Intel386 Family Utilities User's Guide

RadiSys Corporation
5445 NE Dawson Creek Drive
Hillsboro, OR 97124
(503) 615-1100
FAX: (503) 615-1150
www.radisys.com
07-0579-01
December 1999

December 1999

Quick Contents

Notational Conventions

The notational conventions described below are used throughout this manual:

UPPERCASE	Characters shown in uppercase, monospace font must be entered in the order shown. The characters may be entered in either uppercase or lowercase.
italics	Monospaced italics indicate a metasymbol that may be replaced with an item that fulfills the rules for that symbol. Metasymbols in tables are not always shown in italics.
[ ]	Brackets indicate that the enclosed arguments or parameters are optional.
{ }	Braces indicate that one and only one of the enclosed entries must be selected unless the entire field is also surrounded by brackets, in which case choosing an entry is optional.
|	The vertical bar separates options within brackets [ ] or braces { }.
...	Ellipses indicate that the preceding item may be followed by other like items; the items must be separated by single spaces, but no additional punctuation is required.
[,...]	The brackets enclosing comma and ellipsis indicate that the preceding item may be followed by other like items; the items must be separated by commas.
punctuation	Punctuation other than ellipses, braces, and brackets must be entered as shown.
	This typeface represents what you type in and is used for examples of machine response or other computer displays.

Contents

4 Using the Intel386 Mapper

Tables

Figures

Introduction 1

Overview

This manual describes how to use the Intel386™ Family utilities on DOS and iRMX® operating systems. The utilities consist of the following software tools for modular program development:

- The BND386 binder produces a single-task, bootloadable object module in Intel386 Family Object Module Format (OMF386) by linking modules created on Intel compilers or assemblers. Linking is the process of combining segments and resolving references. As an option, BND386 can combine linkable modules, which can then be used as input to BND386 or to the BLD386 System Builder to produce loadable modules.
- The LIB386 librarian organizes linkable modules into libraries and provides utilities for adding, deleting, or replacing library modules. LIB386 supports many useful functions related to library classification, identification, and maintenance.
- The MAP386 mapper generates a variety of listings describing the features of linkable or loadable object files.

Used together to develop both application and system software, BND386, LIB386, and MAP386 provide the flexibility to accommodate a wide variety of system designs.

Program Development

The Intel386 utilities can be most effectively used for modular program development, an efficient, proven approach to writing software. When a large program is written as a set of smaller, inter-related modules, errors are reduced, testing and debugging are simplified, and documentation is made easier. For more efficient project management and product updating, modules can be created by several programmers and can be written in different languages.

Figure 1-1 shows the typical development of an application module using BND386, LIB386, and MAP386.

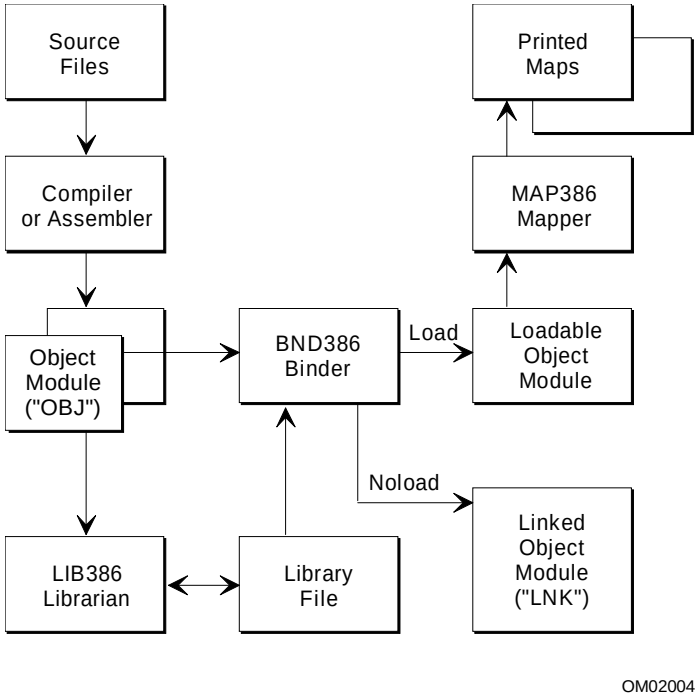

Figure 1-1. Development of an Application Module

Operational Summary: BND386

BND386 produces either linkable or loadable output modules. Options available for use during BND386 invocation allow you to perform the following operations:

- Produce OMF386 bootloadable output for execution on an Intel386 operating system
- Produce linkable output, which can be stored in object libraries or reprocessed with BND386 or with the BLD386 System Builder to create loadable output
- Perform type checking while resolving external public symbols
- Obtain a print file containing a segment map, module information, and any error messages. Error messages can also be directed to a separate file
- Control the contents of the output object module

BND386 enables you to control the contents of the output object module in the following ways:

- Include or exclude debug information and public symbol definitions in the output module
- Change the length of selected input segments
- Change segment names
- Adjust loadable module characteristics related to the operating system on which the modules will execute

Chapter 2 gives additional details and examples on BND386 functions and use. Appendix A explains the BND386 error messages.

Operational Summary: LIB386

LIB386 executes both in interactive mode (for running in the foreground) and non-interactive mode (for running in the background). LIB386 enables you to perform the following operations:

- Create a new library of linkable object modules
- Update an existing library by adding, deleting, or replacing modules
- Display information about each library
- Change library names and version numbers
- Obtain on-line summaries of the syntax of each LIB386 command

Chapter 3 gives additional details and examples on LIB386 functions and use. Appendix B explains the LIB386 error messages.

Operational Summary: MAP386

MAP386 enables you to perform the following operations:

- Obtain a variety of listings, or maps, describing the contents of linkable or loadable input modules. The maps include the following:
  - Table Map lists descriptor names and corresponding indexes for Global Descriptor Tables (GDTs), Interrupt Descriptor Tables (IDTs), and Local Descriptor Tables (LDTs).
  - Segment Map lists names of segments in the input file and characteristics of each segment, such as descriptor table index, access type, base, descriptor privilege level, USE16/32 attributes, align attributes, and others.

  - Gate Map lists symbolic gate names and the characteristics of each gate in the input file, such as descriptor table name, descriptor table index, gate type, and others.
  - Public Map lists public symbols in the input file and their characteristics, such as symbol type, word count, and logical address (and physical address, if applicable).
  - Symbol Map lists names of local symbols in the input file and characteristics of each symbol, such as symbol type, address, and others.
  - Task Map lists, for each task in an input file, task characteristics, such as initial privilege stack, flags, initial values of CS and EIP registers, the task's LDT selector, and others.
  - Cross-Reference Map lists, for each symbol in an input file, the symbol's name and type, as well as the name of the module containing the symbol's public definition and the names of modules containing external declarations for the symbol. This is the only map produced for linkable modules.
- Modify loadable or bootloadable files by selectively purging debug information.
- Add information specific to the operating system to loadable files.
- Control page formatting (e.g., page width and page length).

Chapter 4 gives additional details and examples on MAP386 function and use. Appendix C explains MAP386 error messages.

□□□

Using the Intel386™ Binder 2

The BND386 binder produces loadable or linkable modules by combining separately translated, linkable object modules, including modules in library files.

You can create two kinds of loadable, single-task modules: a loadable module that can be loaded on an Intel386 protected-mode system under control of the operating system or a bootloadable object module in Intel386 Family Object Module Format (OMF386).

As an option, you can create linkable modules. A linkable module can then become input to BND386 or to the BLD386 System Builder for the next incremental linking step or for final binding. In incremental linking two or more linkable modules output by a compiler or assembler are combined into a single linkable module. This linkable module then becomes input for the next incremental linking step or for final binding.

Major Functions of BND386

BND386 performs the following major functions:

- Creates a linkable module by combining linkable input modules. The linkable module can be linked with other linkable modules through incremental linking. A linkable file generated by BND386 can then become input to the BLD386 System Builder.
- Automatically selects required modules from specified libraries to resolve symbolic references
- Resolves symbolic references from one input module to another. As an option, BND386 performs type checking while resolving public-external symbols
- Creates a print file that contains the segment map and other information
- Creates an OMF386 bootloadable module targeted for an Intel386 operating system

Input and Output

BND386 accepts linkable modules from either 80286 or Intel386 software development products, including linkable modules produced by compilers or assemblers such as ASM386 and PL/M-386, linkable files output by BND286 or BND386, library files created by LIB286 or LIB386, or export files produced by BLD286 or the BLD386 System Builder containing entry points to the operating system. Export files are treated as ordinary linkable files and so are not discussed separately in this manual.

BND386 does not accept incrementally built files created with BLD286 or the BLD386 System Builder.

Input files are processed in the same order in which they are specified on the BND386 invocation line.

Each input file can be a library file or a non-library file, and each one can be followed by a list of modules. Both factors determine how BND386 processes the input file.

A library file contains one or more object modules as well as control information. A module in a library file is processed if the module is explicitly listed on the invocation line. If a module list is not specified for a library file, the binder only processes modules if previously processed modules contain one or more unresolved externals. Then the binder scans the library file for modules containing public symbols that match the unresolved externals. Each such module is processed when found. The scanning continues until the modules in the library cannot satisfy any more unresolved externals, including any more unresolved externals encountered while processing modules from the library.

As output, BND386 creates either an OMF386 bootloadable file targeted for an Intel386 operating system, or a linkable file that can become input to BND386 or the BLD386 System Builder in a subsequent step. The BND386 `RCONFIGURE` control produces an OMF386 bootloadable file. The `LOAD` control creates a single-task loadable file that can be loaded on an Intel386 protected-mode system under the control of the operating system.

In addition, BND386 creates a print file that contains a segment map and other information.

Figure 2-1 illustrates BND386 input and output.

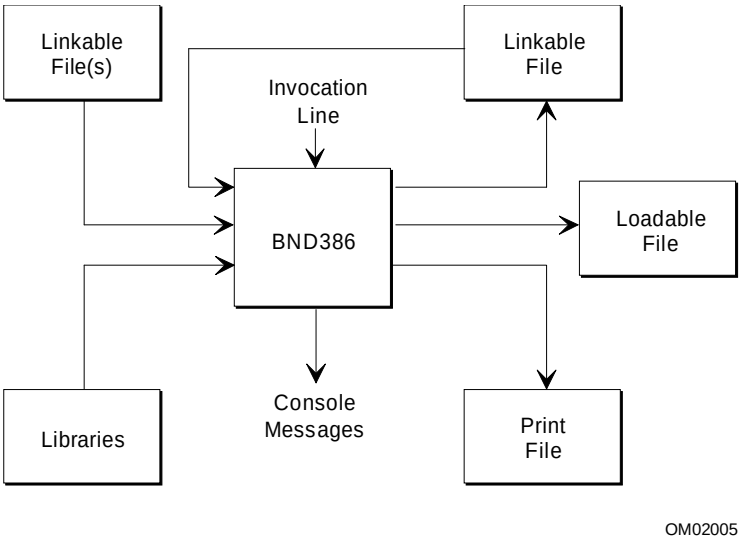

Figure 2-1. BND386 Input and Output

⚠ **CAUTION**

The maximum size of a BND386 output file is 8 Mbytes. If the total of the combined image is greater than that, BND386 issues `Error 118: PAGE FILE OVERFLOW`.

BND386 Processing

When BND386 creates a linkable or a loadable file, it performs the following functions, which are not directed by invocation controls:

- Segment combination
- Reference resolution
- Fix-up processing
- Descriptor table creation (only for loadable files)
- Task state segment (TSS) creation (only for loadable files)

These functions are described in the following sections.

Segment and Segment Combination

The Intel386 processor uses a segmented memory scheme in which program instructions and data are divided into logical units, or *segments*. There are two kinds of segments: USE16 and USE32. A USE16 segment is output from 80286 utilities, compilers, and assemblers and contains up to 64K bytes of code, data, or stack area. A USE32 segment is output from Intel386 utilities, compilers, and assemblers and contains up to 4 gigabytes of code, data, or stack area. The ASM386 assembler can output both USE16 and USE32 segments.

Intel compilers and assemblers support the concept of logical segments and can produce object code that is already segmented. BND386 uses the output produced by a compiler or assembler to take segmentation one step further, combining segments with the same name and similar characteristics. Because similar segments from separately translated modules can be merged, the program as a whole consists of fewer segments and becomes more efficient.

BND386 combines segments of the same kind: code, data, or stack (see Figure 2-2). During processing, BND386 checks the characteristics of each input segment and combines segments that satisfy its combination criteria. The following sections describe these combination criteria in detail.

Segment Attributes

The code, data, stack and other segments that constitute a translated program have a variety of characteristics, such as symbolic name, size, and access rights. These characteristics are based in part on the kind of information in the segment and on the features of the programming language in which the program is written.

The characteristics of a segment determine whether BND386 combines it with other segments in modules processed during the same BND386 invocation. If the characteristics of two or more segments satisfy the combination criteria, BND386 combines them into one segment. The characteristics of the two original segments determine the characteristics of the resulting segment.

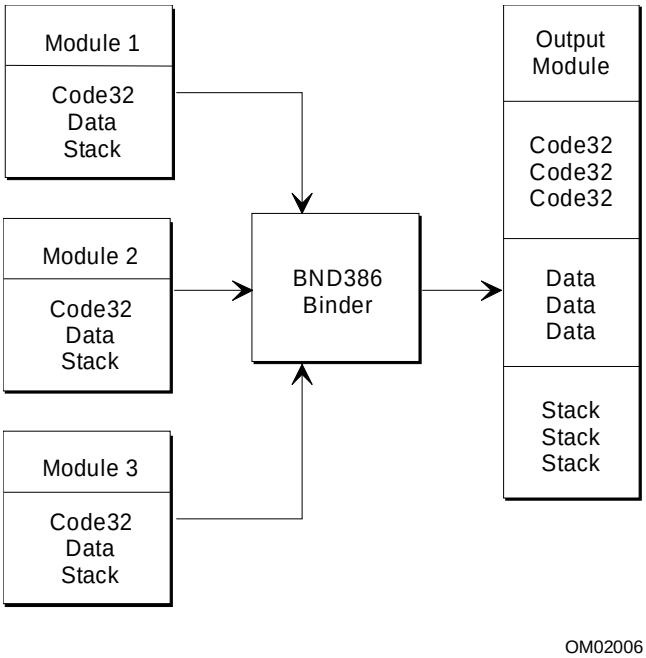

Figure 2-2. BND386 Segment Combination

The following segment characteristics affect the process of segment combination:

- Segment name
- USE attribute
- Length (size)
- Access rights
- Combine type
- Align attribute

Segment Names

The name of a segment is the symbolic name assigned by the programmer in ASM386 programs or by the compiler in high-level-language programs. For high-level-language programs, the compiler assigns segment names according to the model of segmentation used. Refer to the appropriate Intel compiler manual for the names assigned to segments compiled under various segmentation models.

USE Attribute

USE16 segments have up to 64K bytes; USE32 segments have up to 4 gigabytes. A USE16 segment cannot be combined with a USE32 segment that contains code (that is, has an execute access right). A USE32 segment that does not have execute access rights can be combined with a USE16 segment as long as the other combination criteria are met. The resulting segment is a USE16 segment of up to 64K bytes.

Segment Length

Segment length is measured in bytes. The length of a given segment depends on how much and what kind of information it contains. These factors in turn depend on the design of the program or the individual module.

Access Rights

Each segment in a program is assigned one of the following access rights during translation:

- Read-only (RO)
- Read-write (RW)
- Execute-only (EO)
- Execute-read (ER)

Code, data, and stack segments may be assigned access rights as follows:

- Code segments: EO or ER access
- Data segments: RO or RW access
- Stack segments: RW access

Combine Type

Each segment in a program is assigned a combine type during compilation. Segments can have the following combine types:

Combine Type	Segment Description
stack	All stack segments
no-combine	Code or data segments cannot be combined: for example, ASM386 segments that are not PUBLIC and PL/M-286 segments compiled under the LARGE model
common	Segments contain named FORTRAN-386 common blocks
blank common	Segments contain unnamed FORTRAN-386 common blocks
normal	Code or data segments can be combined and do not contain FORTRAN-386 common blocks: for example, ASM386 segments that are PUBLIC or segments created by PL/M386
debug	Segments that contain information for debuggers. They are combined like normal segments

Align Attribute

The align attribute specifies the boundary to which a segment is aligned. A byte-aligned segment can be placed anywhere in memory. A word-aligned segment is placed at an even address, and so on. During segment combination, if necessary, BND386 inserts gaps between segments to comply with alignment requirements.

Privilege Level

Each segment has a privilege level between 0 and 3. Intel386 compilers and assemblers set the privilege level of all segments to 3. BND386 creates a single task with a single privilege level, and all segments have that privilege level, regardless of original privilege level. The exception is gate descriptors exported from the BLD386 System Builder, which retain their original privilege level.

Criteria for Segment Combination

BND386 combines two segments if all the following criteria are met:

- Both segments have the same name.
- Both segments have compatible access rights (see Table 2-1).
- Both segments have compatible USE attributes.
- Both segments have the same combine type and neither is of the type no-combine.

Table 2-1. Matrix of Access Rights Assignments for Combined Segments

Original Segments	RO	RW	EO	ER
RO	RO	RW	ER	ER
RW	RW	RW	•	•
EO	ER	•	EO	ER
ER	ER	•	ER	ER

RO = Read only
RW = Read-write
EO = Execute only
ER = Execute-read

Attributes of the Resulting Segment

The attributes of the original segments determine the attributes of the combined segment. The combined segment is assigned attributes as follows:

- The name of the original segments
- The combine type of the original segments, if both were of the same combine type

The combined segment is also assigned access rights based on the access rights of the original segments (see Table 2-1).

Code segments with different USE attributes cannot be combined. Data segments with different USE attributes are assigned the USE16 attribute when combined.

Length of the Resulting Segment

The length of the combined segment depends on the lengths, combine types, and alignment attributes of the original segments, as explained below:

- When two normal segments are combined, BND386 starts by making the length of the combined segment the sum of the lengths of the original segments. Then BND386 adds bytes between segments, if necessary, to align the second segment according to its alignment attribute. The combined segment may therefore be longer than the sum of the lengths of the original segments.
- When two stack segments are combined, BND386 does not perform offset relocation or alignment. The combined segment is as long as the sum of the lengths of the original segments.
- When two common segments are combined, BND386 expects the segments to be of the same length and does not perform offset relocation. The combined segment is as long as either one of the original segments. An error is issued if the segments are not of the same length.
- When two blank common segments are combined, BND386 makes the combined segment as long as the longer of the two segments and does not perform relocation or offsets.

BND386 can add up to three extra bytes to segments, in addition to the gaps introduced for alignment. This length adjustment is called *padding*. BND386 adds an extra byte to segments with both the USE16 attribute and RO, RW, or ER access rights, so that 16-bit word references to the last byte in the segment remain valid. BND386 adds three extra bytes to segments with both the USE32 attribute and RO, RW, or ER access rights, so that 32-bit word references to the last byte in the segment remain valid.

Reference Resolution

BND386 attempts to resolve symbolic references among input modules. A symbolic reference is a declaration of a variable, procedure, or label, either external or public. Names of variables, procedures, and labels are called symbols. References are resolved when BND386 finds a public declaration in the input modules for each external declaration of the same symbol.

When an external declaration such as an extern in C or a common in ANSI FORTRAN has no matching public, the compiler or assembler gives the total size of the symbol. BND386 then allocates the maximum size of external matching symbols in the input and initializes the allocated area to zero. In effect, this procedure generates a public declaration to resolve the externals.

An unresolved external causes a warning when a loadable module is built and the reference is left undefined. However, unresolved externals are allowed in loadable modules, because in some operating systems they can be resolved by the loader at load-time. Multiple public symbols with the same name cause a warning.

Figure 2-3 shows a resolved public-external procedure pair in two linkable input modules, one of them a main module.

```
MAIN$MOD:                                BUFF$HANDLER:  do ;
                                         *--------------------------------------*
  DECLARE BUFF (5) BYTE;                 * FOO: PROCEDURE (BUFFER$ADDR) PUBLIC;*
                                         *                                      *
                                         *--------------------------------------*
*----------------------------------*
*  FOO: PROCEDURE (BUFPTR) EXTERNAL;*
*                                  *       DECLARE BUFFER$ADDR POINTER;
*    DECLARE BUFPTR POINTER;       *                .
*             .                    *                .
*             .                    *                .
*             .                    *                .
*END FOO;                         *                .
*----------------------------------*
          .                              END FOO;
          .
          .                                        .
  CALL FOO (BUFF);                                 .
          .                                        .
          .                              END BUFF$HANDLER;
          .
END MAIN$MOD;
```

Figure 2-3. A Resolved Public-External Procedure Pair

Fix-up Processing

Input segments may contain references to an unresolved external or to a logical address, which consists of a segment selector and an offset in the segment. The compiler or assembler passes fix-up information on each reference to BND386. BND386 adjusts the offsets of logical addresses in the fix-up information to reflect segment combination.

In generating a linkable module, BND386 performs the following fix-up operations:

- Replaces satisfied external symbol names with the logical addresses of the corresponding public symbols.
- Adjusts all logical addresses referring to the combined segments to reflect segment combinations.

In generating a loadable module, BND386 processes the fix-ups as follows:

- If the target is specified by an external symbol that is satisfied by a matching public symbol (either from the input or allocation by BND386), the external symbol is replaced by the logical address of the matching public symbol.
- If the target refers to an unresolved symbol, the fix-up is output to the loadable file, a warning is issued, and the fix-up is not applied. Fix-ups are output to the loadable file in the section of the object that contains relocation information.
- If the target is specified by a gate, the fix-up is applied using the gate selector.
- If the target is specified by a global descriptor table (GDT) selector [internal name], the fix-up is applied using the GDT selector.

Descriptor Table Creation

BND386 produces a descriptor table for loadable output. The table is either in the format of a final LDT (local descriptor table) ready to be loaded by the loader or in the format of a relocatable descriptor table, in which the entry number of each descriptor is not fixed and is set at load time. When the table is not necessarily final, BND386 adds relocation information to the object file. This procedure allows the loader to update each selector reference.

Task State Segment Creation

A BND386 loadable output module contains the basic elements of a single task. In each loadable module, BND386 provides information that enables the loader to construct a task state segment (TSS) and an LDT.

Invoking BND386

DOS and iRMX OS Invocation Syntax

To invoke BND386 on a DOS and iRMX operating system, use the following syntax:

```
BND386 input_list [controls]
```

Where:

input_list
: is one or more linkable modules or object library modules to be processed by BND386. For DOS, the modules are specified as follows:

  filename [(*module_list* | *])

  filename(*) can replace the complete list of linkable or library modules in the file called *filename*. BND386 then processes all modules in a file.

controls
: consists of one or more of the specifications defined in BND386 Controls.

BND386 processes files in the input list in the order in which they are specified. Therefore, only unresolved external symbols that come before a library file can be resolved by the public symbols in the library.

If modules are specified with a library file in the invocation line, BND386 processes specified modules only.

If *filename*(*) is specified with a library file in the invocation line, BND386 processes all modules in the library.

If no modules are specified with a library file in the invocation line, BND386 processes that library file only if previously processed modules contained at least one unresolved external. The library is scanned for modules containing public symbols that match unresolved externals and each such module is processed as if it were specified. The process continues until the modules in the library cannot satisfy any more unresolved externals (including externals encountered while processing modules from the library).

If no modules are specified with a linkable file in the invocation line, BND386 processes all modules in the file exactly as if you had used *filename*(*).

BND386 issues an error message when it finds duplicate module names.

The input list can be omitted from the invocation command if the list is specified in at least one control file.

You can continue the invocation line on additional lines by entering the ampersand (&) before you enter the line terminator. The continuation line then appears automatically with the DOS or iRMX system prompt character.

> ⚠ **CAUTION**
> Long invocation lines can cause BND386 to fail with a general protection fault. To specify many controls and/or input files, it is better to use a control file than continuations of the command line.

Control Files

The BND386 invocation line is simplified when you can use the `CONTROLFILE` control to include a control file. A control file is a text file containing controls, file names, or controls and file names that would normally appear in the invocation line. For example, instead of listing five controls in the BND386 invocation line, you can place those controls in a single control file and then invoke the control file in place of all five controls.

Using a Control File on DOS and iRMX OS

To include a control file in the BND386 invocation line on a DOS operating system, use the following syntax:

```
BND386 CONTROLFILE (filename[,...])
```

Where:

`filename` is the name of the control file containing controls, file names, or controls and file names for the input list. You cannot nest control files: that is, the `CONTROLFILE` control cannot appear in a control file.

A control file that contains only controls can be specified in any position in the control list. A control file that contains only file names for the input list can be specified in any position in the input list.

In a control file that contains both input files and controls, input files must come before controls. In this case, specify the control file as part of the input list.

The following example shows how to specify the `CONTROLFILE` control in an input list that contains the files named in CF1.DAT:

```
BND386 MOD.OBJ, CONTROLFILE (CF1.DAT) DEBUG
```

Within a control file, use a semicolon before a comment. Use the ampersand (&) to continue to the next line. When the line terminator comes before the ampersand, it is treated as if it were a blank space. BND386 ignores characters between a semicolon or continuation character and the line terminator. Lines in a control file cannot exceed 120 characters in length.

This example control file contains only file names for the input list:

```
util.lib,     ; utility library&
system.lib    ; system library
```

This example control file contains the last file names for the input list and controls for the control list:

```
util.lib,     ; utility library&
system.lib    ; system library&
lo&           ; loadable module&
ep&           ; directs error messages&
              ; to the specified print&
              ; file specified&
oj (lbt.sys)  ; name output file
```

BND386 Defaults

Output File Names

If not specified in the invocation line, output file names are assigned by default, as follows:

- The file that contains the output loadable module has the same name as the first input file. Under DOS and iRMX OS, the output has no extension.
- The file that contains the output linkable module has the same input file name as the first input file, with extension, or file type, .LNK.
- The print file has the same name as the output object file, with extension, or file type, .MP1.

Controls

If no controls are specified on the invocation line, BND386 does the following:

- It creates a loadable output module. The output module is placed in a file having the same file name as the first input module listed in the invocation line. Under DOS and iRMX OS, the output has no extension.
- It creates a print file with the same name as the object file, with extension, or file type, .MP1.
- It performs type checking between public and external symbols in input modules and includes symbol type information in output modules.
- It includes debug information in output modules (if this information is in the input modules).

Some BND386 controls imply that other controls are in effect by default. Refer to BND386 Controls and individual control entries later in this chapter for more information on default implementation.

Console Messages

BND386 displays console messages when signing on and signing off, as well as during processing. During processing, BND386 issues warnings, error messages, or fatal error messages if it encounters any problems.

Fatal error messages are always displayed at the console, even if you have directed error messages to an error print file with the ERRORPRINT control.

When you invoke BND386 on a DOS or iRMX operating system, the binder displays the following sign-on message:

```
system_id
iRMX III 386(TM) BINDER, Vx.y
Copyright years Intel Corporation
```

Where:

system_id is the identifier and version number of the operating system.

V*x.y* is the BND386 version number.

years is the copyright year or years.

When BND386 encounters a fatal error condition, an error message and sign-off message are displayed at the console. The sign-off message is as follows:

```
PROCESSING ABORTED
```

See Appendix A for additional information about error conditions.

When BND386 does not encounter a fatal error condition, BND386 signs off when you exit or after processing is complete, as follows:

```
PROCESSING COMPLETED.n WARNINGS,m ERRORS
```

Where:

n and *m* represent the number of warning and nonfatal error conditions encountered during processing. You can use the ERRORPRINT control to display warning and error messages on the console or direct them to a file.

BND386 Controls

BND386 controls determine the extent and characteristics of BND386 output. Some controls determine what kind of output is produced: a print file, an object file, or a file containing error messages. Other controls affect the characteristics of the output object module: size of segments, privilege level, whether debug information is included, and other features.

Target controls determine the type of object file that BND386 is to produce. `NOLOAD` produces a linkable output module. `LOAD` produces a loadable module. `RCONFIGURE` produces an OMF386 bootloadable module specifically intended for the iRMX III Operating System.

Table 2-2 summarizes BND386 controls for DOS and iRMX Operating Systems. The default column shows the condition in effect when the control is not specified. When an invocation contains duplicate control specifications, BND386 processes only the rightmost specification on the invocation line.

Each BND386 control is listed alphabetically and described in detail in the section following Table 2-2.

