib.PutNumNonshared
      DisplayLib.InitDisplay
```

Library

Library [*libraryName*] [*initialization* [*initialProcedureName*]] [*appType*]

Description

The Library statement specifies that the run file created by the Linker when it links in this object module will be a DLL rather than an application run file. It also specifies the DLL name and its initialization type.

If the Library statement is present in the module definition file, the Name statement may not appear.

If neither a Name statement nor a Library statement appears in the module definition file, the Linker assumes that the resulting run file is an application run file rather than a DLL.

Parameters

libraryName

Any valid file name. It specifies the nondefault name of the DLL as recognized by CTOS. The entry in the *libraryName* field should match the entry in the Linker command field, especially if Export statements are present in the module definition file input.

initialization

Specifies when the initialization procedure is to be called. The initialization procedure may be called either on the first load of the DLL or for each access to the library. Mnemonic values for this field are described below:

Mnemonic	**Meaning**
InitGlobal	(default) *initialProcedureName* is called only on first load of the DLL.
InitInstance	*initialProcedureName* is called on each access to the DLL.

Note: *The parameters initialProcedureName and appType are CTOS extensions to module definition syntax.*

initialProcedureName

Specifies the procedure associated with the initial entry point.

appType

Specifies the application run type environment. This parameter has the same values as *appType* in the Name statement. The Linker uses the value of *appType* to ensure compatibility between object modules and libraries. Mnemonic values for *appType* are described below:

Mnemonic Value	**Meaning**
WindowApi	PM and XVT/PM programs
WindowCompat	Character map, Advanced video (AVIO), Some Window API
NotWindowCompat	(default) Regular video (VIO)
Dynamic Link	Dynamic link library (default)

Example

Library DisplayLib InitInstance InitDisplay

LoadType

LoadType [*loadSpec*]

Description

Note: *This statement is a CTOS extension to the module definition syntax.*

LoadType specifies information used by the system loader. This information indicates where the resulting run file can be executed in memory.

Parameters

loadSpec

Specifies information the loader uses to load and execute the run file. *loadSpec* is one of the mnemonic values listed below:

Protected
GDTProtected
HighMemGdtProtected
LowDataGdtProtected
HighMemProtected
CodeSharingServer
HighMemCodeSharingServer

The Linker command form entry takes precedence over the module definition file. For a definition of each of these values, see Section 3, "Using the Linker Command Forms."

Name

Name [*appName*] [*appType*]

Description

The Name statement specifies that the resulting file created by the Linker when it links in this object module will be an application run file (rather than a DLL). It also specifies the application name and its type.

If the Name statement is present in the module definition file, the Library statement may not appear. (See the Library statement.) If neither a Name statement nor a Library statement appears in the module definition file, the Linker assumes that the resulting file is an application run file rather than a DLL.

Note: *If the resulting application run file calls DLL procedures, its interface to the DLL must be resolved. It is recommended that the application be linked with the Imports library created with the DLL. (For details, see "Defining the Client Interface to a DLL" in Section 11.)*

Parameters

Note: *CTOS ignores the appName field. It is provided for ease of portation. CTOS recognizes the application name specified in the* ***Linker*** *command.*

appName

Any valid file name.

appType

Specifies the application run type environment. This parameter has the same values as *appType* in the Library statement. The Linker uses the value of *appType* to ensure compatibility between object modules and libraries. Mnemonic values for *appType* are described below:

Mnemonic Value	Meaning
WindowApi	PM and XVT/PM programs
WindowCompat	Character map, Advanced video (AVIO), Some Window API
Notwindowcompat	(default) Regular video (VIO)

Note: *For PM and XVT/PM programs, you must specify the value WindowApi for your program to run correctly.*

Example

Name MyApp WindowCompat

Old

Old *'fileName'*

Description

The Linker ignores this statement. It is provided for ease of portation.

ProtMode

ProtMode

Description

The ProtMode statement specifies that the run file produced by the Linker will run in protected mode.

Parameters

This statement has no parameters.

Example

```
ProtMode
```

RealMode

RealMode

Description

The RealMode statement specifies that the run file produced by the Linker will run in real mode.

Parameters

This statement has no parameters.

Example

RealMode

RunType

RunType [*runSpec*]

Description

The RunType statement specifies additional information specific to the run file. The information includes minimum and maximum operating system version, minimum instruction set, minimum and maximum data sizes, and library search specification.

Note: *This statement is a CTOS extension to the module definition syntax. It should be used in place of the ExeType statement in programs to be run on CTOS operating systems.*

Parameters

runSpec

Is one of the items described below. Each item contains an identifying string (for example, Priority) followed by a specific value (for example, 128).

Item	Description
Priority <number>	Run-time priority of run file. Values are in the range 0 to 255 (default: 128)
MinInstruction <number>	Minimum required processor. Values are 80186, 80286, 80386 (default: 80186)
MinMathInstruction <number>	Minimum required math processor. Values are 80287, 80387 (default: 80287)
MinOSVersion <version*>	Minimum required operating system. (default: 0)
MaxOSVersion <version*>	Maximum required operating system. (default: 0)

* The version is the operating system release/revision number. The release is a major release number. Values are in the range 0 to 255. The revision is a minor revision number. Values are in the range 0 to 255.

Example

```
RunType Priority 80
        MinInstruction 80386
```

Segments

Segments [*SegmentDefinitions...*]

[*SegmentDefinitions...*]

segmentName [Class '*className*'] [*attribute...*]

Description

The Segments statement defines attributes for specific, named code or data segments in the run file produced by the Linker.

Parameters

segmentName

Specifies the name of a segment.

className

Specifies the segment class. If class is not specified, class **'CODE'** is assumed. Segments of class **'DATA'** are classed with the automatic data segment (DGroup) but do not need to be part of it if the sharing attributes do not match and the segment is not grouped into DGroup. Multiply defined segments of the same segment name and same class name are not allowed.

FAR_DATA is the class that the Microsoft C compiler assigns to all non-DGroup (i.e., "far") data segments. DGroup itself has a class name of DATA.

attribute

An attribute from the following list of mnemonic values (the default CTOS value is listed first in each case):

Attribute	**Valid Mnemonic Values**
Conforming	Nonconforming, Conforming
Discard*	Discardable, Nondiscardable
ExecuteOnly	ExecuteRead, ExecuteOnly
Iopl*	NoIopl, Iopl
Load*	LoadOnCall, Preload
Movable*	Moveable, Movable, Fixed
ReadOnly	ReadWrite, ReadOnly
Shared**	Shared, Nonshared

*CTOS ignores this attribute.

**CTOS ignores this attribute for ReadOnly segments. (They are always shared.)

The attribute field must contain one of the attributes listed. Each attribute may appear at most one time. Order is not important.

See Table 12-2, “Attribute Definitions,” later in this section for a complete description of each attribute.

Example

Segments 'Nonshared' Class 'Nonshared' Nonshared

StackSize

StackSize *number*

Description

The StackSize statement specifies the stack size of the run file produced by the Linker.

Parameters

number

The size (in bytes) of the stack. This value must be an even number.

Example

StackSize 1024

Stub

Stub *'fileName'*

Description

*The **Module Definition** command ignores this statement. It is included for ease of portation.*

Segment Attributes

The Segments statement defines attributes for specific, named code, or data segments in the run file produced by the Linker. This subsection describes each of the segment attributes that you can specify and summarizes how the attributes are used in each attribute's segment class. For more information on the Segments statement, see the description of that statement earlier in this section.

Following are two tables: the first table presents the attribute definitions, and the second table shows the attribute default values for virtual memory operating systems.

Table 12-2. Attribute Definitions

Attribute	Definition
Conforming	Refers to 286 conforming. Typically used for device drivers.
Discard	Determines whether the operating system can swap this segment out to disk. Ignored by virtual memory operating systems. All statements may be swapped out when using virtual memory operating systems.
ExecuteOnly	Determines whether code can be read-only or executed.
Instance	Applies only to the automatic data segment, which is a physical segment typically called DGroup. DGroup is the physical segment that contains the local stack and heap of the application. SegmentType determines the sharing attributes of DGroup. It may specify that DGroup be copied (not shared) for each instance of the run file. Alternatively, it may specify that DGroup not be copied (shared or global) for each instance of the run file.
Iopl	Determines whether or not a segment has I/O privilege. If a segment has I/O privilege, it can access the hardware directly.
Load	Determines whether the segment is loaded when execution of the module starts, or whether the segment is loaded when the module is accessed.
Movable	Determines whether the segment can be moved around in memory. Ignored by virtual memory operating systems. All segments are movable in virtual memory operating systems.
ReadOnly	Determines whether the segment can be read-only or read from and written to.
Shared	Determines whether instances of a run file can share a ReadWrite data segment. If not shared, each segment must be loaded separately for each process.

Table 12-3. Segment Attribute Default Values

Attribute	Segment Class	CTOS III Default
Conforming	Code	Nonconforming
	Data	*Not applicable*
	other	Nonconforming
Discard	Code	*Ignored*
	Data	*Ignored*
	other	*Ignored*
ExecuteOnly	Code	ExecuteRead
	Data	*Not applicable*
	other	*Not applicable*
Instance*	Code	*Not applicable*
	Data (DGroup dll)	Single (Shared)
	Data (DGroup app)	Single (Shared)
	other	*Not applicable*
Iopl	Code	*Ignored*
	Data	*Ignored*
	other	*Ignored*
Load	Code	LoadOnCall
	Data	LoadOnCall
	other	LoadOnCall
Movable	Code	*Ignored*
	Data	*Ignored*
	other	*Ignored*
ReadOnly	Code	ReadOnly (always)
	Data	ReadWrite
	other	ReadWrite
Shared	Code	Shared (always)
	Data	Shared
	Data ReadOnly	Shared (always)
	other	Shared
	other ReadOnly	Shared (always)

* Within a Data statement in a module definition file, it is recommended you use the attribute Shared.

Instance and Shared Attribute Field Effects

Although the Instance field is supported for compatibility, it is recommended that you use the Shared field for the Data statement *attribute* parameter in new programs. (See the description of the Data statement for segments in "Statements" earlier in this section.)

Using Only the Shared Field

Table 12-4 shows all segment types that can be generated by using only the Shared attribute.

Table 12-4. Shared Field Effects

Shared Field[1]	Automatic Data Segment	Other Segments[2]
<none>[3]	Shared	Shared
Shared	Shared	Shared
Nonshared	Nonshared	Nonshared

1. Cannot be overridden with a Segments statement.
2. Can be overridden with a Segments statement.
3. Neither 'Shared' nor 'Nonshared' is specified.

Using Both the Shared and Instance Fields

Although using both the Shared and Instance field is supported for compatibility, doing so is not recommended. It leads to redundancy and confuses the descriptions of segment types in your program. Table 12-5 shows all the combinations of segment types that can be generated by using both of these fields.

Table 12-5. Instance and Shared Field Effects

Instance Field	Shared Field[1]	Automatic Data Segment	Other Segments[2]
<None>[3]	**<none>** [4]	Shared	Shared
<None>	**Shared**	Shared	Shared
<None>	**Nonshared**	Nonshared [5]	Nonshared
Single	**<none>**	Shared	Shared
Single	**Shared**	Shared	Shared
Single	**Nonshared**	Shared	Nonshared
Multiple	**<none>**	Nonshared	Nonshared [6]
Multiple	**Shared**	Nonshared	Shared
Multiple	**Nonshared**	Nonshared	Nonshared

1. Cannot be overridden with a Segments statement.
2. Can be overridden with a Segments statement.
3. Neither 'Single' nor 'Multiple' is specified as a mnemonic value. 'None' is a valid mnemonic value for the Instance field.
4. Neither 'Shared' nor 'Nonshared' is specified.
5. Because the Instance attribute is not specified, the Shared attribute takes effect.
6. Because the Shared attribute is not specified, the Instance attribute takes effect.

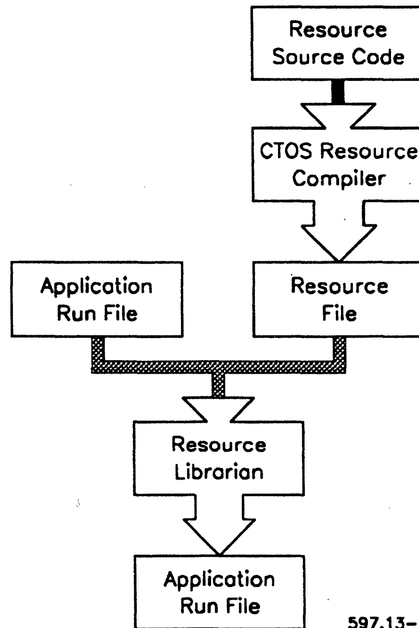
Section 13
What Is the Resource Librarian?

Introduction

The Resource Librarian allows you to maintain a program and its associated data (or resources) in the same run file. This arrangement makes it easy to correctly maintain corresponding versions of program and data files because the information is contained together in one file. For example, you can bind the correct version of a symbol file or a message file together in its corresponding run file.

Figure 13-1 shows how the Resource Librarian adds resources to a run file.

Figure 13-1. Adding Resources to a CTOS Run File



Adding resources to a run file is one part of the application development process. To see how the Resource Librarian works in conjunction with the other building applications utilities, see Figure 1-1 in Section 1, "Introduction to Building Applications."

If you are a Presentation Manager or XVT programmer, you need to bind resources into CTOS run files or dynamic link libraries (DLLs). The first step in this process is to use Presentation Manager tools to create the required resources. The next step is to use the Resource Librarian to bind the resources to a run file or DLL.

The Resource Librarian can manipulate both CTOS and Presentation Manager resources.

The Resource Librarian supports Version 6 run files and Version 8 run files. The Resource Librarian does not support Version 4 run files.

What Are Resources?

On CTOS, the term *resource* has historically meant anything required by an application to carry out its normal execution. Memory, for example, is a typical resource that an application requests from the memory management service.

In the context of the Resource Librarian, resources are simply blocks of data. Using the Resource Librarian, you can bind these resources to an executable run file. As an example, font data included in a run file for printing documents is a resource.

An application can directly access resources in a run file using the resource operations available in the standard operating system libraries. For a detailed discussion on how to use the resource operations, see "Utility Operations" in the *CTOS Operating System Concepts Manual.*

What the Resource Librarian Does

Basically, the functions performed by the Resource Librarian are analagous to those performed by the "regular" Librarian. The Resource Librarian is particularly useful for creating and maintaining sets of resources within run files. It can be used to add localized (translated) resources to run files as well.

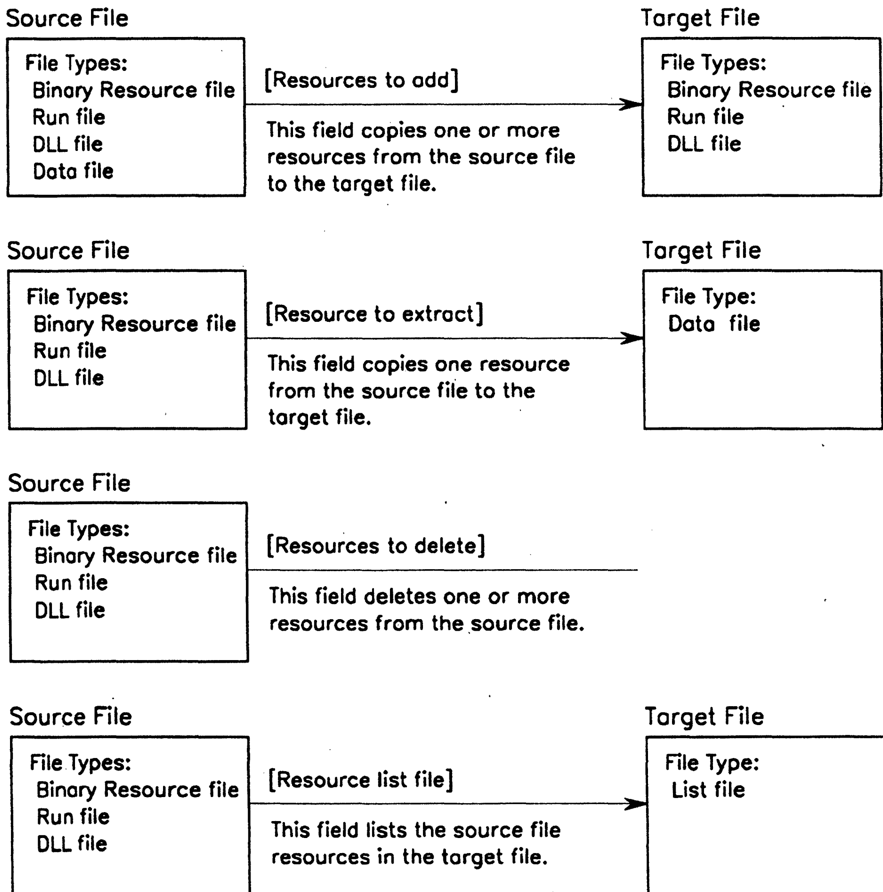
Using the Resource Librarian, you can:

- Add one or more resources to a run file or a DLL. Using as an input source any file that stores resources (a run file, binary resource file, or data file), you can add (copy) one or more resources to an output target run file, DLL, or binary resource file.
- Extract (copy) a specified resource from a run file or a binary resource file and copy it to a data file. (Remember, a data file is considered by the Librarian as a single resource.) Extracting does not modify the run file. You can then examine or modify the resource in the data file, and then if needed, add it back to the run file.

- Delete resources from a run file or binary resource file.
- List the resources included in the resource set of a run file.

Figure 13-2 shows the types of tasks you can perform by using the Resource Librarian.

Figure 13-2. Tasks Performed by the Resource Librarian



How Resources Are Stored

Resources can be stored in four types of files that can be manipulated by the Resource Librarian:

- Run file
- DLL (a type of run file)
- Binary resource file
- Data file

Using the Resource Librarian, you can move resources back and forth between these different types of files.

The files used in the Resource Librarian and how they store resources are summarized in Table 13-1.

Table 13-1. Resource Librarian File Definitions

File Type	Suffix	Description
Run file	*.run*	This file type must be a Version 6 run file created by a Version 12.0 or later version of the CTOS Linker or a Version 8 run file. It is created in run file format. A run file can contain multiple resources.
DLL file	*.dll*	A DLL file is created by the CTOS Linker in Version 8 run file format. A DLL file can contain multiple resources.
Binary Resource file	*.res*	This file type is created by a Resource Compiler or any other resource tool, such as the Presentation Manager Dialog Box Editor. A binary resource file can contain multiple resources. If a file has a *.res* extension, it is assumed by the Resource Librarian that each resource in the file has the following format. Only binary resource files with this format are supported. *TYPE NAME FLAGS SIZE BYTES* *TYPE* — Consists of a first byte (FFh) followed by an integer (2 bytes), which is an ordinal. *NAME* — Consists of a first byte (FFh) followed by an integer (2 bytes), which is an ordinal. There are no predefined ordinals. *FLAGS* — An unsigned value (2 bytes), which contains the memory manager flags. *SIZE* — A long value (4 bytes) showing how many bits are contained in the resource following. *BYTES* — The stream of bytes that comprise the resource.
Data file	*.bin* *.sym* *.cfg* *.txt* etc.	This file type can be either a binary or text file and can be created by any application. It does not have a specified format and can have any file extension. For example, the Linker produces a symbol (*.sym*) file; the Create Message File utility produces *msg.bin* files; specific resource editors, such as the Icon Editor utility, also produce data files. A data file always comprises a single resource.

How the Resource Librarian Identifies Resources

The Resource Librarian identifies a resource by resource type and resource ID using the following syntax:

ResourceType.ID.

The resource type and resource ID are numbers assigned to resources within applications. The type code identifies a general class and the ID number identifies an instance within that class. Default CTOS resource type and ID numbers are listed in Appendix K of the *CTOS Procedural Interface Reference Manual.* You may need to check your application-specific documentation for other resource type and ID numbers.

Resource type and ID numbers can be associated for memory purposes with names. This name/number association is done within the Resource Librarian configuration file (described in more detail in Section 15, "Using the Resource Librarian Configuration File"). For example, the symbol file associated with the Debugger is a resource listed as follows in the Resource Librarian configuration file:

