iAPX 286 EVALUATION BUILDER USER'S GUIDE

Order Number: 121711-001

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's software license, or as defined in ASPR 7-104.9(a)(9).

No part of this document may be copied or reproduced in any form or by any means without the prior written consent of Intel Corporation.

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products:

BXP	Intel	Library Manager	Plug-A-Bubble
CREDIT	int_el	MCS	PROMPT
i	Intelevision	Megachassis	Promware
ICE	Intellec	Micromainframe	RMX/80
iCS	iRMX	Micromap	System 2000
i_m	iSBC	Multibus	UPI
Insite	iSBX	Multimodule	μScope

and the combination of ICE, iCS, iRMX, iSBC, iSBX, MCS, or RMX and a numerical suffix.

REV.	REVISION HISTORY	DATE
-001	Original issue.	9/81

PREFACE

This manual provides operating instructions for the iAPX 286 Evaluation Builder, an 8086-resident vehicle for building descriptor tables and binding relocatable segments of iAPX 286 programs to absolute addresses.

This manual is designed for users who are already familiar with:

- the architecture and features of the iAPX 286 microprocessor as described in the *iAPX 286 User's Manual.*
- the operation of the Series III development system as described in the *Intellec Series III Microcomputer Development System Console Operating Instructions* manual.

To use the iAPX 286 Evaluation Builder you need version V1.0 or later of the BD286E software component.

This manual contains the following chapters and appendices:

- Chapter 1, Overview of the iAPX 286 Evaluation Package, presents an overview of the Evaluation Package, describes the features and the functions of the individual components of the package, and concludes with a presentation of the iAPX 286 Development System.
- Chapter 2, The iAPX 286 Evaluation Builder, provides technical information necessary to use the Evaluation Builder to create gates, descriptor tables and the task state segment, bind addresses, and use the X286E run-time procedures. The listings and contents of the output module are also described.
- Chapter 3, Evaluation Builder Controls, describes the procedures required to invoke the Evaluation Builder and the controls that are provided to specify activities and output of the builder.
- Chapter 4, Evaluation Builder Language, describes the declarative language that allows you to specify initial system configuration and segment attributes and to define stacks, gates, and descriptor tables.
- Chapter 5, X286E Support Package, provides information needed to use the iAPX 286 Evaluation Package. It describes X286E conventions and initialization, descriptor and table management, task state segment management, and free space management.
- Appendix A, Error Messages, gives a listing of error and warning messages and recovery information.
- Appendix B, Status Codes, gives brief descriptions of the iAPX 286 status codes.

Related Publications

For information on the iAPX 286 microprocessor, see the following manual:

- *iAPX 286 User's Manual*, 121749

For information on the iAPX 286 Evaluation Package assembler and simulator, see the following manuals:

- *iAPX 286 Evaluation Macro Assembly Language Reference Manual*, 121708
- *iAPX 286 Evaluation Macro Assembler Operating Instructions*, 121709
- *iAPX 286 Evaluation Macro Assembly Language Pocket Reference*, 121710
- *iAPX 286 Evaluation Simulator Operating Instructions*, 121712

For information on the Series III development system, see the following manuals:

- *Intellec Series III Microcomputer Development System Product Overview*, 121575
- *A Guide to Intellec Series III Microcomputer Development Systems*, 121632
- *Intellec Series III Microcomputer Development System Console Operating Instructions*, 121609
- *ISIS-II CREDIT CRT-Based Text Editor User's Guide*, 9800902
- *Model 740 Hard Disk Subsystem Operation and Checkout*, 9800943

Notational Conventions

UPPERCASE	Characters shown in upper case must be entered in the order shown. You can enter the characters in uppercase or lowercase.
italics	Italics indicate variable information, such as *filename* or *address*.
[ ]	Brackets indicate optional arguments or parameters.
{ }	Braces indicate one and only one of the enclosed entries must be selected. If the field is also surrounded by brackets, the enclosed items are optional.
{ }...	Braces followed by ellipses indicate that at least one of the enclosed items must be selected. If the field is also surrounded by brackets, the enclosed items are optional. The items may be used in any order unless otherwise noted.
...	Ellipses indicate that the preceding argument or parameter may be repeated.
input lines	Examples of user input are printed in white on black to differentiate user entry from system output.
<cr>	The characters "cr" enclosed in angle brackets indicates the RETURN key. Do not enter the angle brackets or the characters "cr".
punctuation	Any other punctuation besides those described above must be entered as shown. For example, all of the punctuation shown in the following commands must be entered:

```
Table GDT(LIMIT = 480, ENTRY = (seg1,seg2));

SEGMENT stack_seg0 (LEVEL = 0),
        stack_seg1 (LEVEL = 1),
        stack_seg2 (LEVEL = 2);
```

CONTENTS

TABLES

ILLUSTRATIONS

CHAPTER 1
OVERVIEW OF THE iAPX 286 EVALUATION PACKAGE

This chapter presents an overview of the iAPX 286 Evaluation Package. It briefly describes the product and specifies its components: an assembler, a builder, and a simulator. It also provides an overview of these components and concludes with a brief presentation of the iAPX 286 Development Package, a future software product.

Introduction

The iAPX 286 Evaluation Package is a software product designed to allow system developers to evaluate the iAPX 286 microprocessor. The Evaluation Package enables you to evaluate the microprocessor's architecture and features such as the instruction set, segmentation, processor timing, and memory mapping and protection. You can experiment with the operating system functions and write and test simple systems that you can later port to the iAPX 286 microprocessor with minor modification. This software product also enables you to begin the development of more complex iAPX 286 programs and operating system nuclei for later execution on the iAPX 286 microprocessor.

This product consists of an assembler, a system builder utility, an iAPX 286 simulator, and a set of run-time support procedures. Table 1-1 shows the software components contained in the Evaluation Package, the mnemonics by which they are called, and the names of the associated files that contain the software components.