Table 2-2. BND386 Controls for DOS and iRMX Operating Systems

Abbr.	Command Line Syntax	Description	Default
CF	CONTROLFILE (*filename* [,...])	Specifies file for input elements	None
DB NODB	DEBUG NODEBUG	Retains or removes debug information	DEBUG
EP NOEP	ERRORPRINT [(*filename*)] NOERRORPRINT	Creates or does not create error print file	NOERRORPRINT
I2	INT286 [*mod_name*[,...])]	Provides interface control to 80286 programs	None
LO NOLO	LOAD NOLOAD	Creates loadable (LOAD) or linkable (NOLOAD) module	LOAD
NA	NAME (*mod_name*)	Creates and names or suppresses creation of object module output	*First input_ filename*
OJ NOOJ	OBJECT [(*filename*)] NOOBJECT	Verifies input object module meets Intel387 requirements or Intel386 requirements	MOD386
PR NOPR	PRINT [(*filename*)] NOPRINT	Creates and names or suppresses creation of print file	PRINT
PL[EC] NOPL[EC]	PUBLICS [EXCEPT (*symbol* [,...])] NOPUBLICS [EXCEPT (*symbol* [,...])]	Retains or removes public symbol definitions in linkable output modules	PUBLICS
RC[DM]	RCONFIGURE [(DYNAMICMEM (*memory_range*))]	Produces loadable output configured for an Intel386 operating system	None
RN	RENAMESEG (*old_seg _name* TO *new_seg_name* [,...])	Renames an input segment	None
SS	SEGSIZE (*seg_name* ([+/-]*size*)[,...])	Changes length of stack or data segment in output object module by specified size	None
TY NOTY	TYPE NOTYPE	Enables or suppresses type checking	TYPE

CONTROLFILE

Specifies file for input elements

Syntax

```
CONTROLFILE (filename[,...])
```

Abbreviation

```
CF
```

Default

`CONTROLFILE` is not in effect.

Description

The `CONTROLFILE` control reads invocation specifications from a control file. Invocation specifications can include input files and invocation controls, but cannot include partial controls or partial input-list elements. BND386 returns to the command line when it encounters the end of the control file.

A control file cannot contain the `CONTROLFILE` control.

See Control Files for content and format of control files.

Example

In the following example, control file CNTL1.DAT contains UTIL.LIB and SYSTEM.LIB, and control file CNTL2.DAT contains PRINT(SAMPLE.MAP) and DEBUG. The following pairs of invocation lines do the same thing:

```
BND386 SAMPLE.OBJ, CF(CNTL2.DAT) CF(CNTL1.DAT)

BND386 SAMPLE.OBJ, UTIL.LIB, SYSTEM.LIB PRINT(SAMPLE.MAP) DEBUG
```

For additional examples, see Chapter 4 on the mapper.

DEBUG/NODEBUG

Retains or removes debug information

Syntax

```
DEBUG
NODEBUG
```

Abbreviations

```
DB
NODB
```

Default

```
DEBUG
```

Description

The `DEBUG` control places information used by symbolic debuggers in the output object module. Debug information consists of the following:

- Symbolic names and source or listing line numbers generated by compilers or assemblers
- Public and external symbols plus type and module information formatted by BND386 for debuggers

The `NODEBUG` control purges symbolic debugging information from the output module.

Debug information is needed by MAP386, so you can leave the information in place and purge it with MAP386 later in the development process.

Examples

1. In the following example, compiler or assembler output for symbolic debugging is included in the output linkable module in the file MOD1.LNK.

   ```
   BND386 MOD1.OBJ, MOD2.OBJ NOLOAD DEBUG
   ```

2. In the following example, debugging information is removed from the output loadable module in the file MOD3.

   ```
   BND386 MOD3.OBJ, MOD4.OBJ NODEBUG
   ```

ERRORPRINT/NOERRORPRINT

Creates or does not create an error BND386:print file

Syntax

```
ERRORPRINT [(filename)]
NOERRORPRINT
```

Abbreviations

```
EP
NOEP
```

Default

```
NOERRORPRINT
```

Description

The `ERRORPRINT` control directs error messages to one of the following:

- The standard output device, if *`filename`* is not specified
- The file called *`filename`*

The `NOERRORPRINT` control does not produce a file containing error messages; error messages appear in the normal print file.

No matter which of the two controls is in effect, fatal error messages are displayed on the standard output device, and error and warning messages are included in the print file. In the sign-off message BND386 reports the number of errors.

Note

If the specified file name matches the name of a file on the input list or the name of a control or output file, BND386 processing aborts.

Examples

1. In the following example, error or warning messages go to the error print file MOD2.LIS and to the print file that contains the segment map.

   ```
   BND386 MOD1.OBJ ERRORPRINT (MOD2.LIS).i.1.
   ```

2. In the following example, no error print file is created; by default, error messages are included in the print file.

   ```
   BND386 MOD3.OBJ, MOD4.OBJ
   ```

3. In the following example, error and warning messages are sent to the standard output device.

   ```
   BND386 MOD5.OBJ, MOD6.OBJ ERRORPRINT
   ```

INT286

Provides interface control to 80286 programs

Syntax

```
INT286 [(mod_name [,...])]
```

Abbreviation

```
I2
```

Default

`INT286` is not in effect.

Description

The `INT286` control allows Intel386 programs to interface with 80286 programs and ensures that data in specified modules resides in the combined USE16 segment in the first 64K bytes of memory. Read-write segments in the specified modules and in all stack segments become USE16 segments.

Only stack segments become USE16 segments when no module is specified.

Examples

1. In the following example, the data segments of the specified module become USE16 segments. USE16 segments are placed below

   ```
   BND386 MOD1.OBJ, MOD2.OBJ INT286 (MOD NAME)
   ```

2. In the following example, the stack is made up of USE16 segments, which are placed below USE32 segments.

   ```
   BND386 MOD.OBJ INT286
   ```

LOAD/NOLOAD

Creates a loadable (LOAD) or linkable (NOLOAD) module

Syntax

```
LOAD
NOLOAD
```

Abbreviations

```
LO
NOLO
```

Default

```
LOAD
```

Description

The LOAD control creates a loadable module containing an executable program and data plus system items such as an LDT. The single-task module can be loaded on an Intel386 protected-mode system under the control of the operating system. (Use the RCONFIGURE control to create a loadable module targeted specifically for an Intel386 operating system.)

The NOLOAD control creates a linkable module that can be used in subsequent BND386 invocations or as input to the builder. By default, the linkable module is placed in the file called *first_input_filename*.LNK.

Refer to the OBJECT entry later in this chapter for information on the assignment of the name of the object file.

Notes

RCONFIGURE is not effective when used with NOLOAD.

The NAME, PUBLICS, and NOPUBLICS controls work only when NOLOAD is in effect.

Examples

1. In the following example, a loadable module is produced; `LOAD` is in effect by default. By default, MOD1 is the output file name.

   ```
   BND386 MOD1.OBJ, MOD2.OBJ
   ```

2. In the following example, a linkable object module is produced. MOD3.LNK is the output file name.

   ```
   BND386 MOD3.OBJ, MOD4.OBJ NOLOAD
   ```

3. In the following example, the `NOLOAD` control produces a linkable object module. Modules MOD1 and MOD2 in MOD3.OBJ are input along with MOD4.OBJ.

   ```
   BND386 MOD3.OBJ (MOD1,MOD2), MOD4.OBJ NOLOAD
   ```

NAME

Names the linkable output module

Syntax

NAME (*mod_name*)

Abbreviation

NA

Default

NAME(*first_input_mod_name*)

Description

The NAME control assigns a name to the linkable output module. When NOLOAD is in effect and the NAME control is not specified, BND386 assigns the name of the first input module (*first_input_mod_name*) encountered in the input list to the linkable output module. NAME does not affect the file name of the file containing the linkable module.

Note

NAME is effective only when used with NOLOAD.

Examples

In the following example, MOD NAME is the output linkable module.

```
BND386 MOD1.OBJ, MOD2.OBJ NOLOAD NAME (MOD NAME)
```

OBJECT/NOOBJECT

Creates and names or suppresses creation of the output object module

Syntax

```
OBJECT [(filename)]
NOOBJECT
```

Abbreviations

```
OJ
NOOJ
```

Default

Loadable output is `OBJECT` (*`first_input_filename`*) and linkable output is `OBJECT` (*`first_input_filename`*`.LNK`)

Description

The `OBJECT` control creates an object file, either assigning the specified file name or using the default file name when *`filename`* is not specified.

The `NOOBJECT` control prevents the creation of an object file.

Note

If the name of the specified file or the name of the default file matches the name of an input file, print file, or control file, BND386 processing aborts.

Examples

1. In the following example, BND386 places the loadable output module in a file with the default file name MOD1.

   ```
   BND386 MOD1.OBJ, MOD2.OBJ OBJECT
   ```

2. In the following example, BND386 outputs an object file with same name as the first file in the control file CNTRL3.CF, with extension, or file type, .LNK.

   ```
   BND386 CF (CNTRL3.CF) NOLOAD OBJECT
   ```

3. In the following example, BND386 outputs a loadable output module called MOD6.LNK.

```
BND386 MOD4.OBJ, MOD5.OBJ OBJECT (MOD6.LNK)
```

4. In the following example, BND386 outputs only a print file.

```
BND386 MOD7.OBJ, MOD8.OBJ NOLOAD NOOBJECT
```

PRINT/NOPRINT

Creates and names or suppresses creation of a print file

Syntax

```
PRINT [(filename)]
NOPRINT
```

Abbreviations

```
PR
NOPR
```

Default

```
PRINT
```

Description

The PRINT control creates and names a print file, which contains a segment map, a module list, a list of unresolved symbols, and warning and error messages. When the file name is specified, it is assigned to the print file. Otherwise, the file name of the print file is assigned by default, as follows:

- When PRINT is not specified or is specified without a file name, the name of the print file is the same as that of the file containing the output object module, with extension, or file type, .MP1.
- When NOOBJECT is in effect, the print file is assigned the name of the first input file, with extension, or file type, .MP1.

The NOPRINT control prevents the creation of a print file.

The contents and format of the print file are described later in this chapter in Print File.

The ERRORPRINT control can be used to create a separate print file for error messages only.

Notes

When the name of the specified file or the name of the default file matches the name of an output file or a file on the input list, BND386 processing aborts.

A message about a fatal error condition is always displayed on the standard output device.

The default file name of the print file is affected by any file name specified with `OBJECT`.

Examples

1. In the following example, BND386 produces a print file called MOD3.MP1.

   ```
   BND386 MOD1.OBJ, MOD2.OBJ OBJECT (MOD3.LNK)
   ```

2. In the following example, no print file is produced.

   ```
   BND386 MOD4.OBJ, MOD5.OBJ NOPRINT
   ```

3. In the following example, BND386 produces a print file called MOD8.LIS.

   ```
   BND386 MOD6.OBJ, MOD7.OBJ PRINT (MOD8.LIS)
   ```

4. In the following example, BND386 produces a print file called MOD9.MP1.

   ```
   BND386 MOD9.OBJ NOOBJECT
   ```

PUBLICS/NOPUBLICS

Retains or removes selected public symbol definitions in linkable output modules

Syntax

```
PUBLICS [EXCEPT (symbol[,...])]
NOPUBLICS [EXCEPT (symbol[,...])]
```

Abbreviations

```
PL [EC], NOPL [EC].
```

Default

```
PUBLICS
```

Description

The PUBLICS control keeps some or all of the public symbol definitions in the linkable output module. You can use the EXCEPT specification to exclude a unique symbol or, with the asterisk (*), to exclude a group of public symbol definitions with a common prefix. For example, to purge all occurrences of publics with the prefix DQ, you would simply enter:

```
DQ*
```

The NOPUBLICS control removes some or all public symbol definitions from the linkable module. Again, you can use the EXCEPT specification to select public symbol definitions to be included in the linkable module.

The NOPUBLICS EXCEPT construction is useful during modular program development. For example, you can link some modules into a subsystem and grant access to the subsystem only through specific entry points. This construction reduces the chance of error and also allows different subsystems to use the same public name for different purposes (just as you might allow local variables in different procedures to have the same name).

The PUBLICS EXCEPT construction is useful when you want to hide a few names from the rest of the application, but still keep most public names visible.

Public symbols that represent gates created by the builder cannot be specified with EXCEPT.

Note

PUBLICS and NOPUBLICS are effective only when used with NOLOAD. When a loadable module is created, public symbol definitions are removed from the output object module and retained only as debug information for debuggers.

Examples

1. In the following example, the linkable module in MOD4.LNK contains no public symbol definitions except SYMBOL2 and SYMBOL3. MOD1.OBJ is the input file.

```
BND386 MOD1.OBJ NOLOAD NOPL EC (SYMBOL2, SYMBOL3) OJ (MOD4.LNK)
```

2. In the following example, public symbol definitions are included in MOD5.OBJ and MOD6.OBJ by default.

```
BND386 MOD5.OBJ, MOD6.OBJ NOLOAD
```

3. In the following example, all publics except those starting with the prefix PLM are purged.

```
BND386 MOD1.OBJ NOLOAD NOPUBLICS EXCEPT (PLM*)
```

RCONFIGURE

Produces bootloadable output for an Intel386 operating system

Syntax

```
RCONFIGURE [(DYNAMICMEM (memory_range))]
```

Abbreviation

```
RC[(DM)]
```

Default

`RCONFIGURE` is not in effect.

Description

The `RCONFIGURE` control produces an output file that can be loaded on iRMX III OS or DOSRMX.

The `DYNAMICMEM` specification selects the memory range, which specifies the minimum and maximum dynamic memory requirements of the output task. Specify memory-range as follows:

```
min[,max]
```

Where:

min is an integer representing the minimum dynamic memory requirement of the output task. When you specify *min* only, BND386 sets minimum and maximum requirements to this value.

max is an integer representing the maximum dynamic memory requirement of the output task. The *max* specification must be greater than or equal to the minimum value or BND386 will issue an error message.

Notes

`LOAD` is automatically in effect with `RCONFIGURE`, and `LOAD` or `NOLOAD` controls in the invocation are ignored.

Examples

1. In the following example, BND386 produces an output module that can be loaded on an Intel386 operating system. BND386 instructs the operating system to allocate 00AFH bytes of dynamic memory for the output task.

   ```
   BND386 MOD1.OBJ RCONFIGURE (DYNAMICMEM(00AFH))
   ```

2. In the following example, BND386 produces an output module that can be loaded on an iRMX operating system. BND386 instructs the operating system to allocate at least 00AFH bytes and at most 0AF45H bytes for the output task.

   ```
   BND386 MOD1.OBJ, MOD2.OBJ RC (DM(00AFH, 0AF45H))
   ```

RENAMESEG

Renames code and/or data segment

Syntax

```
RENAMESEG (old_seg_name TO new_seg_name [,...])
```

Abbreviation

```
RN
```

Default

`RENAMESEG` is not in effect.

Description

The `RENAMESEG` control changes the names of input segments. All input segments named *`old_seg_name`* are renamed *`new_seg_name`* to provide control over segment combination. All input segments are first renamed and then combined as necessary. The specified segment name cannot be longer than 40 characters.

Any reference to a renamed segment in other controls in the command line, should refer to the new segment name.

Examples

In the following example, the segment MODCODE1 in MOD1.OBJ is renamed MODCODE2. The new name is then used in the `SEGSIZE` control.

```
BND386 MOD1.OBJ RN (MODCODE1 TO MODCODE2) SS (MODCODE2 (+20))
```

SEGSIZE

Changes length of stack or data segment in output object module

Syntax

```
SEGSIZE (seg_name ([+/-]size)[,...])
```

Abbreviation

```
SS
```

Default

Segment size reflects combination, if any.

Description

The `SEGSIZE` control changes the length of one or more writable stack or data segments in the BND386 output object module. The length of a segment is the memory space it requires. Segment length can be increased, decreased, or set to a specific decimal value for a specified segment-name, as follows:

- To increase segment length, set the size to a positive number, +*n*, where *n* is the number of bytes by which the segment length is to be increased.
- To decrease segment length, set the size to a negative number, -*n*, where *n* is the number of bytes by which the segment length is to be decreased.
- To specify a particular segment length, set the size to a hexadecimal number in bytes. For USE16 segments, the size must be between 0 and 0FFFFH (64K bytes). For USE32 segments, the size must be between 0 and 0FFFFFFFFH (4 gigabytes)

CAUTION

The iRMX OS assumes stack sizes of at least 1024 bytes and normally provides a minimum stack by increasing a stack of less than 1 Kbyte up to 1 Kbyte. However, it does not do this when it is passed an explicit stack pointer. If you set a SEGSIZE(STACK(x)) directive for a module, where x is less than 1024, the application may fail to load with the Soft-Scope debugger. When you bind an application with the SEGSIZE(STACK(x)) directive, always make sure that x is at least 1024.

BND386 pads all segments except those that are execute-only. The SEGSIZE control supersedes this padding. Note that the addition of bytes due to alignment cannot be suppressed, and the length of a segment is not the exact sum of all combined segments.

Zero-length segments are purged from loadable modules but not from linkable modules. Therefore, zero-length segments are not listed in the segment map in the print file when the LOAD control is in effect.

BND386 issues a warning if the specified segment does not have the write attribute; it also issues a warning whenever the size of a segment is decreased. BND386 ensures that references in the output module to any segment affected by SEGSIZE are within the specified limits.

Examples

In the following example, the length of each loadable module segment is changed as follows:

- SEG1 is increased by 47 bytes.
- SEG2 is decreased by 47 bytes.
- SEG3 is set to 511 bytes.

The length of every other segment depends on segment combination.

```
BND386 MOD3.OBJ, MOD4.OBJ SS (SEG1(+002FH), SEG2(-002FH), &
SEG3(01FFH))
```

TYPE/NOTYPE

Enables or suppresses type checking

Syntax

```
TYPE
NOTYPE
```

Abbreviations

```
TY
NOTY
```

Default

```
TYPE
```

Description

The TYPE control performs type checking between public and external symbols of the same name. A warning is issued when a mismatch is found. When used with NOLOAD, TYPE includes type definitions for all external and public symbols in the linkable output module.

NOTYPE suppresses type checking and causes BND386 to omit type definitions from the linkable output module. When NOTYPE is in effect, there is no type checking for unresolved symbols in a loadable file.

Examples

1. In the following example, type checking is done on all external and public symbols.

   ```
   BND386 MOD1.OBJ, MOD2.OBJ TYPE
   ```

2. In the following example, no type checking is done.

   ```
   BND386 MOD3.OBJ, MOD4.OBJ NOTYPE
   ```

Print File

The print file contains the following sections (see Figure 2-4).

- Header
- Segment map
- Input module list
- Unresolved external symbol list (if BND386 has not been able to resolve one or more external symbol references)
- Error and warning messages if present
- Summary of memory usage

```
386(TM) BINDER                  dd/mm/yy     hh/mm/ss         PAGE  1

system_id iRMX III 386(TM) BINDER, Vx.yVX

INPUT FILES:    MAIN.OBJ,    UTIL.OBJ
OUTPUT FILES:   MAIN.LNK
CONTROLS SPECIFIED:  NOLOAD, NAME=EXAMPLE_MAIN

*** WARNING 151:      UNRESOLVED EXTERNAL SYMBOLS

SEGMENT MAP

LIMIT        ACCESS     ALIGN  USE    COMBINE TYPE    COMBINE NAME

00000236H     ER        BYTE   USE32     NORMAL          CODE
000041CBH     RW        DWORD  USE32     NORMAL          DATA
FFFFDFFFH     RW        DWORD  USE32     STACK           STACK

INPUT MODULES INCLUDED:

MAIN    (MAIN.OBJ)
UTIL    (UTIL.OBJ)

UNRESOLVED EXTERNAL SYMBOLS:

SYMBOL NAME          REFERRING MODULE

DQATTACH             MAIN
DQCLOSE              MAIN
DQCREATE             MAIN
DQDETACH             MAIN
DQEXIT               MAIN
DQOPEN               MAIN
DQREAD               MAIN
DQWRITE              MAIN

PROCESSING COMPLETED.            1 WARNING,               0 ERRORS
```

Figure 2-4. Sample BND386 Print File

The following explanations of the print file sections are accompanied by generic examples of the print file format.

Header

The print file header summarizes the invocation specifications (see Figure 2-5).

```
386(TM) BINDER                  mm/dd/yy   hh:mm:ss      PAGE number

system_id iRMX III 386(TM) BINDER,       Vx.yVX

INPUT FILES:    filename1,     filename2  ...
OUTPUT FILES:   filename_n
CONTROL SPECIFIED:  control1, control2 ...

        <------------ (warnings, if any, will appear here)
```

Figure 2-5. BND386 Print File Header

Segment Map

The segment map provides the following information for each segment in the output object module (see Figure 2-6):

- LIMIT: the segment limit, in bytes, with a suffix of H (for hexadecimal). The limit is the offset of the last byte in a segment. Unless the `SEGSIZE` control is used to specify an exact segment length, the limit may be a few bytes longer than the actual limit (to accommodate references to the last byte of the segment).

  If a segment has the length of zero, then EMPTY is printed in the LIMIT column. Zero-length segments are purged from loadable modules. Therefore, they are not listed in the segment map when `LOAD` is in effect. Segments that have expand down attribute will have a LIMIT of (0 - segment size - 1).

- DPL: the descriptor privilege level.

- ACCESS: one or more of the following segment attributes:
  - — C (conforming): a segment that can be shared by programs that execute at lower (numerically higher) privilege levels without using gates.
  - — D (expand down): a nonexecutable segment whose limit can be extended toward lower-order addresses at run time.
  - — EO (executable only): a code segment that can only be executed, not read.
  - — ER (executable and readable): a code segment that can be executed and read.
  - — RO (readable only): a data segment that can only be read.
  - — RW (readable and writable): a data segment that can be read from and written to.
- ALIGN: two segments are combined according to the stricter of the two alignment types; e.g., if a byte and a word are combined, the new segment is type word. (The following types are valid: byte, word, dword, quad, para, inpage, and page.)
- USE: can be USE16 or USE32. USE16 segments have a limit of 64K bytes and USE32 segments have a limit of 4 gigabytes. For code segments, this attribute determines the default addressing width and the default operation width.
- COMBINE TYPE: one of the following, describing information in the segment:
  - — BLANKCOMMON: segments that contain FORTRAN-386 that are not named.
  - — COMMON: segments that contain FORTRAN-386 common blocks that are named.
  - — NOCOMBINE: code or data segments that cannot be combined: for example, ASM386 segments that are not declared PUBLIC and PL/M-286 segments compiled under the LARGE model.
  - — NORMAL: code or data segments that can be combined; for example, ASM386 segments that are public or PL/M-386 segments.
  - — STACK: stack segments.
- COMBINE NAME: the name of the segment. In assembler programs, this name is given by the user. In high level language programs such as PL/M, the name is given by the compiler, and depends on the model of segmentation used.

```
SEGMENT MAP
LIMIT        ACCESS    ALIGN    USE    COMBINE TYPE    COMBINE NAME
```

```
xxxxxxxxH     xxxx      xxxx     xxxxx  xxxxxxxxxxx  xxxxxxxxxxxxxxxx
xxxxxxxxH     xxxx      xxxx     xxxxx  xxxxxxxxxxx  xxxxxxxxxxxxxxxx
    .          .         .        .        .              .
    .          .         .        .        .              .
xxxxxxxxH     xxxx      xxxx     xxxxx  xxxxxxxxxxx  xxxxxxxxxxxxxxxx
```

Figure 2-6. BND386 Print File Segment Map

Input Modules List

The print file contains a list of all input modules processed by BND386 and the file name in which the modules appeared in the input list (see Figure 2-7). The order is the same in which the input modules were encountered in the input list.

```
INPUT MODULES INCLUDED

"module_name1" ("filename1")
                .
                .
                .
"module_name_n" ("filename_n")
```

Figure 2-7. BND386 Print File Input Module Map

Unresolved Symbols List

Figure 2-8 shows the format of the section that provides information for each unresolved reference in the segments processed:

- SYMBOL NAME: the name of the referenced symbol.
- REFERRING MODULE: the name of the first input module that contains a reference to the symbol.

```
UNRESOLVED EXTERNAL SYMBOLS:

    SYMBOL NAME                 REFERRING MODULE

    xxxxxxxxxxxxxx              xxxxxxxxxxxxx
          .                           .
          .                           .
          .                           .
    xxxxxxxxxxxxxx              xxxxxxxxxxxxx

        <-------------(errors/warnings, if any, will appear here)
```

Figure 2-8. BND386 Print File Unresolved Symbol List

Warning and Error Messages

Appendix A defines the error and warning messages that can appear in the print file.

Using BND386: Examples

The example below illustrates the two stages in developing an application program. First, modules that contain application code are linked together. The resulting linkable module is then linked with a system file to create a loadable module. The loadable module is a program that can be executed on an iRMX III operating system.

Figure 2-9 shows the BND386 print file produced when two linkable iC-386 modules (named MAIN.OBJ and UTIL.OBJ) are processed with BND386. The DEBUG compiler control provides symbolic information useful later during debugging. The following sequence is used to invoke the compiler:

```
iC386 MAIN.C DEBUG
iC386 UTIL.C DEBUG
```

After compilation, the BND386 NOLOAD control is used to create a combined linkable module from MAIN.OBJ and UTIL.OBJ. The linkable module that is produced is placed in the file MAIN.LNK (by default) and named EXAMPLE.

```
BND386 MAIN.OBJ, UTIL.OBJ NOLOAD NAME (EXAMPLE)
```

The linkable output module contains three segments--an executable-readable segment (code), a readable-writable data segment (data), and a readable-writable stack segment (stack) (see Figure 2-9).

The value in the LIMIT column can be different from the sum of the lengths of the input segments. BND386 will change the combined segment lengths as follows:

- When determining the output segment limits during combination, BND386 aligns the second and third segments according to their alignment attribute.
- The segment limit (the offset of the last byte in a segment) is equal to the segment size minus one.
- The segment can be padded by one to six bytes. In this example, code is padded by one byte and data by two bytes.

```
386(TM) BINDER              dd/mm/yy   hh/mm/ss            PAGE   1

system_id       iRMX III 386(TM) BINDER Vx.yVX

INPUT FILES:     MAIN.OBJ,     UTIL.OBJ ...
OUTPUT FILES:    MAIN.LNK
CONTROLS SPECIFIED:  NOLOAD, NAME=EXAMPLE

*** WARNING 151:     UNRESOLVED EXTERNAL SYMBOLS

SEGMENT MAP

LIMIT         ACCESS   ALIGN   USE    COMBINE TYPE   COMBINE NAME

00000236H      ER      BYTE    USE32     NORMAL         CODE
000041CBH      RW      DWORD   USE32     NORMAL         DATA
FFFFBFFFH      RW      DWORD   USE32     STACK          STACK

INPUT MODULES INCLUDED:

MAIN    (MAIN.OBJ)
UTIL    (UTIL.OBJ)

UNRESOLVED EXTERNAL SYMBOLS:

SYMBOL NAME                REFERRING MODULE

DQATTACH                        MAIN
DQCLOSE                         MAIN
DQCREATE                        MAIN
DQDETACH                        MAIN
DQEXIT                          MAIN
DQOPEN                          MAIN
DQREAD                          MAIN
DQWRITE                         MAIN

PROCESSING COMPLETED.           1 WARNING                0 ERRORS
```

Figure 2-9. BND386 Print File for Linkable Output Containing Unresolved Symbols

Because the modules MAIN.OBJ and UTIL.OBJ were linked with no errors, the output module is valid and ready to convert to a loadable module. Linking the module EXAMPLE with UDIIFC32.LIB resolves the external references because UDIIFC32.LIB contains the iRMX OS UDI system call interfaces. The following invocation creates a loadable program and the print file shown in Figure 2-10:

```
BND386 MAIN.LNK, UDIIFC32.LIB CF (EXMPL1.CFL) OBJECT (EXMPL1)
```

BND386 creates a loadable module, because `LOAD` is in effect by default.

As "controls specified" in Figure 2-10 indicates, the control file contains the following controls:

```
DEBUG
SEGSIZE (stack (5000H))
```

Using the control file in this case is equivalent to including `DEBUG` and `SEGSIZE` in the invocation line.

The resulting output object module is placed in a file named EXMPL1, as specified by the `OBJECT` control. This module contains segments from the module created in the first BND386 invocation, EXAMPLE, and from the module in the system library UDIIFC32.LIB.