```
:ResourceType:          Debugger          20000

:ResourceID:            SymbolFile        1
```

This resource can be recognized by the Resource Librarian either by number (20000.1) or by name (*Debugger.Symbolfile*). Note that if you want the Resource Librarian to recognize the names of resources, you must list the name of the Resource Librarian configuration file on the *[Resource config file]* field of the Resource Librarian command form, as described in Section 14.

Section 14
Using the Resource Librarian Command Form

Introduction

This section describes how to use the Resource Librarian command form. The parameters of the command are described later in this section.

The last part of this section includes examples of using the command form to specify adding, extracting, and deleting resources.

Command Form

To use the Resource Librarian to manipulate resources within a given resource set, follow these steps:

1. On the Executive command line, type **Resource Librarian**, and press **RETURN**.
2. Fill in the command form according to your needs. The parameter fields are described in the next subsection.

```
Resource Librarian
 Run or Res file                ______________
 [Resources to add]             ______________
 [Resources to delete]          ______________
 [Resource to extract]          ______________
 [Resource config file]         ______________
 [Resource list file]           ______________
 [Suppress confirmation?]       ______________
```

3. Press **GO**.

By specifying the appropriate fields, you can request multiple operations in one execution of the Resource Librarian. Resources are deleted, added, and extracted, in that order. Specify a file name in the *[Resource list file]* field for a list of resources in the run file after all operations are completed.

If you have revised a resource and want to reinsert it, it is most efficient to use the *[Resources to add]* field and allow the Resource Librarian to overwrite the preexisting resource of the same name. Deleting the old resource and adding the new one takes approximately twice as long.

Note that if you want the Resource Librarian to recognize the names of resources, you must list the name of the Resource Librarian configuration file in the *[Resource config file]* field of the Resource Librarian command form.

Parameter Fields

The parameter fields of the **Resource Librarian** command are described below.

Run or Res file

Enter the name of either a binary resource file (*.res* file) or a run file (including dynamic link library (DLL) files). The run file can be either a Version 6 run file created by a 12.0 or later version of the Linker or a Version 8 run file. You must complete the *Run or Res file* field.

Resources contained in both Version 6 and Version 8 run files can be shared among applications. Resources contained in a Version 8 run file are also treated as virtual storage.

[Resources to add]

Default: None

Enter a list of resource specifications, separated by spaces, that you want to add to the run file. The specified resources are copied from the specified source file (a run file , binary resource file, or data file) in the *[Resources to Add]* field and inserted into the target file (a run file or binary resource file) specified on the *Run or Res file* line. A copy of the original target file is made under *target-old*.

You are prompted for confirmation if a resource that you want to add has the same resource type code and ID as a resource already in the target file. If you confirm the replacement, the resource in the specified resource file replaces the resource of the same type code and ID in the target file.

Note: *The syntax for adding resources shown below differs according to the type of file where the resources are stored. The syntax for adding from a data file is different from adding from a binary resource or a run file. (For an explanation of the different types of files, see "How the Resource Librarian Identifies Resources" in Section 13.)*

Syntax for adding a single resource from a data file:

ResourceType.ID / DataFile . . .

Example:

MENU.DESKTOP / ExecMenu.Res

Syntax for adding one or more resources from a run file:

RunFile(ResourceType.ID) . . .

or

RunFile(ResourceWildcard) . . .

Examples:

PmExec.run(3.1) or *PmExec.run(MENU.Desktop)*
PmExec.run(.*)*

Syntax for adding one or more resources from a binary resource (*.res*) file:

BinaryResourceFile(ResourceType.ID) . . .

or

BinaryResourceFile(ResourceWildcard) . . .

Examples:

ExecMenu.Res(MENU.Desktop)
ExecMenu.Res(.*)*

(In this field, wildcards are not visibly expanded; the Resource Librarian does the expansion instead.)

[Resources to delete]

Default: None

Enter a list of resources that you want to delete from the run file.

Syntax:

ResourceType.ID . . .

or

ResourceType.Wildcard

or

Wildcard.Wildcard

Examples:

MENU.DESKTOP(3.1)
*MENU.**
.

(In this field wildcards are not visibly expanded on the CTOS command line; the Resource Librarian does the expansion instead.)

[Resources to extract]

Default: None

Syntax:

ResourceType.ID / Datafile

Example:

3.1 / X.res

Enter a *ResourceType.ID / Datafile* specification in the *[Resources to extract]* field. The Resource Librarian creates the specified *Datafile* and initializes it with the *ResourceType.ID* that is extracted from the binary resource file or run file you specify on the *Run or Res file* line. Extracting does not modify the run file or binary resource file.

Note that this field is used to extract (copy) a resource from a run file or binary resource file and copy it to a data file only, not to create another resource set.

(In this field wildcards are not visibly expanded on the CTOS Executive command line; the Resource Librarian does the expansion instead.)

[Resource config file]

Default: *ResourceLibConfig.sys*

Enter the name of the default configuration file, *ResourceLibConfig.sys*, or the name of a configuration file you have created.

To specify resources by symbolic names instead of by numbers, include the name of the Resource Librarian configuration file that contains this information. Doing this also causes the Resource Librarian output to use the same symbolic names to identify resources.

If you specify a Resource Librarian configuration file as shown below, processing messages will display resource names instead of numbers.

```
Resource Librarian
 Run or Res file                Test.run
 [Resources to add]             Source.run(*.*)
 [Resources to delete]
 [Resource to extract]
 [Resource config file]         ResourceLibConfig.sys
 [Resource list file]           Test.toc
 [Suppress confirmation?]
```

For more information, see Section 15, "Using the Resource Librarian Configuration File."

[Resource list file]

Default: None

Enter the name of a file that will list the resources contained in the file specified in the *Run or Res file* field. The Resource Librarian writes to this file a list of resource types and IDs, as well as the length in bytes of each resource contained in the run file.

If you want the list file to show the names in addition to the numbers of the resources, you need to specify a configuration file in the *[Resource config file]* field as shown in the following example.

```
Resource Librarian
  Run or Res file               Test.run
  [Resources to add]
  [Resources to delete]
  [Resource to extract]
  [Resource config file]        ResourceLibConfig.sys
  [Resource list file]          Test.toc
  [Suppress confirmation?]
```

[Suppress confirmation?]

Default: No

Enter **Yes** if you do not want prompts for confirmation when replacing existing resources (with the *[Resources to Add]* field). If you enter **No** or leave this field blank, the Resource Librarian enters prompts for confirmation.

Examples of Adding Resources

Use the *[Resources to add]* field to copy resources from one file to another. Using as a source any file that stores resources (a run file, a binary resource file, or a data file), you can add (copy) one or more resources to a target run file.

Note that the syntax you use differs depending on the type of file you are adding (copying) from. Specifically, you need to use a slash (/) so that the Resource Librarian can recognize a data file.

Note that *.res* or *.dll* can be substituted for *.run* in the following examples of adding resources.

Example 1: Adding a Data File

A data file is considered by the Resource Librarian to comprise a single resource.

To add the data file *Test.sym* (the source file), which is comprised of resource *Debug.Symbolfile*, to the resource portion of the run file *Test.run* (the target file), fill in the command form as follows:

```
Resource Librarian
 Run or Res file                    Test.run
 [Resources to add]                 Debug.SymbolFile/Test.sym
 [Resources to delete]
 [Resource to extract]
 [Resource config file]             ResourceLibConfig.sys
 [Resource list file]
 [Suppress confirmation?]
```

Example 2: Adding a Single Resource From a Run File

To add resource *KbdTransTable.4* (Type is KbdTranstable; ID is 4) within the file *Source.run* to the resource portion of *Target.run*, fill in the command form as follows:

```
Resource Librarian
 Run or Res file                     Target.run
 [Resources to add]                  Source.run(KbdTransTable.4)
 [Resources to delete]
 [Resource to extract]
 [Resource config file]              ResourceLibConfig.sys
 [Resource list file]
 [Suppress confirmation?]
```

Example 3: Adding Multiple Resources From a Run File

To add all resources of type *KbdTransTable* within the file *Source.run* to the resource portion of *Target.run*, fill in the command form as follows:

```
Resource Librarian
 Run or Res file                     Target.run
 [Resources to add]                  Source.run(KbdTransTable.*)
 [Resources to delete]
 [Resource to extract]
 [Resource config file]              ResourceLibConfig.sys
 [Resource list file]
 [Suppress confirmation?]
```

Examples of Deleting Resources

Use the *[Resources to delete]* field to delete resources from a run file or a binary resource file.

Example 1: Deleting a Single Resource From a Run File

Note that *Target.res* or *Target.dll* can be substituted for *Target.run* in the following examples of deleting resources.

To delete the resource *Debugger.SymbolFile* within the run file *Target.run*, fill in the command form as follows:

```
Resource Librarian
 Run or Res file                     Target.run
 [Resources to add]
 [Resources to delete]               Debugger.SymbolFile
 [Resource to extract]
 [Resource config file]              ResourceLibConfig.sys
 [Resource list file]
 [Suppress confirmation?]
```

Example 2: Deleting Multiple Resources From a Run File

To delete all resources of type *KbdTransTable* within the run file *Target.run*, fill in the command form as follows:

```
Resource Librarian
 Run or Res file                     Target.run
 [Resources to add]
 [Resources to delete]               KbdTransTable.*
 [Resource to extract]
 [Resource config file]              ResourceLibConfig.sys
 [Resource list file]
 [Suppress confirmation?]
```

Example of Extracting a Resource

Use the *[Resources to extract]* field to extract (copy) a resource from a run file or a binary resource file and place it in a data file.

Note: *Extracting does not modify the source file.*

Example: Extracting a Resource From a Run File

The following example extracts the resource *KbdTransTable.4* from the run file *Source.run*. The resource contents are copied to a new data file *Target.bin* which is created by the Resource Librarian. The file *Source.run* is not modified.

Note that *Source.res* or *Source.dll* can be substituted for *Source.run* in the following example.

```
Resource Librarian
 Run or Res file                        Source.run
 [Resources to add]                     ____________________________
 [Resources to delete]                  ____________________________
 [Resource to extract]                  KbdTransTable.4/Target.bin
 [Resource config file]                 ResourceLibConfig.sys
 [Resource list file]                   ____________________________
 [Suppress confirmation?]               ____________________________
```

Section 15
Using the Resource Librarian Configuration File

Introduction

The Resource Librarian configuration file is used to associate resource types and ID numbers with names. (If you have not already done so, see "How the Resource Librarian Identifies Resources" in Section 13 for more information about resource types and IDs.) It also provides a means by which you can look up resources in the event that you should forget the name or number.

The default configuration file distributed with the software is *[Sys]<Sys>ResourceLibConfig.sys*; however, you can create your own configuration file.

Resource Librarian Configuration File Format

The resource operations used by the Resource Librarian recognize only numbers; the configuration file provides a means of assigning more easily remembered names to these numbers. Specifying the name of a resource configuration file on the *[Resource config file]* line of the Resource Librarian command form instructs the Resource Librarian to display and use resource names instead of resource numbers when reporting about its operations (add, extract, and delete).

If you specify a Resource Librarian configuration file when requesting that the Resource Librarian produce a list file, the list file will list resource names as well as numbers.

The Resource Librarian configuration file uses the standard configuration file format:

```
:Keyword:Value
```

There are two valid entries:

- Resource Type
- Resource ID

Each identifier must define only one number. Any resource ID is assumed to follow its corresponding resource type code. Both must be specified for each resource to uniquely identify it for the Resource Librarian.

Each entry contains a mnemonic name and a decimal value separated by a space. The Resource Librarian uses the decimal value to identify the resource. The mnemonic values can help you; you can enter them in the Resource Librarian command form instead of entering the decimal values. See "How the Resource Librarian Identifies Resources" in Section 13 for more information.

Example Resource Librarian Configuration File

The following example shows a portion of the default configuration file *ResourceLibConfig.sys*.