Table 1-1. iAPX 286 Evaluation Package

Component	Mnemonic	Filename
iAPX 286 Evaluation Macro Assembler	AS286E	AS286E.86
iAPX 286 Evaluation Builder	BD286E	BD286E.86
iAPX 286 Evaluation Simulator	SM286E	SM286E.86
X286E Run-Time Procedures	X286E	BD286E.INC
X286E Source File Declarations		X286E.INC

The following paragraphs provide descriptions of the components of the package.

Assembler Overview

The iAPX 286 Evaluation Macro Assembler accepts a main program module written in assembly language as input, and produces an object file acceptable to the iAPX 286 Evaluation Builder and a program listing. The assembly language program should contain code, data, and stack segments for an initial task, and may also contain additional code, data, and stack segments for tasks to be established at run-time. The program may reference external system functions supported by the iAPX 286 Evaluation Simulator as well as the external X286E procedures, but must otherwise be a completely self-contained module.

AS286E has been produced from the Series III ASM86 to promote portability of programs from the iAPX 86 to the iAPX 286. AS286E supports the full iAPX 286 instruction set, which is a superset of the instructions found in ASM86. Some of the ASM86 directives have been eliminated or modified to suit the iAPX 286. For example, the segment statement has been changed. However, many of the ASM86 features, such as the instruction set syntax, operand typing, structures, and macros, have been carried over to AS286E unchanged.

Builder Overview

The iAPX 286 Evaluation Builder accepts three input files: an object file produced by AS286E, a file containing builder commands, and the X286E run-time procedures file (BD286E.INC). The Evaluation Builder produces an object file acceptable to the iAPX 286 Simulator and a listing containing: a copy of the command file, a map describing the contents of the descriptor tables created and the addresses of segments, public symbols, a symbol table, and error messages.

The builder command language provides the mechanism for building a single task. Descriptor tables and the task state segment are created. To evaluate the protection mechanism, attributes and privilege levels may be assigned to data and executable segments, and gates to public procedures may be created.

BD286E assigns physical addresses to all segments including the descriptor tables and the task state segment and resolves all inter-segment references. BD286E also provides the linkage to the X286E run-time procedures and the system functions supported by SM286E.

Simulator Overview

The iAPX 286 Evaluation Simulator allows you to execute programs on a simulated iAPX 286 architecture, to observe instruction timings, debug symbolically and to examine and modify the contents of tables, registers, and memory. The simulator consists of three major parts - a loader, a simulator, and a monitor/debugger.

The simulator loader loads an executable file produced by the Evaluation Builder. It initializes the descriptor tables and the task state segment and loads the executable and data segments into a simulated iAPX 286 address space. The loader also sets up a table of debug information if the DEBUG control was used with AS286E.

The simulator portion simulates the iAPX286 processor and executes your program. It executes all supported instructions and enforces all protection rules. It verifies segment access types, offsets for segment boundaries, and privileged instructions and gating. It handles all processor traps and faults exactly as they are on the iAPX 286 processor.

The monitor/debugger provides functions for displaying or modifying the contents of registers, memory locations, symbol table, descriptor tables, and task state segment, all of which can be referenced symbolically. It displays instructions in disassembled form, and provides multiple breakpoints, instruction timing information, and an interval timer to generate pseudo-hardware interrupts. Instructions may be executed continuously or in step mode.

X286E Run-Time Procedures Overview

The iAPX 286 Evaluation Package supports a single module, single task created statically. The program module may contain code, data, and stacks for additional tasks, but the creation of additional task descriptors, task state segments, and local descriptor tables must be done at run-time. In other words, in multi-tasking programs the initial task in the system must contain code to set up and dispatch other tasks.

The X286E Run-Time Procedures and Source File Declarations are designed to assist a programmer with the creation of tasks, local descriptor tables, and segments. Procedures are also provided to manage descriptor tables and the system free memory space.

Using the Evaluation Package

There is a straight flow from assembler to builder to simulator with two associated interfaces. Figure 1-1 illustrates the use of these components for software development.

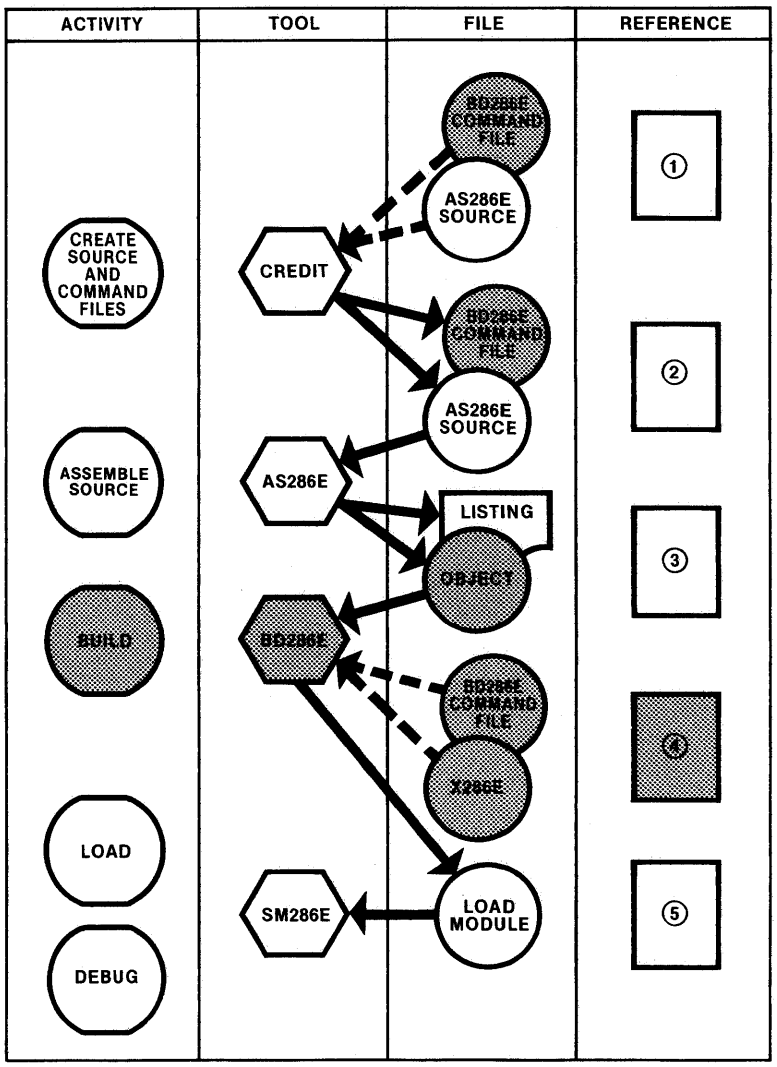
The builder accepts an object module from a single assembly. This module contains names identifying segments, public and external symbols, segment contents, and debugging information.