The segment named font/size has a segment limit of 5000H bytes. This is due to the `SEGSIZE` control used, which specifies that the limit of the segment named stack is to be 5000H bytes.

Because the system library UDIIFC32.LIB contains public definitions for all symbols declared external in EXAMPLE, no references remain unresolved. Any unresolved references are listed in the print file.

The output module in file EXMPL1 is ready to load or debug on an iRMX OS. The `DEBUG` control in effect during both compilation and binding produces symbolic information that can be used by MAP386 and by a symbolic debugger like Soft-Scope.

```
386(TM) BINDER                          dd/mm/yy   hh/mm/ss     PAGE  1
system_id       iRMX III 386(TM) BINDER Vx.yVX

INPUT FILES:     MAIN.LNK,     UDIIFC32.LIB
OUTPUT FILES:    EXMPL1.386
CONTROLS SPECIFIED:   OJ(EXMPL1), DB, SS(STACK(5000H))

SEGMENT MAP

LIMIT         ACCESS   ALIGN   USE    COMBINE TYPE   COMBINE NAME

00000236H      ER      BYTE    USE32     NORMAL         CODE
000041CBH      RW      DWORD   USE32     NORMAL         DATA
FFFFBFFFH      RW      DWORD   USE32     STACK          STACK

INPUT MODULES INCLUDED:

main    (MAIN.LNK)
COPYRIGHT_YEAR(s)_INTEL_CORPORATION (udiifc32.lib)

PROCESSING COMPLETED.  0 WARNINGS, 0 ERRORS
```

Figure 2-10. BND386 Print File for Loadable Modules

❑❑❑

Using the Intel386 Librarian 3

The LIB386 librarian organizes linkable object modules, which have been produced by Intel386 compilers, assemblers, or other Intel386 utilities, into libraries. You can then create, examine, change, search, copy, and otherwise manipulate library files with LIB386 commands.

You can construct object libraries around modules with common characteristics (i.e., modules that perform similar functions or are required for a particular system). For example, one library might contain modules that perform mathematical functions; another, modules that perform I/O routines.

When invoking other Intel386 utilities such as BND386, MAP386, or the BLD386 System Builder, you can gain access to all the applicable modules in a library simply by specifying the library file name, rather than having to type the names of all the modules.

LIB386 operates in two modes: foreground (NOBATCH) or background (BATCH). In foreground mode, LIB386 queries, provides intermediate displays, and expects input from the keyboard. In background mode, LIB386 processes commands that have been redirected from a command file (or some type of batch file), and does not query you or expect interaction. Refer to the individual LIB386 commands for the effects of background execution. See Using LIB386: Examples for a background execution example.

Major Functions of LIB386

LIB386 performs the following major functions:

- Creates new libraries
- Updates existing libraries by adding, deleting, or replacing modules
- Backs up the library currently being processed before it is modified
- Examines information in the libraries (e.g., names of modules or names of public symbols)
- Lists the names of library modules that contain specified public symbols

- Accesses or changes library-related information (e.g., the name or the version number)
- Compresses the data structure of the library being processed, thereby speeding up the retrieval operation of other Intel386 utilities (such as BND386) that access library files
- Requests help and receives a brief description of the specified LIB386 command

Refer to Appendix B for LIB386 error messages.

Input and Output

LIB386 accepts object files as input. Object files contain a single object module, a sequence of object modules, or libraries (see Figure 3-1).

LIB386 output consists of library files, backup library files, console messages, and a listing file, depending on the commands and controls that are specified. A listing file can list modules in a library, public symbols in a library, or public symbols in a particular module.

The Target Library

The library file that LIB386 is processing at any given time is called the target library. The file you specify in the invocation line or with the GET command becomes the target library file. All LIB386 commands act on the target file. (The LIST command is an exception to this processing rule; LIST can operate on library files other than the target library file.)

A specified library file remains the target file until you exit LIB386 or specify another library as the target library.

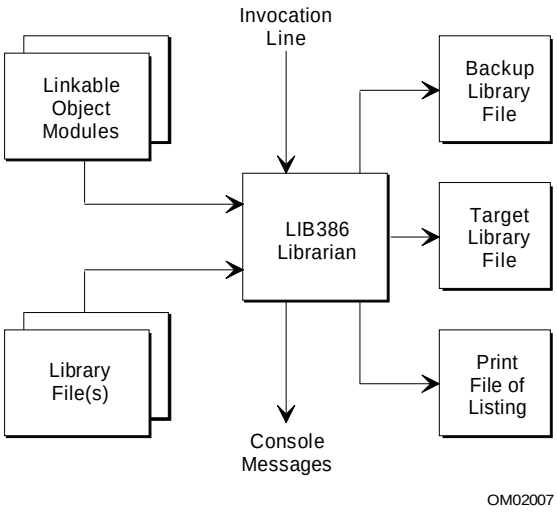

Figure 3-1. LIB386 Input and Output

Library Sessions

Single library file processing is referred to as a single LIB386 session. Only one target library can be processed per session. However, multiple library files can be processed sequentially without exiting LIB386. To do this sequential processing, close the current target library and then initialize another target library file for the next session. For more details on single and multiple sessions, see Using LIB386: Examples.

Invoking LIB386

DOS and iRMX Invocation Syntax

To invoke LIB386 on a DOS or iRMX operating system, use the following syntax:

```
LIB386 [filename] [BATCH | NOBATCH] [BACKUP | NOBACKUP]
```

Where:

filename is the name of the target library file. Specify the file name according to the operating system requirements.

A library session is begun in one of two ways: by specifying a file name, which will initialize the target library, or by using the GET command to initialize the target library file. See the GET entry in this chapter for details on initializing target library files.

`BATCH (BA)`
specifies non-interactive mode. LIB386 processes commands that have been redirected from a command file.

`NOBATCH (NOBA)`
specifies interactive mode. LIB386 runs interactively and expects input from the keyboard. `NOBATCH` is the default.

`BACKUP (BU)`
specifies that LIB386 create a backup file of the target library at the start of a new session. The backup file contains the version of the target library created during the last session, with extension .LBK. `BACKUP` is the default. `BACKUP` can be executed at the command level or as an invocation control.

`NOBACKUP (NOBU)`
specifies that LIB386 does not create a target library backup file when a new session starts. `NOBACKUP` can be executed at the command level or as an invocation control.

Invocation Controls

BACKUP, NOBACKUP, BATCH, and NOBATCH are the only controls that can be specified in the LIB386 invocation line. BACKUP and NOBATCH are the defaults.

BACKUP and NOBACKUP can be specified as invocation controls, and they can also be used as commands.

BACKUP creates a backup copy of a target library file each time a LIB386 session begins. LIB386 gives each backup file the same name as the original target library, with extension .LBK.

When used as commands, BACKUP and NOBACKUP override the BACKUP or NOBACKUP controls used in invoking LIB386. See the BACKUP entry in this chapter for more details on using BACKUP and NOBACKUP.

LIB386 Defaults

When no controls are specified in the invocation line, LIB386 does the following:

- Begins operating in foreground mode
- Displays the sign-on message and the foreground prompt, an asterisk (*)
- At the beginning of a session, displays a line identifying the target library
- If BACKUP is specified or implied, creates a backup copy of the target library at the beginning of a session, with extension .LBK
- Awaits command input

If no file name is specified in the invocation line, LIB386 waits until a target library is initialized with the GET command.

Console Messages

In foreground mode, LIB386 displays sign-on and sign-off messages, queries for user response, and presents display messages and error messages.

In background mode, LIB386 does not display queries for user response, because the utility is not interactive. LIB386 displays all other messages, including sign-on and sign-off messages, display messages, and error messages.

When you invoke LIB386 without specifying a file name, the librarian signs on as follows:

```
system_id iRMX III 386(TM) LIBRARIAN, Vx.y
Copyright years Intel Corporation
*
```

Where:

system_id is the identifier and version number of the operating system.

V*x.y* is the LIB386 version number

years is the copyright year or years.

*** is the LIB386 prompt. The prompt appears only in foreground mode.

When you specify a file name, LIB386 displays the sign-on message and one of two messages, as follows:

- If the file name is the name of a new library:

```
TARGET LIBRARY:[new_library]  date/time_last_modified
**
```

- If the file name is the name of a library that already exists:

```
TARGET LIBRARY: target_lib-name version date/time_last_modified
**
```

Where:

target_lib_name
is the name of the target library in the invocation line.

version_number
is the version number of the target library.

date/time_last_modified
date and time the target library was last processed.

During processing, LIB386 issues warnings, error messages, or fatal error messages if it encounters any problems. When LIB386 encounters a fatal error condition, an error message and the following sign-off message are displayed at the console:

```
PROCESSING ABORTED
```

See Appendix B for additional information about error conditions.

LIB386 signs off when you exit from the librarian or from the current LIB386 session, as follows:

```
library_filename date/time_last_modified
n MODULES ADDED, m MODULES DELETED
```

Where:

n and *m* are the numbers of added and deleted modules.

Queries

The following is an example of a LIB386 query:

```
All Changes Lost, OK? [Y/N]
```

This query appears when you specify the `QUIT ABORT` or `QUIT INITIALIZE` command and the target library has been changed but not updated. See the `QUIT` entry in this chapter for details on responses to this query.

Display Messages

An example of a display message is the target library's identification banner. A banner is displayed in the following form when the target library file for the session is initialized:

`TARGET LIBRARY:` *`target_lib_name version_date/time_last-modified`*

Another display example is the listing of the library module names. See the LIST entry in this chapter for more details.

A third type of LIB386 display message is the help summary. The help summary appears on screen when you enter the HELP command and briefly describes each LIB386 command. See the HELP entry in this chapter for more details.

Error Messages

Error messages are displayed when LIB386 encounters error conditions. Appendix B contains a listing and explanation of the LIB386 error messages.

LIB386 Commands

Hierarchical Levels

The LIB386 command set contains two elements: the invocation line and library commands. These elements are organized into a hierarchical structure. The three hierarchical levels are the operating system level, the initial command level, and the action command level (see Figure 3-2).

The operating system level is the outermost level; you can invoke LIB386 and specify the BATCH, NOBATCH, BACKUP, and NOBACKUP controls. When you exit LIB386, control returns to the operating system.

When no file name is specified in the invocation line, control transfers to the initial level. A single asterisk (*) prompt indicates that control is at the initial level. The BACKUP, NOBACKUP, GET, HELP, LIST, and QUIT commands are available at this level. Use the GET command at the initial level to initialize the action command level. (See the GET entry in this chapter.) Use the QUIT command to return to the operating system level.

If the file name is specified in the invocation line, control transfers directly to the action command level, indicated by the double asterisk (**) prompt. Most of the LIB386 commands are available at this level. Use these commands to alter the target library contents. Use the QUIT command to return to the initial level or to the operating system level.

Transfer of Levels

Control transfers inward from the operating system to the action command level when the target library file is specified on the invocation line. A double asterisk prompt (**) indicates the action command level. All action level commands are available for execution. Invoking LIB386 contains details on specifying the target library file.

Control transfers inward from the operating system to the initial command level if the target library file is not specified with the invocation. A single asterisk prompt (*) indicates the initial command level. All initial level commands (e.g., GET, LIST) can be executed. GET transfers control from the initial command level to the action command level.

The QUIT command sequence must be executed to transfer control outward from the action or initial command level. See the QUIT entry in this chapter for details.

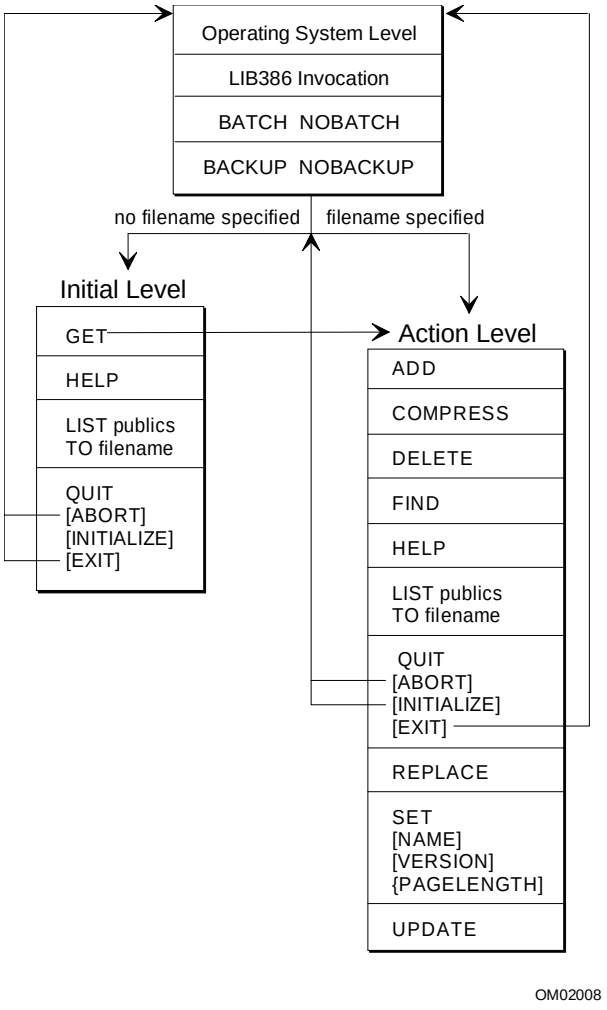

Figure 3-2. Levels of LIB386 Command Set

Effect of Entering the Interrupt Character

The interrupt character terminates LIB386 activities (except for BACKUP), in foreground mode. However, LIB386 completes the portion of the command that was executing before the interrupt character was entered. For DOS systems, the interrupt character is <Ctrl-Break>.

If BACKUP is specified either as a command or as a control and the interrupt character is entered, LIB386 finishes backing up the library before interrupting itself.

Summary of Commands

Table 3-1 summarizes LIB386 commands for DOS and iRMX operating systems. The default column shows the condition in effect when the control is not specified.

Table 3-2 lists abbreviations of LIB386 controls. Each LIB386 control is listed alphabetically and described in detail in the section following Table 3-2.

Command Syntax

To invoke a LIB386 command, use the following syntax:

```
command [parameters] [;comments]
```

Where:

command is the command specified (e.g., ADD).

parameters
are one or more items required by the command. Separate parameters by commas or blank spaces unless otherwise noted under the individual command.

comments is the text between the semicolon (;) and a new line. LIB386 ignores this text.

You can continue the invocation line on additional lines by typing an ampersand (&) before the line terminator. When LIB386 encounters a line terminator, the command executes.

Table 3-1 summarizes LIB386 commands for DOS and iRMX operating systems. The default column shows the condition in effect when the command is not specified.

Each LIB386 command is listed alphabetically and described in detail in the section following Table 3-2.

Table 3-2 lists abbreviations of LIB386 commands. These abbreviations are repeated in the following sections describing each command in detail.

Table 3-1. LIB386 Commands for DOS and iRMX Operating Systems

Command Syntax	Description	Default
ADD {*filename*[*(mod_name* [,...])]}[,...]	Adds object modules to target library (at action level)	None
BACKUP NOBACKUP	Enables or suppresses backup file for target library file (at initial level)	BACKUP
COMPRESS	Physically removes deleted	No compression
DELETE {*mod_name*[,...]|*}	Logically removes modules from target library (at action level)	None
FIND *symbol*[,...]	Searches target library for the public symbol(s) (at action level)	None
GET *filename*	Initializes target library for session (at initial level)	None
HELP	Summarizes each LIB386 command, keyword, and control	None
LIST {*filename1*[(*mod_name* [,...])]}[,...] [TO *filename2*][PUBLICS][1]	Lists module names of library files (at initial level and action levels)	Console is default output device. No publics
QUIT [ABORT | EXIT | INITIALIZE]	Terminates current session (at initial and action levels). ABORT, EXIT, and INITIALIZE are QUIT command controls	EXIT control of QUIT command only in BATCH mode
REPLACE *mod_name1* BY *filename* [(*mod_name2*[,...])]	Replaces one module from the target library with one or more modules from another object file (at action level)	None
SET [N] [V] [PL] [,...]	Changes name (N) or version number (V) of target library or changes the page length (PL) of listings (at action level)	N = 1 blank space V = 4 blank spaces PL = 23 lines
UPDATE	Updates current target library file (at action level)	None

1 filename1 is required at the initial level, but is optional at the action level. The target library is the default for filename1.

Table 3-2. Abbreviations for LIB386 Commands

Commands	Abbr.	Commands	Abbr.
ADD	A	QUIT	Q
BACKUP	BU	ABORT	[A]
COMPRESS	C	EXIT	[E]
DELETE	D	INITIALIZE	[I]
FIND	F	REPLACE	R
GET	G	SET	
HELP	H	NAME	[N]
LIST	L	VERSION	[V]
NOBACKUP	NOBU	PAGELENGTH	[PL]
		UPDATE	U

ADD

Add object modules to target library

Syntax

```
ADD {filename [(mod_name[,...])]} [,...]
```

Where:

filename is the input file. The input file must be a linkable or library file. The file name must be specified according to operating system requirements.

mod_name is the object modules or modules from the input file to be added to the target library.

Abbreviation

A

Default

ADD is not in effect.

Description

The ADD command lets you add input object modules to the target library. Only linkable object modules generated by Intel386 compilers, assemblers, or other Intel386 utilities can be added. The input file specified may contain a single object module or a sequence of object modules; or it may be another library file. The input file cannot have the same name as the library. ADD is executable only at the action command level.

Unless modules are specified in parentheses, all modules in the input file are added to the target library. This is true whether or not the input file is a library. The target library may be initialized with the invocation line, or with the GET command at the initial level for the current LIB386 session. If modules are specified in parentheses, only the specified modules are added to the target library.

Uninitialized global variables in C modules are BSS, not publics. (To make them publics, you must initialize them.) BSS variables are not used by BND386 or the builder to pull in library modules, which are not otherwise required to satisfy known externals.

A module in an object library is active if it is both logically and physically available. (See the DELETE entry in this chapter for details.) When a module specified with the ADD command is not active in the input file, the following error message appears at the console:

```
MODULE NOT FOUND
FILE: filename
MODULE: mod_name
```

If the specified module name is identical to the name of a module already in the target library, the module is not added to the library. The following error message appears at the console:

```
DUPLICATE MODULE
FILE: filename
MODULE: mod_name
```

If a public symbol name in the specified module is already active in the target library, the module is not added to the library. The following error message appears at the console:

```
DUPLICATE PUBLIC
FILE: filename
MODULE: mod_name
PUBLIC SYMBOL: public symbol name
```

If ADD is for a single object file that is invalid, LIB386 does not add any modules from the file. If ADD is for two or more files and only one of the files is invalid, LIB386 adds modules from all files but the invalid file.

In either case, LIB386 continues processing and the following error message appears at the console:

```
INVALID OBJECT FILE
FILE: filename
```

If ADD is attempted for a file generated by an 80286 compiler, assembler, or utility, the following error message appears at the console:

```
INPUT FILE IS A 286 OBJECT FILE
FILE: filename
```

Examples

1. In the following example, all the modules in the files MOD1.OBJ and MOD2.OBJ are added to the target library.

   ```
   **ADD MOD1.OBJ, MOD2.OBJ
   ```

2. In the following example, FILE1.LIB is a library file. MOD1 and MOD2 from this input file are added to the target library.

   ```
   **A FILE1.LIB(MOD1, MOD2)
   ```

BACKUP

Enables or suppresses backup file for target library file

Syntax

BACKUP
NOBACKUP

Abbreviations

BU
NOBU

Default

BACKUP

Description

The BACKUP command tells LIB386 to create a backup file before the start of each library session. When BACKUP is used with the GET command, LIB386 copies the current contents of the target library file to the backup file before the session begins. Backup files have the same file name as the target library, with extension .LBK.

BACKUP and NOBACKUP are the only LIB386 commands that remain in effect during all sessions under a single LIB386 invocation.

The BACKUP command can be executed in the invocation line or at the initial command level. See Invoking LIB386 for more information on using BACKUP or NOBACKUP in the invocation line.

Examples

1. In the following example, the library file MOD2.LIB is initialized, and the GET control identifies this file as the target library file. BACKUP is specified; therefore, LIB386 copies MOD2.LIB to the backup file MOD2.LBK when the library file is initialized for the session.

   ```
   *BU
   GET MOD2.LIB
   target library : FROUTINES X110 05/15/86
   ```

2. In the following example, no backup files are created during this invocation of LIB386.

   ```
   LIB386 MOD1.LIB NOBU
   ```

COMPRESS

Physically removes deleted modules from target library

Syntax

```
COMPRESS
```

Abbreviation

```
C
```

Default

`COMPRESS` in not in effect.

Description

The COMPRESS command physically removes data that has already been logically deleted or replaced. Execute COMPRESS after you have executed several deletes and/or replaces; this will give LIB386 and other Intel386 utilities faster access to the target library.

The DELETE command logically deletes a module but does not necessarily physically remove it from the target library file. You can use the COMPRESS command to physically remove the module from the object library. See the description of the DELETE command for more information.

COMPRESS is executable only at the action command level.

Example

In the following example, the target library is compressed by executing COMPRESS at the action command level. The target library is restructured for maximum efficiency.

```
**C
```

DELETE

Logically removes modules from target library

Syntax

DELETE *mod_name*[,...]

Where:

mod_name	is the module to be deleted from the target library.
*	deletes all modules from the target library.

Abbreviation

D

Default

DELETE is not in effect.

Description

The DELETE command logically removes specified modules from the target library. Public symbols in deleted modules are also logically removed. DELETE does not physically remove the modules, but there is no way to refer to modules that have been deleted. Use the COMPRESS command to physically remove modules that have already been logically deleted.

When you are in foreground mode and specify, "delete all modules" with the construction, DELETE (*), LIB386 queries:

```
Are you sure? [Y/N]
```

Type y or yes to confirm that you want to delete all modules. Any response besides y or yes causes LIB386 to ignore the DELETE command and issue the action command level prompt (**). In batch mode, you will not have the opportunity to confirm; LIB386 will delete all modules upon command.

An object library module is active if the module is both logically and physically available. For each non-active module you specify, the following error message is displayed:

```
MODULE NOT FOUND
FILE: filename
MODULE: mod_name
```

DELETE is executable only at the action command level.

Examples

1. In the following example, MOD1 and MOD2 are logically deleted from the target library.

```
**D MOD1,MOD2
```

2. In the following example, LIB386 deletes all modules from the target library.

```
**D *
Are You Sure? [Y/N] y
```

FIND

Searches target library for public symbol

Syntax

FIND *symbol*[,...]

Where:

symbol is the name of the public symbol that LIB386 is to search for in the target library.

Abbreviation

F

Default

FIND is not in effect.

Description

The FIND command first verifies whether the specified public symbol is active (that is, logically and physically available) in the target library. If the symbol is active, the symbol name and the name of the module containing the symbol are displayed.

If the symbol is not active in the target library, the following message appears at the console:

public symbol, NOT FOUND

Where:

public symbol is the specified public symbol.

FIND is executable only at the action command level.

Examples

1. In the following example, CHKSTATUS is the public symbol that is searched for in the target library. The search finds CHKSTATUS in CHECKMOD.

```
**F CHKSTATUS
CHKSTATUS, IN MODULE CHECKMOD
```

2. In the following example, the search for CONVRBYTE finds that this public symbol is not active in any of the target library's modules.

```
**F CONVRBYTE
CONVRBYTE, NOT FOUND
```

3. In the following example, the search for symbols A and B finds that both are active in the target library's modules.

```
**F A,B
A, IN MODULE X
B, IN MODULE Y
```

GET

Initializes target library file

Syntax

`GET` *`filename`*

Where:

`filename` is the library file that contains the target library. The file name must be specified according to the host operating system requirements.

Abbreviation

`G`

Default

`GET` is not in effect.

Description

The GET command initiates a LIB386 session with the specified library file as the target library. All subsequent modifications, such as deletions and additions, affect the target library.

After executing GET, LIB386 does the following:

- Transfers control to the action command level, giving access to most LIB386 controls.
- Signals the start of the LIB386 session by identifying the target library with the target library banner, which contains the name, version number, and last date of modification of the target library.

The library file is automatically checked for read and write mode. If the target library is write-protected, the GET command is successfully executed, but the following warning appears at the console:

```
***WARNING: TARGET FILE IS WRITE PROTECTED
```

You can use GET to create a new library file: specify a new library file name, and then add modules with the ADD command. Use the SET command to set the name and version number of the library. The banner displayed for a new library is as follows:

`TARGET LIBRARY: [`*`new_library`*`] `*`date_and_time_last_modified`*

To get another library file, you must first terminate the current session with the QUIT command before you can request another GET.

GET is executable only at the initial option level.

Examples

1. In the following example, LIB386 initiates the library file named MOD2.LIB. FPFUNCTIONS is the target library for this session.

```
*G MOD2.LIB
TARGET LIBRARY: FPFUNCTIONS X110 01/01/1986 10:04:55
```

2. In the following example, a new library file, MOD3.LIB, is created, because this file does not already exist.

```
*G MOD3.LIB
TARGET LIBRARY: [new library] 09/17/86 11:45:07
```

HELP

Summarizes LIB386 commands, keywords controls

Syntax

```
HELP
```

Abbreviation

```
H
```

Default

The HELP summary is not displayed.

Description

The HELP command displays a summary of LIB386 commands. The summary lists each LIB386 command and control with the command syntax, abbreviation, and a brief description of its function.

HELP summaries are available on the following topics:

```
ABORT       ADD      BACKUP      COMPRESS  DELETE
EXIT        FIND     GET         HELP      INITIALIZE
LIST        NAME     PAGELENGTH  PUBLICS   QUIT
REPLACE     SET      UPDATE      VERSION
```

The HELP command is available at both the initial and action command levels.

Example

In the following example, the HELP command summary is requested.

```
*H
```

LIST

Lists module names of library files

Syntax

```
LIST {[file1] [(mod_name)]}[,...] [TO file2] [PUBLICS]
```

Where:

file1	is the name of the library file whose module information is to be listed.
mod_name	is the module or modules to be listed. If no modules are specified, all modules in the file are listed. Module names should be separated by commas: for example, *mod_name1*, *mod_name2*.
file2	is the output print file.
PUBLICS	lists all public symbols in the specified modules.

Abbreviation

L

Default

If *file1* is not specified at the action command level, the target library is used. If TO *file2* is unspecified, list information is displayed at the standard output device.

The listing defaults are as follows:

- The names of all modules in the library are listed.
- Public symbol names are not listed unless the PUBLICS control is specified.
- The first portion of the list appears 23 lines per page in NOBATCH mode. The listing is continuous in BATCH mode. The listing is also continuous if *file2* is specified.

Description

The `LIST` command displays the names of library modules and, as a control, public symbols on the standard output device. When using LIST at the initial command level, you must specify *`file1`*, because no target library has yet been initialized. At the action command level, *`file1`* defaults to the target library if not specified.

You can obtain a list of public symbols in an object module by specifying *`mod_name`* and the PUBLICS control. You can also verify that a particular module resides in the specified or default library by specifying the name of only one module without PUBLICS. LIB386 prints the module name if the module is in the library.

In both foreground and background modes, LIST first identifies the library being listed by printing a library identification banner.

In foreground mode, LIST prints the first page of the listing and then waits for input from the keyboard to determine how the next page or line is to be displayed. Use the following display controls:

- P--Displays one page at a time (the default)
- L--Displays one line at a time
- F--Displays with no breaks
- E--Ends display, immediately terminating LIST command processing.

When an input character other than P, L, F, or E is entered, LIB386 continues to list in the previous display mode.

You can set page length with the SET control. See the SET entry in this chapter for details.

In background mode, or when the `TO` *`file2`* control is specified, LIB386 prints the full list of modules and, if requested, public symbols. The display controls are unavailable.

To obtain listings for other libraries without exiting from the current session, specify the desired library's file name with the `LIST` command.

`LIST` can execute at both the initial and action command levels.

Examples

1. In the following example, the target library's three module names are printed at the standard output device.

```
** L
TARGET LIBRARY : FPFUNCTIONS X110 01/01/1986 02:45:07
MOD1
MOD2
MOD3
```

2. In the following example, MOD1.LIB's module names and public symbols are printed to a file.

```
*L MOD1.LIB TO Z.LST P
```

   File Z.LST contains the following:

```
NON_TARGET LIBRARY : LIBABC 1.2 12/04/86 06:54:34
MOD1
  PUBA
  PUBB
MOD2
  PUBC
```

3. In the following example, LIB386 verifies that the specified module is in the target library by displaying its name.

```
**L (MOD1)
TARGET LIBRARY : FPFUNCTIONS X110 01/01/1986 07:07:15
MOD1
```

QUIT

Terminates current session

Syntax

```
QUIT [ABORT | EXIT | INITIALIZE]
```

Abbreviations

```
QUIT [A | E | I]
```

Default

The `QUIT` command is not in effect unless specified. If specified in background mode, `EXIT` is the default control for the `QUIT` command. In foreground mode, you must specify `QUIT ABORT`, `QUIT EXIT`, or `QUIT INITIALIZE`; there is no default control.