```
:ResourceType:Debugger          20000
:ResourceID:      SymbolFile        1
:ResourceType:KbdTransData      20001
:ResourceID:      TrK1US           04     ; 04h
:ResourceID:      TrSG101K         33     ; 21h
:ResourceID:      TrSG102K        101     ; 65h
:ResourceID:      TrK4            144     ; 90h
:ResourceID:      TrOEM23         176     ; B0hFFh
.
.
.
:ResourceType:KbdEmulData       20002
:ResourceID:      EmK1US           04     ; 04h
:ResourceID:      EmSG101K         33     ; 21h
:ResourceID:      EmSG102K        101     ; 65h
:ResourceID:      EmK4            144     ; 90h
. . .
```

For a complete listing of default CTOS resource types and IDs, see Appendix K in the *CTOS Procedural Interface Reference Manual.*

(For application specific resource type and resource ID numbers, you may need to check your application documentation.)

Appendix A
Status Codes

Where to Find Help

When the results of a command are not what you expect, refer to the description of that command earlier in this manual for detailed information about the command and its parameter values.

When error messages occur, see the tables of error messages and explanations. They contain the following:

- Two tables of Linker/Librarian error messages. Table A-1 contains an alphabetized list of error messages that does not have status codes assigned. Table A-2 contains a numerical list of Linker/Librarian error messages with status code identification.
- Table A-3 contains Module Definition utility error messages.
- Table A-4 contains Resource Librarian error messages.

Linker and Librarian Messages

Linker and Librarian messages are similar because the structure and functions of the two programs are related. Throughout this appendix, references to Linker messages and solutions are also applicable to the Librarian unless an exception is noted.

If an error occurs during linking, the following message appears:

```
There were x errors detected.
```

The map file includes descriptions of the errors.

Levels of Linker Errors

The Linker can encounter three levels of problems:

- Violation of a Linker convention that still allows the Linker to produce a valid run file (program results can be affected)
- Violation of a Linker convention that produces a run file that you cannot run (the system crashes if you try to run the file)
- Fatal errors that cause the Linker to abort the linking process (the Linker does not produce a run file)

The Linker cannot always provide a complete diagnosis for each problem because it may not have enough information. For some of the complex problems, you must examine your program using clues from the Linker messages.

Linker Compatibility

The Linker is compatible with only certain versions of *Ctos.lib,* the compilers, and the Assembler. If you use an incompatible compiler, Assembler, or *Ctos.lib*, errors can occur. See the *Development Utilities Software Release Announcement* for details.

Causes of Linker Errors

Linker messages result from:

- Linker command input problems, such as erroneous file names or a missing entry from a required field

  These problems prevent the Linker from producing a run file.

- Capacity limitations, such as too many public symbols or not enough memory

  These limitations prevent the Linker from producing a run file.

Note: *If the problem is a lack of memory, try running the program in a larger partition or on a workstation with more memory.*

- Relocation or overlay problems

  If you have a relocation error, you should try rearranging the input modules listed in the Linker command form.

  If the error persists, you must determine the program's segment size requirement and reduce it. You can use the Linker map file (*filename.map*) to determine segment lengths. You can allocate large buffers to decrease the data segment memory requirements.

- I/O problems, such as an inability to create, read, write, or perform other operations on disk files

  These problems prevent the Linker from producing a run file.

  A CTOS error code accompanies most I/O problems; refer to Table A-2, or refer to your *CTOS Status Codes Reference Manual.*

- Compiler/Assembler problems, such as using the latest version of the Linker on object modules produced by earlier versions of a compiler or the Assembler.

Linker/Librarian Error Messages

This section contains two tables of Linker/Librarian error messages.

- Table A-1 is a list of messages that do not have status code identification. The table provides an explanation/action for each message.

- Table A-2 contains a numerical list of messages that have status code identification. These messages also appear in your *CTOS Status Codes Reference Manual.*

Table A-1. Linker Messages

Message	Explanation/Action
Attribute for segment of Name *<segment name>* Class *<segment class>* conflicts with other segments within its group.	Other segments of this group do not have the same attributes. Attributes are specified in the module definition file. See Section 12, "Module Definition Statements," for a list of segment attributes. You change the attributes in the module definition file. Groups may be set by your compiler or in a *First.asm* module. See "Arranging Object Module Components" in Section 5, "How the Linker Works."
Bad libraries parameter. Right paren expected after library reference	Invalid library reference on library parameter line input.
Bad max parameter	You entered a minimum higher than a maximum for the maximum array size, or the maximum data size in the Link, Link V6 or Link V8 command form fields.
Bad numeric parameter	You entered a nonnumeric character in a *Link,* Link V6 or Link V8 command form field that requires a numeric value.
Bad OS version parameter	Invalid operating system version parameter on the Run file mode parameter line to specify conditional run file mode.
Bad yes/no parameter	You entered something other than yes or no in a Link, Link V6 or Link V8 command form field that requires a yes/no response.
Cannot prohibit DS Allocation and use High C small or compact model.	Setting DsAllocation to 'NO' is incompatible with the object modules you are using.

continued

Table A-1. Linker Messages (cont.)

Message	Explanation/Action
Code reader confused at procedure start. Referenced in <*object module name*> near module offset xxxx. OR Code reader found illegal instruction <*instruction*>. Referenced in <*object module name*> near module offset xxxx.	The code reader is looking for IDIV instructions. If this error occurs, all instances of IDIV instructions may not be identified. Not identifying these instructions will not affect your resulting run file unless you are running on an 80186 processor that does not correctly process IDIV instructions.
Code segment of Name <*segment name*> Class <*segment class*> conflicts with other segments within its group.	The other segments of this group are not code segments. Groups may be set by your compiler or in a *First.asm* module.
Data segment of Name <*segment name*> Class <*segment class*> conflicts with other segments within its group.	The other segments of this group are not data segments. Groups may be set by your compiler or in a *First.asm* module.
DGroup is maximum value; DSAllocation turned off; DSAllocation segment ordering maintained. (Warning)	A segment of maximum length cannot expand any more, so DSAllocation (an expand down selector) is not required.

continued

Table A-1. Linker Messages (cont.)

Message	Explanation/Action
DGroup too large; it is xxxh bytes long.	DGroup must be less than 65535. ***Note:*** *If DGroup is maximized by the Linker, its maximum size is 65520. This is for backwards compatibility with previous versions of the operating system.*
DGroup too large; it is XXXXh bytes.	Offsets can only be 64K bytes; the group to which this segment belongs will cause an offset greater than 64K bytes.
Duplicate exported ordinal XXX.	Ordinal *xxx* was already specified.
Duplicate segment attribute directives ignored (default data segment). Duplicate segment attribute directives ignored (default code segment). Duplicate segment attribute directives ignored (Segment of name *<segment name>* class *<segment class>*).	Segment attribute directives have been processed more than once for this segment and they differ.
Export *<exported procedure name>* specified in module *<module name>*. Procedure exportation is INVALID for non dll.	Procedure is specified as an Export either in the module definition object module or in the object module where it is public.

continued

Table A-1. Linker Messages (cont.)

Message	Explanation/Action
IDIV instruct in overlay	When a Pascal or FORTRAN program contains code that results in an IDIV (integer division) instruction within an overlay, this error results. It indicates a real problem only if you plan to run the resulting run file on one of the affected systems (one which uses an early production 80186 processor chip). Move the code containing IDIV into the resident or ensure that all integer-division operands are positive. 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 1	The Linker has not created a stub in the overlay stub array data structure for a procedure you called in an overlay (normally this is an Assembly program problem). If you are trying to link an Assembly program: • If the message "Warning proc near xxxxx in xxxxx doesn't follow CALL/RET conventions" appeared during the link, examine that location in your Assembly program. • If the message did not appear, examine your entire Assembly program for call/return violations. The location cited with the message indicates where the call occurred. You can use this location to refer to a compilation listing to see what was called. **Note:** *Some run-time library modules in noncurrent versions of high-level language compilers generate code that violates the Linker call/return conventions. Either place such modules and the calls to them in the resident portion of your code or upgrade your compiler to the current level.*

continued

Table A-1. Linker Messages (cont.)

Message	Explanation/Action
Illegal segment address reference type 2	Parts of a procedure address have been separated. In a swapping program, it is illegal to use only one part of a two-part procedure address. In PL/M you can generate this error by using the construction p=@ProcedureName, which generates the statement MOV AX, SEG ProcedureName. To find the overlay address of a PL/M procedure name, you must define the procedure as a static constant in a DECLARE statement.
Illegal segment address reference type 5	Your Assembly program uses segment and offset in other than the two allowed ways: • A long CALL instruction • A DD instruction Examine your Assembly code. This error usually results from using a far JMP. This is illegal in an overlay program.
Incompatible initialization instructions file *<object module name>*.	A prior object module had a conflicting initialization instruction.
Incompatible window access mode encountered processing *<object module name>*.	A prior object module had a conflicting window access mode.
Input library is not case sensitive.	The Convert Public Case utility must be run on input library, or input library must be created with case-sensitive requested.
Invalid CharacterCodeSet *<CodeSet parameter>* specified in the configuration file for Version 4 run file format.	The configuration file parameter *xxx* for CharacterCodeSet found in the Linker configuration file is not appropriate for the target run file version.

continued

Table A-1. Linker Messages (cont.)

Message	Explanation/Action
Invalid character code parameter in config file - xxxx.	*xxxx* is not a valid character code parameter.
Invalid configuration file parameter *<parameter name>* for non Version 8 run file ignored OR Invalid configuration file parameter *<parameter name>* for Version 8 run file ignored.	The configuration file parameter *<parameter name>* found in the Linker configuration file is not appropriate for the target version of the run file.
Invalid library block size in *<library file name>*.	Library was made with invalid block size.
Invalid library header.	Linker is unable to locate dictionary information for looking up unresolved externals.
Invalid object module; Read error (Input file)	You specified an input file that is either corrupt, not a valid object module, or not a library file. Check your file name entry. Make sure your compiler or assembler is current.
Invalid source debugger parameter	Incorrect name for source debugger on source debugger parameter line.
Invalid value in configuration file for parameter *<parameter name>*.	An invalid value was specified in the configuration file for the *<parameter name>* parameter.
Invalid yes/no parameter in configuration file for parameter *<parameter name>*.	The Linker configuration file contains an invalid entry for *<parameter name>*; either yes or no is expected.

continued

Table A-1. Linker Messages (cont.)

Message	Explanation/Action
Length for near communal *<communal symbol>* is xxxh which exceeds 64Kb.	Size of near communal symbols exceeds maximum of 64K bytes. Near communal symbols are allocated for uninitialized variables in small and medium models.
Library reference expected	
Library reference not found	Library reference was not found in the configuration file. See "Library References" in Section 3, "Using the Linker Command Forms."
LoadOnCall attribute for segment of Name <segment name> Class <segment class> conflicts with other segments within its group.	This is an error for a Version 8 run file. The other segments of this group do not have the LoadOnCall attribute. You can change the attributes in the module definition file.
Module compiled with Publics is not resident	This error message is applicable only for programs generated by the BASIC compiler. You cannot locate BASIC modules that contain public symbols in overlays. Move the module to the resident segment, or remove the data definitions from the module.
Multiple definition of communal symbol *<communal symbol>* already defined as public symbol.	Errors occur for non-case-sensitive links to aid in identifying symbols that have the same letters but not the same case.
Multiple definition of public symbol *<public symbol>* already defined as communal symbol *<communal symbol>*.	Errors occur for non-case-sensitive links to aid in identifying symbols that have the same letters but not the same case.

continued

Table A-1. Linker Messages (cont.)

Message	Explanation/Action
Multiple definition of symbol *<public symbol name>* in file *<object module name>*.	The same public symbol is defined in two or more modules; the Linker uses the first definition it encounters and issues this error. You can determine which symbol the Linker encounters first; proceed as follows: 1. List the location of each multiply defined symbol using the Librarian. 2. List the object modules in the Link, Link V6, or Link V8 command form such that the Linker encounters the symbol first.
No Overlay Fault procedure loaded	In a program with overlays, no call to InitOverlays or InitLargeOverlays exists, so the Overlay Handler is not loaded. Add the call to your program.
No run file	You must specify a run file name (or dynamic link library name) in the Link, Link V6, or Link V8 command form.
No STACK segment	You must provide a stack segment for Assembly language programs. If you do not, the Linker creates a run file, but the system crashes when you run it. This error can be ignored in linked objects with no code segments such as "nls.sys". A dynamic link library also usually does not have a stack segment.
Non "CODE" class loaded into overlay	An overlay cannot contain a segment with a class other than CODE. Segments in overlays can contain only executable instructions. The program may run if the affected overlay is not actually used as an overlay.
Non contiguous GROUPS not pMode compatible. (Selectors nnn and mmm)	This error message is printed when the protected mode requirement that all code segments on all data segments be contiguous is violated, for example, binding modules in which the original order of groups has not been preserved. This message often occurs when binding assembler modules with various compiler-generated modules. Segments can be rearranged with either the *First.asm* file or the Linker configuration file option *:ClassOrder:*.

continued

Table A-1. Linker Messages (cont.)

Message	Explanation/Action
NonShared attribute for segment of Name <segment name> Class <segment class> conflicts with other segments within its group.	This is an error for a Version 8 run file. The other segments of this group do not have the NonShared attribute. You can change the attributes in the module definition file.
NonShared DGroup cannot be expanded if DsAllocation is requested. (Warning)	DGroup may not be LoadOnCall data if DSAllocation is requested.
Object module *<object module name>* generated by Microsoft C does not support Version 4 run files format OR Object module *<object module name>* generated by MetaWare High C does not support Version 4 run files format.	Use Link V6 or Link V8 to create run files for Microsoft C or MetaWareHigh C.
Odd copyright string length; truncated	Copyright string must have an even number of bytes.
Odd DGroup heap size requested; rounded up to xxxx	DGroup heap size on the *[Stack]* parameter line must be even.
Odd length STACK in <filename>; rounded up. (Warning)	This is a compiler error; make sure that you have the current version. All stack lengths must be an even number of bytes. The Linker adds one byte to the length of any stack that is odd. The run file should execute correctly.

continued

Table A-1. Linker Messages (cont.)

Message	Explanation/Action
Odd size stack requested rounded up	You requested an odd-length stack in the stack size parameter of the Linker command or assembler. The Linker adds one byte to the length of any stack that is odd; the run file should execute correctly.
Overlay with zero length will cause real mode Virtual Code Segment Management to fail.	The overlay specified by the number xxx has zero length. Zero-length overlays cannot be used with real mode virtual code management.
Proc near xxxxx in *<object module name>* doesn't follow CALL/RET convention	The Linker call/return conventions have been violated. If the message "Illegal segment address reference of type x" appears, a fatal error has occurred. If the call/return convention is not followed in a module in an overlay, real mode overlay management may not work correctly. Refer to the Explanation/Action for the "Illegal segment address reference type x" message. This violation can result from the use of a noncurrent compiler, from placing a noncurrent run time library module in an overlay, or from an Assembly program with a call/ret problem. For more information about language calling conventions, see the *CTOS/Open Programming Practices and Standards, Application Design Manual.*
Procedure Exportation is INVALID for non V8 run files.	Procedure is specified as an Export either in the module definition object module or in the object module where it is public.
Procedure importation is INVALID for non V8 run files	Procedure is specified as an Import either in the import library or in the module definition object module.
Program size exceeds Linker capacity	Insufficient memory is available to the Linker in the partition. Try running the Linker in a larger partition. 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, this error results.

continued

Table A-1. Linker Messages (cont.)

Message	Explanation/Action
Re-assigning STACK segment in <filename>.	Only one segment can have the STACK combine type.
Relocate offset from group is too large	Group size exceeds 64K bytes, or, noncontiguous segments occurred. If the error is caused by noncontiguous segments, use an assembly language program to declare the class names of the segments in a different order and place this module first in the Linker object modules field. This first module serves as a template; the Linker orders segments from the following modules in the same way.
Relocate offset is too large	Refer to the explanation and action for the message "Relocation offset from group is too large" above.
Relocate offset of near reference is too large	The procedure call or data reference uses a 16-bit address, but the target object is too far away to be reached using only 16 bits. A near call requires that the called address be less than 64K bytes from the caller's address and that a 16-bit address be used. The run file produced is invalid. You can either make your program smaller or reorder the object modules to bring references and addresses closer together. If the message identifies a public symbol, you can use it to identify the call. If the message identifies a hexadecimal address, you can examine a compilation list to identify the call. If the caller and called address are from a high level language, this error probably results from a data segment variable reference. If the caller or the called address are in Assembly language, change the near call to a far call. If you cannot do this, make sure that both addresses are in the same group.

continued

Table A-1. Linker Messages (cont.)

Message	Explanation/Action
Requested heap size exceeds 64K.	You requested a heap size that exceeds 64K bytes. You must reduce your heap requirements.
Requested stack size exceeds 64 Kb	You requested a stack size that exceeds 64K bytes. You must reduce your stack requirements.
Run file header exceeds 64Kb; Run file cannot run on OS prior to CTOS II 3.3 (Warning)	Enhancements to both the Linker and the operating system will allow some run files with run file headers greater than 64K bytes to run that previously could not. A run file that gets this message falls into that category and will not execute on older operating systems.
Segment of absolute or unknown type	All segments must be relocatable. This message can result from using a nonsupported compiler. The run file the Linker produced may be invalid.
Segment of name *<segment name>* class *<segment class>* not found; attributes specifying characteristics for that segment ignored.	Segment declared in module definition input, but no segment definition exists in the object modules.
Shared attribute for segment of Name <segment name> Class <segment class> conflicts with other segments within its group.	This is an error for a Version 8 run file. The other segments of this group do not have the shared attribute. You can change the attributes in the module definition file.
Size mismatch for communal symbol *<communal symbol>* in file *<object file name>* vs. communal symbol in file *<object file name>* (warning)	Uninitialized variables have a different total number of elements or a different number of bytes in the specified modules. You may want to make them the same size.

continued

Table A-1. Linker Messages (cont.)

Message	Explanation/Action
Symbol file hash table overflow	The program requires more table space than is currently available to the Linker. The upper limit on the symbol table is 997 pages. This message can also appear if you have many long names for public symbols. You must reduce the number of public symbols, or the name length, before the Linker can produce a run file.
Too many libraries.	Maximum number of libraries is 100.
Too many parameters on stack line	Too many parameters entered on the *[Stack]* parameter line. The maximum number allowed is 2.
Too many public symbols	Insufficient memory is available. 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, this error results. If you are using the Linker, increase the Linker's available memory or link the files on a workstation with more memory. If you are using the Librarian, 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. To list the cross-reference of 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.
Total group size exceeds 64k - exceeded size is XXXXh bytes. OR DGroup too large; it is XXXXh bytes.	Offsets can only be 64K bytes; the group to which this segment belongs will require an offset of greater than 64K bytes.

continued

Table A-1. Linker Messages (cont.)

Message	Explanation/Action
Total size of resident segments is xxxh paragraphs at segment name *<segment name>* class *<class name>*.	Size of resident segments exceed 64K paragraphs. This is a warning for applications linked so that they can run in protected mode. It is fatal for applications linked so that they can run only in real mode. It is not an error at all if either Link V8 or NRelProtected has been requested in the *[Run file mode]* field.
Unresolved export *<export symbol name>*.	Export symbol is an unresolved external. (Its public declaration has not been linked in.)
Unresolved externals	Your program contains references to external names that do not have public definitions in any other module. The map file contains an undefined symbol list. The Linker produces a run file. For direct calls, the Linker modifies the call to reference the Debugger. You can run the program; however, the program response is questionable. The system may crash. You should add the definitions to an existing module or provide a new module containing the definitions. ***Note:*** *If you do not specify a version when you are linking the operating system, or any system that uses a version number, this error results. The unresolved external's name will be SBVERRUN in this case.*
Warn: Module *<object module name>* compiled with Publics is not resident.	Warning emitted if an object module produced by the BASIC compiler is placed in an overlay.
Warn: No First.Obj found before Fortran-86 module.	Warning emitted for FORTRAN object module.

Linker/Librarian Status Codes

Status codes from 1380 to 1390 indicate some internal inconsistency in the software, perhaps because of some work area or table being exceeded. You should note the operation you were doing just before the problem occurred. This information will be helpful in determining the error.

Status codes from 4400 to 4427 are specific to the Linker/Librarian utility. The status codes from this range that do not appear on the list in Table A-2 are part of internal Linker error checking; if you see an unlisted status code displayed, report it to Unisys, because it results from a Linker or compiler error.

Table A-2. Linker Status Codes

Code	Explanation
400	There is not enough memory available in a specific partition to satisfy a memory allocation request. The Linker does not have enough memory available to link the file. You commonly encounter this status code by generating a run file improperly using the default for the *[Max array, data]* parameter, or by setting a parameter too small to meet program needs. To link the file: Properly determine and use valid parameters. If you are running the Linker under the Context Manager, reconfigure the partition size. Link the run file on a system with more memory.
1380	Bad heap node pointer. An internal memory management error has occurred. If you observe such an error, report it to your technical representative.
1381	Bad node link. An internal memory management error has occurred. If you observe such an error, report it to your technical representative.

continued

Table A-2. Linker Status Codes (cont.)

Code	Explanation
1382	Bad node tag. An internal memory management error has occurred. If you observe such an error, report it to your technical representative.
1383	Count of register pointers node overflow. An internal memory management error has occurred. If you observe such an error, report it to your technical representative.
1384	Count of register pointers node underflow. An internal memory management error has occurred. If you observe such an error, report it to your technical representative.
1385	Dangling node backpointer. An internal memory management error has occurred. If you observe such an error, report it to your technical representative.
1386	Double node registry. An internal memory management error has occurred. If you observe such an error, report it to your technical representative.
1387	No node backpointers. An internal memory management error has occurred. If you observe such an error, report it to your technical representative.
1388	No free node backpointer. An internal memory management error has occurred. If you observe such an error, report it to your technical representative.
1389	Node not busy. An internal memory management error has occurred. If you observe such an error, report it to your technical representative.
1390	Node not free. An internal memory management error has occurred. If you observe such an error, report it to your technical representative.

continued

Table A-2. Linker Status Codes (cont.)

Code	Explanation
4400	Attempt to access data outside of segment bounds, possibly bad object module. Virtual address exceeds virtual area bounds. 1. Some of the virtual memory areas are configurable; if so, there are directions in the map file above this message that specify which virtual memory area size to increase in the Linker configuration file. An example of such a message is: Size of configurable virtual Linker area aaaa exceeded; default setting in Linker configuration file is: Maxaaaa:xxxx aaaa is the area and xxxx is the current setting. 2. If you did not use a segment directive in your Assembly program, or if you declare code or data outside any segment, the Assembler supplies a segment named ??SEG. The resulting object module is invalid and the Linker cannot produce a run file. In Assembly programs, make sure you include a segment directive. This error can also result from a compiler error.
4401	Virtual address is out of bounds..
4402	Fatal error. An internal failure has occurred. If you observe such an error, report it to your technical representative.
4403	Too many segment or class names. You cannot declare more than 511 segments or different segment names in one module; however, the program can contain more than 511 segments. The Linker does not produce a run file. If necessary, divide the module.
4404	Too many segments. You cannot declare more than 511 segments or different segment names in one module; however, the program can contain more than 511 segments. The Linker does not produce a run file if you receive this message. If necessary, divide the module.

continued

Table A-2. Linker Status Codes (cont.)

Code	Explanation
4405	Segment size exceeds 64K bytes. Each segment cannot be larger than 64K bytes. This error pertains only to a single segment, not to a group or sum of segments (for example, DATA, CONST, and STACK). The link is aborted. The Linker does not produce a run file. If you are writing in Assembly language or Pascal, reduce the size of the segment to less than 64K bytes.
4406	Too many groups. Each module can contain a maximum of 10 groups, and the program can contain a maximum of 512 groups.
4407	Too many external symbols in one module. The Linker does not have sufficient memory to link these modules. To link the file: If you are running the Linker under the Context Manager, reconfigure the partition size. Link the run file on a system with more memory.
4408	Too many public symbols in one module. The Linker does not have sufficient memory to link these modules. To link the file: If you are running the Linker under the Context Manager, reconfigure the partition size. Link the run file on a system with more memory.
4409	Invalid object module. A file you specified as an object module is not in object module format. This could result from: Compiler error Damage to the file Specification of a text file (such as the source file) instead of an object module.

continued

Table A-2. Linker Status Codes (cont.)

Code	Explanation
4410	Linker discovered segment in second pass that it had not seen or processed correctly in the first pass. Call your Unisys technical representative, and be prepared to send your linking environment.
4411	Too many common symbols in one module. The Linker does not have sufficient memory to link the run file. To link the file: If you are running the Linker under the Context Manager, reconfigure the partition size. Link the run file on a system with more memory.
4412	Linker discovered common symbol in second pass that it had not seen or processed correctly in the first pass. Call your Unisys technical representative.
4413	Bad object module, segment, or group index out of range. You included an invalid object module. Usually, this is the result of a compiler error.
4414	Too many public procedures in resident/overlay. The resident portion and any single overlay can have a maximum of 4,096 procedures. Divide the code into more overlays.
4415	Data offset exceeds the current maximum of 64K bytes. CTOS and the Linker do not support this addressing mode. Rewrite the module to keep all offsets to less than 64K bytes.
4416	Object module has code that refers to a segment that is an overlay which is illegal. It is likely that a construct was used that determines a pointer to a procedure and then passes or otherwise uses the pointer. This is illegal if the procedure pointed to is in an overlay or if the procedure that does this is in an overlay. Do not do this.

continued

Table A-2. Linker Status Codes (cont.)

Code	Explanation
4417	Object module has code that tries to use the selector of an absolute segment, which is illegal. Do not do this.
4418	Too many global segments. The Linker does not have sufficient memory to link the run file. To link the file: If you are running the Linker under the Context Manager, reconfigure the partition size. Link the run file on a system with more memory.
4419	Too many segments. The Linker does not have sufficient memory to link the run file. To link the file: If you are running the Linker under the Context Manager, reconfigure the partition size. Link the run file on a system with more memory.