The simulator accepts a loadable file from the builder. This file contains a single-task program with all descriptor positions established in the LDT, GDT, and IDT. Physical addresses are assigned and descriptors are completely initialized. The file consists of descriptor tables and task state segment along with the contents of all segments in the program. Debug information if requested at assembly time will also be included.

iAPX 286 Development Package

The iAPX 286 Development Package is a future product that is to be designed for both systems and applications developers. With the Development Package, you will be able to develop and test large complex applications. The following additional capabilities are planned for the Development Package:

- High level languages
- Math Processor support
- A system binder for combining separately generated object modules
- A more complete System Builder, capable of creating multiple static tasks
- A librarian



① ISIS-II CREDIT CRT-Based Text Editor User's Guide
② iAPX 286 Evaluation Macro Assembly Language Reference Manual
③ iAPX 286 Evaluation Macro Assembler Operating Instructions
④ iAPX 286 Evaluation Builder User's Guide
⑤ iAPX 286 Evaluation Simulator Operating Instructions

121711-16

Figure 1-1 Software Development Using the iAPX 286 Evaluation Package

CHAPTER 2
THE iAPX 286 EVALUATION BUILDER

This chapter provides technical information necessary for using the iAPX Evaluation Builder (BD286E). It describes:

- Creation of gates, descriptor tables and the task state segment
- Address binding
- Selector resolution
- Use of the X286E run-time procedures
- Contents of the listing file:
  - Command File Listing
  - Map
  - Symbol Table
- Contents of the BD286E output module

Overview

The Evaluation Builder builds descriptor tables and binds relocatable segments to absolute addresses and also produces a load module containing absolute text and, optionally, debug information.

BD286E takes as input a file containing an object module produced by the iAPX 286 Evaluation Macro Assembler (AS286E).

The output from the Evaluation Builder consists of a loadable object module, an optional map, and an optional symbol table. The object module contains the memory image of an AS286E program, which can be loaded by the iAPX 286 Evaluation Simulator (SM286E).

The Evaluation Builder:

- Allows assignment of segment attributes and privilege levels.
- Allows the explicit creation of call gates for inter level transfers, and interrupt and trap gates for the Interrupt Descriptor Table (IDT).
- Creates the Global Descriptor Table (GDT), Interrupt Descriptor Table (IDT), and the Local Descriptor Table (LDT).
- Creates a Task State Segment for a single-LDT, single-task program.
- Allows you to control whether descriptor entries are placed in the GDT, LDT or IDT.
- Assigns absolute addresses to relocatable segments.
- Performs selector resolutions to segments and to a predefined gate for system functions.
- Creates gates for X286E run-time procedures contained in a predefined file (BD286E.INC) and performs selector resolutions to these gates.
- Produces a map summarizing the results of segment, gate, and public symbol processing and a symbol table showing the logical addresses of symbols produced by the assembler.
- Detects and lists errors found in the input module, the invocation line and the command file.

These functions are described in the following sections.

Assignment of Segment Attributes and Protection Levels

Using the command language specified in Chapter 4, you may specify a segment privilege level and change a non-conforming executable segment into a conforming one.

Gate Creation

By using Evaluation Builder commands, you create gates from public symbols declared in your assembly language program. You can specify gate type (call, interrupt or trap gate only) and gate privilege level. The default type is a call gate and the default privilege level is 3. While creating a gate, the Builder searches the public symbol table for a matching name. The Builder issues a warning if either the symbol can not be found or the symbol does not represent a gatable procedure in an executable segment.

If the symbol is found and it represents a gatable procedure entry, the Evaluation Builder builds a gate pointing to the public entry, sets the gate's word count equal to the entry word count, sets its type to the specified (or default) type, and sets its privilege level equal to the specified (or default) value.

Besides gates created under your control, the builder also creates a predefined gate for the system entry point called DQ_SIM and public gates for the X286E run-time procedures using the information found in a predefined file named BD286E.INC. These gates are all included in the GDT.

Descriptor Table and Task State Segment Creation

To specify entries in the GDT, LDT, or IDT table, you specify the table name (GDT, LDT, or IDT) and list the entry names. All entries not specifically assigned to the GDT or IDT will be placed in the LDT. You can include interrupt and trap gates only in the IDT table. The Evaluation Builder issues a warning if:

- You try to put an interrupt gate or a trap gate in either the GDT or the LDT.
- You try to include segment or call gate descriptors in the IDT.

You may specify table limits if you want extra blank entries. The Evaluation Builder issues a warning if the specified table becomes too large. By default, there are no extra entries in the IDT, 16 in the GDT, and 16 in the LDT.

The Evaluation Builder reserves the first 17 entries (entries 0-16 inclusive) in each of the descriptor tables for Intel use. If any of these entries are used, a warning is issued but the specified entry is used.

NOTE

The development package will reserve a larger number of entries for Intel use.

The following paragraphs describe the detailed structure of each table.

Global Descriptor Table

The Evaluation Builder builds the GDT according to the list of elements specified in your command. The Evaluation Builder issues a warning if a specified segment or gate does not exist. Besides the entries specified by you, the GDT also contains

preassigned entries, reserved entries, and some extra entries. The Evaluation Builder issues a warning if you use a preassigned entry or a reserved entry. The Evaluation Builder fills all uninitialized entries with zeros.