Description

The `QUIT` command terminates the current LIB386 session. Three optional controls, `ABORT`, `EXIT`, and `INITIALIZE`, are available.

When `QUIT` is specified at the action command level, LIB386 identifies the current target library by displaying the target library file and the library banner, as follows:

`filename library_name version_number date_time_last_modified`

If the `QUIT` command is issued from the initial command level, no sign-off message is displayed.

In foreground mode, LIB386 queries for a control if none is specified with the QUIT command. The QUIT command is ignored if any character other than A, E, I, or the unabbreviated form of these controls is entered. If A, E, or I is specified, LIB386 proceeds through the appropriate quit sequence as explained below.

In background mode, LIB386 does not query for a control if none is specified with the QUIT command. If changes were made to the library file, an update is made automatically before the exit. `ABORT` and `INITIALIZE` are available only if specified with the `QUIT` command (for example, `QUIT ABORT`).

The `ABORT` control aborts update and transfers control to the operating system level. However, in foreground mode, if an update is required, the following query is displayed:

```
All Changes Lost, OK? [Y/N]
```

Select yes or y to complete the abort process and transfer control to the operating system level. All changes to the target library since the last update are lost. Select no, n, or any character other than y to cancel the QUIT command. Control remains at the action command level.

The EXIT control updates the target library. LIB386 does not query, and control is transferred to the operating system. In BATCH mode, EXIT is the default control.

The INITIALIZE control begins another session, transferring control from the action command level to the initial command level. At this point you can initialize another library file at the initial command level with the GET command.

When the INITIALIZE control is specified, LIB386 checks to see if the target library needs to be updated. If so, the following query is displayed:

```
All Changes Lost, OK? [Y/N]
```

Select yes or y to transfer control from the action command level to the initial command level. All changes made to the target library since the last update are lost and a new LIB386 session is initiated.

Enter no, n or any character other than y to direct LIB386 to ignore the QUIT command. Control remains at the action command level. (You can issue the UPDATE command, then reissue the QUIT command.)

The QUIT command is executable at the initial and action command levels.

Examples

1. In the following example, the current session is aborted. LIB386 discovers that NEWLIB_NAME has been altered since the last update and queries:

   ```
   All Changes Lost, OK? [Y/N]
   ```

   Yes is selected, instructing LIB386 to ignore the changes and complete the abort process. LIB386 returns control to the operating system.

   ```
   **Q A
   MOD1.LIB, NEWLIB_NAME V1.1 11/25/1986 01:03:08
   All Changes Lost, OK? [Y/N] y
   ```

2. In the following example, DRIVERS is the target library name. In foreground mode, LIB386 prompts for a control. INITIALIZE is specified and y is the response to the query:

   ```
   All Changes Lost, OK? [Y/N]
   ```

   LIB386 returns to the initial command level from the action command level.

   ```
   **QUIT
   MOD2.LIB, DRIVERS ZZ93 11/25/1986 12:05:45
   A(bort)/E(xit)/I(nitialize) =  I
   All Changes Lost, OK? [Y/N] y
   ```

REPLACE

Replaces module from target library with module(s) from another object file

Syntax

REPLACE *mod_name1* BY *filename*[(*mod_name2*[,...])]

Where:

mod_name1 is the name of the module to be replaced in the target library.

filename is the input object file to be added to the target library, or the file containing *mod_name2*.

mod_name2 is the name or names of the module or modules to be added to the target library.

Abbreviation

R

Default

REPLACE is not in effect.

Description

The REPLACE command logically deletes a module from the target library and replaces it with one or more modules from the input object file. Unless specific module names are listed with the input object file, all modules in that file are added to the target library.

If *mod_name1* and *mod_name2* are not found, the replacement is not made; the target library remains unchanged. The following error message is displayed at the console:

```
MODULE NOT FOUND
FILE: filename
MODULE: mod_name
```

If a public symbol name in the input module is already in the target library, the module is not added to the target library.

REPLACE is executable only at the action command level.

Example

In the following example, LIB386 replaces MOD5 in the target library with MOD1 from the input object file.

```
**R MOD5 BY MOD1.LIB (MOD1)
```

SET

Changes name, version number, or page length of listing

Syntax

```
SET [NAME = library_name |
     VERSION = version_number |
     PAGELENGTH = lines]
```

Where:

`NAME` is the keyword that sets the name of the target library.

`library_name`
is the name of the library. The *`library_name`* can be up to 40 characters long.

`VERSION` is the keyword that sets the version number of the target library.

`version_number`
is the version number, up to four characters long.

`PAGELENGTH`
is the keyword that sets the length of a page of listing.

`lines` is the number of lines in a page of listing, from 1 to 65535 (decimal).

Abbreviations

```
T [N | V | P]
```

Default

`NAME`: one blank space in the library banner. Before the `SET` command is used to name the target library, LIB386 displays the target library name as blank spaces in the library banner.

`VERSION`: four blank spaces in the library banner.

`PAGELENGTH`: 23 lines.

Description

The `NAME` command lets you change the name of the target library. The library name can be a string up to 40 characters long. A syntax error occurs if the specified name exceeds the maximum number of characters. When this happens, LIB386 does not change the old name.

You can change the version number of the target library using the VERSION control. The version number can be a string up to four characters long. A syntax error occurs if the specified number exceeds the maximum number of characters. When this happens, LIB386 does not change the old number. If less than the maximum number of characters is used, the version number is left-justified and padded with blanks.

You can specify the number of lines in a page generated by the LIST command using the PAGELENGTH control. The page length can be from 1 to 65,535 lines. A syntax error occurs if the specified page length exceeds the maximum number of lines. When this happens, LIB386 does not change the old page length.

To make multiple specifications (e.g., for a library name and a version number) with the same execution of SET, separate one specification from the next with a comma. If the same control is specified twice in one line, the rightmost specification takes effect.

When you enter the SET command without controls, LIB386 responds by prompting for SET specifications. Press the return key to cause LIB386 to issue successive prompts for NAME, VERSION, and PAGELENGTH. LIB386 displays the following prompts, in the sequence shown:

```
NAME = current_library_name.  NEW VALUE:
VERSION = current_version_number.  NEW VALUE:
LIST command PAGELENGTH = current_page_length.  NEW VALUE:
```

To change a specification, enter the new name, version number, or page length at the prompt.

To retain the current specification and proceed to the next prompt, press the return key. After the PAGELENGTH prompt, LIB386 displays the library banner and returns to the action command level.

LIB386 displays the new banner for the target library after the SET command has successfully completed.

Once specified, the target library name and number can be changed only with the SET command. The following characters can be used in library names and versions:

- All alphanumeric characters
- The following characters: @ . ? _
- Other characters provided they are enclosed by apostrophes (') or double quotation marks (").

It is recommended that alphanumeric characters be used, because some operating systems may not accept non-alphanumeric characters.

The SET command is executable only at the action command level.

Examples

1. In the following example, X201 is set as the version number of the target library. LIB386 immediately displays the new banner for COMMONLIB.

```
**SET VERSION = X201
TARGET LIBRARY: COMMONLIB X201 04/23/1986 06:04:05
```

2. In the following example, PASCALIB and X100 are set as the name and version number of the target library.

```
**S N = PASCALIB, VERSION = X100
TARGET LIBRARY: PASCALIB X100 05/06/1986 11:09:45
```

3. In the following example, a carriage return is entered after the SET command. LIB386 prompts for the name, version number, and page length of the target library. In this example, only the version number is changed.

```
**S
NAME = LIB1.  NEW VALUE:
VERSION = 1.  NEW VALUE: 2
LIST COMMAND PAGE LENGTH = 23.  NEW VALUE:
TARGET LIBRARY: LIB1 2 12/02/1986 10:45:07
```

UPDATE

Updates current target library file

Syntax

```
UPDATE
```

Abbreviation

```
U
```

Default

`UPDATE` is not in effect.

Description

The `UPDATE` command writes the contents of the target library to the target library file. A target library file can be updated at any time during a session.

When you attempt to update a write-protected target file, LIB386 does not update the file and issues the following message:

```
ATTEMPT TO UPDATE WRITE-PROTECTED FILE
```

Updating the target library does not terminate the session: you must use the `QUIT` command after `UPDATE` to end the session.

`UPDATE` is executable only at the action command level.

Example

In the following example, the `GET` command initializes FILE1.LIB as the target library file. The object file MOD2.OBJ is added to the target library FPFUNCTIONS. `UPDATE` incorporates this change into FILE1.LIB. The `QUIT` command ends the session.

```
**G FILE1.LIB
TARGET LIBRARY: FPFUNCTIONS X201 04/05/1986 10:22:43
**A MOD2.OBJ
**U
**Q I
```

Using LIB386: Examples

Figures 3-3 and 3-4 show examples of LIB386 use.

Single Session

Figure 3-3 is an example of a single library session in foreground mode. In this example, only one target library is processed before the QUIT command is invoked and LIB386 is exited.

```
LIB386 COMMON.LIB <cr>
system_id iRMX III 386(TM) LIBRARIAN.  Vx.yVX
Copyright year(s) Intel Corporation
TARGET LIBRARY : COMMONLIB X010 02/02/86
**ADD INIT.OBJ, ERRH.OBJ <cr>
**ADD UTIL.LIB(GETFLP,GETINT) <cr>
**DELETE STRCHK <cr>
**REPLACE START BY OLD.LIB(STARTUP) <cr>
**FIND OVRERR <cr>
OVERR, IN MODULE SETLST
**LIST PUBLICS <cr>
COMMONLIB X010 2/2/1986
       IOMOD
              PUB1INIOMOD
              PUB2INIOMOD
              PUB3INIOMOD
              PUB4INIOMOD
       PACMAN
              WEIRDNOISE
              COLOREXTRAVAGANZE
              WASTEMONEY
       SETLST
              OVRERR
              PUBINSETLST
       GETFLP
              PUB1INGETFLP
              PUB2INGETFLP
              PUB3INGETFLP
       GETINT
              PUBINGETINT
       STARTUP
              STARTANDKILL
**SET VERSION = X011 <cr>
COMMONLIB X011 02/02/1986 10:48:20
A(BORT)/E(XIT)/I(NITIALIZE) = E <cr>

5 MODULES ADDED, 2 MODULES DELETED
```

Figure 3-3. Interactive Execution Example: A Single Session

Multiple Session

Figure 3-4 is an example of a multiple library session in foreground mode. In this example, several libraries are sequentially processed, using the QUIT command with the INITIALIZE command to transfer control from the action command level to the initial command level. At the initial command level, the GET command initializes another library as the target library for the next session.

```
LIB386 <cr>
system_id iRMX III 386(TM) LIBRARIAN, Vx.yVX
Copyright year(s) Intel Corporation

*GET FIRST.LIB <cr>
TARGET LIBRARY : MAINLIB X001 04/03/1986 9:40:07
**ADD IOPROC.OBJ (IO_DRIVER) <cr>
**DELETE DUMMY_IO_PROC <cr>
**SET VERSION = X002 <cr>
MAINLIB X002 04/03/1986 9:40:20
**UPDATE <cr>
**QUIT <cr>
FIRST.LIB, MAINLIB X002 04/03/1986 9:41:16
A(bort)/E(xit)/I(nitialize) = I <cr>

1 MODULE ADDED, 1 MODULE DELETED

**GET OVERLY.LIB <cr>

TARGET LIBRARY :SUBROUTINES X001 03/02/1986 9:42:42
**ADD NEWMOD.OBJ (LPDRIVE) <cr>
**UPDATE <cr>
**QUIT <cr>
OVERLY.LIB, SUBROUTINES X001 04/03/1986 9:43:42
A(bort)/E(xit)/I(nitialize) = E <cr>

1 MODULE ADDED, 0 MODULES DELETED
```

Figure 3-4. Interactive Execution Example: Multiple Sessions

DOS Batch Session

In this example, a batch file called MAKELIB.BAT and an input file called MAKLIB.DAT contain the following:

- The LIB386 invocation line, which initializes NEW.LIB as the target library and invokes BATCH mode
- The processing commands `ADD` (used twice) and `COMPRESS`
- The QUIT command to terminate execution

The contents of the DOS command file MAKELIB.BAT are as follows:

```
LIB386 NEW.LIB BATCH < MAKLIB.DAT
```

The contents of the DOS input file MAKLIB.DAT are as follows:

```
ADD UDIIFC32.LIB
ADD\MARK\LIB386\TEST\LKS\LKSLRG.T01
COMPRESS
QUIT EXIT
```

❑❑❑

Using the Intel386 Mapper 4

The MAP386 mapper produces printed information about object modules, including cross-reference maps, and, at your request, purges debug information from the object modules.

MAP386 accepts, as input, object files created by other Intel386 utilities, BND386, BLD386, and LIB386, or by 80286 utilities (except for 80286 loadable modules). Input files can be linkable files, library files containing linkable modules, or loadable files containing loadable or bootloadable modules. You can also specify individual linkable modules within linkable or library files as input to MAP386.

Major Functions of MAP386

MAP386 performs the following major functions for loadable input files:

- Removes selected debug information, that is, information about public, external and local symbol declarations, and lines of source code that are used by software debuggers.
- Generates printed that describes the contents of the input file. This output can include:
  - — Module list
  - — Table map
  - — Segment map
  - — Gate map
  - — Symbol map
  - — Public map
  - — Task map
  - — Cross-Reference map
- Inserts information for the target operating systems.

Input and Output

MAP386 accepts the following as input:

- One loadable file (containing a single loadable or bootloadable module) produced by one of the Intel386 utilities, or one or more linkable and/or library files (containing one or more linkable modules. (See Figure 4-1.) For non-loadable object modules, you may specify Intel386 and 80286 object files in the same invocation.
- One or more MAP386 invocation controls described later in this chapter.
- Operating system information (osinfo) file if the input is loadable.

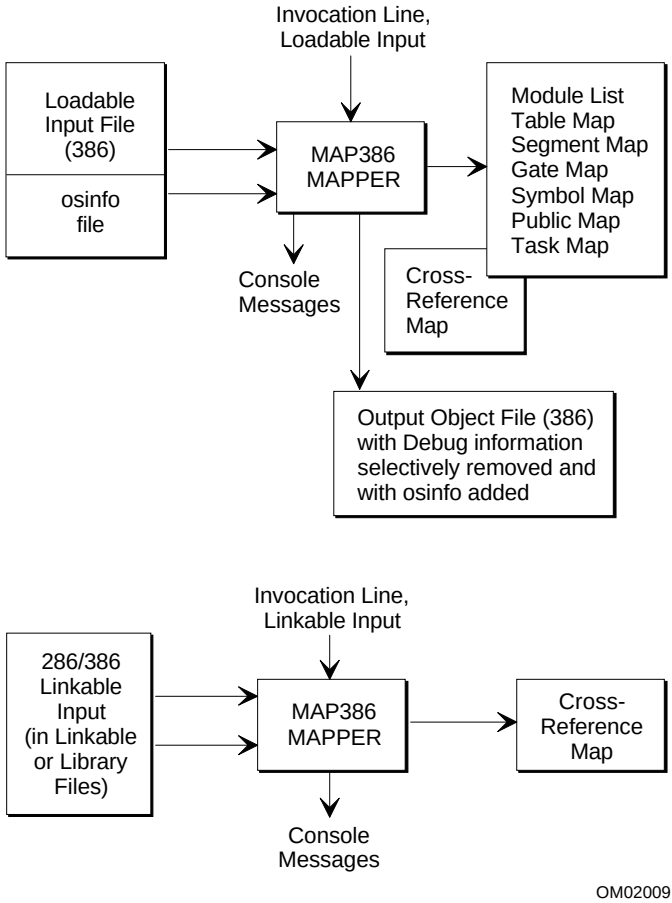

Figure 4-1. MAP386 Input and Output

For loadable and linkable input files, MAP386 outputs the following lists, as shown in Figure 4-1:

Module List

For linkable input, lists modules input to MAP386. For loadable modules, lists linkable modules that make up the expandable file

Cross-Reference

Provides name and type of each symbol in the input file, name of the module containing the public definition, and names of modules containing external declarations for the symbol. The cross-reference map is the only map produced for linkable modules

For loadable files, MAP386 includes the following information in the print output, as shown in Figure 4-1:

Table Map	Descriptor names and corresponding indexes for global descriptor table (GDT), interrupt descriptor table (IDT), and local descriptor tables (LDTs)
Segment Map	Names of segments in the input file and characteristics of each segment, such as descriptor table index, access type, base, descriptor privilege level (DPL), USE16/32 attributes, align attributes, and others
Gate Map	Symbolic gate names and characteristics of each gate in the input file, such as descriptor table name, descriptor table index, gate type, and others
Symbol Map	Names of local symbols in the input file and characteristics of each symbol, such as symbol type, address, and others
Public Map	Public symbols in the input file and their characteristics, such as symbol type, word count, and logical (and if applicable, physical) address
Task Map	Task characteristics for each task such as initial privilege stack, flags, initial values of CS and EIP registers, LDT selector of the task, and others

The module list and each of these maps are described in more detail in MAP386 Print Files later in this chapter.

If the file input to MAP386 contains a loadable module, MAP386 produces some or all of the following output, as specified by the input controls:

- An output object file. Debug information, such as information about symbols and line numbers, can be removed from the output file. Information relating to the operating system can be added or updated.
- A print file containing a module list and one or more of the following maps:
  - Table map
  - Segment map
  - Gate map
  - Symbol map
  - Public map
  - Task map
- A cross-reference map that can be directed to the print file or another specified file.

If the input file is linkable, MAP386 produces only a cross-reference map. The file containing the cross-reference map includes a module list.

MAP386 Module Processing

Executable Modules

Executable files input to MAP386 contain a single loadable module produced by a Intel386 utility. MAP386 accepts only one loadable input file per invocation.

Linkable Modules in Linkable Files

MAP386 processes modules in linkable files as specified in the invocation line. MAP386 can process linkable files produced by any 80286 or Intel386 compiler, assembler, or utility.

Linkable Modules in Library Files

MAP386 processes linkable modules in library files if they are explicitly specified in the invocation line or if they resolve external references made in modules previously processed during the same MAP386 invocation.

The processing of an input file varies depending on whether it is a library or non-library file. A non-library file may contain one object module or several object modules concatenated by BND386 or the BLD386 System Builder. Since several modules cannot reliably be concatenated with a copy command, a library file contains control information in addition to object modules. A module in a non-library file is processed by MAP386 if it is explicitly listed in the module list or if the module list is not specified.

Processing a library file is more complicated. If a module list is specified for the library file, it is processed in the same manner as a non-library file. If a module list is not specified, the library file is processed only if the previously processed modules contain an unresolved external. The library is scanned for modules containing public symbols that match as-yet unresolved externals. Each such module is processed as if it has been explicitly specified. The selection process continues until the modules in the library cannot satisfy any more unresolved externals (including externals encountered while processing modules from the library).

Invoking MAP386

DOS and iRMX Invocation Syntax

To invoke MAP386 on a DOS or iRMX operating system, use the following syntax:

```
MAP386 input_list [controls]
```

Where:

input_list
: is one or more linkable modules or object library modules, specified as follows:

  filename [(*mod_name* | *)]

*
: An asterisk (*) specifies all modules in the file named by *filename*.

controls
: is one or more of the MAP386 specifications described later in this chapter.

When you are working with loadable files, you must specify the input list with the file name only; you cannot specify module names. MAP386 accepts only one input file in this case.

You can continue the invocation line on additional lines by typing the ampersand (&) before the line terminator. This character causes the continuation line to appear with the DOS or iRMX prompt character.

Control Files

The MAP386 invocation line is simplified when you can use the CONTROLFILE control to invoke a control file. A control file is a text file containing any file names or controls that would normally appear in the invocation line. For example, instead of listing five controls in the MAP386 invocation line, you can place those controls in a single control file and then invoke the control file in place of all five controls.

Using a Control File on DOS and iRMX

To include a control file in the map386 invocation line for a DOS or iRMX operating system, use the following syntax:

```
map386 CONTROLFILE (filename[,...])
```

Where:

filename is the name of the control file containing controls, file names, or controls and file names for the input list. You cannot nest control files: that is, the CONTROLFILE control cannot appear in a control file.

A control file that contains only controls can be specified in any position in the input list. A control file that contains only file names for the input list can be specified in any position in the input list.

In a control file that contains both input files and controls, input files must come before controls. In this case, specify the control file as part of the input list.

The following example shows how to specify the CONTROLFILE control in an input list that contains the files named in *cf1.dat*:

```
MAP386 MOD.OBJ, CONTROLFILE (CF1.DAT) DEBUG
```

Within a control file, use a semicolon before a comment. Use the ampersand (&) to continue to the next line. When the line terminator comes before the ampersand, it is treated as if it were a blank space. MAP386 ignores characters between a semicolon or continuation character and the line terminator. Lines in a control file cannot exceed 120 characters in length.

This example control file contains only file names for the input list:

```
util.lib,       &       ; utility library
system.lib              ; system library
```

This example control file contains the last file names for the input list and controls for the control list:

```
util.lib,&      &       ; utility library
system.lib      &       ; system library
lo              &       ; loadable module
ep              &       ; directs error messages to the specified print
                &       ; file specified
oj (lbt.sys)            ; name output file
```

MAP386 Defaults

If the input file is a loadable module, MAP386 processes the single module.

If the input file is linkable and no modules are specified in the input list, MAP386 processes all modules.

If the input file is an object library and no modules are specified in the input list, MAP386 processes only the modules that satisfy external symbol references already encountered in the input. For a reference to be satisfied, a symbol must be declared public in a module in the library.

Output Identifiers

If no output file names are specified in the invocation line (with controls `PRINT`, `OBJECT` and `ERRORPRINT`), MAP386 creates output files as follows:

- MAP386 produces a print file with the same file name as the input file and assigns the extension .MAP.
- If the input file is loadable, and `OBJECT`, `OBJECTCONTROL` or `OSINFO` have been specified, MAP386 produces an output object file with the same file name as the input file and assigns the extension .OUT.
- If `ERRORPRINT` is specified without a file name, all the error messages are printed to the console.

Controls

When no controls are specified in the MAP386 invocation line, MAP386 performs the functions described below:

When the input file is loadable, MAP386 does the following:

- Produces a module list, table map, symbol map, gate map, public map, task map, and cross-reference map. These items and error messages are placed in a file with the same name as the input object file, with extension .MAP.

When the input is a linkable file or modules, MAP386 does the following:

- Produces a cross-reference map and places it along with a module list and error messages in a file with the same name as the input file and with the extension .MAP.

Regardless of whether file input is loadable, bootloadable, or linkable MAP386 does the following by default:

- Displays only fatal error messages on the console.
- Produces output maps 120 columns wide with 60 lines per page. A header on each page includes a title, date, and page number.

Console Messages

MAP386 generates sign-on/sign-off messages and error messages on the console.

MAP386 signs on to the system console with the following message:

```
system_id iRMX III 386(TM) MAPPER, Vx.y
Copyright years, Intel Corporation
```

Where:

system_id identifies the host operating system.

V*x.y* is the MAP386 version number

years is the copyright year or years.

After processing is complete and if no fatal errors have occurred, MAP386 signs off as follows:

```
PROCESSING COMPLETED.n WARNING(S),m ERROR(S)
```

Where:

n and *m* are the number of warnings and errors.

Fatal error messages are always displayed. If MAP386 encounters a fatal error condition, an error message and the following sign-off message appears at the console:

```
PROCESSING ABORTED
```

Although nonfatal error messages are included in the print file by default, they can be directed to the console with the ERRORPRINT control. See the ERRORPRINT entry in this chapter for more information.

MAP386 Controls

Table 4-1 summarizes MAP386 controls for DOS and iRMX operating systems. The default column shows the condition in effect when the control is not specified. When an invocation contains duplicate control specifications, MAP386 processes only the rightmost specification on the invocation line.

Table 4-2 lists abbreviations of MAP386 controls. Each MAP386 control is listed alphabetically and described in detail in the section following Table 4-3.

Table 4-1. MAP386 Controls for DOS and iRMX Operating Systems

Command Syntax	Description	Default
CONTROLFILE (*filename*[,...])	Specifies file for input elements	None
ERRORPRINT (*filename*) NOERRORPRINT	Creates or suppresses creation of error print file	NOERRORPRINT
OBJECT [(*filename*)] NOOBJECT	Creates output object file from loadable or bootloadable input files	If OBJECT CONTROLS or OSINFO specified OBJECT = (*input_filename.*OUT); otherwise, NOOBJECT
OBJECTCONTROLS (*objectcontrol* [EXCEPT] (*mod_name*[,...])][,...]) object controls: DEBUG NODEBUG EXTERNALS NOEXTERNALS LINES NOLINES PUBLICS NOPUBLICS SRCLINES NOSRCLINES SYMBOLS NOSYMBOLS	Includes or removes debug information in output object file	OBJECTCONTROL (DEBUG) (includes debug information)
OSINFO (*filename*)	Creates or updates operating system information section of object file	None
PAGELENGTH (*length*)	Sets lines per page for output listing	PAGELENGTH (60)

continued

Table 4-1. MAP386 Controls for DOS and iRMX Operating Systems (continued)

Command Syntax	Description	Default
PAGEWIDTH (*width*)	Sets characters per line for output listing	PAGEWIDTH (120)
PAGING NOPAGING	Creates page breaks in print files	PAGING
PRINT (*filename*) NOPRINT	Creates or suppresses creation of print file	PRINT(*first_input_filename*.MAP)
PRINTCONTROLS (*printcontrol*) [EXCEPT}(*mod_name* [,...])[,...]) print controls: DEBUG NODEBUG LINES NOLINES PUBLICS NOPUBLICS SRCLINES NOSRCLINES SYMBOLS NOSYMBOLS TABLES NOTABLES TASKS NOTASKS	Includes or omits selected maps from print file	PRINTCONTRTOLS (DEBUG, TABLES, TASKS)
SYMBOLSORT NOSYMBOLSORT	Prints symbol names in alphabetical order (SYMBOLSORT) or in order of occurrence in (NOSYMBOLSORT)	SYMBOLSORT
TITLE (*title*)	Places header line at top of each print file page	No title
TYPE NOTYPE	Ignores types	TYPE
TYPECHECK NOTYPECHECK	Enables or suppresses type checking	TYPECHECK
XREF [(*filename*)] NOXREF	Directs intermodule cross-reference map between public and external symbols to specified file	XREF(*print_file filename*)

Table 4-2. Standard Abbreviations for MAP386 Controls

Commands	Abbr.	Commands	Abbr.
CONTROLFILE	CF	OBJECTCONTROLS	OC
ERRORPRINT	EP	OSINFO	OI
NOERRORPRINT	NOEP	PAGELENGTH	PL
NOOBJECT	NOOJ	PAGING	PG
NOPAGING	NOPG	PRINT	PR
NOPRINT	NOPR	PRINTCONTROLS	PR
NOSYMBOLSORT	NOSS	SYMBOLSORT	SS
NOTYPE	NOTY	TITLE	TT
NOTYPCHECK	NOTC	TYPE	TY
NOXREF	NOXREF	TYPECHECK	TC
OBJECT	OJ	XREF	XREF

CONTROLFILE

Specifies file for input elements

Syntax

```
CONTROLFILE (filename[,...])
```

Abbreviation

```
CF
```

Default

`CONTROLFILE` is not in effect.

Description

The `CONTROLFILE` control directs MAP386 to the specified file for controls or elements of the input list. A partial control or input list element is not allowed. Nested control files are not allowed. MAP386 returns to the command line when it encounters the end of a control file.

Refer to Control Files in this chapter for the content and format of control files.