4420	Too many areas. The Linker does not have sufficient memory to link the run file. To link the file: If you are running the Linker under the Context Manager, reconfigure the partition size. Link the run file on a system with more memory.
4421	This is an internal error generated when sorting procedures in one of several ways. Either the sort procedure started sorting past the end of the allowable range, or the flags set for the procedure comparison step are not correct. Call your Unisys technical representative.
4422	Bad object module, external index out of range. You included an invalid object module. Usually this is the result of a compiler error.

continued

Table A-2. Linker Status Codes (cont.)

Code	Explanation
4423	Bad object module, name index out of range. You included an invalid object module. Usually this is the result of a compiler error.
4424	Bad sector.
4425	Fatal error. Cannot open the file. You may have used an illegal library format.
4426	Bad Index. The index read-in is out of range or does not follow OMF rules for indexes greater than 128.
4427	Run file header too large. The run file header exceeds 64K bytes.
4430	Multiple definition of IMPORT procedure. An IMPORT procedure is PUBLIC in more than one object module.
4431	Duplicate segment attribute directives ignored this code segment. You probably specified segment attributes for code segments in a Code record in a module definition file and then specified other segment attributes in the same module definition file in a Segment statement or in another file. Do one of the following: You can fix your module definition file so that it does not define attributes for segments two ways, or, make sure that the desired definition comes first in the file. Or, if the definitions are in two object modules, insure that the object module with the desired definition is processed first by the Linker.
4432	Duplicate segment attribute directives ignored this default data segment.
4433	Duplicate segment attribute directives ignored this segment flag.

Module Definition Utility Messages

Table A-3 contains Module Definition utility error messages. A common source of errors is incorrect syntax in the module definition input file for Statements or Segment attributes. For information on syntax, see Section 12, "Module Definition Statements."

Table A-3. Module Definition Utility Messages

Message	Explanation/Action
<attribute symbol> attribute has already been set.	The *<attribute symbol>* attribute has already been set for this segment.
<attribute symbol> ignored; LoadOnCall, ReadOnly, Shared, NoIOPL, Discardable, and Movable always set for Code segments	The *<attribute symbol>* attribute is one of the attributes set for Code Segments.
<attribute symbol> ignored; LoadOnCall, NoIOPL, Discardable, and Movable always set for Data segments	The *<attribute symbol>* attribute is one of the attributes set for Data Segments.
Code statement should only appear once.	A Code statement has been specified more than one time.
Data statement should only appear once.	A Data statement has been specified more than one time.
Import library not created; Input file did not contain Exports	An import library has not been created from the module definition input since there were no exported entry names specified.
Identifier truncated to *<truncated identifier>*	The maximum size of an identifier is 80 characters.
Internal entry name required with an ordinal entry.	An ordinal import entry name has been specified with no internal name specified to invoke it.

continued

Table A-3. Module Definition Utility Messages (cont.)

Message	Explanation/Action
Invalid LoadSpec in LoadType statement.	An invalid loadspec has been specified in the LoadType statement.
MinInstruction should be 8086, 80186, 80286, or 80386.	An invalid MinInstruction has been specified.
MinMathInstruction should be 8087, 80187, 80287, or 80387.	An invalid MinMathInstruction has been specified.
Multiply entered segment *<sement name>* class *<sement class>*	Segment *<sement name>* class *<sement class>* has been specified more than one time.
Ordinal value *<number>* defined for *<export-1>* and *<export-2>*.	The ordinal value *<number>* has been defined for two export procedures; their names are *<export-1>* and *<export-2>*.
'pwords' field ignored for xxx	The current implementation ignores *pwords* field. This is not an error.
'pwords' field required.	Although the current implementation ignores the *pwords* field, according to the syntax diagram, it should be here. This is not an error.
<symbol> has been defined as an initialization procedure, but not exported.	The procedure *<symbol>* must be defined in the list of exported procedures if it is to be accessed as an initialization procedure.
<symbol> is Multiply defined as an exported entryname.	The symbol *<symbol>* has been specified more than once as an EXPORT in an export definition.
<symbol> is Multiply defined as an internal entryname.	The symbol *<symbol>* has been specified more than once as an internal entry name in an export definition or in an import definition.
Token size exceeded: *<symbol>*	The token *<symbol>* has exceeded maximum token size of 255; only the first 50 characters of *<symbol>* are displayed.

Resource Librarian Messages

Table A-4 contains Resource Librarian error messages. A common source of errors is incorrect syntax in the Resource Librarian command form. For information on syntax, see Section 14, "Using the Resource Librarian Command Form."

Table A-4. Resource Librarian Messages

Message	Explanation/Action
Identifier truncated to *<truncated identifier>*	Maximum identifier size is 80 characters.
Internal error	An internal failure has occurred. If you observe such an error, report it to your technical representative.
Invalid add parameter *<copy of invalid add parameter>*	Parameter in *[Resources to add]* field of command form is incorrect.
Invalid delete parameter *<copy of invalid delete parameter>*	Parameter in *[Resources to delete]* field of command form is incorrect.
Invalid extract parameter *<copy of invalid extract parameter>*	Parameter in *[Resources to extract]* field of command form is incorrect.
Missing file name for add parameter *<copy of invalid add parameter>*. Missing file name for extract parameter *<copy of invalid extract parameter>*.	The *[Resources to add]* and *[Resources to extract]* fields require file names.
Missing Resource identifier in *<copy of invalid parameter>*	A "Resource ID" could not be found in the parameters input *<copy of invalid parameter>*.

continued

Table A-4. Resource Librarian Messages (cont.)

Message	Explanation/Action
Missing Resource Type in *<copy of invalid parameter>*	A "Resource Type" could not be found in the parameters input *<copy of invalid parameter>*.
Missing) in *<copy of invalid parameter>*	There should be a right parenthesis in parameter *<copy of invalid parameter>*
Resource Type is non integer: *<symbol>*. Resource Id is non integer: *<symbol>*.	*<symbol>* is not an integer. *<symbol>* is either misspelled or is not in the Resource Libarian configuration file.
Too many resources specified by wild card *<wild card parameter>*	The number of resources specified by the wild card *<wild card parameter>* exceeds the capacity of the Resource Librarian. Try enumerating them.

Appendix B
Run File Reference

Introduction

This appendix is a run file reference. The first part describes the fields in the run file headers. The second part of this appendix compares the run file formats of Version 8 and Version 6 run files.

Run File Header Fields

The header fields in Version 8, Version 6, and Version 4 run files are described in the tables that follow.

Version 8 Run File Header

The Version 8 run file header fields are described below.

Table B-1. Version 8 Run File Header Fields

Offset	Field	Size (bytes)	Description
0	wSignature	2	Run file signature = "WG"
2	ver	2	Run file format version = 08
4	cpnFile	2	Size of the data (in pages) in the run file after this header (does not include resources).
6	wRev	2	Run file format subversion number.
8	wFlags	2	Additional space for flags that may need to be passed to the loader.
10	wMinOSVersion	2	Minimum version of the operating system that must be running for this run file to load and run correctly.
12	wMaxOSVersion	2	Maximum version of the operating system that must be running for this run file to load and run correctly.
14	wPriority	2	Priority to be set for this run file when it is loaded.
16	bInstMin	1	Value that indicates the minimum assembly language instruction set that must be handled to execute this run file.
17	bMathMin	1	Value that indicates the minimum assembly language floating point math instruction set that must be handled to execute this run file.
18	cpnDir	2	Number of pages in the run file before the data element.
20	cpnHdr	2	Number of pages in the header (before the first run file table).
22	wChkSumHdr	2	Checksum of the header.

continued

Table B-1. Version 8 Run File Header Fields (cont.)

Offset	Field	Size (bytes)	Description
24	cModify	2	Number of times this run file was modified (such as by the **Debug File** command) since it was created.
26	snFirst	2	First (lowest) selector index in the run file.
28	cSn	2	Number of prototype descriptors.
30	fill	2	Reserved.
32	verAlt	2	Word of flags specifying the run file mode and how code and data are handled.
34	dateTime	4	Date and time the run file was created: this same value goes into the symbol file for this run file.
38	qbMinData	4	Minimum number of bytes of memory that must be available for the run file to be loaded.
42	qbMaxData	4	Maximum number of bytes of memory that will be allocated if available when the run file is loaded.
46	wCodeSet	2	Character code set used for nationalization.
48	cbHeap	2	Number of bytes in Dgroup Heap.
50	rgbFutureUse	30	Pad with nulls to paragraph boundary. Reserved for extensions.
78	cRfeDesc	2	Number of possible entries in the run file table descriptor array (36).

continued

Table B-1. Version 8 Run File Header Fields (cont.)

Offset	Field	Size (bytes)	Description
80	rgRfeDesc	36	An array of 12-byte element descriptors. Each descriptor describes an element of the run file and provides a checksum for it. The format of a descriptor is shown below:

Name	Size (bytes)	Description
plaElement	4	Paragraph element address
cparElement	4	Count of paragraphs
checksum	4	Checksum

Each element in a run file header is a single function table. The descriptor numbers are listed below. For a conceptual description of each table, see Table B-4 later in this appendix.

Index	Table Name
0	Register Initialization Table
1	Strings Table
3	Prototype Descriptor Table
4	Export Information Block
10	Import Information Block
11	Relocation Entries Table
12	Resident (Preload) data
14	Nonshared data
20	Resident (Preload) code
29	Resources data
30	Resources Descriptor Table

Version 6 Run File Header

The Version 6 run file header fields are described below.

Table B-2. Version 6 Run File Header Fields

Offset	Field	Size (bytes)	Description
0	wSignature	2	Run file signature = "WG"
2	ver	2	Run file version = 06
4	cpnRes	2	Run file size (in paragraphs).
6	iRleMax	2	Maximum relocation entry index (maximum number of fixups minus 1).
8	cParDirectory	2	Size of run file header (in paragraphs) including all tables.
10	cParMinAlloc	2	Minimum memory array size (in paragraphs).
12	cParMaxAlloc	2	Maximum memory array size (in paragraphs).
14	saStack	2	Initial stack segment: initial value put in the SS register.
16	raStackInit	2	Initial stack offset: initial value put in the SP register.
18	wChkSum	2	Run file checksum.
20	raStart	2	Initial code offset: initial value put in the IP register.
22	saStart	2	Initial code segment: initial value put in the CS register.
24	rbrgRle	2	Offset from the start of the run file to the start of the of Relocation Entries Table.
26	iovMax	2	Maximum overlay index: one less than the number of overlays.
28	saData	2	Initial value of the data segment: the initial value put in the DS register.

continued

Table B-2. Version 6 Run File Header Fields (cont.)

Offset	Field	Size (bytes)	Description
30	allFs	2	0FFFFh.
32	verAlt	2	Contains information that is passed to the loader, such as the run file mode, where data is to be located, and whether code is to be shared.
34	rbRgpIDiv	2	Offset from the start of the run file to the start of the IDIV Instruction Table.
36	cldiv	2	Number of entries in the table of IDIV Instruction Table.
38	qbDataMin	4	Minimum number of bytes of memory that must be available for the run file to be loaded.
42	qbDataMax	4	Maximum number of bytes of memory that will be allocated if available when the run file is loaded.
46	rbRgPdh	2	Offset from the start of the run file to the start of the table of segment prototype descriptors.
48	iPdhMax	2	Maximum Prototype Descriptor Table index.
50	rbRgRqLablE	2	Offset from the start of the run file to the start of the table to Call Gate Reference Table.
52	iRqLablEMax	2	Maximum Call Gate Reference Table index.
54	rbMpSnSa	2	Offset from the start of the run file to the start of the table of Selector to Segment Mapping Table.
56	iSnMax	2	Maximum Selector to Segment Mapping Table index.
58	snFirst	2	Selector index of the first prototype descriptor.
60	EndRunFile	4	lfa to the start of the legalese at the end of the run file. This is used by the resource compiler.
64	slData	2	First data segment selector.

continued

Table B-2. Version 6 Run File Header Fields (cont.)

Offset	Field	Size (bytes)	Description
66	cslData	2	Count of data segments.
68	slStack	2	Stack segment selector.
70	cslStack	2	Constant 1.
72	lfaSbVerRun	4	Offset from the start of the run file to the version string *SbVerRun* in DGroup.
76	dateTime	4	Date and time when run file was created: this same value goes into the symbol file for this run file.
80	cModify	2	Number of times this run file was modified (such as by the **Debug File** command) since it was created.
82	qbCodeMin	4	Reserved.
86	qbCodeMax	4	Reserved.
90	rbrgRlePStub	2	Offset from the start of the run file to the start of the Relocation Entries Table.
92	iPStubMax	2	Maximum Protected Mode Overlay Fixup Table index.
94	wOsVersion	2	Operating system version for the Linker command *[Run file mode]* option, Conditional CTOS Protected.
96	ParaPad	2	Pad with nulls to paragraph boundary.

Run File Formats

This subsection describes Version 8 and Version 6 run file formats. (For information on Version 4 run file formats, see Appendix E, "Version 4 Link Command.")

Version 8 Run File Format

Both executable application run files and dynamic link libraries have the same Version 8 run file format.

Not all Version 8 run files must have all run file tables described in Table B-4. The Linker creates a run file table only if it is needed.

Table B-4. Version 8 Run File Formats

Table Name	Description
Header	See the discussion earlier under "Run File Header Fields."
Register Initialization Table	Contains four byte values used by the loader to initialize the following hardware registers: EIP, EFL, EAX,ECX, EBX, EDX, ESP, EBP, EDI, ESI, EES, ECS, ESS, EDS, EFS, and EGS.
Strings Table	Stores strings. If the run file contains export or import information blocks, for example, these blocks contain fields that may refer to strings stored in this table.
Prototype Descriptor Table	Contains the data fields that will be placed in the system segment descriptor when the run file is loaded. The segment address field contains the lfa of the start of the segment in the run file. When the loader loads the segment into memory, it writes the memory address where the segment, identified by a selector, is loaded into this field. There is a prototype descriptor in the table for each separately addressable segment in the run file, including call gates. (For details on the format of the prototype descriptor table, see the Intel manuals listed in "About This Manual" following the Table of Contents in this manual.) The Module Definition utility can set all the flags in this table. (For details, see Section 12, "Module Definition Statements.")

continued

Table B-4. Version 8 Run File Formats (cont.)

Table Name	Description
Export Information Block	Consists of information the Linker builds into a DLL from a module definition file Exports statement. (For details, see Section 12, "Module Definition Statements.") This information is used by the loader to provide clients with addresses of procedures or data in the DLL. An export descriptor for each DLL procedure or data contains the address of a DLL procedure or data and an index (into the strings table) to the name string used to access the procedure.
Import Information Block	Consists of information the Linker builds into a client run file from a module definition file Imports statement or an import library. The loader uses this information to locate a DLL procedure a client calls. If the procedure is not already in memory, the loader loads the DLL the procedure is in. (For details, see Section 12, "Module Definition Statements.")
Relocation Entries Table	Contains the locations in the run file code or data to which the loader writes data when a run file is loaded. The data written can be the address of a segment in a long pointer or a reference to a DLL procedure. Because of paging, the loader fixes up only a single 4K-byte page of locations pointed to by the relocation entries at a time. To expedite loading and fixing up pages, relocation entries for any 4K-page are kept together.
Resident (Preload) Data	Contains all the data that is loaded when the run file is loaded. This data is loaded once. In the case of a DLL, the data is shared.
Nonshared Data	Contains all the nonshared data of a procedure. A copy of this data is created for each client that calls a DLL procedure accessing the nonshared data.
Resident (Preload) Code	Contains all the code that is loaded when the run file is loaded. The code is loaded once.
Resource Descriptor Table	Used by the resource management tools to access resources.
Resources Data	Contains resource data used by the resource management tools to access resources.

Version 6 Run File Format

Version 6 run files have the tables described in Table B-5.

Table B-5. Version 6 Run File Formats

Table Name	Description
Header	See the discussion earlier under "Run File Header Fields."
IDIV Instruction Table	Contains the locations of IDIV instructions. (This is needed to handle a bug in early versions of the 80186 chip.)
Prototype Descriptor	Contains the data fields that will be placed in the system segment descriptor when the run file is loaded. The segment address field contains the lfa of the start of the segment in the run file. When the loader loads the segment into memory, it writes the memory address where the segment, identified by a selector, is loaded. There is a prototype descriptor in the table for each separately addressable segment in the run file, including call gates. (For details on the format of the prototype descriptor table, see the Intel manuals listed in "About This Manual" following the Table of Contents in this manual.) The Module Definition utility can set all the flags in this table. (For details, see Section 12, "Module Definition Statements.")
Call Gate Reference	Contains locations of references to call gates (system-common procedure calls).
Selector to Segment Mapping	Contains the real mode segment addresses that correspond to the specified segment selectors.
Protected Mode Overlay Fixup	Contains the locations of selectors of procedures called in overlay programs.
Relocation Entries Table	Contains the locations in the run file code or data to which the loader writes data when a run file is loaded. The data written is the address of a segment in a long pointer.

Appendix C
Object Module Formats (OMF)

Introduction

This appendix lists all of the Intel object module formats (OMF) recognized by the Linker. The object module formats are described in the *MS-DOS Encyclopedia* and Intel documentation, which are listed in "About this Manual."

Table C-1. OMF Formats Recognized by Linker

Format Name	Acronym	Code	Class	Sub-Class	Source
Intel Link-86 product	RHEADR	6E			Intel
Register initialization	REGINIT	70			Intel
Block definition record to specify local symbols	BLKDEF	7A			Intel
Object module Header	THEADR	80			Intel
Intel pre v1.3Link-86 product	LHEADR	82			Intel
Absolute Data	PEDATA	84			Intel
Absolute Data Arrays	PIDATA	86			Intel
Comment Records	COMENT	88	None	None	Intel
Module end	MODEND	8A			Intel
External names definition	EXTDEF	8C			Intel
Public names definition	PUBDEF	90			Intel
Line numbers	LINNUM	94			Intel
Compiler generated names	LNAMES	96			Intel
Segment Definition	SEGDEF	98			Intel
Group definition	GRPDEF	9A			Intel

continued

Table C-1 OMF Formats Recognized by Linker (cont.)

Format Name	Acronym	Code	Class	Sub-Class	Source
Fixup data	FIXUPP	9C			Intel
Logical relocatable data	LEDATA	A0			Intel
Logical relocatable data (32-bit length)	DLEDATA	A1			Microsoft (1990)
Logical relocatable data arrays	LIDATA	A2			Intel
Common Data		C0			Microsoft (1982)
Communal names	COMDEF	B0			Microsoft C (5.0)
Local external names definition	LEXTDEF	B4			Microsoft C (5.0)
Local public names definition	LPUBDEF	B6			Microsoft C (5.0)
Local communal names	LCOMDEF	B8			Microsoft C (5.0)
Import definition	IMPDEF	88	A0	01	Microsoft for OS/2, Windows
Export definition	EXPDEF	88	A0	02	Microsoft for OS/2, Windows
Module name in library	LIBMOD	88	A3		Microsoft

Communal Name Records

The Microsoft C compiler generates two types of communal name (COMDEF) records: *Near* and *Far*. Near COMDEF records are generated for Small and Medium memory models. Far COMDEF records are generated Compact, Large, and Huge memory models. Only Far COMDEF records have an element size field.

The Linker generates three types of communal segments: *Near*, *Far*, and *Huge*. Near communal segments are created for near COMDEF records. Far communal segments are created for far COMDEF records where total size is less than 64K bytes. Huge communal segments are generated for far COMDEF records where total size is greater than 64K bytes.

Local Variables

The local variable records LEXDEF, LPUBDEF, and LCOMDEF, shown in Table C-1, are for local variables that are to be treated as external, public, and communal records, respectively. However, it is assumed that there are no constraints on the uniqueness of variable names, since they are local variables. The Linker does the following with variables specified in these records:

1. It creates unique internal names for Microsoft local variables.
2. It treats these variables as if they were in an EXTDEF, PUBDEF or COMDEF record, respectively.

Import and Export Definition Records

The import and export definition records are the EXPDEF and IMPDEF record types shown in Table C-1. They actually are subsets of (have the same or fewer fields than) the CTOS implementation of these records and are handled in the same way the CTOS versions for the Module Definition utility are handled. (For details, see "Import Definition Records" and "Export Definition Records" later in this appendix.)

Library Module Records

This record is the LIBMOD record type shown in Table C-1. It is used to set the flag that specifies that a module is from the Microsoft C compiler.

CTOS Module Definition Utility

The OMF formats emitted by the Module Definition utility are shown in Table C-2. They are used in Version 8 run files only. If one of these records is encountered while making a Version 4 or Version 6 run file, the Linker ignores the record.

Table C-2. OMF Formats From CTOS Module Definition Utility

Format Name	Acronym	Code	Class	Sub-Class	Source
Application name	APPNME	88	C1	00	CTOS for PM port
Library name	LIBNME	88	C1	00	CTOS for PM port
Attributes for code segments	CODATR	88	C0	03	CTOS for PM port
Attributes for DGroup segments	DGRATR	88	C0	04	CTOS for PM port
Individual segment attributes	SEGATR	88	C0	05	CTOS for PM port
CTOS Export definition	CTEXPD	88	C0	06	CTOS for PM port
CTOS Import definition	CTIMPD	88	C0	07	CTOS for PM port
Run file defaults	RUNDEF	88	C0	09	CTOS for PM port

Application and Library Name Records

The first two OMF records shown in Table C-2 are used by the Linker to determine whether an application run file (default) or a dynamic link library (DLL) should be built.

Segment Attribute Records

The OMF records for segment attributes are CODATR, DGRATR, and SEGATR:

CODATR — Allows the user to specify segment attributes for all segments of class "CODE".

DGRATR — Allows the user to specify segment attributes for all segments in DGroup.

SEGATR — Allows the user to specify segment attributes for a segment the name of which is specified. If it is not specified, the segment is assumed to be class Code.

These records pass choices about segment attributes to the Linker. All supported choices are shown in Table C-3 .

Table C-3. Segment Attribute Alternatives Used by the Linker

Attribute	Mnemonic Value	Segment Types to Which Attribute Applies
Code readability	ExecuteRead ExecuteOnly	Code segments
Code 286 conformance	Conforming Nonconforming	Code segments
Instance	Multiple (Instance) Single (Shared) None	DGroup data segments only
Shared	Shared Nonshared	Any data segments
Readonly	ReadOnly ReadWrite	Any segment

The Linker can do any of four things with this segment attribute information:

1. Use the data specified in the CODATR and DGRATR records to specify attributes for segments of class Code or in DGroup, respectively.
2. Check segments within a group for consistency of attributes. If a conflict is encountered, the link fails.
3. Use this data to determine which DLL segments are not intended to be shared. Nonshared segments are placed in a separate element in the Version 8 run file.
4. Enter this information in the appropriate flags fields in the Prototype Descriptors for the segments to which the data applies.

Export Definition Records

The CTEXPD OMF record type in Table C-2 provides the name of a procedure in a DLL and a way for a client to access that procedure, using either an ordinal number or a name. The Linker uses this information to create the Export Information Block in a Version 8 run file.

Import Definition Records

The import definition record type CTIMPD in Table C-2 provides a way for a client to access a DLL procedure. It gives the name of the DLL and either the procedure name recognized by the DLL or an ordinal value. (The ordinal is another way to identify an import and is used for portation.) The import definition record also contains the name by which the client accesses a procedure.

Run Type Records

The run type record RUNDEF in Table C-2 provides information on the following run file data:

1. Run-time priority
2. Minimum and maximum required operating system
3. Minimum instruction set needed by the run file
4. Minimum floating-point math instruction set needed by the run file
5. Valid load types are these values (the first four force the run file to be of protected mode):

   GDT
   Protected
   LowData
   HighMem
   CodeSharingServer
6. Stack size (bytes)
7. Heap size (bytes)

Appendix D
Calling Medium Model Procedures from a DLL

Introduction

The Intel medium model of computation works differently from the large model. In medium model, the value of the address of the Data Segment (DS) is equal to the value of the Stack Segment (SS); in large model these values can differ. This means that serious problems might occur if a client routine compiled in large model directly calls a function compiled in medium model. Since DLLs must be compiled in large model, this is of particular concern for programmers who write dynamic link libraries (DLLs)

Note that this discussion uses the term *large model* to refer to all those models of computation that do not assume equality of the DS and SS registers. Some examples of members of this group are the compact, large, and huge models of computation. *Medium model* is used to denote those models, which assume DS is equal to SS. Small and medium models are examples of this type.

DLLs are required to be linked from large model object modules. Therefore, they generally cannot be linked with medium model objects. Instead, all medium model objects need to be placed in their own special DLL. In this special DLL, each medium model procedure call is surrounded with code that "fixes" the incompatibility. A function surrounded by code that fixes the DS/SS equality problem is called a *mediated* function. Similarly, a DLL that contains a collection of these mediated functions is called a *mediated DLL.*

This appendix describes when to use mediation, gives step-by-step instructions on creating mediation code and linking mediated DLLs, and explains what happens at execution.

This description assumes that you already know how to build a DLL, as described earlier in this manual. For additional background information on this subject see your compiler manual and the *CTOS Operating System Concepts Manual.*

Large Model, Medium Model, and DLLs

A DLL is a type of a Version 8 run file, and, as such, it can have all the elements of a loadable executable entity: code, stack, and data. Public and static data usually are grouped together in one segment, the DLL's DGroup. (The Linker performs this grouping for every run file, including DLLs.) For this data to be accessible, DS must be set to the same value as the DLL's DGroup.

When an application run file is loaded, the operating system sets the values of CS, DS, and SS to the proper addresses. When the application calls a procedure contained in a DLL; however, the operating system leaves the current values of DS and SS. To access data that belongs to the DLL, therefore, the called procedure itself must set the value of DS to the value of the DLL's DGroup. At that point, DS and SS are not the same, so the code must not assume that DS and SS are the same. To insure that the code does not assume DS and SS are not the same, the objects must be compiled as large model. CTOS Microsoft C generates code to do this when a procedure is declared with the "_loadds" option.

Note that since the operating system does not change the values of SS and the routines within the DLL cannot change it (and still execute properly), the value of SS remains the same. This means that the procedures in the DLL use the client's stack to reference arguments and automatic (local) data. Therefore, a normal DLL needs no stack. Indeed, any stack space just wastes space. (Since a DLL should be large model, DS and SS are disjoint.) Later, we will see that a mediated DLL must have stack.

Until the advent of DLLs, it was unnecessary for any library code to be compiled in large model. Developers used medium model when they compiled the code that is put into the various libraries (for example, *ISAM.lib, Ctos.lib, Mouse.lib*, and so on.) Since it does not cause problems to do so in medium model, code generated by some less than careful compilers use SS and DS interchangeably. For example, in preparation to making a call to another function, a compiler may generate a "PUSH DS" instruction to push the segment address of an automatic variable.

When to Use Mediation

You should assume that all object module libraries contain procedures compiled in medium model, unless the software vendor specifically makes it clear that they can be called from CTOS DLLs. Specifically, Unisys object module libraries such as *Ctos.lib* 12.3 must be mediated and placed into a DLL before their procedures can be accessed from other, user-written DLL's.

Some newer libraries may not need to be mediated. Check the *Software Release Announcement* or other release documentation for the library for specific information.

Determining Which Procedures to Mediate

As stated above, the only procedures that you need to mediate are object module procedures. You can look at your DLL's map and tell which procedures are of what type; see Example D-1. When you link your DLL, enter **Yes** in the *[Publics?]* field of the Linker command form.

Example D-1. DLL Map File