BD286E formats the GDT as follows:

Entry	Descriptor
0	All zeros
1	Writable data segment descriptor for GDT
2	Writable data segment descriptor for IDT
3	Table descriptor for LDT
4	Writable data segment descriptor for LDT
5	Gate descriptor for system routine DQ_SIM
6	Task State Segment descriptor
7-16	Reserved entries
17-n	Segment and gate descriptors for X286E run-time procedures. 'n' is determined by BD286E using information in the BD286E.INC file (if any)
	Segment and gate descriptors as specified by you (if any)
	Extra entries (if any)

Interrupt Descriptor Table

As with the GDT, the Evaluation Builder builds the IDT according to the list of elements you specify. The Evaluation Builder issues a warning if a specified gate does not exist. Besides the entries specified by you, the builder always includes entries 0 through 16 for interrupts 0-16 into the IDT (see *iAPX 286 User's Manual*). Some of these entries are reserved for simulator use. The Evaluation Builder fills all uninitialized entries with zeros.

Local Descriptor Table

You may specify the first part of the LDT. The builder creates the table by including the specified entries followed by all other segments and gates not already included in either the GDT or the IDT. LDT also contains a number of reserved entries as specified above.

BD286E formats the LDT as follows:

Entry	Descriptor
0	All zeros
1	Writable data segment descriptor for LDT
2-16	Reserved entries
17-n	Segment descriptors and gate descriptors (if any)
	Extra entries (if any)

Task State Segment

The builder creates the initial task state segment (TSS) only if you have initialized the input module by including register initialization values in the assembly language END statement. You must specify a stack segment for each protection level that contains code. The builder issues a warning and creates a corresponding invalid entry in the task state segment if:

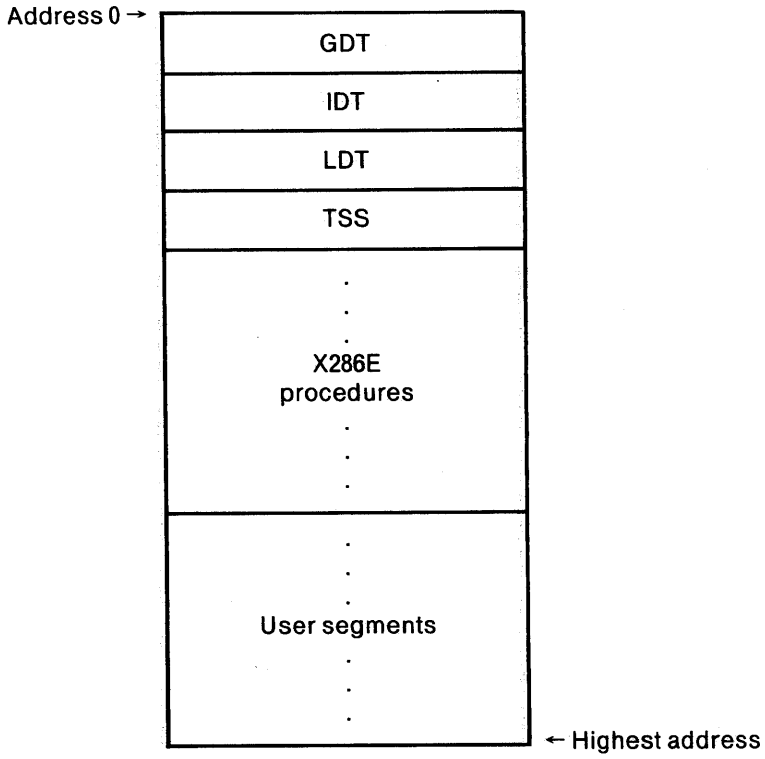
- the information for the initialization of a segment register (CS-IP, SS-SP, or DS) is not present in the object module, or
- you do not specify a stack segment for any of the privilege levels 0, 1, and 2 (the warning does not necessarily imply user error), or
- a specified segment for a stack is not a stack segment or
- the privilege level for CS is not equal to the privilege level of SS or
- the privilege level of CS is numerically greater than the privilege level of DS.

ES is initialized equal to DS, BP equal to SP. The link field is initialized to 0, the nested context flag to 0, the I/O privilege level to 1, interrupt enable flag to 1, all other flags to zero, and all other registers also equal to zero.

Address Binding

The Evaluation Builder assigns paragraph addresses to all segments (including tables and the task state segment) so that the Simulator can use the 8086 base registers as iAPX 286 base registers. The addresses are assigned starting at zero in the following order: GDT, IDT, LDT, TSS, X286E procedure segments (if any), and all user's segments.

The iAPX 286 memory is organized as follows:



Selector Resolutions

The Evaluation Builder provides selector resolutions for two types of program references:

- inter-segment references and
- external references to predefined publics such as the DQ_SIM system entry point and the X286E run-time procedures.

The builder sets the requested privilege level (RPL) of each reference equal to the descriptor privilege level (DPL) of the referencing segment.

Inter-Segment References

The builder determines the privilege level of the referencing segment (X) and the privilege level of the referenced segment (Y). Recall that in the iAPX 286, a segment X is said to be more privileged than a segment Y if the privilege level of X is numerically less than the privilege level of Y (see figure 2-1 Segment Reference Example). The builder resolves inter-segment references in the following manner.

- If Y is a data segment, then the reference to the data segment is replaced with the corresponding descriptor table selector. In addition, a warning is issued if Y is more privileged than X (see figure 2-1(d)).
- If Y is an executable segment, then:
  a. If X and Y are at the same privilege level (see figure 2-1(a)), the resolution is performed in the same manner as for a data segment.
  b. If X is more privileged than Y (see figure 2-1(b)), then the resolution is performed in the same manner as for a data segment and a warning is issued.
  c. If X is less privileged than Y and Y is a conforming segment (see figure 2-1(c)), then the resolution is performed as for a data segment.
  d. If X is less privileged than Y and Y is not a conforming segment (see figure 2-1(d)), BD286E searches its gate table for a matching selector, offset pair. If a gate is found, the segment reference is replaced with the corresponding gate selector. If a gate is not found, a warning is issued and the resolution is performed as for a data segment.