Invocation Examples

In all of the following examples, four control files are invoked in three different combinations. Control file CF1.DAT contains files and controls, as follows:

```
FILE1.OBJ, FILE2.OBJ, FILE3.OBJ NOSS EP
```

Control file CF2.DAT contains files only:

```
FILE4.OBJ, FILE5.OBJ, FILE6.OBJ
```

Control file CF3.DAT contains controls only:

```
NOSS PR (A1.MAP) TT ("THIS IS A TITLE")
```

Control file CF4.DAT contains files only, but with a comma after the last file name:

```
FILE7.OBJ, FILE8.OBJ, FILE9.OBJ,
```

Examples

1. In the following example, the PAGEWIDTH and NOXREF controls and the A.OBJ input file are specified on the command line with CF4.DAT and CF2.DAT.

   ```
   MAP386 A.OBJ, CF (CF4.DAT) CF (CF2.DAT) PAGEWIDTH (90)
   NOXREF
   ```

2. In the following example, the PAGEWIDTH and NOXREF controls and the B.OBJ input file are specified on the command line with CF4.DAT and CF1.DAT.

   ```
   MAP386 B.OBJ, CF (CF4.DAT) CF (CF1.DAT) PAGEWIDTH (90)
   NOXREF
   ```

3. In the following example, all input files and controls except C.OBJ are specified in the control files CF2.DAT and CF3.DAT.

   ```
   MAP386 C.OBJ, CF (CF2.DAT,CF3.DAT)
   ```

ERRORPRINT/NOERRORPRINT

Creates or suppresses creation of error print file

Syntax

```
ERRORPRINT [(filename)]
NOERRORPRINT
```

Abbreviations

```
EP, NOEP
```

Default

```
NOERRORPRINT
```

Description

The `ERRORPRINT` control directs all error messages, including warnings, errors, and fatal errors, to one of the following:

- The console, if no file name is specified
- The error print file specified by *`filename`*

The `NOERRORPRINT` control prevents nonfatal errors and warnings from being sent to the error print file.

When the file name is the same as the name of an input file, control file, or output file, processing aborts.

Whether `ERRORPRINT` is in effect or not, fatal error messages are displayed at the console and all error and warning messages are included in the print file. The number of error and warning messages is reported in the sign-off message.

Examples

1. In the following example, MAP386 sends all error messages and warnings to the error print file MOD2.LIS. MOD1.OBJ is the input file.

   ```
   MAP386 MOD1.OBJ ERRORPRINT (MOD2.LIS)
   ```

2. In the following example, MAP386 sends all error messages and warnings to the console by default.

   ```
   MAP386 MOD1.OBJ ERRORPRINT
   ```

OBJECT/NOOBJECT

Creates output object file from loadable or bootloadable input files

Syntax

```
OBJECT [(filename)]
NOOBJECT
```

Abbreviations

```
OJ, NOOJ
```

Default

When OBJECTCONTROLS or OSINFO is specified: OBJECT (*input_filename*.OUT). When neither OBJECTCONTROLS nor OSINFO is specified: NOOBJECT

Description

The OBJECT control produces an output object file from a loadable input file. MAP386 copies all sections of the input file to the output file, except when OBJECTCONTROLS directs MAP386 to remove debug information. Refer to the description of OBJECTCONTROLS for more information.

When only the output object file is required, specify NOPRINT; this suppresses the print file and saves execution time and space. Otherwise, a print file is created as well.

The NOOBJECT control suppresses creation of a loadable output module.

Examples

1. In the following example, MAP386 generates an output object file named MOD2.DAT. By default, MAP386 also creates a print file. MOD1.OBJ is the input file name.

   ```
   MAP386 MOD1.OBJ OBJECT (MOD2.DAT)
   ```

2. In the following example, MOD3.OBJ is the input file name. Because no output file name is specified, MAP386 generates an output object file named MOD3.OUT.

   ```
   MAP386 MOD3.OBJ OBJECT
   ```

OBJECTCONTROL

Includes or removes debug information in output object file

Syntax

```
OBJECTCONTROLS (objctrl [[EXCEPT]
(mod_name[,...])][,...])
```

Where:

objctrl Is one of the following:

Object Control	Abbreviation
DEBUG	DB
NODEBUG	NODB
EXTERNALS	ET
NOEXTERNALS	NOET
LINES	LI
NOLINES	NOLI
PUBLICS	PL
NOPUBLICS	NOPL
SRCLINES	SL
NOSRCLINES	NOSL
SYMBOLS	SB
NOSYMBOLS	NOSB

mod_name is the name of a separately translated module that has been input to BND386 or to the BLD386 System Builder to create the loadable file named in the invocation line.

Abbreviations

```
OC [EC]
```

Default

`OBJECTCONTROLS` (`DB`). All debug information is retained.

Description

The `OBJECTCONTROLS` control removes specified debug information from a loadable input file. MAP386 creates an output object file that is just like the input file, except that the specified debug information is omitted. `OBJECTCONTROLS` has no tables or tasks controls.

The DEBUG, EXTERNALS, LINES, PUBLICS, SRCLINES, and SYMBOLS object controls direct MAP386 to retain various kinds of debug information; the NODEBUG, NOEXTERNALS, NOLINES, NOPUBLICS, NOSRCLINES, and NOSYMBOLS object controls direct MAP386 to remove certain information. Object controls may be specified in any order. When invoking MAP386 to purge debug information, also specify NOPRINT to save time and memory.

DEBUG includes all debug information in the output object file, including external symbol definitions, line number definitions, public symbol definitions, source line numbers, and symbol definitions. NODEBUG removes this information from the output object file.

The construction EXCEPT *mod_name* [,...] excludes the listed modules from the effects of the preceding object control.

EXTERNALS includes external symbol definitions in the output object file. NOEXTERNALS removes external symbol definitions from the output object file.

LINES includes line number definitions in the output object file. NOLINES removes line number definitions from the output object file.

PUBLICS includes public symbol definitions in the output object file. NOPUBLICS removes public symbol definitions from the output object file.

SRCLINES includes source line numbers in the output object module. NOSRCLINES omits source line numbers from the output object module.

SYMBOLS includes local symbol definitions in the output object file. NOSYMBOLS removes local symbol definitions from the output object file.

You can select which debug information is to be retained or removed by specifying the module name or list of module names plus the desired object control. Only the information previously contained in the module is affected.

You can exclude selected modules from the effect of the object control. Specify EXCEPT plus the module name or list of module names.

If you specify an object control without any module names, the control affects all the debug information in the file.

OBJECTCONTROLS has no effect if the modules input to MAP386 are linkable modules.

Because an output object file is produced with OBJECTCONTROLS, it is not necessary to also specify the OBJECT control, except to direct the output to *filename*.

Note

OBJECTCONTROLS is not effective when NOOBJECT is used, and no object file is produced.

Examples

1. In the following example, MAP386 removes all debug information contained in the input file. The output file is MOD1.OUT.

   ```
   MAP386 MOD1.OBJ OBJECTCONTROLS (NODEBUG)
   ```

2. In the following example, MAP386 removes line numbers, local symbol definitions originally contained in MOD3, and public symbols definitions originally contained in all modules except MOD2 and MOD3.

   ```
   MAP386 MOD2.OBJ OC (NOLI, NOSB(MOD3), NOPL EC(MOD2,MOD3))
   ```

OSINFO

Creates or updates operating system information section of object file

Syntax

```
OSINFO (filename)
```

Abbreviation

```
OI
```

Default

The operating system information field in the object file is not created or updated.

Description

The `OSINFO` control creates or updates the operating system information field in the object file by copying the contents of the file name (including its path) into the `OSINFO` field in the object file. For a large file, only the first 4K bytes are written to `OSINFO` and a warning is issued. See documentation on your target operating system for more details.

Note

`OSINFO` is not effective when used with `NOOBJECT`.

Examples

In the following example, operating system information for the file MOD1.OBJ is copied into the osinfo field in the object file. MOD2.OBJ is the input file name.

```
MAP386 MOD2.OBJ OSINFO (MOD1.OBJ)
```

PAGELENGTH

Sets lines per page for output listing

Syntax

```
PAGELENGTH (length)
```

Abbreviation

```
PL
```

Default

A page is 60 lines long.

Description

The PAGELENGTH control specifies the number of lines per page in MAP386 output listings (e.g., print and cross-reference files). Headings are included in the lines per page specification. Page length may be set to from 10 to 65535 lines.

Note

PAGELENGTH is not effective when used with NOPAGING.

Examples

1. In the following example, the page length in output listings is set to 120 lines per page.

   ```
   MAP386 MOD1.OBJ PAGELENGTH (120)
   ```

2. In the following example, the page length in output listings is set to 60 lines per page by default.

   ```
   MAP386 MOD1.OBJ
   ```

PAGEWIDTH

Sets characters per line for output listing

Syntax

```
PAGEWIDTH (width)
```

Abbreviation

```
PW
```

Default

A page is 120 characters wide.

Description

The PAGEWIDTH control specifies the maximum number of characters per line in MAP386 output listings (e.g., print and cross-reference file). Page width may be set to from 80 to 132 characters per line. (The system will accept widths from 72 to 80 as well, but these settings are not recommended, due to the width of the map.) If the number of characters in a line exceeds the physical width of the page, the extra characters wrap to the following line.

Note

PAGEWIDTH is not effective when used with NOPAGING.

Examples

1. In the following example, the line width in output listings is set to 100 characters.

   ```
   MAP386 MOD1.OBJ PAGEWIDTH (100)
   ```

2. In the following example, the line width in output listings is set to 120 characters by default.

   ```
   MAP386 MOD1.OBJ
   ```

PAGING/NOPAGING

Creates page breaks in print file

Syntax

```
PAGING
NOPAGING
```

Abbreviations

```
PG, NOPG
```

Default

```
PAGING
```

Description

The PAGING control prints the output map with page breaks. Each output page contains the number of lines specified by the PAGELENGTH control (with default at 60 lines). Each new output page has a heading containing the label 386(TM) MAPPER, a title (specified with the TITLE control), the date, and a page number.

NOPAGING prints the output file printed continuously and the heading appears on the first page only.

With NOPAGING, PAGELENGTH and PAGEWIDTH are not in effect, and page length and width are unspecified.

Examples

1. In the following example, MAP386 produces a print file with pages that are 55 lines long. A heading appears at the top of each page.

   ```
   MAP386 MOD1.OBJ PAGELENGTH (55)
   ```

2. In the following example, MAP386 produces a print file that is not separated by page breaks. MOD2.OBJ and MOD3.OBJ are input files.

   ```
   MAP386 MOD2.OBJ, MOD3.OBJ NOPAGING XREF (MOD1.LIS)
   ```

PRINT/NOPRINT

Creates or suppresses creation of print file

Syntax

```
PRINT [(filename)]
NOPRINT
```

Abbreviations

```
PR, NOPR
```

Default

```
PRINT (first_input_filename.MAP)
```

Description

The PRINT control creates a print file, which contains the Module List and output maps, including the Table Map, Segment Map, Gate Map, Symbol Map, Public Map, Task Map, and error messages. The Cross-Reference Map is included in the print file if directed there with the XREF control. You can also use XREF to create a separate cross-reference file for the cross-reference map. If XREF and the file name for the cross-reference file are specified, the cross-reference map is placed in that file rather than the print file.

The print file begins with a heading containing the label 386(TM) MAPPER, a title (specified with the TITLE control), the date, and page number. The file also includes error and warning messages.

When *filename* is specified, MAP386 assigns that name to the print file. Otherwise, by default, if the PRINT control is not specified or if PRINT is specified without a file name, the name of the print file is the same as *input_filename*, with extension .MAP. The contents and format of the print file are described later in this chapter.

The NOPRINT control suppresses creation of a print file. NOPRINT has no effect on the cross-reference map when the print file and cross-reference file are different. You can use NOPRINT to save time when purging debug information or adding the osinfo section.

Use ERRORPRINT to create a separate error print file for error messages. Should a fatal error condition occur, the fatal error message is displayed on screen.

Note

If the default file name or a specified file name matches the name of the input file or any other output file, MAP386 processing aborts.

Examples

1. In the following example, the cross-reference map, error messages, and print file information (e.g., input and output identifiers, control list and modules processed) are directed to the print file MOD2.LIS. MOD1.OBJ is the input file.

   ```
   MAP386 MOD1.OBJ PRINT (MOD2.LIS)
   ```

2. In the following example, MAP386 print file information and cross-reference map are sent to the default print file. MOD1.MAP is the default print file.

   ```
   MAP386 MOD1.OBJ
   ```

3. In the following examples, the file is written to the standard output device.

   ```
   MAP386 MOD.OBJ PRINT (:CO:)
   ```

PRINTCONTROLS

Includes or omits selected maps from print file

Syntax

```
PRINTCONTROLS (prtctrl [[EXCEPT] (mod_name [,...])][,...])
```

Where:

prtctrl is one of the following print controls:

Print Control	Abbreviation
DEBUG	DB
NODEBUG	NODB
LINES	LI
NOLINES	NOLI
PUBLICS	PL
NOPUBLICS	NOPL
SRCLINES	SL
NOSRCLINES	NOSL
SYMBOLS	SB
NOSYMBOLS	NOSB
TABLES	TB
NOTABLES	NOTB
TASKS	TA
NOTASKS	NOTA

mod_name is the name of a separately translated module that has been input to BND386 or to the BLD386 System Builder to create the loadable file named in the invocation line.

Abbreviation

```
PC
```

Default

```
PRINTCONTROLS (DEBUG, TABLES, TASKS)
```

Description

The PRINTCONTROLS control modifies the contents of the print file. (It does not affect the contents of the cross-reference map.) Specified print controls include or omit specified maps from the print file. The print controls DEBUG, LINES, PUBLICS, SRCLINES, SYMBOLS, TABLES, and TASKS cause certain maps to be included. Specify a module name or list of module names after a print control to print maps describing only the information originally contained in that module or modules. If you do not specify a module name, the print controls affect the contents of the entire file, regardless of the linkable module from which the information originated. The controls NODEBUG, NOLINES, NOPUBLICS, NOSRCLINES, NOSYMBOLS, NOTABLES, and NOTASKS cause certain maps to be omitted. Print controls can be specified in any order.

You can exclude a module or modules from the effect of a print control with EXCEPT, by specifying a print control followed by the EXCEPT control followed by a module name or list of module names. The information previously contained in the module or modules is excluded from the effect of the print control. For example, PRINTCONTROLS can be specified in the invocation line as follows:

```
PRINTCONTROLS (NOPAGELENGTH EXCEPT(MOD1, MOD3))
```

In this case, the public map would include public symbol information for symbols originally declared in modules MOD1 and MOD3.

You cannot limit TASKS, NOTASKS, TABLES, and NOTABLES by specifying a module name; all modules will be affected by the specified print control.

DEBUG includes a symbol map and public map in the print file. NODEBUG omits the symbol map and public map from the print file.

LINES includes a section in the symbol map that contains line number definitions and their logical addresses. NOLINES omits this section.

PUBLICS includes a public map in the print file. NOPUBLICS omits the public map.

SRCLINES includes a section in the print file that contains source line definitions. NOSRCLINES omits this section.

SYMBOLS includes a section in the symbol map that lists symbols and their types and addresses. NOSYMBOLS omits this section.

TABLES includes a table map, gate map, and segment map in the print file. NOTABLES omits these maps. You cannot specify *mod_names* for TABLES or for NOTABLES.

TASKS includes a task map in the print file. NOTASKS omits the task map. You cannot specify module names for TASKS or NOTASKS.

Notes

PRINTCONTROLS is not effective when used with NOPRINT.

SYMBOLS, NOSYMBOLS, LINES and NOLINES are not effective when used with NODEBUG.

Any input file specified on the MAP386 invocation line must be loadable.

Examples

In the following examples, MOD1.OBJ is a loadable input file. A print file is created and contains the following:

- Module list
- Public map for all original input modules except MOD5 and MOD6
- Symbol map for all original input modules except MOD5 and MOD6, which will, however, include lines. The symbol map describes local symbols for all original modules, and contains source line numbers and addresses for all original modules except MOD3.
- Table map, gate map, and segment map for all modules. The task map is not be printed.

```
MAP386 MOD1.OBJ PC (DB EC(MOD5,MOD6), LI EC(MOD3), NOTA)
```

SYMBOLSORT/NOSYMBOLSORT

Prints symbol names in alphabetical order or in order of occurrence

Syntax

```
SYMBOLSORT
NOSYMBOLSORT
```

Abbreviations

```
SS, NOSS
```

Default

```
SYMBOLSORT
```

Description

The `SYMBOLSORT` control prints lists of symbol names in alphabetical order in output maps.

`NOSYMBOLSORT` prints symbol names in the order in which they occur in the input object file. This preserves scoping information.

Examples

In the following example, symbol names appear in the symbol map and cross-reference map in alphabetical order.

```
MAP386 MOD1.OBJ SYMBOLSORT
```

TITLE

Places header line at top of each print file page

Syntax

```
TITLE (title)
```

Abbreviation

```
TT
```

Default

`TITLE` is not in effect and the title string is left blank.

Description

The `TITLE` control specifies the page title of the page heading. The title must be an alphanumeric string of 80 characters or less. When you use spaces or other delimiters in the title, you must enclose the whole title in apostrophes (').

The title is truncated on the right when the specified page width does not allow enough room for the complete title.

If the cross-reference map and the print file are separate files, the same title appears in the headings of both files.

Examples

1. In the following examples, when the file is printed, the title that appears at the top of each page is CROSS-REF MAP FOR PL/M PROG XYZ.

   ```
   MAP386 MOD1.OBJ TITLE ('CROSS-REF MAP FOR PL/M PROG XYZ')
   ```

2. In the following examples, when the file is printed, the title CROSS-REF MAP appears at the top of each page. A cross-reference map is generated and sent to the cross-reference file MOD2.LIS. MOD1.OBJ is the input file and MOD1.MAP is the print file.

   ```
   MAP386 MOD1.OBJ TITLE ('CROSS_REF MAP') XREF (MOD2.LIS)
   ```

TYPE/NOTYPE

Ignores types

Syntax

```
TYPE
NOTYPE
```

Abbreviations

```
TY, NOTY
```

Default

```
TYPE
```

Description

The TYPE control specifies that type information be printed. NOTYPE suppresses the checking and printing of type information.

Note

TYPECHECK is not effective when used with NOTYPE.

Examples

In the following example, NOTYPE is specified so that type information is neither checked nor printed.

```
MAP386 MOD.OBJ NOTYPE
```

TYPECHECK/NOTYPECHECK

Enables or suppresses type checking

Syntax

```
TYPECHECK
NOTYPECHECK
```

Abbreviations

```
TC, NOTC
```

Default

```
TYPECHECK
```

Description

The TYPECHECK control performs type checking between public and external symbols of the same name in the input file. If a mismatch is found, MAP386 issues a warning message. Type mismatches are detected even if the BND386 or BLD386 NOTYPE control has been used to purge type information from the input file. This type checking is less comprehensive than the type checking performed by BND386.

NOTYPECHECK suppresses type checking, and the error file will not notify you of mismatches between public and external symbol types

TYPECHECK and NOTYPECHECK do not affect the information about public and external symbols in output maps. For example, the cross-reference map lists symbol types even when NOTYPECHECK is specified.

Examples

In the following example, type checking is suppressed; MAP386 does not issue warnings about type mismatches. MOD2.LIS is the print file; MOD1.OBJ is the input file.

```
MAP386 MOD1.OBJ NOTYPECHECK PRINT (MOD2.LIS)
```

XREF/NOXREF

Directs intermodule cross-reference map between public and external symbols to a specified file

Syntax

```
XREF [(filename)]
NOXREF
```

Abbreviations

```
XR, NOXR
```

Default

XREF and the cross-reference map file have the same name as the print file.

Description

The XREF control generates an intermodule cross-reference map between public and external symbols. When a file name is specified, the map is sent to that file. Otherwise, the map is sent to the print file.

NOXREF suppresses the generation of cross-reference maps.

Note

The cross-reference map is not printed when XREF is used with NOPRINT.

Examples

1. In the following examples, the cross-reference map is generated and sent to the XREF file MOD2.LIS. Other maps and print file information are sent to the default print file, MOD1.MAP.

   ```
   MAP386 MOD1.OBJ XREF (MOD2.LIS)
   ```

2. In the following examples, the generation of cross-reference maps is suppressed and the remaining print file information is directed to MOD2.MAP. MOD1.OBJ is the input file.

   ```
   MAP386 MOD1.OBJ NOXREF PRINT (MOD2.MAP)
   ```

MAP386 Print Files

MAP386 produces maps each time it is invoked, unless the NOPRINT control is specified. MAP386 places the cross-reference map in the file specified by the XREF control or, by default, in the file implied by the PRINT control.

When the input file is linkable, MAP386 produces only one print file containing a module list and the cross-reference map. When the input file is loadable, MAP386 can create any of the following maps: table map, segment map, gate map, task map, symbol map, public map, and cross-reference map.

All maps except the cross-reference map appear in the print file, whose file name is specified with the PRINT control. When PRINT does not provide a file name, MAP386 by default gives the print file the same file name as the input file, with extension .MAP. The cross-reference map can be included in the print file, but it can also be directed to another file with the XREF control.

The print file can contain the following sections, in this order:

Header
Module list
Table map
Segment map
Gate map
Symbol map
Public map
Task map
Cross-reference
Error messages

The cross-reference map can be directed to another file or excluded altogether with NOXREF. Error messages can be directed to another file with ERRORPRINT and also appear in the print file. Selected maps (table, segment, gate, symbol, public, or task) can be excluded from the print file by the use of print controls.

The following sections describe and give format examples of each part of the print file. In the examples, Xs indicate the space allotted to an entry; they are replaced by names or numbers in actual print files. Minuses are printed in the field if the absolute value has not yet been defined.

Header

The print file header summarizes the MAP386 invocation specifications by listing input and output file names and controls specified. Figure 4-2 shows the format of the print file header.

```
386(TM) MAPPER              title                   date    PAGE number

system_id      iRMX III 386(TM) MAPPER, Vx.yVX

INPUT FILE:  filename [, . . .]
INPUT OSINFO FILES:  filename
OUTPUT OBJECT FILE:  filename
OUTPUT PRINT FILE:  filename
OUTPUT XREF FILE:  filename
CONTROLS SPECIFIED:  control1, control2 ...

<------------(warnings, if any, appear here)
```

Figure 4-2. MAP386 Print File Header

Module List

The module list names the input files and the modules contained in each file. Files are listed in the order in which they appear in the input list. MAP386 prints a question mark (?) to the left of the names of modules containing no debug information. Figure 4-3 shows the format of the module list.

```
MODULES INCLUDED:
     FILE NAME           MODULE NAME(S)

     xxxxxxxxx           xxxxxxxxxxxxxx          xxxxxxxxxxxxxx
                         xxxxxxxxxxxxxx          xxxxxxxxxxxxxx
     xxxxxxxxx           xxxxxxxxxxxxxx          xxxxxxxxxxxxxx
                         xxxxxxxxxxxxxx          xxxxxxxxxxxxxx
        .                      .                       .
        .                      .                       .
```

Figure 4-3. MAP386 Module List

Table Map

The table map contains information about GDTs, IDTs, and LDTs. For each descriptor table in the input, the table map lists table indexes, selector values (except IDT) for table descriptors, and descriptor names. Table indexes appear in decimal notation in ascending numerical order. Selector values appear in hexadecimal notation.

The table map lists information about the GDT first, information about the IDT second, and information about the LDT third. LDT descriptions include a decimal value labeled SEQ. NO.; this value is the sequence number of the LDT in the GDT. MAP386 lists only indexes that have valid descriptor entries. Figure 4-4 shows the format of the table map.

```
DESCRIPTOR TABLE MAP

TABLE = GDT       BASE = xxxxxxxxH        LIMIT = xxxxH

TABLE INDEX            SELECTOR           DESCRIPTOR NAME

    xxxx                xxxxH             xxxxxxxxxxxx
     .                    .                    .
     .                    .                    .
     .                    .                    .
    xxxx                xxxxH             xxxxxxxxxxxx

TABLE = IDT       BASE = xxxxxxxxH        LIMIT = xxxxH

TABLE INDEX       DESCRIPTOR NAME

    xxxx          xxxxxxxxxxxx
     .                  .
     .                  .
     .                  .
    xxxx          xxxxxxxxxxxx

TABLE = xxxxxxxxx    SEQ. NO. = n     BASE xxxxxxxxH   LIMIT = xxxxH

TABLE INDEX             SELECTOR            DESCRIPTOR NAME

    xxxx                  xxxxH             xxxxxxxxxxxx
     .                      .                    .
     .                      .                    .
     .                      .                    .
    xxxx                  xxxxH             xxxxxxxxxxxx
```

Figure 4-4. MAP386 Table Map

Segment Map

The segment map lists each input segment alphabetically and according to descriptor table. For each segment, the map includes the segment name, its index (slot number) in the descriptor table, present bit (PBIT), descriptor privilege level (DPL), USE16/32 attribute, align attribute, access type, base or relocation information (an eight-digit hexadecimal number), limit of the page-fixed part (an eight-digit The segment map lists each input segment alphabetically and according to hexadecimal number for USE32 or a four-digit hexadecimal number for USE16), limit (an eight-digit hexadecimal number for USE32 or four-digit hexadecimal number for USE16). If the map does not fit on one line, the `SEGMENT` name will be printed on a separate line. Access types (listed under the heading ACCESS) are denoted as follows:

- EO Executable only
- ER Executable and readable
- C Conforming
- RO Read only
- RW Readable and writable
- D Expand-down

For example, ERC indicates that a segment is loadable, readable, and conforming. RWD indicates that a segment is readable, writable, and expand-down. Figure 4-5 shows the format of the segment map.

```
SEGMENT MAP

TABLE : xxxxxxxx

SEGMENT NAME  TABLE INDEX  PBIT  USE    DPL  ALIGN  ACCESS     **     LIMIT

xxxxxxxx        xxxx        x    xxxxx  x    xxxx   xxx     xxxxxxxxH xxxxxxxH
   .            .          .      .    .      .    .          .        .
   .            .          .      .    .      .    .          .        .
   .            .          .      .    .      .    .          .        .
xxxxxxxx      xxxx       x    xxxxx  x    xxxx   xxx    xxxxxxxxH xxxxxxxH

TABLE : xxxxxxxx

SEGMENT NAME  TABLE INDEX  PBIT  USE    DPL  ALIGN  ACCESS     **     LIMIT

xxxxxxxx        xxxx        x    xxxxx  x    xxxx   xxx     xxxxxxxxH xxxxxxxH
   .            .          .      .    .      .    .          .        .
   .            .          .      .    .      .    .          .        .
   .            .          .      .    .      .    .          .        .
xxxxxxxx      xxxx       x    xxxxx  x    xxxx   xxx    xxxxxxxxH xxxxxxxH
```

Figure 4-5. MAP386 Segment Map

Gate Map

The Gate Map lists each gate alphabetically and according to descriptor table. For each gate, the map contains the gate's name, descriptor table name, index (slot number) in the descriptor table, present bit (PBIT), descriptor privilege level (DPL), type of gate, word count (WC), selector of the gate entry point, and offset of the gate entry point. Gate types are listed as 286CALL, 386CALL, 286INTR (for interrupt), 386INTR, 286TRAP, 386TRAP, or TASK. Figure 4-6 shows the format of the gate map.

```
GATE MAP

TABLE: xxxxxxxxx

GATE NAME       TABLE INDEX   PBIT   DPL   TYPE     WC   SELECTOR    OFFSET

xxxxxxxxxxxxx     xxxx         x      x    xxxxxxx   xx   GDT(xxxx)   xxxxxxxxH
xxxxxxxxxxxxx     xxxx         x      x    xxxxxxx   xx   GDT(xxxx)   xxxxxxxxH
xxxxxxxxxxxxx     xxxx         x      x    xxxxxxx   xx   LDT(xxxx)   xxxxxxxxH
       .            .          .      .       .      .       .          .
       .            .          .      .       .      .       .          .
xxxxxxxxxxxxx     xxxx         x      x    xxxxxxx   xx   GDT(xxxx)   xxxxxxxxH

TABLE: xxxxxxxxx

GATE NAME       TABLE INDEX   PBIT   DPL   TYPE     WC   SELECTOR    OFFSET
xxxxxxxxxxxxx     xxxx         x      x    xxxxxxx   xx   GDT(xxxx)   xxxxxxxxH
xxxxxxxxxxxxx     xxxx         x      x    xxxxxxx   xx   GDT(xxxx)   xxxxxxxxH
xxxxxxxxxxxxx     xxxx         x      x    xxxxxxx   xx   LDT(xxxx)   xxxxxxxxH
       .            .          .      .       .      .       .          .
       .            .          .      .       .      .       .          .
xxxxxxxxxxxxx     xxxx         x      x    xxxxxxx   xx   GDT(xxxx)   xxxxxxxxH
```

Figure 4-6. MAP386 Gate Map

Symbol Map

The symbol map, which MAP386 produces on a per-module basis, consists of three sections: the first section describes local symbols; the second section describes line numbers and their offsets in the object code; the third section describes line numbers and their offsets in the source code.