```
Linker 12.2.0

Run file                  :NonMediated.dll
Link Start Time           :04/13/93 10:27:08

Config file               :LinkerConfig.Sys

Start       Stop        Length           Name        Class

00000000h   00000000h   0000h   (0084h)  _DATA       DATA(NonShared)
00000000h   00000000h   0001h   (0084h)  DATA        DATA(NonShared)
00000002h   00000002h   0000h   (0084h)  CONST       CONST(NonShared)
00000010h   00000013h   0004h   (0084h)  STATICS     CONST(NonShared)
00000014h   00000014h   0000h   (0084h)  _BSS        BSS(NonShared)
00000020h   00000020h   0000h   (008Ch)  ??SEG       ??SEG(NonShared)
00000020h   0000041Fh   0400h   (0094h)  STACK       STACK(NonShared)
00000420h   0000043Dh   001Eh   (009Ch)  CALL_TEXT   CODE
00000440h   00000474h   0035h   (009Ch)  FatalPro    CODE

Publics by name           Address                                Overlay

BEEP                      0000FFD0:0518h     (0080:0034h)        CallGate
CHECKERC                  00000042:0048h     (009C:0048h)        Res
ErrorExit                 0000FFEF:0334h     (00AC:800Ah)        CallGate
exit                      0000FFEF:0336h     (00A4:800Bh)        CallGate
FatalError                00000042:0020h     (009C:0020h)        Res
fDevelopement             00000000:0000h     (0084:0000h)        Res
sbVerRun                  00000000:0010h     (0084:0010h)        Res
TheCall                   00000042:0000h     (009C:0000h)        Res

Publics by value          Address                                Overlay

fDevelopement             00000000:0000h     (0084:0000h)        Res
sbVerRun                  00000000:0010h     (0084:0010h)        Res
BEEP                      0000FFD0:0518h     (0080:0034h)        CallGate
ErrorExit                 0000FFEF:0334h     (00AC:800Ah)        CallGate
exit                      0000FFEF:0336h     (00A4:800Bh)        CallGate
TheCall                   00000042:0000h     (009C:0000h)        Res
FatalError                00000042:0020h     (009C:0020h)        Res
CHECKRC                   00000042:0048h     (009C:0048h)        Res

No warnings detected

No errors detected
```

There are four possible values for the "Overlay" column. They are *Abs*, *Imp*, *CallGate*, and *Res*. *Abs* and *CallGate* denote procedures that do not need to be mediated. They are requests, system common, or kernel calls. *Imp* denotes an imported procedure (or data). They are procedures or data from another DLL. They do not need to be mediated. *Res* denotes resident code or data. These might need to be mediated, depending on what model of computation they were compiled.

You should be concerned with procedures, not data. This is explained a little later. Resident data can be readily differentiated from resident procedures. Data will have the selector value of the data segment. This can be found by looking at the "Class" column in the first part of the map. Large model DGroup consists of (at least) Data and Constants. As can be seen in the "Name" column of the Data class (and Const class), the selector value is 0084h.

The second part of the map file (headed by "Publics by name") identifies all the resident procedures. (Those publics that are overlay type *Res* and do not have a selector value of 0084h. This is the 4-digit value before the colon, in the third column.) In this example, the resident procedures are CheckErc, FatalError, and TheCall. (Note that public resident data are fDevelopment and sbVerRun, but you do not need to be concerned with these.)

Having identified the resident procedures, you can eliminate those that are compiled in large model (and hence do not need to be mediated).

In this case, the only procedure compiled in large model is TheCall.

Our list consists of CheckErc and FatalError, which need to be placed in a mediated DLL. If you want to check this, look up the calls in the *CTOS Procedural Interface Reference Manual.* The documentation will verify that they are indeed object module procedures.

The reason you do not need to be concerned with resident data is that all data associated with CheckErc and FatalError will go away (to the mediated DLL) when the procedures are placed in the mediated DLL.

Using a New Mediated DLL

After removing the appropriate procedures to a mediated DLL (see the next section), your DLL must link with the DLL's import library. This is done by adding the library name to the *[Libraries]* field in the Linker command form or in the *:LibraryReferences:* entry in the Linker configuration file.

As the last step, use the map file again to verify that the only resident procedures are those that you wrote. The mediated procedures will have an overlay of type *Imp*.

Creating a New Mediated DLL

This procedure describes how to set up a mediated DLL when the DLL is first created.

The procedure described below shows an example that creates a mediated DLL called *Ctos.dll*. It is suggested you follow the same naming conventions used in the example, replacing "*Ctos*" with the name of the library you are mediating.

The Development Utilities installs the following files in the directory *[Sys]<Mediator>*:

MediatorFirst.obj
Mediator.asm
Mediator.mdf
Mediator.def

To create a mediated DLL:

1. Start the Editor.
2. In the Editor, copy the *Mediator.asm* file to a new file, for example *CtosMediator.asm*.
3. Open the new file.

4. *CtosMediator.asm* must contain a macro for each procedure you want to place in your mediated DLL.

   Add one new line for each procedure you intend to export . The format is

   `%Mediate(`*`ProcedureName, N`*`)`

   where

   N is the total number of bytes in the parameter list for the procedure. Parameters that are described as a byte should be counted as two (2) bytes because parameters are always pushed onto the stack with word-alignment. *N* should never be an odd value.

   ProcedureName is the name of the procedure.

   Example:

   ```
   %Mediate(CloseByteStream, 4)
   %Mediate(OpenByteStream, 24)
   ```

5. Assemble the file, creating a new mediator object file, as shown in the example below.

   ```
   Assembler
     Source Files                        CtosMediator.Asm
     [Errors only?]                      ________________
     [Macro expansion (GenOnly)]         ________________
     [object file]                       CtosMediator.Obj
     [Print file                         ________________
     [Error file                         ________________
     [List on pass 1?                    ________________
     [:f1:                               ________________
     [:f0:(Sys]<Edf>)                    ________________
   ```

6. Using the Editor, copy *Mediator.def* to a new file, for example *Ctos.def*.

7. Open the new module definition file.

8. Change the entry MediatedDll, as shown below, to use the name of your DLL. We are using *Ctos* as the example here.

```
LIBRARY MediatedDll INITINSTANCE InitMediator
```

becomes

```
LIBRARY Ctos INITINSTANCE InitMediator
```

9. Add one new entry to the EXPORTS section for each of the procedure names that you added to mediator file in step 4. The format is

```
ProcName = 1_ProcName
```

where

ProcName is the name of the procedure, and

```
1_ProcName
```

is the name of the mediated procedure which eventually calls *ProcName*. *1_ProcName* was created when you assembled *CtosMediator.Asm*.

Clients of the mediated DLL call CloseByteStream, but (transparently to the client,) the DLL routes the call to 1_CloseByteStream which in turn calls CloseByteStream.

Note: *Mediation requires that the mediated DLL's stack segment be big enough to accommodate all calls that any single one of the medium-model procedures you include might make. The* Mediator.def *file sets stack size to 4096 to allow for this. This is the recommended size for CTOS III. If you experience problems in using the medium-model routines which you think may be stack related, try increasing 4096 to a larger value.*

Note also that reducing the size of this parameter does not actually reduce memory requirements since the paging service allocates memory in 4K-byte increments even if only one byte is needed.

10. Use the **Module Definition** command to create the object and imports files.

```
Module Definition
  Input file (.def)              Ctos.def
  [Object file]                  CtosDef.obj
  [Imports file]                 CtosImp.Lib
  [List file]                    CtosDef.list
  [Suppress warnings?]           ____________
  [Suppress ordinals?]           ____________
  [Upper case?]                  ____________
```

11. Link the DLL. A sample command form is shown below:

```
Link V8
  Object modules                 @LinkCtosDll.fls
  Run file                       Ctos.dll
  [Map file]                     ____________
  [Publics?]                     ____________
  [Line numbers?]                ____________
  [Stack, Dgroup heap size]      ____________
  [Max array, data]              0 0
  [Min array, data]              0 0
  [Run file mode]                Protected
  [Version]                      ____________
  [Libraries]                    (CtosDll)
  [DS allocation?]               ____________
  [Symbol file]                  ____________
  [Copyright notice?]            ____________
  [File to append]               ____________
  [Linker config file]           ____________
```

LinkCTOSDll.fls contains the following object module files:

MediatorFirst.obj, *CtosMediator.obj*, *CtosDef.obj*

Ensure that the segment ordering module, *MediatorFirst.obj*, is listed first in the *Object modules* field of the **Link V8** command.

In the example above the file *LinkerConfig.sys* contains:

```
:LibraryReference:CtosDll
:LIbraryFile:[Sys]<Sys>Ctos.lib
:LIbraryFile:[Sys]<Sys>CtosToolKit.lib
:LIbraryFile:[Sys]<Sys>Enls.lib
```

Note: *For some compiler-specific object modules, the Linker sets* [DSAllocation?] *to Yes. Therefore, it is recommended that you explicitly set* [DSAllocation?] *to* **No,** *even though this is the default.*

What Does the Mediator Do?

The mediator changes the large model environment (where DS doesn't equal SS) to a medium model environment. It does this by setting DS and SS equal to the value of the mediated DLL's DGroup. Note that this implies that the mediated DLL must have some stack. The following discussion gives details.

Mediated procedures must be kept in a mediated DLL. Mediated procedures cannot be linked into a client DLL. Instead, the client DLL links with the mediated DLL's import library (in the previous example, *CtosImp.lib*.)

In the previous example given for the procedure, an envelope is created in *CtosMediator.obj* for each procedure listed in *CtosMediator.asm*. These are each named l_ProcName, where ProcName is the name of the medium model procedure.

The mediator consists of a series of steps that are sandwiched around the call to the actual medium model procedure. In effect, the mediator changes the large model environment into a medium model environment in which the medium model procedure can be executed, and restores it upon completion.

For example, a function in the client DLL calls CloseByteStream. *Ctos.dll* is loaded and the address of the call is fixed up to the address of l_CloseByteStream. The reason this occurs is that when the client DLL was linked with *CtosImp.lib*, the Linker put in information that equated CloseByteStream with l_CloseByteStream. Control then passes to l_CloseByteStream which:

1. Saves DS and SS.
2. Changes DS to the DGroup of *Ctos.dll*.

   This is necessary because the medium model procedure may use module-level data. This data is made part of the DGroup for the DLL at link time.
3. Locks out other callers by calling SemLock.
4. Copies parameters from the caller's stack to the *Ctos.dll's* stack. This allows the mediator code to change the value of SS to that of *Ctos.dll*.

   Since the *Ctos.dll* stack segment is part of its DGroup, SS will equal DS, thus allowing the medium model code to execute correctly.
5. Sets the value of SS:SP to the proper address in the *Ctos.dll* stack.
6. Calls the medium model function CloseByteStream.

After CloseByteStream returns, the following steps are executed:

1. Restores the caller's stack, including restoring DS and SS:SP to the original values.
2. Releases the semaphore acquired in step 3 by calling SemClear.

Control then returns to the client DLL.

Appendix E
Version 4 Link Command

Introduction

This appendix describes those features of the **Link** command and related files where there are differences from the **Link V6** and **Link V8** commands. (For information on **Link V6** and **Link V8**, see Section 3, "Using the Linker Command Forms.")

When to Use the Link Command

It is generally recommended that you use the **Link V8** (Version 8) or **Link V6** (Version 6) commands instead of the **Link** (Version 4) command, since these commands allow you to take better advantage of the newest features of the operating system.

However, the **Link** command is supported for backwards compatibility. You can use Link to link real mode programs that run on all of the following systems:

- Virtual memory operating systems
- Protected or real mode workstation operating systems without virtual memory
- CTOS/XE real mode processors

Table E-1 shows the capabilities of the **Link** command in detail.

Table E-1. Capabilities of the Link Command

Linking Command	Link V4
Creates	V4 Run File
Executes On	CTOS I BTOS XE/BTOS CTOS/SRP CTOS II BTOS II CTOS/VM CTOS/XE CTOS III
Execution Mode	Real
DLL Support	No

Link User Interface

As explained in Section 3, there are three command forms: **Link V8**, **Link V6**, and **Link**. Each produces a different run file. However, if you already know about one command, you will be familiar with the others even if you haven't used them. The few differences in the command form fields are discussed below.

Differences Between Linker Command Forms

- The command form names are **Link V8, Link V6**, and **Link**. The names are different because each command is a different command case (produces a different default run file type).
- The **Link** command contains the parameter fields *[Max memory array size]* and *[Min memory array size]* in place of the **Link V8** and **Link V6** command parameter fields *[Max array, data]* and *[Min array, data]*.

All the other command form parameter fields are the same. A description of each field is given in Section 3, "Using the Linker Command Forms."

Validity of Run File Mode Parameters With the Link Command

Table E-2 indicates whether or not a run file mode parameter is valid with a Link command form. The value **Yes** means that the keyword is valid with the Link command form. The value **No** means that the keyword is invalid.

Table E-2. Validity of Run File Mode Parameters with Link

Run File Mode Parameter	Valid
Keyword	
V4	Yes
No	Yes
Yes	Yes
Real	Yes
Protected	No
GDTProtected	No
HighMemGDTProtected	No
HighMemProtected	No
LowDataGDTProtected	No
NRelProtected	No
CodeSharingServer	No
HighMemCodeSharingServer	No
SuppressStubs	Yes
PMOS	No

Using Overlays

For Version 4 run files, the Linker can create overlays for use with the virtual code management facility.

Under CTOS III, all run files are paged into memory on demand. Under CTOS III, the virtual code (overlay) segments of Version 4 run files are treated the same as the resident segments; they are paged in on demand. (For more information on the paging service, see the *CTOS Operating System Concepts Manual.*)

For information on how to create overlays, see "Examples" in Section 3 and "Program Memory Requirements" in Section 6.

Grouping Segments

The Linker combines all the segments of a group into one segment, which is addressed with one selector. However, 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.

Version 4 Run File Formats

Link creates a Version 4 run file by default. A Version 4 run file is executed in real mode only. For this reason, its run file format is fairly straightforward. (It is similar to the Version 6 run file format, but it has fewer tables.) See Figure E-1. The run file header consists of regions 1 and 2 in the figure. These regions correspond to the numbered comments below:

1. The run file header fields are described in detail later in Table E-4.
2. Two tables in the header are an IDIV Instructions Table and the Relocation Entries Table. (See Table E-3 for a description of each of these tables.)

Following the run file header, the run file code and data start. If there are any overlays, these are contained in the overlay region beyond the code and data.

Figure E-1. Version 4 Run File Format

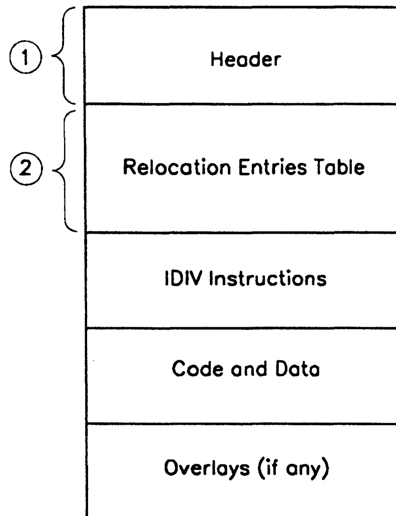


Table E-3. Version 4 Run File Header Tables

Table Name	Description
Header	See Table E-4 below.
IDIV Instructions Table	Contains the locations of IDIV instructions. (This is needed to handle a bug in early versions of the 80186 chip.)
Relocation Entries Table	Contains the locations in the run file code or data to which the loader writes data when a run file is loaded. The data written is the address of a segment in a long pointer.

Version 4 Run File Header

The Version 4 run file header fields is shown in Table E-4.

Table E-4. Version 4 Run File Header Fields

Offset	Field	Size (bytes)	Description
0	wSignature	2	Run file signature = "WG"
2	ver	2	Run file version = 04
4	cpnRes	2	Run file size (in paragraphs).
6	iRleMax	2	Maximum relocation entry index (maximum number of fixups minus 1).
8	cParDirectory	2	Size of run file header (in paragraphs), including all tables.
10	cParMinAlloc	2	Minimum memory array size (in paragraphs).
12	cParMaxAlloc	2	Maximum memory array size (in paragraphs).
14	saStack	2	Initial stack segment: initial value put in the SS register.
16	raStackInit	2	Initial stack offset: initial value put in the SP register.
18	wChkSum	2	Run file checksum.
20	raStart	2	Initial code offset: initial value put in the IP register.
22	saStart	2	Initial code segment: initial value put in the CS register.
24	rbRgRle	2	Offset from the start of the run file to the start of the Relocation Entries Table.
26	iovMax	2	Maximum overlay index (one less than the number of overlays).

continued

Table E-4. Version 4 Run File Header Fields (cont.)

Offset	Field	Size (bytes)	Description
28	saData	2	Initial value of the data segment: initial value put in the DS register.
30	allFs	2	0FFFFh.
32	verAlt	2	0000h.
34	rbrglDiv	2	Offset from the start of the run file to the start of the IDIV Instruction Table.
36	cldiv	2	Number of entries in the IDIV Instruction Table.
38	ParaPad	10	Pad with nulls to paragraph boundary.
48	rgrleTable		Relocation Entries Table Its size is [4 * cldiv] bytes. It starts at an offset of rbrglDiv = 48 + [4 * (iRleMax + 1)] bytes.
	IdivTable		IDIV Instruction Table. starts here. Its size is 4 * (iRleMax + 1)] bytes.

Version 4 Map File

The Linker produces a map file containing information about the link. You can specify whether or not the map file displays public symbols or line numbers.

A Simple Map File

If you use the default values for *[Publics?]* and *[Line numbers?]* in the Link command form and omit the Linker configuration file entry *:Details:*, the Linker produces a simple map file. (Another way to generate this map file is to enter the default value **No** for each parameter just mentioned.)

Example E-1 shows a simple map for a Version 4 run file. It consists of three main components:

- Segment entries describing the size, location, and name of each segment
- The program entry point
- A breakdown of the warnings and errors detected in the link

Addresses

From left to right in Example E-1, the first three columns show the beginning and ending addresses and the length of each segment. The beginning addresses under the column heading *Start* are offsets. 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.

Example E-1. Sample Version 4 Map File