External References

When the Evaluation Builder encounters an external reference, it searches its gate table for a matching name. The gate table contains one predefined entry named DQ_SIM for service routines and a number of entries extracted from information contained in the X286E run-time procedures file (if any). If BD286E finds no matching name for the external reference, it issues a warning. If it finds a gate name, it replaces the external reference with the corresponding gate selector.

Use of the X286E Run-Time Procedure File

When building your program, the builder automatically searches for the file whose name is the same as the builder file name with the extension "INC". It builds the first part of the GDT and the first part of its gate table using information in that file. If the file does not exist, the GDT will be initialized to empty and no X286E procedure gates are included into the initial public table. This is the case when you do not want to include the X286E procedure file.

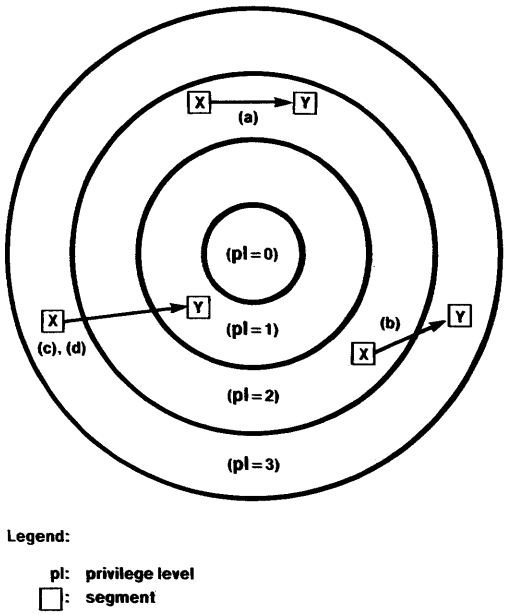


Figure 2-1. Segment Reference Example 121711-15

The following examples illustrate building programs with and without run-time procedures.

Example 2-1

```
RUN BD286E :F5:OSFILE.OBJ     ;Run-time procedures are included
```

The above example presumes that the run-time procedure file (BD286E.INC) is available to BD286E. The run-time procedures are included in the program.

Example 2-2

```
RENAME BD286E.INC TO X286E
RUN BD286E :F1:SIMPLE.OBJ     ;Run-time procedures not included
RENAME X286E TO BD286E.INC    ;Restore file name
```

The above example builds a program that does not include the run-time procedures.

Command File Listing

The builder outputs a copy of the command file to the listing file. This portion of the listing file contains the command lines preceded by line numbers, and error messages following the line (if any).

Map

BD286E outputs a segment map and a public table to the listing file.

For each segment, the listing shows the following:

- Owning descriptor table and table index
- Base address as a six digit hexadecimal number

- Limit as a four digit hexadecimal number
- Privilege level
- Access type
- Segment name

For each public symbol, the listing shows the following:

- Symbol name
- Type (gate or procedure)
- Owning descriptor table and table index (if a gate)
- Privilege level (if a gate)
- Word count (if any)
- Segment selector
- Segment offset as a four digit hexadecimal number

Symbol Table

The builder outputs a symbol table by default. The symbol table contains the symbol names defined in your assembly language program, one block per segment. The symbol listing contains the

- segment name
- list of alphabetized symbols declared in the segment along with their offsets in the segment.

Output Module

BD286E outputs a load module. Debug information is also output by default.

CHAPTER 3
EVALUATION BUILDER CONTROLS

This chapter provides information that you need to invoke the iAPX 286 Evaluation Builder (BD286E). The chapter also describes the controls that you can specify as part of the invocation to control the activities of the builder and the output generated by these controls. Finally, the chapter is concluded with three examples illustrating invocations of the Evaluation Builder.

Evaluation Builder Invocation

BD286E operates on a Series-III Microcomputer Development System. You can invoke the iAPX 286 Evaluation Builder in either of the two ways shown below. The syntax notation used to define invocation is given in the preface.

When the system is under the control of ISIS-II, you can invoke the Evaluation Builder by typing:

```
[:Fn:]RUN [:Fn:]BD286E[.86] commandtail <cr>
```

where

.86 — The specification '.86' is optional.

commandtail = [:Fn:]*infile* [TO [:Fn:]*outfile*][*controllist*]

:Fn: — references the directory of the disk in drive 'n' that contains the target file. The value is an integer between 0 and 9. If *:Fn:* is not specified, :F0: is assumed.

infile — the file that contains the iAPX 286 object module produced by the iAPX 286 Evaluation Assembler that is the program input to the Evaluation Builder and must be specified. The input file may reside on any direct access storage medium supported by the host operating system.

outfile — the file to be created by the Evaluation Builder for use by the simulator. *outfile* is determined in one of two ways: you can specify the outfile in the command tail with the TO token after the input file or it is determined by the Evaluation Builder to be the same name as the input file but with no extension. A fatal error results if the output file name is the same as the input file. If a file with the same name as the output file exists, it will be deleted and replaced with the specified output file unless protected at which time an error will be issued.

controllist — a specification of a set of controls for the activities of the Evaluation Builder. These controls are described later in this chapter.

When the system is already under the control of the RUN program, you can invoke the Evaluation Builder by typing:

```
[:Fn:]BD286E[.86] commandtail <cr>
```

The Evaluation Builder always signs on to the system console. The sign-on message is shown below:

```
<O.S. name> iAPX286 EVALUATION BUILDER, VX.Y
```

Where O.S. name is a string, if any, returned by the host operating system.