The first section of the symbol map lists the name of each local symbol, its logical address (given by values in the columns labeled BASE and OFFSET), and its type (such as word, byte, selector, pointer, procedure, etc.). If no type has been assigned to the symbol, the word NULL appears in the column labeled TYPE. The symbol map includes absolute addresses if they have been assigned to listed symbols by BLD386. Symbols are listed by name (under the SYMBOL NAME column) in the order in which they occur in the input module.

If a module contains a GDT, the first section of the symbol map indicates the GDT slot (index) that points to the module's LDT.

The symbol map indicates the logical address of each symbol, in the columns labeled BASE and OFFSET. The BASE column lists the symbol base, which represents the selector of the symbol, in terms of the symbol's descriptor table and the table slot (index) within that descriptor table. The OFFSET column lists the offset in the segment indicated by the selector. Together the offset and selector form the logical address of the symbol.

If the offset of the symbol is relative to the current stack, SS:EBP is printed in the BASE column.

The symbol map indicates when a symbol is a based symbol, that is, a symbol whose logical address is determined by a value residing at the address of another symbol. For example, PL/M based variables are based symbols.

Depending on the declarations made in the source program, the contents of the symbol's base can be any of the following:

- The value of the selector portion of the symbol's address. The symbol is referred to as a selector-based symbol.
- The value of the offset portion of the symbol's address. The symbol is referred to as an offset-based symbol.
- The value of both the selector and the offset of the symbol's address. The symbol is referred to as a pointer-based symbol.

The symbol map indicates that a symbol is based by displaying one of the following characters to the right of the hexadecimal value in the OFFSET column:

P Indicates a pointer-based symbol; the symbol's full logical address (composed of a selector and offset) is the value found at run time at the location given by the BASE and the OFFSET listings.

O Indicates an offset-based symbol; the base of the symbol's logical address is the selector contained at location listed in the BASE column; the offset of the symbol's logical address is given by the value found at run time at the location given by BASE and OFFSET listings.

S Indicates a selector-based symbol; the value found at run time at the location given by the BASE and OFFSET listings contains the selector of the symbol. The symbol's implicit offset is zero.

If a module has symbols from more than one LDT, the absolute addresses for the symbols are unknown and are marked with a question mark (?). The absolute address of symbols that refer to the IDT or constant symbols are also unknown.

The absolute address of a symbol that refers to an unknown entry in the LDT or GDT is marked `** ERROR **`. An undefined absolute address from a loadable file created by the BLD386 System Builder that has no base value is marked "--------". The absolute address of a gate symbol (the gate selector entry in the GDT or IDT) is the address of the gate entry.

The second section of the symbol map lists lines of loadable source code and their offsets within code segments. The column labeled LINE lists line numbers (in multiples of five) corresponding to the line or statement numbers of the original source program. Under the heading OFFSET, MAP386 prints the offset portions of the logical addresses of the of loadable code that corresponds to each original source line (or source statement, depending on the compiler or assembler). Note that for highly optimizing compilers this information can be misleading. The logical address of each line is the code segment (given at the top of this section of the symbol map as a descriptor table name and index) plus the offset. Line numbers to which the offsets correspond ascend from left to right, beginning with the line listed in the LINE column.

The third section of the symbol map lists source lines and their offset to the source file. The column labeled LINE gives line numbers in multiples of five that correspond to the source line. The column labeled OFFSET prints the offset portions of the logical addresses containing each source line. Line numbers to which the offsets correspond ascend from left to right, beginning with the line listed in the LINE column. Figure 4-7 shows the format of the symbol map.

```
SYMBOL MAP

MODULE = xxxxxxxxx

SYMBOL NAME    BASE          OFFSET        TYPE     ABSOLUTE ADDRESS

xxxxxxxxxx     LDT(xxxx)     xxxxxxxxH     xxxxx        xxxxxxxxH
xxxxxxxxxx     CUR. STACK    xxxxxxxxH     xxxxx        xxxxxxxxH
xxxxxxxxxx     GDT(xxxx)     xxxxxxxxH     xxxxx        xxxxxxxxH
    .            .               .           .              .
    .            .               .           .              .
MODULE = xxxxxxxxxx     CODE SEGMENT = "table(index)"

LINE   OFFSET IN CODE SEGMENT

xxxx    xxxxxxxxH    xxxxxxxxH    xxxxxxxxH    xxxxxxxxH   xxxxxxxxH
xxxx    xxxxxxxxH    xxxxxxxxH    xxxxxxxxH    xxxxxxxxH   xxxxxxxxH
xxxx                 xxxxxxxxH    xxxxxxxxH    xxxxxxxxH
xxxx                 xxxxxxxxH    xxxxxxxxH    xxxxxxxxH   xxxxxxxxH
xxxx    xxxxxxxxH    xxxxxxxxH    xxxxxxxxH
 .                                    .
 .                                    .

MODULE = xxxxxxxxx

SOURCE PATHNAME = xxxxxxxxxxxxxxx

LINE   OFFSET IN SOURCE FILE

xxxx    xxxxxxxxH    xxxxxxxxH    xxxxxxxxH    xxxxxxxxH   xxxxxxxxH
xxxx    xxxxxxxxH    xxxxxxxxH    xxxxxxxxH    xxxxxxxxH   xxxxxxxxH
xxxx                 xxxxxxxxH    xxxxxxxxH    xxxxxxxxH
xxxx                 xxxxxxxxH    xxxxxxxxH    xxxxxxxxH   xxxxxxxxH
xxxx    xxxxxxxxH    xxxxxxxxH    xxxxxxxxH
 .                                    .
 .                                    .
```

Figure 4-7. MAP386 Symbol Map

Public Map

The public map alphabetically lists the name of each public symbol in input modules. For each public symbol, the map includes its symbol name, symbol type, word count (if applicable), and the logical address, including selector and offset. The absolute address, if any, is displayed with the logical address. NULL in the type column indicates that the symbol type was not declared.

Figure 4-8 shows the format of the public map.

```
PUBLIC MAP

MODULE = xxxxxxxxx
PUBLIC NAME     BASE          OFFSET      TYPE     WC    ABSOLUTE ADDRESS
xxxxxxxxx       LDT(xxxx)     xxxxxxxH    xxxxx    xx      xxxxxxxxH
xxxxxxxxx       GDT(xxxx)     xxxxxxxH    xxxxx    xx      xxxxxxxxH
    .               .             .          .     .           .
    .               .             .          .     .           .
```

Figure 4-8. MAP386 Public Map

Task Map

For each task in an input module, the task map contains the initial stacks for all privileges, flags, initial CS and EIP values, the task's LDT selector, initial segment register specifications, and the task's page directory register.

The task map lists tasks alphabetically. Figure 4-9 shows the format of the task map.

```
TASK MAP

 task name   TSS      = GDT (index)  DPL = number     PBIT = x
             BASE     = xxxxxxxxH
             LIMIT    = xxxxxxxxH
             SSO:ESPO = "table (index)";xxxxxxxxH
             SS1:ESP1 = "table (index)";xxxxxxxxH
             SS2:ESP2 = "table (index)";xxxxxxxxH
             SS:ESP   = "table (index)";xxxxxxxxH
             CS:EIP   = "table (index)";xxxxxxxxH
             DS       = "table (index)";
             FLAGS    = xxxxxxxxH
             LDTSEL   = ldt_name AT GDT (index)
             PDR      = xxxxxxxxH
             DEBUG TRAP NOT ENABLED
.
.
.
```

Figure 4-9. MAP386 Task Map

Cross-Reference Map

The cross-reference map can be included in the print file or directed to a separate file with the XREF control. If the NOPRINT control is specified and XREF is not specified with a different file name, the map is not printed.

The cross-reference map lists the name of each symbol in the specified input module or modules, the type of each symbol, the name of the module containing the symbol's public definition, and the names of any modules containing external references for the symbol.

A warning is issued if a type mismatch occurs between two symbols. If a type mismatch occurs between a public definition and an external reference, the type used in the public definition is the symbol type listed in the cross-reference map. If a symbol has not been assigned a type, NULL appears in the type column. If the type is not defined in the public definition, the cross-reference map lists the type used in the external declaration.

If the public definition of a symbol is not found in the input modules, MAP386 prints the following message in the column labeled DEFINING MODULES:

```
**** UNRESOLVED ****
```

If a public symbol is defined more than once, MAP386 prints the following message:

```
**** DUPLICATE DECLARATION **** : mod_name.
```

Figure 4-10 shows the format of the cross-reference map.

```
CROSS REFERENCE MAP FOR:  filename

SYMBOL NAME         TYPE           DEFINING MODULE    REFERRING MODULES

xxxxxxxxxxxxxx      xxxxxxxx       xxxxxxxxx          xxxxxxx   xxxxx
                                                      xxxxxx   xxxxxx
      .                 .              .                       .
      .                 .              .                       .
      .                 .              .                       .
```

Figure 4-10. MAP386 Cross-Reference Map

Warning and Error Messages

The last item in the print file is the message `PROCESSING COMPLETED` and the number of error and warning messages generated during processing. See Appendix C for descriptions of MAP386 error and warning messages.

Descriptor Segment Naming

If descriptors are unnamed, MAP386 assigns qualified names with a question mark (?) as the first character. The format of the qualified descriptor is as follows:

```
?xxxx.n
```

Where:

xxxx is the descriptor type

n is a sequence number assigned to the descriptor.

Descriptor types can be one of the following:

SEGMENT (code and data segments)
LDT
TASK
286CALL_GATE
386CALL_GATE
TASK_GATE
286INTR_GATE
386INTR_GATE
286TRAP_GATE
386TRAP_GATE

The sequence numbers start from one and increase in order of the segment's occurrence in the object file. The LDT segments are an exception; their names are qualified as ?LDT.n where n is the sequence number assigned to the LDT in the table map.

DOS and iRMX Examples Using MAP386

The following example shows a sample MAP386 invocation and the resulting print file.

```
MAP386 TESTC2.386 PRINT (TEST.MAP) NOTYPECHECK&
PAGEWIDTH (100) OSINFO (OSFILE)&
OBJECTCONTROLS (NOLINES, NOSRCLINES)
```

MAP386 uses TESTC2.386 as input modules. The output consists of all maps, including a cross-reference map, and the maps are directed to the print file, TEST.MAP. Because the input file is linkable, MAP386 does not create an output object file. The default `PAGELENGTH` and `PAGING` controls are in effect. Page width is set to 100 with the `PAGEWIDTH` control, resulting in printed pages 60 lines long and 100 columns wide, with headings, but no title, at the top of each page. The contents of the file `OSFILE` are included in the operating system information field of the output object file named TESTC2.OUT. The object controls specify that no line or source line information is included in the output file. No type checking is done. The name of the print file is TEST.MAP.

Figure 4-11 shows the print file for DOS and iRMX.

```
386(TM) MAPPER

system_id iRMX III 386(TM) MAPPER, Vx.yVX

INPUT FILE(S):        TESTC2.386
INPUT OSINFO FILE:    OSFILE
OUTPUT OBJECT FILE:   TESTC2.OUT
OUTPUT PRINT FILE:    TEST.MAP
OUTPUT XREF FILE:     TEST.MAP
CONTROLS SPECIFIED: PR((COST.MAP) NOTC PW(100) OSINFO(INFILE) OC(NOLI.NOSL)

MODULES INCLUDED:

     FILE NAME     MODULE NAME(S)

     TESTC2.386    MAIN           HELP            MOD2           ?DEBUG_INFO

DESCRIPTOR TABLE MAP

TABLE = GDT        BASE = ---------    LIMIT = 002FH

TABLE INDEX       SELECTOR     DESCRIPTOR NAME

      1             0008H      CODE
      2             0013H      PLMPROC1
      3             0018H      PLMPROC2
      4             0020H      LDT?
      5             0028H      TASK1

TABLE = IDT       BASE = ---------    LIMIT = 00F7H

TABLE INDEX      DESCRIPTOR NAME

      0            TRAPGATE

TABLE = LDT?           SEQ. NO. = 1   BASE = ---------  LIMIT = 0027H

TABLE INDEX       SELECTOR     DESCRIPTOR NAME
      1             000CH      LDT?
      2             0017H      CODE32
      3             001FH      DATA
      4             0027H      DATA
```

Figure 4-11. Print File Example on DOS and iRMX

```
SEGMENT MAP

TABLE = GDT

SEGMENT NAME  TABLE INDEX  PBIT   USE  DPL  ALIGN  ACCESS   BASE       LIMIT  FIX

CODE             1         1    USE16  3   WORD   ER     00003000H     004BH   0

TABLE = LDT?

SEGMENT NAME  TABLE INDEX  PBIT   USE  DPL  ALIGN  ACCESS   BASE       LIMIT  FIX

CODE32           2         1    USE32  3    PARA   ER    00001000H 000000DAH 00000
DATA             3         1    USE32  3    PARA   RWD   00005000H FFFFFFFFH 00000
DATA             4         1    USE16  3    WORD   RWD   FFFF4012H     FFFDH     0
LDT?:            1         1    USE16  0    NONE   RW    ---------     0027H     0

GATE MAP

TABLE = GDT

GATE NAME     TABLE INDEX  PBIT  DPL  TYPE      WC  SELECTOR  OFFSET

PLMPROC1           2       1     3   CALL286    4  GDT(1)     0000H
PLMPROC2           3       1     3   CALL286    4  GDT(1)     0000H

TABLE = IODT

GATE NAME     TABLE INDEX  PBIT  DPL  TYPE      WC  SELECTOR  OFFSET

TRAPGATE           0       1     3    INTR386 ---  LDT(2)     0000H

SYMBOL MAP

MODULE = MAIN, LTD = GDT (4)

SYMBOL NAME     BASE      OFFSET    TYPE                          ABSOLUTE ADDRESS

A . . . . . SS:EBP    FFFFFFFEH   INTEGER(2)                        ---------
AB. . . . . LDT(3)    FFFFFFE6H   INTEGER(4)                        00004FE6H
CFUNC . . . . LDT(2)  0000005BH   C FUNCTION INTEGER(4) NEAR32      0000105BH
IN1 . . . . . SS:EBP  00000008H   INTEGER(4)                        ---------
IN2 . . . . . SS:EBP  0000000CH   INTEGER(4)                        ---------
MAIN. . . . . LDT(2)  00000000H   C FUNCTION INTEGER(4) NEAR32      00001000H
```

Figure 4-11. Print File Example on DOS and iRMX (continued)

```
MODULE = MAIN, CODE SEGMENT = LDT(2)

LINE     OFFSET IN CODE SEGMENT

    5               00000000H             00000006H  00000010H
   10    000000019H  0000002FH  00000038H  00000044H  00000059H
   15    00000005BH             0000005EH  00000066H  00000068H

MODULE - MAIN

SOURCE FILENAME = MAIN.C

LINE     OFFSET IN CODE SEGMENT

    5               00000028H  00000033H  0000003CH  0000004BH
   10    00000005EH  00000078H  00000087H  0000009DH  000000BBH
   15    0000000C5H  000000E5H  000000EDH  00000106H  0000010EH


MODULE = HELP, LDT = GDT(4)

SYMBOL NAME     BASE    OFFSET    TYPE                          ABSOLUTE ADDRESS

A . . . . . SS:EBP  FFFFFFFEH   INTEGER(2)                       ---------
ABC . . . . . LDT(3)  FFFFFFE6H   INTEGER(4)                     00004FF6H
CFUNC1. . . . LDT(2)  000000CBH   C FUNCTION INTEGER(4) NEAR32   000010CBH
HELPFUNC. . . LDT(2)  00000070H   C FUNCTION INTEGER(4) NEAR 32  00001070H
IN1 . . . . . SS:EBP  00000008H   INTEGER(4)                     ---------
IN2 . . . . . SS:EBP  0000000CH   INTEGER(4)                     ---------

MODULE = HELP, CODE SEGMENT = LDT(2)

LINE     OFFSET IN CODE SEGMENT

    5               00000070H             00000076H  00000080H
   10    000000089H  0000009FH  000000A8H  000000B4H  000000C9H
   15    0000000CBH             000000CEH  000000D6H  000000D8H

MODULE = HELP

SOURCE FILENAME:  HELP.C

LINE     OFFSET IN CODE SEGMENT

    5               00000029H  00000038H  00000041H  00000050H
   10    000000063H  0000007EH  0000008DH  000000A3H  000000C1H
   15    0000000CBH  000000EBH  000000F3H  0000010CH  00000114H
```

Figure 4-11. Print File Example on DOS and iRMX (continued)

```
MODULE = MOD2, LDT = GDT(4)

SYMBOL NAME     BASE    OFFSET    TYPE                          ABSOLUTE ADDRESS

B . . . . . LDT(3)  00000022H   INTEGER(4)                       00005022H
C . . . . . LDT(3)  00000026H   INTEGER(2)                       00005026H
PLMPROC1. . . GDT(2)  00000000H   NULL                           00003000H
PLMPROC2. . . GDT(3)  00000000H   NULL                           00003000H
TASK1 . . . . GDT(5)  00000000H   NULL                           00002000H
TRAPGATE. . . IDT(?)  00000000H   NULL                               ?

PUBLIC MAP


MODULE = MAIN

PUBLIC NAME     BASE    OFFSET    TYPE

CFUNC2. . . . LDT(2)  0000005BH   C FUNCTION INTEGER(4) NEAR32 255 0000105BH
MAIN. . . . . LDT(2)  00000000H   C FUNCTION INTEGER(4) NEAR32 255 00001070H

MODULE = HELP

PUBLIC NAME     BASE    OFFSET    TYPE                       WC ABSOLUTE ADDRESS

CFUNC1. . . . LDT(2)  000000CBH   C FUNCTION INTEGER(4) NEAR32 255 000010CBH
HELPFUNC. . . LDT(2)  00000070H   C FUNCTION INTEGER(4) NEAR32 255 00001070H

MODULE = MOD2

PUBLIC NAME     BASE    OFFSET    TYPE                       WC ABSOLUTE ADDRESS

PLMPROC . . . GDT(1)  0000H       C FUNCTION FAR16            2      00003000H

MODULE = ?DEBUG_INFO

PUBLIC NAME     BASE    OFFSET    TYPE                       WC ABSOLUTE ADDRESS

B . . . . . LDT(3)  0000003CH   INTEGER(4)                     0   0000503CH
C . . . . . LDT(3)  00000040H   INTEGER(2)                     0   00005040H
PLMPROC1. . . GDT(2)  00000000H   NULL                         -   00003000H
PLMPROC2. . . GDT(3)  00000000H   NULL                         -   00003000H
TASK1 . . . . GDT(5)  00000000H   NULL                         -   00002000H
TRAPGATE. . . IDT(?)  00000000H   NULL                         -       ?
```

Figure 4-11. Print File Example on DOS and iRMX (continued)

```
TASK MAP

TASK1         TSS      = GDT(5)     DPL = 0  PBIT = 1
                 BASE  = 00002000H
              LIMIT    = 00000067H
              SS0:ESP0 = ----------------
              SS1:ESP1 = ----------------
              SS2:ESP2 = ----------------
              SS:ESP   = ----------------
              CS:EIP   = LDT(2):00000000H
              DS       = LDT(3)
              FLAGS    = 00000200H
              LDTSEL   = ------
              PDR      = 00000000H
              DEBUG TRAP NOT ENABLED

CROSS REFERENCE MAP FOR:  TESTC2.386

SYMBOL NAME     TYPE                          DEFINING MODULE  REFERRING MODULE

B . . . . . .   INTEGER(4)                    ?DEBUG_INFO       MOD2    HELP

C . . . . . .   INTEGER(2)                    ?DEBUG_INFO       MOD2    HELP
CFUNC1. . . .   C FUNCTION INTEGER(4) NEAR32 HELP                        MAIN
CFUNC2. . . .   C FUNCTION INTEGER(4) NEAR32 MAIN                        HELP

HELPFUNC. . .   C FUNCTION INTEGER(4) NEAR32 HELP;

MAIN. . . . .   C FUNCTION INTEGER(4) NEAR32 MAIN;

PLMPROC . . .   C FUNCTION FAR16             MOD2;
PLMPROC1. . .   NULL                         ?DEBUG_INFO              MAIN
PLMPROC2. . .   NULL                         ?DEBUG_INFO              HELP

TASK1 . . . .   NULL                         ?DEBUG_INFO;
TRAPGATE. . .   NULL                         ?DEBUG_INFO;
PROCESSING COMPLETED.     0 WARNINGS,    0 ERRORS
```

Figure 4-11. Print File Example on DOS and iRMX (continued)

❑❑❑

BND386 Error Messages A

BND386 issues a message when it encounters one of the following conditions:

- WARNING: Although a questionable condition exists, the output object file is valid.
- ERROR: The output object file is probably invalid even though BND386 processing can continue.
- FATAL ERROR: BND386 processing aborts. All open files are closed. The object file created, if any, is not complete.

Numbers that accompany the messages indicate the location of the exception, as follows:

- No number: Exception is at the system interface level.
- 100-199: Exception is in the invocation line or in an input object file.
- 300-399: Exception is in internal BND386 processing.

Messages appear in the print file and any file created with ERRORPRINT. Fatal error messages also appear at the console.

This appendix provides up to four kinds of information for each message:

- MEANING: how to interpret the message.
- CAUSE: the probable reason for the message.
- EFFECT: the state of the output file(s) and the status of BND386.
- ACTION: suggestions for correcting the condition.

Messages are listed in numerical order.

System-Level Exceptions

```
SYSTEM INTERFACE ERROR
error text
FILE: filename
```

MEANING: This fatal error occurs in a call to the host operating system. The error text contains a message issued by the operating system. The *filename* is present only if the error is an I/O error.

CAUSE: Such problems as an I/O error, invalid parameters, or insufficient memory can cause this condition.

EFFECT: BND386 processing is aborted, and control is returned to the operating system.

ACTION: Refer to host operating system documentation for interpretation. Correct the error and reinvoke BND386.

Invocation or Input Object Exceptions

```
ERROR 100: INPUT FILE MISSING
```

MEANING: This fatal error occurs because no linkable input file is provided.

EFFECT: Execution aborts and control returns to the operating system.

ACTION: Specify at least one linkable input file, and then reinvoke.

```
ERROR 101: FILENAME TOO LONG
NEAR: token string
```

MEANING: This fatal error occurs because there are too many characters in a *filename* in the invocation line near the *token string*.

EFFECT: Execution aborts and control returns to the operating system.

ACTION: Use a valid filename, and then reinvoke.

```
ERROR 102: MISSING LEFT PARENTHESIS
NEAR: token string
```

MEANING: This fatal error occurs because a left parenthesis is missing after the *token string*.

EFFECT: Execution aborts and control returns to the operating system.

ACTION: Insert a left parenthesis in the proper location, and then reinvoke.

```
ERROR 103: MISSING RIGHT PARENTHESIS
NEAR: token string
```

MEANING: This fatal error occurs because a right parenthesis is missing after the *token string*.

EFFECT: Execution aborts and control returns to the operating system.

ACTION: Insert a right parenthesis in the proper location, and then reinvoke.

```
ERROR 105: FILE ALREADY SPECIFIED IN COMMAND TAIL
FILE: filename
```

MEANING: This fatal error occurs because the *filename* is already specified in the input file list. One of the duplicate filenames is explicit in or implied by the controls.

EFFECT: Execution aborts and control returns to the operating system.

ACTION: Ensure that none of the explicit or default file names match, and then reinvoke.

```
ERROR 106: INVALID DELIMITER IN COMMAND TAIL
NEAR: token string
```

MEANING: This fatal error occurs because the invocation line contains an improperly placed delimiter or uses an illegal character as a delimiter. The invalid delimiter is detected either before or after the token string.

EFFECT: Execution aborts and control returns to the operating system.

ACTION: Use valid delimiters, including properly placed left and right parentheses and commas, and then reinvoke.

```
ERROR 107: LINE TOO LONG IN CONTROL FILE
FILE: filename
```

MEANING: This fatal error occurs because the control file named by *filename* contains a line longer than 128 characters.

EFFECT: Execution aborts and control returns to the operating system.

ACTION: Correct the control file, making sure that all lines, except for comments, are less than 128 characters long, and then reinvoke.

```
ERROR 108: TOKEN TOO LONG
NEAR: token string
```

MEANING: This fatal error occurs because the specified *token string* contains too many characters, e.g., a module name exceeding 40 characters.

EFFECT: Execution aborts and control returns to the operating system.

ACTION: Use a token string of valid length, and then reinvoke.

```
ERROR 109: UNKNOWN CONTROL IN COMMAND TAIL
NEAR: token string
```

MEANING: This fatal error occurs because the control indicated in *token string* in the invocation line is invalid.

EFFECT: Execution aborts and control returns to the operating system.

ACTION: Spell the control correctly or use the proper abbreviation, and then reinvoke.

```
ERROR 110: SYNTAX ERROR
NEAR: token string
```

MEANING: This fatal error occurs because the structure of the invocation line near *token string* is incorrect.

EFFECT: Execution aborts and control returns to the operating system.

ACTION: Make sure that the information in the invocation line is complete, appears in the proper order, and is spelled correctly, and then reinvoke.

```
ERROR 112: NUMBER OF SYMBOLS EXCEEDS INTERNAL LIMIT
```

MEANING: This fatal error occurs because the maximum number of symbols BND386 can process has been exceeded.

EFFECT: Execution aborts and control returns to the operating system.

ACTION: Invoke BND386 with fewer input modules, or use incremental binding with the NOPUBLICS control.

```
ERROR 114: INVALID OBJECT FILE
FILE: filename
MODULE: mod_name
```

MEANING: This fatal error indicates that the *mod_name* contained in the file referred to by the file name has an invalid format.

CAUSE: This condition occurs because a non-object file, such as a source file, or a loadable module has been specified as the input file. It can also occur because of an internal translator or utility error.

EFFECT: Execution aborts and control returns to the operating system.

ACTION: Verify that the input list specifies the correct files and retranslate the source files with 80286/Intel386 translators to create linkable input modules for BND386. If the problem persists, contact RadiSys.

```
ERROR 115: DUPLICATE MODULE NAME
NEAR: mod_name
```

MEANING: This fatal error occurs because the specified module appears more than once in the input list.

EFFECT: Execution aborts and control returns to the operating system.

ACTION: Make sure that *mod_name* is selected only once in the input list, and then reinvoke. If you need both selected modules, pre-link one of them, changing its name with the NAME control.

```
ERROR 116: NESTED CONTROL FILES
FILE: filename
```

MEANING: This fatal error occurs because the control file identified by the *filename* contains a reference to another control file.

EFFECT: Execution aborts and control returns to the operating system.

ACTION: Eliminate the nesting in the control file, and then reinvoke BND386.

```
ERROR 118: PAGE FILE OVERFLOW
```

MEANING: This error occurs because an attempt was made to build an image greater than 8M bytes.

EFFECT: Execution aborts and control returns to the operating system.

ACTION: Rewrite your program so that it occupies less than 8M bytes, and then reinvoke.

```
ERROR 121: MISMATCHED SEGMENT ATTRIBUTES
FILE: filename
MODULE: mod_name
SEGMENT: seg_name
```

MEANING: This error condition exists because segments with the same seg_name but with incompatible segment attributes are detected. The segment that triggered this error condition is in *mod_name* and *filename*.

EFFECT: Processing continues, but the output object module is not usable.

ACTION: Assign compatible attributes. Reinvoke BND386 with the adjusted input file or files.

```
ERROR 122: SAME SEGMENT PLACED AT TWO DIFFERENT ENTRIES IN LDT
FILE: filename
MODULE: mod_name
SEGMENT: seg_name
```

MEANING: This error condition occurs because duplicate selectors for *seg_name* in the *input_list* are detected. The segment whose selector caused this condition is in *mod_name* and *filename*.

CAUSE: The *input list* may contain multiple system files (e.g., export modules from the system builder) that have ambiguities.

EFFECT: Processing continues, but the output object module is not usable.