```
Linker (version)

Run file        : V4>Sample.run
Link Start Time : 01/30/92 13:51:31

Config File     : TestConfig.sys

Start    Stop     Length Name        Class

00000h  00000h  0000h  ??SEG        ??SEG
00000h  00000h  0000h  MEMORY       MEMORY
00000h  007FFh  0800h  STACK        STACK
00800h  00810h  0011h  DATA         DATA
00812h  0081Dh  000Ch  CONST        CONST
00820h  00822h  0003h  STATICS      CONST
00830h  0086Dh  003Eh  SAMPLE_CODE  CODE
00870h  008A4h  0035h  FatalPro     CODE

Program entry point at 0082:0002

No warnings detected
No errors detected
```

Names

The fourth column in Example E-1 lists the name of each segment. Note that in the example case of the name SAMPLE_CODE, class Code segment, the name shown is not the file name of the module.

In most high-level language programs, you assign the module name at the beginning of the module. The compiler creates the code segment name by appending the suffix *_CODE* to the assigned module name. The resulting name is reported in the map file by the Linker.

In assembly language, you can directly name each segment as you wish. The Assembler 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, because file names, and not internal module names, are entered in the *Object modules* field of a Link command form. You are not required to use this convention, however.

Classes

The fifth (rightmost) column in the map lists the class of each segment. The Linker groups segments by class and uses class to assign order in the program.

Map Files With Public Symbols, Line Numbers, and Details

Example E-2 shows the same basic map file just shown in Example E-1. Now, however, the map file shows public symbols, line numbers, and other details about parameter information. You can generate this additional information by specifying **Yes** for the **Link** command line parameters *[Publics?]* and *[Line numbers?]* and for the configuration file entry *:Details:*.

Library References

Example E-2 shows library reference information. This information appears in the map file when you enter **Yes** for the *:Details:* option in the Linker configuration file. Library information can include

- The library name and page size (in bytes) for each library the Linker searches
- The library version, if available

Public Symbols

Following the segment entries, public symbols are first sorted alphabetically and then numerically. To request that public symbols be displayed, enter **Yes** for the *[Publics?]* parameter in the Link command form.

Example E-2. Sample Map for a Version 4 Run File Showing Lists of Public Symbols, Line Numbers, and Details (Part 1 of 3)

```
Linker (version)

Run file         : V4>Sample.run
Link Start Time : 01/30/92 13:51:31

Config File      : TestConfig.sys

Library Reference:(Default) from file TestConfig.sys

  Library: [Sys]<Sys>ENLS.Lib
    Block size: 00512
    Version:    x13.0.E (tuesday january 7, 1992, 16:16)

  Library: [Sys]<Sys>Ctos.lib
    Block size: 00512
    Version:    x13.0.E (tuesday january 7, 1992, 16:12)

  Library: [Sys]<Sys>CtosToolKit.lib
    Block size: 00512
    Version:    x13.0.E (tuesday january 7, 1992, 16:14)

Start       Stop        Length      Name          Class

00000h      00000h      0000h       ??SEG         ??SEG
00000h      00000h      0000h       MEMORY        MEMORY
00000h      007FFh      0800h       STACK         STACK
00800h      00810h      0011h       DATA          DATA
00812h      0081Dh      000Ch       CONST         CONST
00820h      00822h      0003h       STATICS       CONST
00830h      0086Dh      003Eh       SAMPLE_CODE   CODE
00870h      008A4h      0035h       FatalPro      CODE
```

Example E-2. Sample Map for a Version 4 Run File Showing Lists of Public Symbols, Line Numbers, and Details (Part 2 of 3)

```
Publics by name              Address          Overlay

AllocPSub                    FF1F:103Ah       Abs
AltDmaMapBufferFast          FE5F:1C32h       Abs
AltDmaUnmapBuffer            FE5F:1C34h       Abs
AltMapDmaBufferFast          FE5F:1C32h       Abs
AltUnmapDmaBuffer            FE5F:1C34h       Abs
AsiaNub                      FE7F:1A3Eh       Abs
AssignKbd                    FFEF:0330h       Abs
AssignVidOwner               FFEF:033Eh       Abs
BitBlt                       FFCF:0532h       Abs

Publics by value             Address          Overlay

SystemCommonConnect          FDFF:2230h       Abs
ConnectProcedure             FDFF:2230h       Abs
SetVideoLocators             FE3F:1E30h       Abs
GetModuleAddress             FE4F:1D30h       Abs
UpdateStatistics             FE5F:1C30h       Abs
CodeIkbd                     FE6F:1B30h       Abs
DiscardLocalPageMap          FE7F:1A30h       Abs
SemQuery                     FE8F:1930h       Abs

Line numbers for SAMPLE_CODE

00002  0083:0000H  00005  0083:0000H  00008  0083:0000H
00010  0083:0000H  00013  0083:0000H
00015  0083:000DH  00016  0083:0010H  00017  0083:002DH
00018  0083:003CH  00019  0083:0000H
00020  0083:0008H  00021  0083:000DH
Program entry point at 0083:0000
```

Example E-2. Sample Map for a Version 4 Run File Showing Lists of Public Symbols, Line Numbers, and Details (Part 3 of 3)

```
Linker Details

Linker Information:

Run file         : V4>Sample.run
Run file format  : Version 4
Stack            : 02048
Run file mode    : 0001h; Real   (default)
CharacterCodeSet: SingleByte

Version          : xC
Max array = 00000
Min array = 00000
DS Allocation not used(default)

Class ordering requested       :
  ??SEG  MEMORY  STACK  DATA  CONST  CODE

Configurable Linker work areas:
Area Name    Sectors Used  Max Sectors
RgIdct       00000         00032 (default)
RgPdhCG      00000         00032 (default)
RgRlePStub   00000         00032 (default)
RgRle        00001         00127 (default)
WorkingData  00044         01024 (default)
RgRqLable    00000

00127 (default)