Completion of Evaluation Builder processing is indicated by the sign-off message sent to the system console. If there are no fatal errors, the builder will sign off as shown below:

```
PROCESSING COMPLETE, n WARNINGS
```

Where n is either the word 'NO' or the number of warnings found (for N = 1 the plural 'S' will be dropped). If a fatal error occurs, the sign-off message will be:

```
PROCESSING ABORTED
```

Controls

The user interface to the Evaluation Builder consists if a set of controls and a command language. The command language is described in the following chapter. The controls to the Evaluation Builder are contained in the controllist:

controllist = COMMAND [([:Fn:]*commandfile*)] | NOCOMMAND
DEBUG | NODEBUG
PRINT [([:Fn:]*listfile*) | (*:device:*)] | NOPRINT

Each of the keywords in the Evaluation Builder syntax has an abbreviated form. If the keyword has a prefix 'NO', then the abbreviated form also has the same prefix. The keywords and their abbreviated forms are listed below:

COMMAND	CM
DEBUG	DB
PRINT	PR

COMMAND and NOCOMMAND

You enter the COMMAND control followed by an optional pathname (e.g., "COMMAND(:F1:PROG.CMD)") to specify the file name where the builder will find all the commands for the creation of segment descriptors, gates, descriptor tables, and task state segment. Entering COMMAND without a pathname indicates that the command file is the file whose name and drive are the same as the input file but with the extension "CMD". You enter NOCOMMAND to specify that no command file is to be used. COMMAND, without a pathname, is the default.

DEBUG and NODEBUG

You enter the DEBUG control word to specify that the debug information such as symbol information, is to be output to the object file and the list file. You enter NODEBUG to suppress all debug output. DEBUG is the default.

PRINT and NOPRINT

You enter PRINT followed by an optional pathname (e.g., "PRINT (:LP:)") to direct the command file listing, the map, the symbol table, and warning messages to the indicated file. Entering PRINT without a pathname directs the above listings to the default list file whose name and drive are the same as the output file but with the extension "MP2". Entering NOPRINT suppresses all listing output. Fatal error messages always appear on the output console. PRINT, without a pathname, is the default. A fatal error will occur if the print file name is the same as the input, output, or command file name. The list file (if any) will contain the information shown below:

Command File Listing

A copy of the command file is output to the list file, with appropriate warnings (if any). The format of the command file listing is shown below:

```
COMMAND FILE: filename

   1          SEGMENT xxxxx(LEVEL=n), ... ;
   .                  ...
   .                  ...
   .                  ...       ;
  16          TABLE ... ;
  17          STACK ... ;
  18          END

COMMAND FILE PROCESSING COMPLETED
```

Where the first column indicates the command line number (for error reporting), and the last line indicates that the command file processing is complete.

Map

The map is output to the list file (if any). It contains information about segments and public symbols in the module.

The segment map lists information about each segment in the input module. The information consists of: the owning descriptor table and the segment index in that table, the base, the limit in bytes, the privilege level, the access type, and the combine name. The access type is given as a string of at most three characters (EO for executable only, ER for executable and readable, C for conforming, RO for read-only, RW for readable and writable, S for stack (expand down)). For example, 'ERC' indicates an executable readable conforming segment, 'EO' indicates an executable only segment, and 'S' indicates a writable stack segment. Segments are listed by descriptor tables and segment indices.

The public table lists all public symbols in alphabetical order. For each symbol the entry consists of: the symbolic name, the type (CALL or INTR or TRAP or PROC), the owning descriptor table and the gate index (if a gate), the privilege level (if a gate), the word count, the segment selector, and the offset. If the public symbol does not represent a gate, then the two fields TABLE and DPL will be filled with dashes. If the word count is greater than 31 or there is no word count, the word count field WC will be filled with dashes. For the special DQ_SIM symbol, the segment selector and the offset will not be listed.

The format of the map is shown below:

```
MODULE: module name

SEGMENT MAP

TABLE          BASE    LIMIT   DPL   ACCESS    COMBINE NAME

GDT(xxxx)   xxxxxxH   xxxxH    x      xxx     xxxxxxxxxxxx
GDT(xxxx)   xxxxxxH   xxxxH    x      xxx     xxxxxxxxxxxx
    .          .        .      .       .          .
    .          .        .      .       .          .
    .          .        .      .       .          .
LDT(xxxx)   xxxxxxH   xxxxH    x      xxx     xxxxxxxxxxxx

PUBLIC TABLE

SYMBOL NAME     TYPE  TABLE        DPL  WC  SELECTOR   OFFSET

xxxxxxxxxxxxxx xxxx  GDT(xxxx)     x   xx  GDT(xxxx)  xxxxH
xxxxxxxxxxxxxx xxxx  ---------    ---  --  LDT(xxxx)  xxxxH
       .         .        .       .    .       .         .
       .         .        .       .    .       .         .
       .         .        .       .    .       .         .
xxxxxxxxxxxxxx xxxx  LDT(xxxx)     x   xx  LDT(xxxx)  xxxxH
```

Symbol Table

The symbol table is output if the control DEBUG is set and if the input module contains debug information. For each segment defined in the program, the segment name is listed followed by information for symbols defined in the segment. Symbol information consists of the symbol offset and its symbolic name listed on the same line.

The format of the symbol table is shown below:

```
SYMBOL TABLE

SEGMENT: segment name

  OFFSET  SYMBOL                      OFFSET  SYMBOL

  xxxxH   xxxxxxxxxxxxxx              xxxxH   xxxxxxxxxxxxxx
  xxxxH   xxxxxxxxxxxxxx              xxxxH   xxxxxxxxxxxxxx
    .           .                       .           .
    .           .                       .           .
    .           .                       .           .
    .           .                       .           .
  xxxxH   xxxxxxxxxxxxxx              xxxxH   xxxxxxxxxxxxxx
```

Examples of Evaluation Builder Invocation

The following examples show some typical usage of the Evaluation Builder program under the Series III Operating System. Each example shows the invocation line as it would appear on the system console followed by the content of portions of the list file.