ACTION: Use only the essential system files, and then reinvoke.

```
ERROR 123: SEGMENT OVERFLOW DUE TO SEGSIZE VALUE
SEGMENT: seg_name
```

MEANING: This error occurs because the SEGSIZE specification is too large. The limit for USE16 segments is 64K bytes; the limit for USE32 segments is 4G bytes.

EFFECT: Processing continues, and the output module may be valid; however, other errors may occur while processing fixups.

ACTION: Ensure that the size value used in the SEGSIZE specification is accurate. Reinvoke if necessary.

```
ERROR 124: SEGMENT UNDERFLOW DUE TO SEGSIZE VALUE
SEGMENT: seg_name
```

MEANING: This error indicates that the SEGSIZE specification caused the segment size to go below zero.

EFFECT: Processing continues; however, later processing may be affected, e.g., when the sum of all segment sizes in input is calculated.

ACTION: Use a different segment size control, and reinvoke BND386 if necessary.

```
ERROR 125: SAME SYMBOL DEFINED TO BE IN DIFFERENT SEGMENTS
FILE: filename
MODULE: mod_name
SEGMENT: seg_name
SYMBOL: symbol name
```

MEANING: This error condition occurs because the specified symbol is defined in a segment different from the segment used in the public definition. The module and file containing the symbol name declaration are defined by *mod_name* and *filename*. This usually occurs because of incompatibilities in the segmentation models.

EFFECT: Processing continues, but the output object module is invalid. The first symbol address remains effective.

ACTION: Verify that the correct libraries for the model of segmentation are used. Correct the source file public-external declarations, retranslate, and then reinvoke.

```
WARNING 126: SYMBOL TYPES MISMATCH
FILE: filename
MODULE: mod_name
SYMBOL: symbol name
```

MEANING: This warning condition occurs because there are two symbols of different types with the same name.

EFFECT: Processing continues.

ACTION: Set the symbols to the same type (special care must be taken when the modules are produced by different translators), recompile the modules, and then reinvoke.

```
WARNING 127: DUPLICATE PUBLIC SYMBOL
FILE: filename
MODULE: mod_name
SYMBOL: symbol name
```

MEANING: This warning condition exists because a symbol is defined as public in more than one input module. The module containing the definition causing this warning is identified by *mod_name*, which is in *filename*.

EFFECT: Processing continues, and the output is valid. The first instance of the public symbols remains effective for later processing.

ACTION: Remove the unneeded public definition and reinvoke.

```
WARNING 128: SPECIFIED MODULE NOT FOUND IN INPUT FILE
FILE: filename
MODULE: mod_name
```

MEANING: This warning occurs because a module is specified in the input list, but cannot be found in the associated file, identified by *filename*.

EFFECT: Processing continues as if the module had not been specified.

ACTION: If it is necessary to include the referenced module in the output module, reinvoke BND386, using a file containing the module.

```
ERROR 129: CS REGISTER INITIALIZED BY NON-EXECUTABLE SEGMENT
FILE: filename
MODULE: mod_name
SEGMENT: seg_name
```

MEANING: This error condition occurs because a segment is not executable, but is identified in the main module named by *mod_name*, as the initial code segment. The module named by *mod_name* resides in *filename*.

EFFECT: Processing continues, but the output object module is not usable.

ACTION: Ensure that CS register initialization requirements are correctly specified in the input module or modules, and then reinvoke BND386.

```
ERROR 130: SS REGISTER INITIALIZED BY NON-WRITABLE SEGMENT
FILE: filename
MODULE: mod_name
SEGMENT: seg_name
```

MEANING: This error condition occurs because *seg_name* is not writable, but is identified in the specified main module (named by *mod_name*) as an initial stack segment. The module named by *mod_name* resides in *filename*. This is almost always due to an invalid END directive in the assembly program.

EFFECT: Processing continues, but the output module is unusable.

ACTION: Ensure that SS register initialization requirements are correctly specified in the input module or modules, and then reinvoke BND386.

```
WARNING 131: REFERENCE TO UNRESOLVED EXTERNAL SYMBOL
FILE: filename
MODULE: mod_name
REFERRING LOCATION: location
REFERENCED LOCATION: target
```

MEANING: This warning occurs when BND386 creates loadable output and finds reference to an unresolved external symbol in an input file. The *target* specifies the unresolved external symbol name.

EFFECT: Processing continues, but the output module might not be usable if the loader does not resolve the external.

ACTION: If the symbol cannot (or should not) be resolved at load time, reinvoke BND386, using a file that resolves the external reference.

```
ERROR 132: REFERENCED LOCATION BEYOND LIMIT
FILE: filename
MODULE: mod_name
REFERRING LOCATION: location
REFERENCE LOCATION: target
```

MEANING: This error occurs because the target information is not contained in the referenced segment, probably because one of the segment limit values is too small. The referring location is in *mod_name* and *filename*.

EFFECT: Processing continues, but the output module is not usable.

ACTION: Correct the size of the segment being referred to, e.g., by using the SEGSIZE control when you reinvoke BND386.

```
WARNING 133: SEGMENT LIMIT DECREASED DUE TO SEGSIZE VALUE
SEGMENT: seg_name
```

MEANING: This warning occurs because the size of the segment named by *seg_name* is decreased because of a SEGSIZE specification.

EFFECT: Processing continues, and the output module is valid. The user-specified value overrides the existing segment size.

ACTION: Ensure the segment size specification is accurate, reinvoking BND386 if necessary to respecify the size.

```
ERROR 135: ENTRY POINT SPECIFIED IN GATE IS NON-EXECUTABLE
FILE: filename
MODULE: mod_name
REFERRING LOCATION: location
REFERENCE LOCATION: target
```

MEANING: This error condition occurs because the entry point identified by *target* is in a nonexecutable segment. Gates are used to mediate access to code segments only.

EFFECT: Processing continues, but the output module is not usable.

ACTION: Reinvoke BLD386 to revise the gate and/or segment in the input system file or files to make the access possible; e.g., make the segment containing the entry point executable. Reinvoke BND386 using the corrected input module or modules.

```
ERROR 136: ENTRY POINT SPECIFIED IN GATE IS LESS PRIVILEGED
THAN GATE
FILE: filename
MODULE: mod_name
REFERRING LOCATION: location
REFERENCED LOCATION: target
```

MEANING: This error occurs because a gate is not pointing to an executable entry point with a higher privilege level (numerically smaller) than the gate itself.

EFFECT: Processing continues, but the output module is not usable.

ACTION: Reinvoke BND386 with the correct system interface file.

```
ERROR 137: BAD SELF RELATIVE REFERENCE
FILE: filename
MODULE: mod_name
REFERRING LOCATION: location
REFERENCE LOCATION: target
```

MEANING: This error occurs because target and location are not in the same segment. The location is in *mod_name* in *filename*.

CAUSE: This condition may be caused by a translator error or an erroneous ASM386 code.

EFFECT: Processing continues, but the output module is not usable.

ACTION: Retranslate the module, and then reinvoke BND386 with the new input file.

```
ERROR 138: REFERRING LOCATION BEYOND LIMIT
FILE: filename
MODULE: mod_name
REFERRING LOCATION: location
REFERENCE LOCATION: target
```

MEANING: This error occurs because the location referring to target is outside the limits of the segment. The location is in *mod_name* in *filename*.

CAUSE: This condition may exist because the SEGSIZE control specified a segment size that is too small, or it may be due to a translator error.

EFFECT: Processing continues, but the output module is not valid.

ACTION: Retranslate the module if necessary, and then reinvoke BND386 using a valid SEGSIZE control specification.

```
ERROR 139: CONSTANT VALUE OVERFLOW
FILE: filename
MODULE: mod_name
REFERRING LOCATION: location
REFERENCE LOCATION: target
```

MEANING: This error occurs because the sum of two constant values, represented by a public symbol at *target*, and an incremental value in the instruction at location, cause a byte or word overflow, depending on the input specification.

EFFECT: Processing continues, and the output module is valid.

ACTION: Reassign values if required, and then reinvoke.

```
ERROR 140: TEXT LENGTH IS GREATER THAN SEGMENT LENGTH
FILE: filename
MODULE: mod_name
SEGMENT: seg_name
```

MEANING: This fatal error occurs because the limit of the segment named by *seg_name* is not large enough. The text may be an actual text item or may be represented by a fixup. The segment is contained in *mod_name* in filename.

EFFECT: Execution aborts and control returns to the operating system.

ACTION: Increase the limit of the segment, e.g., by reinvoking BND386 with an appropriate SEGSIZE control parameter.

```
ERROR 141: CS REGISTER NOT INITIALIZED
FILE: filename
MODULE: mod_name
```

MEANING: This error occurs because BND386 cannot find an initialization value for the CS register in the module named by *mod_name* in *filename*. This usually happens with input that includes ASM386 main modules, but it may also happen if the input does not include a main module.

EFFECT: Processing continues, but the output module is not valid.

ACTION: Include a main module or provide ASM386 CS initialization information, and then reinvoke.

```
WARNING 142: SS REGISTER NOT INITIALIZED
FILE: filename
MODULE: mod_name
```

MEANING: This error occurs because the module named by *mod_name* does not contain SS register initialization information. This usually happens with input that includes ASM386 main modules; it may also happen if the input does not include a main module.

EFFECT: Processing continues, but the output module is not valid.

ACTION: Include a main module or provide ASM386 SS initialization information, and then reinvoke.

```
WARNING 143: DS REGISTER NOT INITIALIZED
FILE: filename
MODULE: mod_name
```

MEANING: This warning occurs because the module identified by *mod_name* does not contain DS initialization information. This warning usually happens with input that includes ASM386 main modules, but it may also happen if the input does not include a main module.

EFFECT: Processing continues, but the output module is not usable.

ACTION: Include a main module or provide ASM386 DS initialization information. and then reinvoke.

```
ERROR 144: SEGMENT OVERFLOW DUE TO SEGMENT COMBINATION
FILE: filename
MODULE: mod_name
SEGMENT: seg_name
```

MEANING: This error occurs because the combination of segments identified by *seg_name* results in a physical segment larger than 64K bytes for USE16 segments, or larger than 4G bytes for USE32 segments. The segment causing the overflow condition is in *mod_name*.

EFFECT: Processing continues, but the output module is not usable. Such overflow may cause errors during fixup processing.

ACTION: Remove modules and/or segments that are not needed. Modify the size of the input segments or rearrange their contents. Use RENAMESEG to prevent unnecessary segment combination. Reinvoke.

```
ERROR 145: TEXT FOUND IN STACK SEGMENT
FILE: filename
MODULE: mod_name
SEGMENT: seg_name
```

MEANING: This error occurs because the stack segment named by *seg_name* in *mod_name* contains text.

EFFECT: Processing continues, but the effect on the output module is not known.

ACTION: Ensure that the result is acceptable, reinvoking BND386 if necessary.

```
ERROR 146: TYPE DESCRIPTION TOO LONG
FILE: filename
MODULE: mod_name
```

MEANING: This fatal error indicates that the type definition is too long and requires space that exceeds an internal limit of BND386.

EFFECT: Processing is aborted, and control returns to the operating system.

ACTION: Simplify the type of the symbol and retranslate. Use the NOTYPE control.

```
WARNING 147: SEGMENT SPECIFIED IN SEGSIZE CONTROL NOT FOUND
NEAR: token string
```

MEANING: This warning indicates that the seg_name specified in a SEGSIZE control specification is not in any of the input modules.

EFFECT: Processing continues, and the output module is valid.

ACTION: Use a different input list or a different SEGSIZE control and reinvoke BND386.

```
WARNING 149: SEGMENT IN SEGSIZE CONTROL IS NON-WRITABLE
SEGMENT: seg_name
```

MEANING: This error occurs because *seg_name* is found in a SEGSIZE control specification, but the segment is not writable.

EFFECT: Processing continues, and the output module is valid.

ACTION: Reinvoke BND386 after changing the attribute of the segment, if required, using the correct SEGSIZE specification.

```
ERROR 150: INPUT HAS TWO MODULES WITH SAME NAME
FILE: filename
MODULE: mod_name
```

MEANING: This error occurs because the input-list contains two modules with the same name.

EFFECT: Processing continues, but the output module is not usable.

ACTION: Eliminate the duplication of names and reinvoke.

WARNING 151: UNRESOLVED EXTERNAL SYMBOLS

MEANING: This warning occurs because the input contains references to public symbols that are not defined in the input.

EFFECT: Processing continues, and the output module is valid.

ACTION: If this message is not accompanied by "WARNING 131: REFERENCE TO UNRESOLVED EXTERNAL SYMBOL", the symbol is never used and the only thing you need to do is remove its definitions. Otherwise, when creating a loadable module, ensure that BND386 is invoked with input modules that contain public definitions for all external declarations in the input, unless this symbol is supposed to be resolved at load time.

WARNING 152: PUBLIC GATE CANNOT BE EXCLUDED FROM OUTPUT
SYMBOL: *symbol name*

MEANING: This warning occurs because an attempt has been made to purge a public symbol associated with a gate.

EFFECT: Processing continues, and the output module is valid. The public symbol is not purged.

ACTION: Respecify the PUBLICS control to ensure that the public gates are not excluded from the output.

ERROR 153: REGISTER INITIALIZED BY AN EMPTY SEGMENT
FILE: *filename*
MODULE: *mod_name*
SEGMENT: *seg_name*

MEANING: This error occurs because the segment identified by *seg_name* is empty.

EFFECT: Processing continues, but the output module is not valid. Message 141, 142, or 143 also appears.

ACTION: Provide adequate initialization information in the input segments, and then reinvoke.

```
ERROR 154: REFERENCED GATE IS AT HIGHER PRIVILEGE
FILE: filename
MODULE: mod_name
GATE: gate name
REFERRING LOCATION: location
REFERENCED LOCATION: target
```

MEANING: This error occurs if a reference to a gate at a higher privilege level (that is, numerically lower) is found in a loadable case.

EFFECT: Output may not be valid.

ACTION: Reinvoke BND386 with the correct system interface file.

```
ERROR 156: MULTIPLE REGISTER INITIALIZATIONS IN INPUT
FILE: filename
MODULE: mod_name
```

MEANING: This error occurs because the input list contains more than one main module.

EFFECT: Processing continues, and the output module is valid. BND386 extracts initialization information from the first main module in the input.

ACTION: Use input containing only one main module, and then reinvoke BND386.

```
ERROR 157: REFERENCE TO SYMBOL ON STACK FOUND
FILE: filename
MODULE: mod_name
REFERRING LOCATION: location
REFERENCED LOCATION: target
```

MEANING: This error occurs because a public symbol is defined to be in stack and this public symbol is referred to in a fixup.

EFFECT: Processing continues, but BND386's actions are undefined and the output module is not valid.

ACTION: Correct the input module, and then reinvoke BND386.

```
ERROR 159: REFERENCE TO AN EMPTY SEGMENT
FILE: filename
MODULE: mod_name
SEGMENT: seg_name
```

MEANING: This error occurs because an empty segment, identified by *seg_name*, is referred to in the input.

EFFECT: Processing continues, but the output may not be valid.

ACTION: Ensure that the reference is correct, reinvoking BND386 with different input modules if necessary.

Note

If a module contains an empty data segment and the selector of the segment is used as a parameter in a PUSH instruction (e.g., PUSH DS), the output contents may still be usable. This is legal if no other references to the specified segment exist. This case occurs by default if a main module is written in a high-level language (e.g., PL/M) and the module contains no data. Only in these specific cases may this error message be ignored.

```
WARNING 161: SPECIFIED SYMBOL IN PUBLICS CONTROL NOT FOUND
SYMBOL: symbol name
```

MEANING: This warning occurs because BND386 cannot find *symbol name*, named in a PUBLICS control, in the input modules.

EFFECT: Processing continues and the output module is valid. BND386 ignores the control referring to this symbol.

ACTION: Ensure that the desired effect has been achieved, reinvoking BND386 with a different PUBLICS control specification if required.

```
ERROR 165: NO REGISTER INITIALIZATION IN INPUT
```

MEANING: This error occurs because the input contains no main module.

EFFECT: Processing continues, but the loadable output module cannot be executed.

ACTION: Reinvoke BND386 using input that contains one main module.

```
ERROR 166: INCREMENTAL BUILD INPUT FILES ARE NOT ALLOWED
FILE: filename
```

MEANING: This fatal error occurs because the file in input is an incrementally built file produced by BLD286/386 and cannot be processed by BND386.

EFFECT: Execution aborts and control returns to the operating system.

ACTION: Remove the incrementally built file from the input list. You may need to use an export file in the input list. Reinvoke.

```
ERROR 167: OFFSET OF SYMBOL BEYOND SEGMENT LIMIT
FILE: filename
MODULE: mod_name
SEGMENT: seg_name
SYMBOL: symbol name
```

MEANING: This error condition occurs because the offset of the specified symbol is located beyond the segment limit.

EFFECT: Processing continues, but the output may not be usable.

ACTION: Reposition the symbol offset and check the SEGSIZE control. Reinvoke.

```
ERROR 168: SEGMENTS WITH COMMON ATTRIBUTE HAVE DIFFERENT
LENGTHS
FILE: filename
MODULE: mod_name
SEGMENT: seg_name
```

MEANING: This error occurs because the input module contains FORTRAN common segments that do not have the same length. The module name identifies the second module containing a common segment.

EFFECT: Processing continues and the output module is usable. The lengths of the segments remain unchanged.

ACTION: Make the segments the same length or define them as blank common segments, and then reinvoke.

```
WARNING 170: SEGMENT SPECIFIED IN RENAMESEG CONTROL NOT FOUND
SEGMENT: seg_name
```

MEANING: This warning occurs because the segment to be renamed with the RENAMESEG control is not found in the input.

EFFECT: Processing continues and the output module is usable.

ACTION: Specify the correct seg_name in the RENAMESEG control, and then reinvoke.

```
ERROR 171: INVALID PRIVILEGE LEVEL
NEAR: token string
```

MEANING: This fatal error occurs because the privilege level is not within the range 0 to 3.

EFFECT: Execution aborts and control returns to the operating system.

ACTION: Use a privilege level in the range 0 to 3, and then reinvoke.

```
ERROR 172: SYMBOL TYPE INFORMATION IS TOO COMPLEX
FILE: filename
MODULE: mod_name
```

MEANING: This error occurs because the length of the type definitions in the input exceeds internal limits.

EFFECT: No type checking on these type definitions is performed. The output module is usable.

ACTION: Recompile the module in *filename* using the NOTYPE control and reinvoke BND386, or reinvoke BND386 using the NOTYPE control.

```
WARNING 174: CONFLICT IN SYMBOL SIZE REQUIREMENTS
FILE: filename
MODULE: mod_name
SYMBOL: symbol name
```

MEANING: This warning occurs because external symbols of the same name have different lengths.

EFFECT: BND386 allocates the longer length to the symbol.

ACTION: Make sure multiple declarations of the same symbol have the same length.

```
ERROR 177: INVALID USAGE OF THE RCONFIGURE CONTROL
NEAR: token string
```

MEANING: This error occurs because of invalid usage of the `RCONFIGURE` control.

EFFECT: Processing aborts and control returns to the operating system.

ACTION: Respecify the `RCONFIGURE` control.

```
ERROR 180: FIXUP OVERFLOW
FILE: filename
MODULE: mod_name
SEGMENT: seg_name
```

MEANING: This error occurs because an overflow due to segment combination occurred during fixup.

EFFECT: Processing continues; the fixup is not applied, and the output module is usable.

ACTION: This might happen in ASM386 programs, when, for example, a WORD is used to hold an address of an object whose address is above 64K bytes. It can also happen when 16-bit addressing is used for such objects. The first example is corrected by using DWORDS. The second example is corrected by using 32-bit addressing.

```
WARNING 181: SEGMENTS WITH DIFFERENT USE32 ATTRIBUTES WERE
COMBINED
FILE: filename
MODULE: mod_name
SEGMENT: seg_name
```

MEANING: This warning occurs because an attempt was made to combine data or stack segments with different `USE` attributes.

EFFECT: Processing continues; the combined segment has `USE16` attributes.

ACTION: If segment must be larger than 64K bytes, use a different name for the segment to avoid segment combination.

```
WARNING 182: SEGMENT ACCESS RIGHTS WERE RELAXED
FILE: filename
MODULE: mod_name
SEGMENT: seg_name
```

MEANING: This warning occurs because the access rights of the specified segments were changed due to segment combinations. For example, when a segment that was read-only is combined with a segment that is writable, the new segment is writable; thus the protection against accidental writing is lost.

EFFECT: Processing continues and the output module is valid.

ACTION: If you need the original access rights, change the input so that the segments will not be combined.

ERROR 183: INPUT MODULE HAS DUPLICATE SEGMENT NAMES

MEANING: This error occurs because the specified module has more than one segment with the same combine name.

EFFECT: Processing aborts and control returns to the operating system.

ACTION: This is usually an error in the translator, the input module, or the controls used with the translator. Check the translation process.

Internal Processing Exceptions

If you encounter an error message that has the following format, contact RadiSys for assistance:

```
* * * ERROR 3xx: INTERNAL PROCESSING ERROR, message
```

Where:

3*xx* is the error number

message is a string containing the text of the message

All these messages are caused by fatal errors.

Following is the list of internal processing exceptions, along with their assigned numbers:

```
ERROR 300: INTERNAL ERROR IN PASS1

ERROR 301: INTERNAL ERROR IN PASS2

ERROR 302: INTERNAL ERROR IN OUTPUT

ERROR 303: INTERNAL ERROR IN OBJECT MODULE INTERFACE

ERROR 304: INTERNAL ERROR IN SYMBOL PROCESSING

ERROR 305: UNKNOWN ERROR
```

❑❑❑

LIB386 Error Messages B

LIB386 issues two types of error messages:

- Error and warning messages related to LIB386 processing
- Messages related to the operating system interface

All messages are sent to the current console.

Processing Errors

These errors are detected when LIB386 encounters an invocation error or an illegal input file. LIB386 processing errors also occur because of improper command syntax or semantics. A problem that is treated as a warning is any condition that could lead to an error, such as an empty target library. Each processing error message contains the following:

- The word *** ERROR or *** WARNING
- A unique error number
- A brief explanation of the problem

An error message may also contain one or more of the following items that describe the problem:

- *filename*
- *mod_name*
- *public_symbol_name*

LIB386 has two types of processing error messages:

- 1*xx*: syntax or semantic errors and object-file errors, and warnings
- 3*xx*: internal processing errors

Errors designated 1*xx* may or may not be fatal, depending on the particular problem. Errors designated 3*xx* are always fatal; these are internal to LIB386. You should report errors designated 3*xx* to an RadiSys representative. The following message is displayed for fatal errors:

```
PROCESSING ABORTED
```

This appendix provides up to three kinds of information for each message:

MEANING: how to interpret the message

EFFECT: the status of LIB386

ACTION: suggestions for correcting the condition. Error messages are listed in numerical order.

LIB386 Processing Error Messages

`ERROR 100: INVALID DELIMITER IN INVOCATION LINE`
`NEAR: token`

MEANING: This error occurs because the delimiter used in the invocation line is not valid. Spaces and carriage returns are the only valid delimiters for the invocation line.

EFFECT: Execution aborts and control is returned to the operating system.

ACTION: Retype the command and then reinvoke.

`ERROR 101: UNKNOWN CONTROL IN INVOCATION LINE`
`NEAR: token`

MEANING: This error occurs because an invalid control is specified in the LIB386 invocation line. The only valid controls for the LIB386 invocation line are `BACKUP`, `NOBACKUP`, `BATCH`, and `NOBATCH`.

EFFECT: Execution aborts and control is returned to the operating system.

ACTION: Retype the command and then reinvoke.

ERROR 102: PATHNAME TOO LONG
NEAR: *filename*

MEANING: This error occurs because the specified *filename* is too long. A filename cannot have more than 45 characters.

EFFECT: The command is not executed.

ACTION: Retype the command line with a shorter filename and then reinvoke.

ERROR 103: NUMBER OF SYMBOLS EXCEEDS INTERNAL PROCESSING LIMIT

MEANING: This error occurs because virtual memory is not sufficient for the number of symbols used.

EFFECT: Execution aborts.

ACTION: Reduce the number of public symbols in the target library or allocate more memory, and then reinvoke.

ERROR 104: PREMATURE END OF COMMAND STREAM

MEANING: This error occurs because the command input file ended before the LIB386 session ended.

EFFECT: Execution aborts.

ACTION: In batch mode, make sure a correct QUIT command is present. In interactive mode, exit with a QUIT command, not a Ctrl-Break. Reinvoke.

ERROR 105: INVALID OBJECT FILE
FILE: *filename*

MEANING: This error occurs because the specified input file does not conform to the Intel386 object module format specifications.

EFFECT: The portion of the command pertaining to the offending file is not executed.

ACTION: Verify that the file was created properly. If the file was improperly created, recreate the file with the appropriate translator or Intel386 utility. Reinvoke.

```
ERROR 106: MODULE NOT FOUND
FILE: filename
MODULE: mod_name
```

MEANING: This error occurs if the module specified in the command line is not in the indicated file.

EFFECT: The portion of the command that pertains to the unfound module is not executed.

ACTION: Use the LIST command to verify that the module exists in the specified file, and then reinvoke.

```
ERROR 107: DUPLICATE MODULE
FILE: filename
MODULE: mod_name
```

MEANING: This error occurs because the module specified to be added or replaced already exists in the target library.

EFFECT: The portion of the command pertaining to the offending module is not executed.

ACTION: Delete the existing module before adding the specified module to the library.

```
ERROR 108: DUPLICATE PUBLIC
FILE: filename
MODULE: mod_name
PUBLIC SYMBOL: public_symbol_name
```

MEANING: This error occurs because the module specified to be added or replaced has a public symbol that already exists in the target library.

EFFECT: The portion of the command pertaining to the offending public symbol is not executed.

ACTION: Delete the existing module before adding the module containing the public symbol to the library.

```
ERROR 109: UNABLE TO CREATE BACKUP FILE
FILE: filename
```

MEANING: This error occurs because LIB386 is unable to create the backup file. This can be caused if the file already exists and is write-protected.

EFFECT: No backup file is created, but the session proceeds.

ACTION: If a backup file is desired, exit LIB386 and either delete or write-enable the backup file, then reinvoke LIB386.

`ERROR 110: UNABLE TO ACCESS NON-TARGET LIBRARY FILE`
`FILE: filename`

MEANING: This error occurs when LIB386 is processing the LIST command, and LIB386 cannot open the specified file. This error occurs because the file does not exist or is already open.

EFFECT: The command is not executed.

ACTION: Check that the file is closed and exists and then reinvoke.

`ERROR 111: NON-TARGET FILE IS NOT A LIBRARY`
`FILE: filename`

MEANING: This error occurs because an attempt was made to list a file that is not a library file.

EFFECT: The command is not executed.

ACTION: Files that are not libraries cannot be listed.

`ERROR 112: UNABLE TO CREATE LIST FILE`
`FILE: filename`

MEANING: This error occurs because the LIST TO *filename* command is specified, and LIB386 is unable to create the *filename*. Probable causes are that *filename* is either already open or is an existing, write-protected file.

EFFECT: The command is not executed.

ACTION: Check that the file is closed and write-enabled or make sure that the file does not exist, and then reinvoke.

`ERROR 113: PATHNAME EXPECTED`

MEANING: This error occurs because the LIST command is issued at the initial command level with no *filename* specified.

EFFECT: The command is not executed.

ACTION: Specify a *filename* with the LIST command at the initial command level or initialize a library with the GET command, and then obtain a listing of the library.

`WARNING 114: TARGET LIBRARY FILE IS EMPTY`

MEANING: This warning occurs because an UPDATE command is issued and the target library is empty.

EFFECT: No update is performed.

ACTION: Use the ADD command to add modules to the target library.

```
ERROR 115: ATTEMPT TO UPDATE WRITE PROTECTED FILE
FILE: filename
```

MEANING: This error occurs because an attempt was made to update a file that is write protected.

EFFECT: The UPDATE command is not executed.

ACTION: Do not specify `UPDATE`, `QUIT EXIT`, or `QUIT INITIALIZE` for the file, or remove the write protection.

```
WARNING 116: TARGET LIBRARY NOW EMPTY
```

MEANING: This warning occurs because the COMPRESS command was specified after all modules were deleted from the target library.

EFFECT: If no other modules are added, the next update produces an empty and invalid library file.

ACTION: Use the ADD command to add modules to the target library.

```
ERROR 117: INPUT FILE IS A 286 OBJECT FILE
FILE: filename
```

MEANING: An ADD or REPLACE was attempted using a 286 object file.