Link End Time    : 01/30/92 13:51:36

No warnings detected
No errors detected
```

The Address column in Example E-2 contains the notation *XXXX:YYYY*h. This is the hexadecimal address of the public symbol.

The Overlay column contains *Res* if the symbol is in the resident portion of the program, an integer (*n*) if it is in the *n*th overlay, and *Abs* if it is a call gate to an operating system procedure.

Line Numbers

The public symbol lists are followed by a list of line numbers. To request line numbers separately, enter **Yes** in the *[Line Numbers?]* field in the Link command form.

Line numbers are intended for use during debugging. They allow you to examine a known part of a 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 to use them.

Command Form Parameter Details

Towards the end of the map file in Example E-2, there are other details about Link command form parameters. (See the portion of the map entitled "Linker Information." It is located between the program entry point line and the error and warning messages at the very end of the map.) These details are displayed along with the library reference information when you specify **Yes** for the *:Details:* option in the Linker configuration file.

Command form parameter details can include the following information:

Run file version
Stack size
Heap size
Maximum array
Maximum data
Minimum array
Minimum data
Run file mode
Version (as specified on the command line)
Whether DS Allocation is used

Configurable Linker Work Areas

Finally, just before the error and warning messages at the very end of the map file is the portion of the map called "Configurable Linker work areas," This portion contains information about the Max Table parameters that you can define in the Linker configuration file.

Glossary

A

absolute segments

An absolute segment that has a negative frame value is considered to be a call gate. Each system common, kernel and request procedure called in a program receives a separate segment.

An absolute segment whose frame value is zero or greater is considered to be an absolute segment at the physical segment address specified by the frame value.

absolute symbol

A symbol that is a call gate to an operating system procedure.

alignment attribute

Specifies whether a segment element can be aligned on a byte, word, or paragraph boundary.

archive (POSIX/UNIX)

An archive file or archive library is a collection of data gathered from several files. Each of the files within an archive is called a member. The command ar(1) collects data for use as a library.

C

Character Code Set

An entry in the runfile header that specifies the character set designator for the application. It may be specified to the Linker using *LinkerConfig.sys*. Character Code Set values may specify character sets such as Japanese, Korean, Chinese, or single byte (the default).

class name

An arbitrary symbol used to designate a class.

CodeSharingServer

The runfile mode that allows multiple installations of the same server on the same processor to share code (local descriptor table-based only).

CodeView

A window-oriented menu-driven debugger implemented by Microsoft that allows tracing and debugging for high-level-language and assembly-language programs.

Conditional Protected

The run-file mode that specifies that the run file runs protected if the internal operating system version is the specified level or greater; otherwise, it runs in real mode.

D

DGroup

The name of the group of logical segments that make up the automatic data segment. Each of the logical segments stores a specific type of data. Typical contents of DGroup may include initialized external and static data, the stack, constants. *See also* group.

DS allocation

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

Dynamic Link Library (DLL)

A dynamic linked library is a loadable file that contains code that an application can call and cause to be run. The code in a dynamic linked library would in the past have had to be linked into the application. The format of a dynamic linked library is NOT the same as the format of a CTOS object module library. The format of a dynamic linked library IS identical to the format of a version 8 runfile.

E

export

A procedure in a dynamic link library (DLL) that may be used by another program or DLL.

external reference

A reference from one object module to variables and entry points of other object modules.

extraction

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.

F

fixups

Sometimes a compiler or an assembler encounters addresses whose values cannot be determined from the source code. One example is addresses of external symbols.

Addresses for fixups can be determined either at link time or load time.

Typically, a translator emits fixup information for the Linker to determine the correct address. The Linker can fully resolve all references except those that refer to the segment address of a relocatable segment.

Since run-file code and data is relocatable, the Linker does not have enough information to determine the segment location portion of a long address. The Linker fills in all known portions of the address and records the location of the indeterminate portions. When the segment is loaded, the Loader has the information needed to complete the address. The Loader "fixes up" the address by writing the address at the location recorded by the Linker.

G

GDTProtected

The run-file mode of a run file produced by the Linker can be executed in protected mode under protected mode version of the operating system and will get its segment selectors from the global descriptor table.

global data segment

A data segment defined by a dynamic-linked library module, one copy of which is shared by all client processes.

global descriptor table (GDT)

A table that is used to address segments within a protected mode system. Within the GDT, there are segment descriptors that describe all of the segments that comprise a system's global address space. There is always one GDT, which is always in effect. See the *iAPX 286 Programmer's Reference Manual* and the *80386 Programmer's Reference Manual.*

global initialization

DLL initialization procedures called once by the first DLL client.

group

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.

The run-file mode of a run file that can be executed on the NGEN Series 386i processor in memory configurations that exceed 16M bytes.

H

HighMem

The runfile mode of a runfile that can be executed on the NGen Series 386i processor in memory configurations that exceed 16Mb.

HighMemCodeSharingServer

The runfile mode that allows multiple installations of the same server on the same processor to share code (local descriptor table-based only). Can be executed on the NGEN Series 386i processor in memory configurations that exceed 16M bytes.

HighMemGdtProtected

The run-file mode of a run file that can be executed in protected mode under protected mode version of the operating system, will use the global descriptor table, and can be executed on the NGEN Series 386i processor in memory configurations that exceed 16M bytes.

HighMemProtected

The run-file mode of a run file that can be executed in protected mode under protected mode version of the operating system, will use the local descriptor table, and can be executed on the NGEN Series 386i processor in memory configurations that exceed 16M bytes.

I

import

A procedure call. When invoked, it accesses code in a DLL.

import library

A library file that contains the names of dynamic link libraries (DLLs) and their procedures that can be called by a client. There is a separate object module for each procedure. Each module contains the name of the DLL and the procedure. The Linker uses this library to resolve external references to DLL procedures in client programs.

instance initialization

DLL initialization procedures called each time a client calls a DLL procedure.

L

Librarian

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.

library block

A basic library unit. Each object module in a library is aligned with the beginning of a block. The default block size is 512 bytes, but the size can be changed through a parameter to the Librarian command.

Linker

A program development utility that combines object modules (files produced by compilers and assemblers) into run files.

Linker segment

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

Loader

A part of the operating system that reads a run file, copies part of it to memory, gives it the resources it needs to execute, and puts it on the run queue.

local descriptor table (LDT)

A table that is used to address segments; it is maintained for each program executing in protected mode. Each code or data segment in the program has a unique selector (an index into the table) and a corresponding unique entry in the LDT. The LDT is an array of these entries, called descriptors, which are eight bytes long and contain various information about the segment. Each program has its own LDT, which is in effect when the program is executing. See the *iAPX 286 Programmer's Reference Manual* and the *80386 Programmer's Reference Manual.*

LowDataGdtProtected

The runfile mode that specifies that the runfile's data should be made accessible to Real mode programs. The runfile produced by the Linker can be executed in Protected mode under Protected mode version of the operating system and will use the global descriptor table.

M

map file

A file, created by the Linker, that 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, library references specified in the Linker configuration file, and information on other options specified in a link.

memory array

Data space allocated at load time above the highest address of a program.

module

Generally means an object file, the contents of an object file within a library file, or a DLL itself.

module definition file

A file that defines the specific requirements of a dynamic link library (DLL) or DLL client.

N

Nonshared segment

A data segment defined by a dynamic-link library that is not shared by multiple client processes; the system loads a separate copy of an Nonshared data segment for each new client. The selector for such a segment comes from the client's LDT. All copies of this segment have the same selector value. Every time a procedure in a dynamic linked library with an Nonshared segment is called, a copy of the Nonshared segment is made for the caller's exclusive use. The converse is shared data.

NRelProtected

The run-file mode that specifies that the run file should be run only in Protected mode. Since the relocation reference table is not required in the run file header in Protected mode, specifying NRelProtected can reduce the size of the run file header by the size of the relocation reference table.

O

object module

A file that 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*).

Object Module Format (OMF)

The set of data formats of the various types of records that can make up an object module file. (See Appendix B.)

object module libraries

An object module library is a file that contains distinguishable object modules and a table that lists public names and the location in the library of the module in which they occur. This is what historically the term CTOS library has meant. An object module library is created by the Librarian from one or more object modules.

offset

The relative distance in bytes of a memory location from the beginning of a segment.

overlay

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 resident in memory. Overlays are not supported in V8 and V6 run files on CTOS V Series operating systems: the paging service swaps program sections into and out of memory instead.

P

page

A 512-byte section of memory or a file; also called a sector.

paragraph

A 16-byte section of memory or a file.

Protected Mode Operating System (PMOS) server

A product that makes it possible for real mode versions of CTOS to run servers in protected mode. PMOS is included in this manual for historic reasons only. For details, see your PMOS documentation.

protected virtual address mode (protected mode)

A mode of operation for the 80286 and 80386 microprocessors. It determines how memory is addressed. In protected mode you can address more than 1M byte of memory, and programs are prevented from accidentally using or modifying the code or data of other programs. 80286 and 80386 microprocessors can operate in either protected mode or real mode.

Prototype Descriptor

A data structure used by the protected mode operating system that contains a prototype of the hardware System Segment Descriptor. The information contained in the data structure includes the segment base (i.e., the memory address at which the segment starts), the size of the segment, and the segment usage flags. In the run file, the segment base field of the prototype of the System Segment Descriptor contains the address (lfa) in the run file where the segment starts.

Note that the memory address at which the segment starts is not known when the run file is created; this data is written in by the Loader when it loads the run file. Therefore, anything in the segment base fields in a prototype descriptor in the runfile is overwritten when the prototype descriptor is loaded into memory. Since this data is lost, the segment base fields in a prototype descriptor for a segment in the runfile are used to hold the logical file address in the runfile of the start of that segment.

Some prototype descriptor information may be specified in the module definition file.

public symbol

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

public variable

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

R

real address mode (real mode)

A mode of operation for the 8086 and 80186 microprocessors. It determines how memory is addressed. In real mode, you can address up to 1M byte of memory. 8086 and 80186 microprocessors operate in real mode at all times. 80286 and 80386 microprocessors can operate in either real mode or protected mode.

relocation

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

relocation directory

An array of locators used by the operating system in relocating the program image.

resident

The resident portion of a program remains in memory throughout execution. *See also* overlay and Virtual Code Management.

resource data

An item of data that is not part of the runfile proper but is inserted into the run file so that file transfer processes (i.e., Copy, Backup, Restore, etc.) that move the run file also move the resource. Resources are added to a file, deleted from a file or edited by the Resource Librarian. Examples of resources could include the symbol file, a nationalization file, etc.

Resource Descriptor Table

A table that contains a information about resources inserted into a file. It also specifies the type of resource, the offset to the start of the resource and the number of bytes in the resource.

run file

The image of a program (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.

S

sector

A 512-byte section of memory or a file.

segment

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

Shared Segment

Only one copy of this type of segment is made by the loader when a runfile is loaded. All users that access the runfile access this one copy of the segment. A shared segment can contain either code or data. The converse is a NonShared segment.

short-lived memory

An area of memory in an application partition. When a program 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.

U

unresolved external reference

A public symbol that is used by some module but not defined by any of the modules being linked.

V

Virtual Code Management

A method of virtual memory supported by the CTOS operating systems. (On Virtual Memory CTOS III systems, the paging service performs this function instead if the run file is a Version 6 or Version 8. In these cases, calls to virtual code management are ignored.) The code of each program is divided into variable-length segments that reside on disk in a run file. As the program 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.

Index

B

binary resource file, 13-5, 13-6

C

class order
  configuration file, 6-16
  specifying, 6-16
code segment, 6-9
Code statement, 12-2, 12-4
  example, 12-5
  parameters, 12-5
communal name records, C-4
  import and export definition, C-4
  library module, C-4
  local variables, C-4
communal segment
  far, C-4
  huge, C-4
  near, C-4
cross-reference listing
  example, 8-5
  parts, 8-4
CTOS Microsoft C, D-2
  stack segment (SS), D-1

D

data file, 13-5, 13-6
data segment (DS), D-1
Data statement, 12-2, 12-6
  example, 12-7
  parameters, 12-7
Description statement, 12-2, 12-8
  example, 12-8
  parameters, 12-8
development utilities, 1-1, 1-2
DGroup, 3-7, 4-3, 5-11, 6-6
  DS register, 5-11
  errors, A-5, A-12
  locating, 3-15
  uninitialized variables, 5-11
DLL, 13-5, 13-6, 5-1, D-1, D-2
  accessing, 9-3, 9-4
  building, 9-1
  calling medium model procedures, D-1
  creating, 9-3
  creating a mediated, D-6
  creating client interface, 11-3
  large model, D-1, D-10, D-2
  map file, D-4
  mediated, D-1, D-6
  medium model, D-1, D-10, D-2
  porting, 9-1
DLL client, 9-6
DLL procedures, 9-6
DS allocation, 3-5, 6-3, 6-5, 6-6
  advantages, 6-7
  errors, A-4, A-5
  specifying, 3-15, 3-34
dynamic link library *(See* DLL)

E

ExeType statement, 11-5, 12-2, 12-9
Exports statement, 12-10, 12-2
  example, 12-11
  parameters, 12-11

F

First.asm file, 5-9, 6-10, A-11, A-4, A-5
  example, 6-12, 6-13, 6-15
fixed partition, 6-1
function
  mediated, D-1

H

HeapSize statement, 12-2, 12-12
  example, 12-12
  parameters, 12-12

I

import definition records, C-7
import library, 9-2, 9-7
  contents, 9-7
  creating, 10-2, 11-4
  using, 11-4
Imports statement, 11-4, 12-2, 12-13
  example, 12-14
  parameters, 12-13
Intel object module formats (OMF), C-1

L

libary files
  examples, 3-20
  searching, 3-20
Librarian
  blocks, 8-6
  command form, 8-1
  cross-reference listing, 8-4
  error messages, A-1, A-3
  library index, 8-9
  managing object modules, 7-2
  public symbols, 8-9
  status codes, A-18
  uninitialized variables, 8-9
  using, 1-4, 7-1, 7-3
Librarian command form
  parameter fields, 8-2
  using, 8-1
Librarian, 7-1
library
  components, 8-7
  conserving space, 8-7
library block size, 8-6
library file
  creating, 3-14
  specifying, 3-35, 8-2
library index, 8-9
  procedures, 8-9
  public symbol names, 8-9
library reference, 3-28
  examples, 3-29
  specifying, 3-35
library search list, 3-29
  searching, 3-30

Library statement, 12-2, 12-15
 example, 12-16
 parameters, 12-15
Link command, E-1
 capabilities, E-2
 run file mode parameters, E-3
 user interface, E-2
 using, E-1
Link command form
 parameter details, E-14
Link V6, 3-3
Link V8, 3-3
Linker
 code segment, 6-9
 command form selection, 3-3
 command form, 3-3
 command forms, 2-3
 configuration file, 2-3, 3-1, 3-24
 configuring, 3-24
 customizing segment ordering, 6-10
 DGroup, 5-11
 error messages, A-1, A-3, A-4
 file types, 2-4
 First.asm file, 6-10
 group, 5-11
 heap size, 3-7
 input, 9-2
 library files, 3-14
 library reference, 3-28
 library search list, 3-29
 map file, 3-6, 4-1
 memory array, 6-7
 memory requirements, 6-1
 memory space, 6-1
 object module formats, C-1
 object module, 2-3, 3-6
 overlays, 6-1
 overview, 2-1
 program memory requirements, 6-1
 run file mode, 3-5
 run file, 3-6
 search path, 3-27
 segment attribute alternatives, C-6
 segment class order, 3-32
 segment limits, 5-15
 segment order, 5-15, 6-1, 6-16
 segments, E-4
 stack size, 3-7, 6-1
 status codes, A-18
 user interfaces, 2-3, 3-1
 using, 1-4
 virtual memory management, 6-9
 virtual memory size, 3-40
Linker command forms
 Link V6, 2-3
 Link V8, 2-3
 Link, 2-3, E-1
 parameter fields, 3-5, 3-6
 using, 3-4
Linker commands
 differences, E-2
 Link, E-1
 Version 4, E-1
Linker configuration file, 3-1, 3-24, 6-16
 example, 3-25
 parameters, 3-31
 specifying, 3-17, 3-33
Linker errors
 causes, A-2
 levels, A-2
Linker map file, 4-1
 reading, 4-1
 simple, 4-1
 Version 6, 4-1
 Version 8, 4-3

Linker segment order
correcting, 5-9
First.asm file, 5-9
specifying, 5-9
Linker segments
addressing, 5-11
combining, 5-9
creating, 5-8
linking, 9-4
DLL, 5-1
dynamic, 5-1
dynamic and static, 5-2
library search algorithm, 5-2
object module, 5-3
overview, 5-1
pass one, 5-1
pass two, 5-3
passes, 5-1
run file, 5-3
static, 5-1
loader, 2-5
invoking, 2-5
LoadType statement, 12-2, 12-17
parameters, 12-17

M

map file, 2-5, 3-6
mediation, D-1, D-10
using, D-3
memory array, 6-7
memory requirements
estimating, 6-1
memory space
allocating, 6-1, 6-5
memory array, 6-5
Microsoft C compiler
communal name records, C-4
Module Definition command, 9-2
Module Definition command form, 10-1
input, 9-2
object modules, 9-2
output, 9-5, 9-6
output files, 9-2
parameter fields, 10-2
porting programs, 11-5
module definition file, 11-1
Code statement, 12-2, 12-4
CTOS extensions, 11-6
Data statement, 12-2, 12-6
Description statement, 12-2, 12-8
example, 11-2
ExeType statement, 11-5, 12-2, 12-9
Exports statement, 12-2, 12-10
guidelines, 11-1
HeapSize statement, 12-2, 12-12
Imports statement, 11-4, 12-2, 12-13
Library statement, 11-2, 12-2, 12-15
LoadType statement, 12-2, 12-17
Name statement, 11-2, 12-3, 12-18
Old statement, 11-5, 12-3, 12-20
ProtMode statement, 12-3, 12-21
purpose, 11-1
RealMode statement, 12-3, 12-22
RunType statement, 12-3, 12-23
segment attributes, 11-6
Segments statement, 12-3, 12-25
StackSize statement, 12-3, 12-27
statements, 11-2, 12-1
Stub statement, 11-5, 12-3, 12-28
syntax, 11-2
writing, 11-1

module definition statements, 11-2, 12-1
module definition text file
specifying, 10-2
Module Definition utility, 9-1, 10-1
command form, 10-1
error messages, A-25
Intel object module formats (OMF), C-5
producing DLLs, 9-2
using, 1-4

N

Name statement, 12-3, 12-18
example, 12-19
parameters, 12-18

O

object module
arranging components, 5-4
contents, 9-6
creating, 10-2
examples, 3-17
Intel Object Module Format (OMF), 9-2
linking, 2-2
segment element, 5-4, 5-5
object module procedures
mediating, D-3
object module, 2-1, 2-4, 3-6, 5-3, 7-3, 9-6
in blocks, 8-8
linking, 7-3
managing, 7-2
Old statement, 11-5, 12-3, 12-20
overlays, 6-9
example, 3-20
linking with, 6-9
sorting procedure names in, 3-23
using, 3-20, 3-21, 6-1

P

Presentation Manager, 9-1
creating resources, 13-2
Presentation Manager programs
porting, 9-1
procedure names
sorting, 3-23
ProtMode statement, 12-3, 12-21
example, 12-21
parameters, 12-21
public symbols, 8-9

R

RealMode statement, 12-3, 12-22
example, 12-22
parameters, 12-22
resource
adding to run file, 13-2, 13-3, 14-2, 14-7, 14-8
defined, 13-3
deleting, 13-4, 14-4, 14-9
extracting, 13-3, 14-4, 14-10
ID, 13-7
identifying, 13-7
listing, 13-4, 14-5
storing, 13-5
type, 13-7
resource ID, 13-7

Resource Librarian
  adding resources, 13-2
  binary resource file, 13-5, 13-6
  command form, 14-1
  configuration file, 14-5, 15-1
  data file, 13-5, 13-6
  DLL, 13-5, 13-6
  error messages, A-27
  file definitions, 13-6
  file support, 13-3
  identifying resources, 13-7
  run file, 13-5, 13-6
  tasks, 13-4
  using, 1-4, 13-3
Resource Librarian command form, 14-1
  adding resources, 14-2
  deleting resources, 14-4
  extracting resources, 14-4
  listing resources, 14-5
  parameter fields, 14-2
Resource Librarian configuration file, 15-1
  example, 15-2
  format, 15-1
  specifying, 14-5
Resource Librarian, 13-1
resource type, 13-7
resource, 13-1
run file, 2-1, 2-4, 3-6, 5-3, 13-5, 13-6
  building, 5-6, 5-13
  format, B-8
  header fields, B-1
  linking, 2-4
  version, 3-14
run file mode, 3-9
  options, 3-11
  specifying, 3-9, 3-36
  validity of parameters, E-3
run type record, C-8
RunType statement, 12-3, 12-23
  example, 12-24
  parameters, 12-24

S

segment attributes, 12-29
  default values, 12-31
  definitions, 12-30
  describing, 11-6
  field effects, 12-32
  recommendations, 11-6
segment attribute records, C-6
segment element, 5-4, 5-12
  class, 5-5
  name, 5-5
  alignment attributes, 5-12
segment limits, 5-15
segment ordering, 5-15
  correcting, 6-14
  customizing, 6-1, 6-10
  example, 6-14
  First.asm file, 6-10
Segment statement, 12-3, 12-25
  example, 12-26
  parameters, 12-25
stack
  correcting overflow, 6-5
  reducing, 6-4
stack segment (SS), 5-11
stack size
  adjusting, 6-1, 6-4
  specifying, 3-37
StackSize statement, 12-3, 12-27
  example, 12-27
  parameters, 12-27
status codes, A-1

Stub statement, 11-5, 12-3, 12-28
symbol file, 2-5
  specifying, 3-16

U

uninitialized variables, 8-9

V

variable partition, 6-1
Version 4 Link command, E-1
Version 4 map file, E-8
  addresses, E-8
  classes, E-10
  details, E-10
  example, E-9, E-11
  library references, E-10
  line numbers, E-10, E-14
  names, E-9
  public symbols, E-10
Version 4 run file
  format, E-4
  header fields, B-1, E-6
  header tables, E-5
  using overlays, E-4
Version 6 map file, 4-1
  addresses, 4-2
  classes, 4-3
  components, 4-1
  details, 4-5, 4-10
  example, 4-2, 4-6
  library reference, 4-5
  line numbers, 4-5, 4-9
  names, 4-3
  protected mode selectors, 4-2
  public symbols, 4-5, 4-9
Version 6 run file, 13-3
  format, B-10
  header fields, B-1, B-5
Version 8 map file, 4-3
  example, 4-4, 4-12
Version 8 run file, 13-3, C-5
  format, B-8
  header fields, B-1
virtual memory management, 6-9
virtual memory size
  customizing, 3-40

43594969-100