Example 1: NOCOMMAND control

```
-RUN BD286E MYPROG.OBJ NOCOMMAND

SERIES-III iAPX286 EVALUATION BUILDER, V1.0

  INPUT:  MYPROG.OBJ
  OUTPUT: MYPROG
  DATE:   03/19/81

CONTROLS SPECIFIED:
  NOCOMMAND

COMMAND FILE: (none)

MODULE: MYPROG

SEGMENT MAP

TABLE       BASE     LIMIT   DPL   ACCESS    COMBINE NAME

GDT(1)     000000H   0177H    0      RW       :GDT
GDT(2)     000180H   0087H    0      RW       :IDT
GDT(4)     000210H   0147H    0      RW       :LDT

LDT(1)     000210H   0147H    0      RW       :LDT
LDT(17)    000540H   0019H    3      RW       MYDATA
LDT(18)    000560H   0044H    3      ER       MYCODE

PUBLIC TABLE

SYMBOL NAME      TYPE   TABLE       DPL   WC   SELECTOR    OFFSET

ALLOCATE         PROC   -------     ---    2   LDT(18)     0010H
CREATE_SEGMENT   CALL   GDT(26)      0    10   GDT(17)     0060H
DELETE_SEGMENT   CALL   GDT(20)      0     5   GDT(17)     0020H
     .             .        .        .          .           .
```

This example shows the use of BD286E in the simplest way. There is no command file, the user does not specify any tables. All segments in the input module are at level 3. The entries in the GDT are created from the BD286E.INC file, and the LDT contains only entries for user's segments. The print file is MYPROG.MP2, and the output file is MYPROG. BD286E assigns absolute addresses to all segments including the GDT, IDT, and LDT segments. Call gates are obtained from BD286E.INC.

Example 2: PRINT and COMMAND controls

```
-RUN BD286E SAMPLE.OBJ PRINT(:LP:) COMMAND(MYCMD)

SERIES-III iAPX286 EVALUATION BUILDER, V1.0

 INPUT:  SAMPLE.OBJ
 OUTPUT: SAMPLE
 DATE:   03/19/81

CONTROLS SPECIFIED:
  PRINT(:LP:) COMMAND(MYCMD)

COMMAND FILE: MYCMD

   (command file listing)

COMMAND FILE PROCESSING COMPLETED

MODULE: SAMPLE

SEGMENT MAP

TABLE       BASE     LIMIT  DPL   ACCESS   COMBINE NAME

  .          .         .     .       .         .
LDT(17)    001000H  01FFH   3       RW       USER_DATA
LDT(18)    001200H  03FFH   3       ERC      USER_CODE
LDT(19)    FF1800H  FDFFH   3       S        STACK

PUBLIC TABLE

SYMBOL NAME       TYPE  TABLE       DPL  WC  SELECTOR   OFFSET

ERROR_HANDLER     INTR  IDT(10)      3    0  GDT(30)    0100H
GET_NEXT_ITEM     PROC  -------     ---   5  LDT(18)    0200H
PROCESS_SEGMENT   CALL  LDT(20)      3   10  LDT(18)    00E0H
     .             .      .         .    .     .          .
```

In this example the user specifies a command file (MYCMD in drive 0), and the listing is sent to the printer. The builder creates descriptor tables according to the specifications in the MYCMD file. For example an interrupt gate is defined in the command file using the public symbol ERROR_HANDLER, and a call gate is defined using the public symbol PROCESS_SEGMENT. The base and limit of the segment STACK have to be adjusted because the segment is an expand-down segment. If its length is 200H bytes and its top is at address 1800H, then the limit should be 64K-00200H -1 = FDFFH and the base should be 16M-64K + 1800H = 0FF1800H.

Example 3: Construction of an X286E file.

```
-RENAME BD286E.INC TO BD286E.SAV
-RUN BD286E OS.OBJ TO BD286E.INC COMMAND(OS.CMD)

SERIES-III iAPX286 EVALUATION BUILDER, V1.0

  INPUT:  OS.OBJ
  OUTPUT: BD286E.INC
  DATE:   03/19/81

CONTROLS SPECIFIED:
  COMMAND(OS.CMD)

***WARNING :104: MODULE NOT A MAIN MODULE
   MODULE: X286E

COMMAND FILE: OS.CMD

   1          segment
   2                    X286E_DATA  (level = 0),
   3                    X286E_CODE  (level = 0),
   4                    X286E_STACK (level = 0);
   5
   6          gate
   7                    CREATE_SEGMENT (level = 0),
   8                    DELETE_SEGMENT (level = 0);
   9
  10          table
  11                    GDT (limit = +0,
  12                         entry = (X286E_CODE,
  13                                  X286E_DATA,
  14                                  X286E_STACK,
  15                                  CREATE_SEGMENT,
  16                                  DELETE_SEGMENT))
  17          end

COMMAND FILE PROCESSING COMPLETED

MODULE:X286E

SEGMENT MAP

TABLE       BASE      LIMIT   DPL  ACCESS   COMBINE NAME

  .          .         .      .      .          .
GDT(17)     0002A0H   0069H   0      RW       X286E_CODE
GDT(18)     000310H   00FDH   0      ER       X286E_DATA
GDT(19)     FF0450H   FFBFH   0      S        X286E_STACK


PUBLIC TABLE

SYMBOL NAME      TYPE   TABLE       DPL  WC  SELECTOR   OFFSET

   .              .      .          .    .     .          .
CREATE_SEGMENT   CALL   GDT(20)      0   10   GDT(17)    0050H
DELETE_SEGMENT   CALL   GDT(21)      0    5   GDT(17)    0060H
    .             .      .          .    .     .          .
```

```
SYMBOL TABLE

SEGMENT:  X286E_CODE

  OFFSET  SYMBOL                                    OFFSET  SYMBOL

  0050H   CREATE_SEGMENT                              .       .
    .       .                                         .       .

SEGMENT:  X286E_DATA

  OFFSET  SYMBOL                                    OFFSET  SYMBOL
    .       .                                         .       .
```

This example gives a sample of the command file that can be used to create the X286E file needed by BD286E to resolve all external references to X286E run-time procedures. Only the GDT is defined with no extra entries. Segments are assigned privilege level 0, gates are assigned privilege level 0, and all segments and gates are included in the GDT. The output is directed to the file named BD286E.INC, the command file is OS.CMD, and (by default) the list file is BD286E.MP2.