EFFECT: The file is not updated.

ACTION: Recompile the file with a 386 translator or relink it with BND386, and then try again.

```
ERROR 118: UNKNOWN COMMAND, TRY AGAIN
```

MEANING: This error occurs because LIB386 does not understand the command.

EFFECT: The command is not executed.

ACTION: Check the command syntax and then reinvoke. Use the HELP command if necessary.

```
ERROR 119: MODULE OR SYMBOL NAME TOO LONG
```

MEANING: This error occurs because the *mod_name* specified with the LIST command or the public symbol name specified with the FIND command is too long. The maximum number of characters is 40.

EFFECT: The command is not executed.

ACTION: Specify a module or symbol name that has 40 characters or less and then reinvoke.

ERROR 120: IMPROPER COMMAND SYNTAX

MEANING: This error occurs because LIB386 detects a command syntax error.

EFFECT: The command is not executed.

ACTION: Check the syntax, and then reinvoke.

ERROR 121: INPUT FILE IS NEITHER LINKABLE NOR LIBRARY

MEANING: This error occurs because the file specified with the ADD or REPLACE command contains no library module or linkable module. Only Intel386 libraries or linkable modules can be added to the target library.

EFFECT: The portion of the command pertaining to the offending file is not executed.

ACTION: Specify an input file that contains either linkable modules or a library, and then reinvoke.

ERROR 122: COMMAND NOT ALLOWED AT CURRENT LEVEL

MEANING: This error occurs because the command specified is not valid at the current command level (e.g., the initial level GET command is specified at the action command level).

EFFECT: The command is not executed.

ACTION: Use the GET or the QUIT command to return control to the proper command level, and then reinvoke.

ERROR 123: UNKNOWN OPTION FOR QUIT

MEANING: This error occurs because LIB386 does not understand the response to the query of the QUIT command.

EFFECT: The command is not executed.

ACTION: Check the command syntax, and then reinvoke.

`ERROR 124: TARGET FILE IS NOT A LIBRARY`

MEANING: This error occurs because the file specified in the GET command is a 386 object file, but it is not a library. LIB386 can output only libraries.

EFFECT: The command is not executed.

ACTION: Specify a valid, existing library file or specify a name for the new library file, and then reinvoke.

`WARNING 125: TARGET FILE IS WRITE-PROTECTED`

MEANING: This warning occurs because the target library file specified with the GET command is write-protected.

EFFECT: The session proceeds and you can inspect the library, but an attempt to update the target library file will cause an error.

ACTION: To make updates, write-enable the target library file.

`ERROR 126: TARGET FILE IS A 286 LIBRARY`
`FILE: `*`filename`*

MEANING: An attempt was made to use GET to call a 80286 library.

EFFECT: The command is not executed.

ACTION: Specify a valid, existing Intel386 library file or specify a name for a new library file, and then reinvoke.

`ERROR 127: ATTEMPT TO LIST TO TARGET FILE`

MEANING: This error occurs because the LIST TO *filename* command is issued and *filename* is the name of the target library file.

EFFECT: The command is not executed.

ACTION: Choose another filename for the LIST TO command, and then reinvoke.

`ERROR 128: VERSION MAY BE UP TO 4 CHARACTERS IN LENGTH`

MEANING: This error occurs because too many characters have been specified for the version number.

EFFECT: The version of the library is not updated.

ACTION: Respecify with a version of up to four characters (e.g., V1.0), and then reinvoke.

ERROR 129: INTERACTIVE SET IS UNAVAILABLE IN BATCH MODE

MEANING: An interactive SET (i.e., SET without any additional parameters) was specified in batch mode.

EFFECT: The command is not executed.

ACTION: In batch mode, specify all items you wish to set with their new values, using the SET command.

System Interface Messages

These errors may be I/O errors, illegal file name problems, or other problems related to the host operating system. These error messages have the following format:

```
*** SYSTEM INTERFACE ERROR
UDI error code and text
FILE: filename
```

Refer to the operating system reference manual for the cause of the problem.

❑❑❑

MAP386 Error Messages C

MAP386 issues a message when it encounters one of the following types of errors in the invocation line or a control file:

- System interface level errors
- Semantic and object-file errors
- Internal processing errors

This appendix provides up to three types of information for each semantic and object-file error:

- MEANING: how to interpret the message
- EFFECT: the status of MAP386
- ACTION: suggestions for correcting the condition

Messages are listed in numerical order.

System Interface Level Errors

If an error in a call to the host operating system is detected, MAP386 issues a fatal error message to the console file, to the file created by ERRORPRINT (if open), and to the print file (if open). The error has the following format:

```
SYSTEM INTERFACE ERROR
error text
FILE: filename
```

The error text is operating-system dependent. The file name is present only if the error is an I/O error.

Semantic and Object-File Errors

Messages appear in the print file and any ERRORPRINT file, if specified. Fatal error messages also appear at the console.

`ERROR 100: INPUT FILE MISSING`

MEANING: This fatal error occurs because no input file is provided in the invocation line or control file command-tail.

EFFECT: Processing aborts, and control is returned to the operating system.

ACTION: Reinvoke, specifying the filename of an input file.

`ERROR 101: FILENAME TOO LONG`

`NEAR:` *`token string`*

MEANING: This fatal error occurs because the indicated filename is too long.

EFFECT: Processing aborts, and control is returned to the operating system.

ACTION: Reinvoke, using a shortened *`filename`*.

`ERROR 102: MISSING LEFT PARENTHESIS`

`NEAR:` *`token string`*

MEANING: This fatal error occurs because a left parenthesis was expected, but was not found.

EFFECT: Processing aborts, and control is returned to the operating system.

ACTION: Reinvoke MAP386, using the expected left parenthesis.

`ERROR 103: MISSING RIGHT PARENTHESIS`

`NEAR:` *`token string`*

MEANING: This fatal error occurs because a right parenthesis was expected, but was not found.

EFFECT: Processing aborts, and control is returned to the operating system.

ACTION: Reinvoke, using the expected right parenthesis.

ERROR 105: FILE ALREADY SPECIFIED IN COMMAND TAIL

FILE: *filename*

MEANING: This fatal error occurs because the indicated *filename* was already specified.

EFFECT: Processing aborts, and control is returned to the operating system.

ACTION: Reinvoke, being sure that the indicated filename is specified only once.

ERROR 106: INVALID DELIMITER IN COMMAND TAIL

NEAR: *token string*

MEANING: This fatal error occurs because the indicated invalid delimiter was used in the invocation line or control file.

EFFECT: Processing aborts, and control is returned to the operating system.

ACTION: Reinvoke, using only valid delimiters.

ERROR 107: LINE TOO LONG IN CONTROL FILE

FILE: *filename*

MEANING: This fatal error occurs because the control file named by *filename* contains a line that is too long. The maximum length is 128 characters.

EFFECT: Processing aborts, and control is returned to the operating system.

ACTION: Prepare the control file again, shortening the indicated line. Then reinvoke MAP 386 with CONTROLFILE, using the corrected filename of the control file.

ERROR 108: TOKEN TOO LONG

NEAR: *token string*

MEANING: This fatal error occurs because the indicated token is too long.

EFFECT: Processing aborts, and control is returned to the operating system.

ACTION: Shorten the indicated token to the proper length, then reinvoke MAP386.

ERROR 109: UNKNOWN CONTROL IN COMMAND TAIL

NEAR: *token string*

MEANING: This fatal error occurs because the indicated control is invalid.

EFFECT: Processing aborts, and control is returned to the operating system.

ACTION: Reexamine the input, being sure that only valid controls (as described in Chapter 4) are used, then reinvoke MAP386.

ERROR 110: SYNTAX ERROR

NEAR: *token string*

MEANING: This fatal error occurs because a mistake or typographical error in syntax was made.

EFFECT: Processing aborts, and control is returned to the operating system.

ACTION: Reexamine your input, being sure that only correct syntax is used, then reinvoke MAP386.

ERROR 112: NUMBER OF SYMBOLS EXCEEDS INTERNAL LIMIT

MEANING: This fatal error occurs because the available memory space is not sufficient for the number of symbols used.

EFFECT: Processing aborts, and control is returned to the operating system.

ACTION: Reduce the number of symbols used if you still want a cross-reference map, then reinvoke MAP386.

ERROR 114: INVALID OBJECT FILE

FILE: *filename*

MODULE: *mod_name*

MEANING: This fatal error occurs because the object file module in the input object file is invalid or is specified incorrectly.

EFFECT: Processing aborts, and control is returned to the operating system.

ACTION: Reconsider your input, trying to determine in what way the object file is invalid. Revise, then reinvoke MAP386.

ERROR 116: NESTED CONTROL FILES

FILE: *filename*

MEANING: This fatal error occurs because the control file specifies control file.

EFFECT: Processing aborts, and control is returned to the operating system.

ACTION: Reconstruct your input, making sure your control file does not refer to a second control file. Reinvoke MAP386.

ERROR 121: PAGE WIDTH OUT OF RANGE

NEAR: *token string*

MEANING: This fatal error occurs because width is too large or small. The value must be between 72 and 132.

EFFECT: Processing aborts, and control is returned to the operating system.

ACTION: Specify a correct width value, then reinvoke MAP386.

ERROR 122: PAGE LENGTH OUT OF RANGE

NEAR: *token string*

MEANING: This fatal error occurs because page length is too large or small. The value must be between 10 and 65535.

EFFECT: Processing aborts, and control is returned to the operating system.

ACTION: Use a correct *length* value, then reinvoke MAP386.

ERROR 123: OSINFO FILE IS GREATER THAN 4K BYTES

FILE: *osinfo filename*

MEANING: This error occurs because the osinfo file is larger than 4K bytes.

EFFECT: Only the first 4K bytes of the osinfo file are written into the osinfo section.

ACTION: Make sure that you do not need any information that comes after the first 4K bytes.

`ERROR 124: NO MODULES IN INPUT FILE FOUND`

`FILE: filename`

MEANING: This fatal error occurs because the executable input file does not contain a module.

EFFECT: Processing aborts, and control is returned to the operating system.

ACTION: Correct the input file and reinvoke MAP386.

`WARNING 126: SYMBOL TYPE MISMATCH`

`FILE: filename`

`MODULE: mod_name`

`SYMBOL: symbol name`

MEANING: This warning occurs because two symbols of the same name have different types.

EFFECT: None, advisory information only.

ACTION: None.

`ERROR 128: INPUT CONTAINS DUPLICATE MODULES`

`FILE: filename`

`MODULE: mod_name`

MEANING: This error occurs because two modules of the same name occur in MAP386's input.

EFFECT: None, advisory information only.

ACTION: None.

`WARNING 129: EXPORTED OR UNRESOLVED EXTERNAL SYMBOLS`

MEANING: This warning occurs because external symbols found in MAP386's input have no corresponding public symbol definition or referred-to export modules from BLD386.

EFFECT: None, advisory information only.

ACTION: None.

WARNING 130: ASSIGNING UNKNOWN DESCRIPTOR NAMES

MEANING: This warning is issued because of unnamed descriptors in the input file. It is always issued for boot-loadable files, since there is no name information and all names are assigned by MAP386. For loadable files, it is issued if there is no name information in DESNAM.

EFFECT: None, advisory information only.

ACTION: None.

ERROR 131: OVERLAPPING DESCRIPTORS IN TABLE

MEANING: This error occurs because there is more than one descriptor assigned to a slot in a table.

EFFECT: The descriptor information in the map is incorrect.

ACTION: The build language contains an error. Correct the error and then reinvoke MAP386.

WARNING 132: NO DEBUG INFORMATION IN INPUT FILE

FILE: *filename*

MEANING: This warning is issued because there is no debug information and consequently no information for the cross-reference map.

EFFECT: There will be no information on symbols, publics or externals.

ACTION: If this information is required, run BND386 or BLD386 with the DEBUG control.

WARNING 133: MODULE NOT FOUND IN INPUT FILE

FILE: *filename*

MODULE: *mod_name*

MEANING: This warning is issued because the *mod_name* specified in the invocation line is not found in the input file.

EFFECT: None-advisory information only.

ACTION: None.

ERROR 134: MORE THAN ONE FILE FOUND WITH EXECUTABLE INPUT

FILE: *filename*

MEANING: This fatal error occurs because more than one executable file was specified in the invocation line.

EFFECT: Processing aborts, and control is returned to the operating system.

ACTION: Specify only one executable file in the invocation line, and then reinvoke MAP386.

ERROR 135: EXECUTABLE FILE FOUND IN INPUT FILE LIST

FILE: *filename*

MEANING: This fatal error occurs because an executable file was found after one or more nonexecutable files.

EFFECT: Processing aborts, and control is returned to the operating system.

ACTION: Correct the input file list, and then reinvoke MAP386.

ERROR 136: ERRONEOUS SYMBOL INFORMATION IN OBJECT MODULE

FILE: *filename*

MODULE: *mod_name*

MEANING: This error occurs because there is an error in symbol debug information, nesting mismatch or missing block end.

EFFECT: Processing continues, but there may be errors in the debug information.

ACTION: None.

WARNING 137: SYMBOL INFORMATION MISSING IN SOME MODULES

FILE: *filename*

MEANING: This warning is issued because there is no public or external symbol information in a linkable file.

EFFECT: None-advisory information only.

ACTION: None.

WARNING 138: PUBLIC INFORMATION MISSING IN SOME MODULES

FILE: *filename*

MEANING: This warning is issued because external symbols were encountered in the input to MAP386 with no corresponding public symbol definition. Also, no indication for unresolved external symbols was found. Public symbol information is probably missing in the input file.

EFFECT: None, advisory information only.

ACTION: If the information is required, find out whether BND386 was invoked with NODEBUG, or whether the information was purged.

Internal Processing Errors

Internal processing errors are unusual occurrences. You can not correct these errors. If such errors occur, you should report the problem to an RadiSys representative.

When internal processing errors occur, the format of the error message is as follows:

ERROR 3*xx*: INTERNAL PROCESSING ERROR, *message*

The following error messages are 3*xx* errors. The assigned numbers accompany the error message.

ERROR 300: IN OBJECT MODULE INTERFACE

ERROR 301: IN SYMBOL PROCESSING

ERROR 302: UNKNOWN ERROR

❑❑❑

Glossary

absolute address	the physical location at which code or data resides in memory; in protected virtual address mode, the Intel386 processor supports 32-bit absolute addresses.
absolute object code	code or data to which absolute addresses have been assigned.
access rights	the attributes that describe how a segment can be accessed by other segments; access rights for stack and data segments include read-only and read-write; access rights for code segments include execute-only, executable and readable, and conforming.
action level	the structural level at which the majority of LIB386 commands are available. This command level includes, for example, commands for adding, deleting, and replacing modules in the target library.
active module	when using the LIB386 utility, a module is designated as active if it is both logically and physically available.
ASM386	see Intel386 Macro Assembler.
attribute	an item defined by a descriptor: base address, limit, and access byte parameters; segment use that violates an attribute causes an exception or interrupt.
base address	the 32-bit address at which a segment starts.
based symbol	a symbol whose logical address is determined by a value residing at the address of another symbol.
binder	see Intel386 binder.
blank common	the combine-type applied to segments containing unnamed FORTRAN common blocks.
BLD386	see Intel386 System Builder.
BND386	see Intel386 binder.

bootloadable module	a module that contains absolute object code in a simple format, to expedite the loading of coldstart modules. BLD386 is the only Intel386 utility that can be used to create bootloadable modules.
builder	see Intel386 System Builder.
call gate	the gate used to transfer control to more privileged code.
combine-name	the symbolic name of a segment. During binding, segments with the same combine-name and compatible combine-types are combined to form a single segment.
combine-type	the attribute specifying how a segment is to be combined. No-combine, normal, stack, common, blank common, and debug are the acceptable combine-types.
common	the combine type associated with segments that contain named FORTRAN common blocks.
conforming segment	the segment that can be shared by programs that execute at different privilege levels without using gates.
control	any of several binder, librarian, or mapper options, each of which performs a specific function when used in the invocation line.
control file	the input file containing the file names of input files and/or controls.
control transfer descriptors	call gates, task gates, interrupt gates, and trap gates.
current privilege level	the privilege level of a task at a specific instant, indicated by the CS register's lower two bits and by its descriptor privilege level.
debug information	symbolic information used by debuggers. This information, which is housed in an object module, includes symbol names, line numbers, and public symbol information.
default value	the value assigned automatically if no value is specified.
descriptor	the eight-byte item that defines memory use in an Intel386 protected-virtual-address-mode system. Descriptors include segment descriptors and system control descriptors.
descriptor privilege level	the privilege level defined in the descriptor for a segment or in a gate. Also called DPL.

DPL	see descriptor privilege level.
dynamically loadable module	an executable object module created by BND386. The module can be loaded onto an Intel386-based system running under control of an iRMX operating system.
entry point	the code segment offset that represents the starting point for execution.
error condition	a condition that causes BND386, LIB386, or MAP386 to issue a warning or error message, or that causes BND386 or MAP386 to issue a fatal error message.
error message	a BND386, LIB386, or MAP386 message that flags a condition that may cause the output object module to be invalid.
executable and readable segment	a code segment that can be read and executed.
executable file	a file containing a loadable or bootloadable module.
executable module	a loadable module.
executable segment	a code segment.
execute-only segment	a code segment that can be executed but not read.
expand-down segment	a nonexecutable segment whose limit can be extended toward lower-order addresses at run time.
expand-up segment	a nonexecutable segment whose limit can be extended toward higher-order addresses at run time.
export file	a file created with the Intel386 System Builder containing system interface. See exportation.
exportation	information placed in a file system to be used when creating loadable modules; this process makes system interface available to application programs.
extension	a term used in DOS operating systems. The extension is a period and three letter acronym added to a file name to identify the type of file. An extension is added by the user, or, in certain cases, by a utility if no extension is specified.
external reference	a reference to a symbol, procedure, or location that must be defined as public in another module.

external symbol	an address imported by a linkable module. The address imported is that of a public symbol with the same symbolic name that occurs in a different module.
fatal error message	a BND386 or MAP386 message that indicates an error condition prohibiting the completion of processing. The result is that processing aborts.
file name	a generic term for *pathname*. The name of a file.
file-spec	see file name.
file-type	see extension.
gate	a descriptor used to mediate access to code at a higher privilege level (call gate), to code in a different task (task gate), or to interrupt service routines (interrupt or trap gates).
GDT	see global descriptor table.
generalized address	an address that has an internal name and an offset in the item specified by the internal name.
global descriptor table	a table that houses descriptors available to all tasks; this table can contain as many as 8190 descriptors of the following types segment descriptors, TSS descriptors, LDT descriptors, call gates, and task gates. Also called GDT.
IDT	see interrupt descriptor table.
illegal access	an attempted code or data access that causes an exception condition because it violates hardware protection mechanisms; protection is implemented by enforcing segment access rights, page access rights, and privilege and descriptor usage rules.
incremental linking	the process in which several linkable modules are merged into a single linkable module. The module that results from linking may be used as input to another stage of incremental linking or to final binding.
index	the portion of a selector that points to a location in a descriptor table.
initial level	the structural level at which a LIB386 session with a new target library can begin.
initialization information	register values that must be established before task execution can begin.

Intel386 binder	the Intel386 program development utility used to link modules and/or create loadable, single-task output modules. Also called BND386.
Intel386 librarian	the interactive Intel386 program development utility used to organize linkable modules into libraries and to allow existing libraries to be modified by adding, deleting, or replacing modules. Also called LIB386.
Intel386 macro assembler	the assembler used to produce linkable object modules executable on Intel386 and Intel387 processors in protected mode. Also called ASM386.
Intel386 mapper	the Intel386 program development utility used to generate intermodule cross-reference maps between public and external symbols. Also called MAP386.
Intel386 microprocessor	the Intel386, an advanced, 32-bit high-performance microprocessor optimized for multiple-user and multitask systems; the processor has built-in virtual memory support and memory protection for isolating memory space from task to task. The 80287 or the Intel387 Numeric Processor Extension provides high-speed floating-point capabilities.
Intel386 System Builder	the system configuration utility for Intel386 protected-mode systems.
Intel386 Utilities	the Intel386 binder, librarian, and mapper.
interactive	the execution mode in which input in accepted from the keyboard.
internal-name	a fixed-length name, having as scope a single module. Segments, externals, types, gates, and descriptors all have internal names.
interrupt descriptor table	the descriptor table that houses up to 256 interrupt, trap, and/or task gates. It is used to mediate access to routines that handle interrupt and exception conditions. Also called IDT.
interrupt gate	the descriptor that points to an interrupt service routine, the use of which disables interrupts.
intersegment reference	a reference to a location in a different segment from the segment that contains the reference.
intrasegment reference	a reference to a location in the same segment as the segment that contains the reference.

LDT	see local descriptor table.
LDT selector	a selector installed in a TSS that points to a particular LDT.
LIB386	see Intel386 librarian.
librarian	see Intel386 librarian.
library	an object library consisting of one or more sets of linkable modules. Object libraries are produced by LIB386 and reside in library files.
library command levels	the hierarchical structure of the LIB386 command set. The three levels are operating system level, initial command level, and action command level.
library file	a file that contains a collection of linkable object modules indexed by module names at least.
library name	the name of a library that resides in a library file.
library section	a portion of an object library containing a set of linkable modules and corresponding directories.
limit	the segment attribute that defines the offset of the last byte in the segment.
linkable file	a file containing a sequence of linkable modules. Through incremental linking, BND386 can combine linkable modules produced by translators into a single linkable module.
linkable modules	an object module created by Intel386 translators or by BND386 during incremental linking; a linkable module requires further processing before it can be executed.
linking	a process of BND386, in which segments from one or more linkable input modules are combined and references between them are resolved.
loadable file	a file containing a single loadable module. Loadable files are produced by BND386 or by the BLD386 System Builder. They are consumed by loaders, debuggers, and mappers.
loadable module	see dynamically loadable module. A module for loading onto a running system.
loader	a system utility that loads the user's program into the system's memory and initiates its execution.
load-time-expandable segment	a segment whose limit can be extended up or down at load time.

local descriptor table	a table that houses up to 8191 descriptors that can be private to a task; an LDT can contain only segment descriptors, task gates, and call gates.
local symbols	all symbols that are found in the SYMBOLS subsection of the DEBTXT section in an input object module.
logical segment	portions of object code that contain logically similar information created by translators.
MAP386	see Intel386 mapper.
mapper	see Intel386 mapper.
module name	the programmer-assigned name for a linkable module.
no-combine	the combine-type associated with code and data segments that BND386 does not combine; these include ASM386 segments that are not PUBLIC, and PL/M-286 segments compiled under the LARGE model of segmentation.
non-executable file	an object file in a non-loadable format. Files in the "linkable" or "library" format.
non-executable module	an object module in a non-loadable format. Modules in files in the "linkable" or "library" format.
non-interactive	the execution mode in which input is redirected from a command (batch) file.
non-sharable segment	a segment whose access attribute is read-write.
normal	the combine type associated with code or data segments that contain ASM386 PUBLIC information or PL/M-286 information not compiled under the LARGE model of segmentation.
object library	a set of library sections.
object module format	the structure of an object module.
offset	a byte address in a segment.
osinfo	the operating system information field in the object file.
page heading	information placed at the top of each page, including product name, optional title string, system date and time, page number, and several blank lines.
pathname	see file name.

physical segment	a contiguous piece of memory that cannot exceed 64 K bytes in length for USE16 attribute or 4 gigabytes for USE32 attribute.
PL/M-286	the compiler used to generate 80286 modules from source programs written in the high-level block-structured PL/M-286 language. These programs can run on the 80286 as well as on the Intel386.
PL/M-386	the compiler used to generate Intel386 modules from source programs written in the high-level block-structured PL/M-386 language. These programs can run on the 80286 as well as on the Intel386.
privilege hierarchy	an aspect of the Intel386 protected-mode memory protection scheme that provides up to four different levels of access to segments.
privilege level	an attribute that ranges from 0 to 3 and controls the use of privileged instructions as well as access to descriptors and their segments; access in a protected-mode system uses three kinds of privilege levels current privilege level, descriptor privilege level, and requested privilege level.
privilege rules	rules that govern how and when a task can access a segment These rules employ the following parameters: the type of segment to be accessed, the instruction used, the type of descriptor used, the current privilege level, the requested privilege level, and the descriptor privilege level.
protected mode	see protected virtual address mode.
protected mode architecture	the Intel386 processor configuration that supports virtual memory addressing and protection.
protected virtual address mode	the Intel386 processor mode of operation that provides virtual memory addressing and memory protection; system data structures recognized by the processor are implemented in this mode.
protection	Intel386 protected-mode mechanisms ensuring that code and data segments are insulated from improper usage and that the critical CPU execution state control instructions are properly implemented.
public	(1) a symbol or procedure available for intersegment or intrasegment references; (2) a kind of ASM386 segment.

public definition	a public symbol's translator definition.
public symbol	an address exported by a linkable module. The address is imported by modules through use of an external symbol with matching symbolic name.
readable segment	a code segment that can be read.
read-only segment	a data segment that can be read only.
read-write segment	a data or stack segment that can be written to.
real address	an address that specifies an absolute location in memory: in Intel386 protected mode, the real address has 32 bits.
reference resolution	the process by which public definitions are paired with external references.
relocatable information	code or data whose location is defined at load- or run-time.
requested privilege level	the privilege level defined by a selector's two least-significant bits; it is used with the descriptor privilege level to establish the privilege levels a task can access.
RPL	see requested privilege level.
run time	the time of program execution.
section	a portion of an object module containing all information of a particular kind about that module; debug information, descriptors for segments and gates, program text (i.e., code and data), symbolic names for descriptors, program text, debug text, fix-ups, etc.
segment	see logical segment and physical segment.
segment attributes	see attributes.
segment base	the 32-bit address at which a segment begins.
segment descriptor	a descriptor referring to code, stack, and data segments in a program.
segment limit	the offset of the last byte in a segment.
segment map	the BND386 print file section that provides information for all segments in the output module.
selector	an index into a descriptor table; GDT and LDT selectors are 16-bit pointers that index the GDT and LDT.

sequence number	a decimal number indicating the sequence of local descriptor tables in the GDT.
session	creating a new library, and adding, replacing, or deleting modules to/from a single target library file with the LIB386 Utility. Multiple sessions can occur sequentially in a single LIB386 invocation.
sharable segment	a segment whose access attribute is read-only, execute-only, or execute-read.
slot	a location in a descriptor table.
special system data segment descriptors	TSS and LDT descriptors.
standard output device	a generic term for *console*, used in DOS operating systems, and *:co:* in iRMX operating systems. The system display or print medium.
symbol	(1) a variable in a module; (2) an internal representation in BND386 of an object module entity.
stack	a combine type associated with stack segments.
system builder	see Intel386 System Builder.
system building	the configuration of a system, especially the selective definition of system data structures and tasks and the allocation of privilege among segments and descriptors.
system data structures	descriptor tables, segment and system descriptors, and task state segments.
system descriptors	special system data segment descriptors and control transfer descriptors.
system file	the file that contains the information an operating system must have for system calls to be possible; in the Intel386 Utilities, system files are linkable files created by Intel386 System Builder; they are called export files.
table indicator	a bit in a 16-bit selector that defines whether the selector points to the GDT or an LDT.
target library	the library file specified by LIB386 in the invocation line or specified in the latest GET library command.

task	a single sequence of execution. A task has an associated processor state and a well-defined address space that has specific access parameters. The processor state is defined by the contents of the TSS; the address space and access parameters are defined by descriptors.
task gate	a gate used to transfer control to another task; a task gate refers to a TSS.
task state segment	the special system segment that stores a task's initialization and restart values; the TSS saves a task's entire execution state, e.g., registers, address space, and a link to the previous task.
text	program code and data.
translator	an assembler or compiler.
trap gate	a descriptor that points to an interrupt service routine. A trap gate does not disable interrupts.
TSS	see task state segment.
TSS descriptor	a descriptor that defines and points to a TSS.
USE16	a segment with 32-bit attribute off. The segment is limited to 64K bytes. For code segments, the default data width and address width is 16 bits.
USE32	a segment with 32-bit attribute on. The segment is limited to 4G bytes. For code segments, the default data width and address width is 32 bits.
virtual address	an address that contains a selector and an offset value.
warning message	a BND386, LIB386, or MAP386 message indicating that a user error may have occurred. The output object file is valid.
writable segment	a stack or data segment that can be written to.

❑❑❑

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

W

X

Z