CHAPTER 4
EVALUATION BUILDER LANGUAGE

This chapter describes the declarative language supported by the iAPX 286 Evaluation Builder. This language allows you to specify a simple initial system configuration. It allows you to specify attributes for segments and to define stacks for the initial task state segment, gates, and descriptor tables.

The syntax for an Evaluation Builder program is:

builder-program = *definition* [; ...][END]

where *definition* can be one or more of the following:

SEGMENT *segment-definition*[, ...]
GATE *gate-definition*[, ...]
TABLE *table-definition*[, ...]
STACK *stack-definition*[, ...]

All of the definitions but STACK accept keyword parameters. The definitions can be specified in any order but parameters can not be duplicated in the same definition and a descriptor can not be referenced before it is created. The optional END directive indicates the end of the command stream in a file. If this declaration is not present, the builder assumes the end of the program when it reaches the end of the command file.

Each definition in the above syntax format is described in the following paragraphs.

Segment Definition

To modify one or more segment descriptors, enter the keyword SEGMENT and a *segment-definition* for each segment. Each *segment-definition* is entered in the following format:

segment-name ({ LEVEL = *privilege-level* / CONFORMING | NOCONFORMING } [, ...])

where

segment-name — the symbolic name of a segment defined in the assembly language module

privilege-level — an allowable segment privilege level (0, 1, 2, or 3)

You modify segment descriptors by defining the privilege level and conformance attributes. If you specify the LEVEL parameter, you must also provide the privilege level value. No numeric value is allowed with the CONFORMING/NOCONFORMING parameter. By default, all attributes are taken from the input object file. Segment bases are set by the builder.

Example:

```
SEGMENT seg1 (LEVEL = 0, CONFORMING),
        seg2 (LEVEL = 1);
```

In the example segment 'seg1' has a privilege level 0 and is a conforming segment (it should be an executable segment). Segment 'seg2' has a privilege level 1. All other attributes are extracted from the input object file.

Gate Definition

To create one or more gates, enter the keyword GATE and a *gate-definition* for each gate. You enter each *gate-definition* in the following format:

gate-name [({ LEVEL = *privilege-level* / CALL | INTERRUPT | TRAP } [, ...])]

where

gate-name	the symbolic name of a public symbol
privilege-level	an allowable gate privilege level (0, 1, 2, or 3)

This declaration allows you to define gates and set their privilege levels and types. The gate name must be a public identifier for a procedure in an executable segment. The word count for all call gates are extracted from the input object file. No procedure type checking is performed for interrupt and trap gates. Note that task gate is not supported because the Evaluation Builder will only be used to create single-LDT, single-task programs. As with the segment definition, only values for the parameter LEVEL are required. By default, the privilege level of a gate is set to 3 and its type is a call gate.

Example:

```
GATE pub1 (LEVEL = 2, INTERRUPT),
     pub2;
```

In this example, two gates are defined: one is a level 2 interrupt gate and its entry point is the virtual address of the public procedure 'pub1', and the other is a level 3 call gate (both default values) and its entry point is the virtual address of 'pub2'.

Table Definition

To create one or more tables, enter the keyword TABLE and a *table-definition* for each table. You would enter each *table-definition* in the following format:

{ GDT / IDT / LDT } ({ LIMIT = [+] *num-of-entries* / ENTRY = ([*index:*] { *segment-name* / *gate-name* / *null-name* } [, ...]) } [, ...])

where

num-of-entries	the number of entries or extra entries in the table
index:	references a particular entry in the table
segment-name	the symbolic name assigned to identify a segment
gate-name	the symbolic name assigned to identify a gate
null-name	designates that no name has been assigned (blank entry)

The name of the table determines its type (GDT, IDT, and LDT are reserved words for the Evaluation Builder).

The LIMIT parameter allows you to specify the number of entries for the indicated table. The Evaluation Builder reserves the first 17 entries in each table. Using the '+' symbol enables you to specify the number of extra entries you desire in a given table. A LIMIT specification not containing the '+' symbol specifies the total number of entries for the table. The minimum number of entries allowed is 0. The maximum number of entries allowed is 8191. This allows for the dynamic construction of new entries. The size of the specified table is equal to

```
(8 bytes) * (number of entries
          + number of reserved entries
          + number of extra entries)
```

The ENTRY parameter allows you to specify the content of a table. An optional index may be used. The selector index for each entry is assigned sequentially in the order that the entries are specified. If you specify an index for a given entry, that entry will be placed at the corresponding table location and all entries between the previously assigned index and specified index will not be allocated. If you do not provide a name for the entry (null name), the selector index for that entry will be skipped and the corresponding entry will be marked invalid. These entries can be used by calling the X286E run-time procedures. The minimum allowable index value is 1 and the maximum allowable value is 8191. Reserved entries will not be used unless specified using *index:*. In this event a warning will be issued.

Example:

```
TABLE GDT(LIMIT = +80, ENTRY = (seg1,seg2,pub1,pub2)),
      IDT(ENTRY = (5:pub3, 10:pub4, 20:pub5));
```

In the example the GDT contains descriptors for the segments 'seg1', 'seg2', and descriptors for gates 'pub1', 'pub2'. It has 80 extra entries. The IDT contains a gate descriptor for 'pub3' at entry 5, 'pub4' at entry 10, and 'pub5' at entry 20. Entries 0 through 4, 6 through 9, and 11 through 19 will be included in the IDT but will be marked invalid.

Stack Definition

To specify stack entries for the task state segment,enter the keyword STACK and a *stack-definition* for each stack. Each *stack-definition* is entered in the following format:

segment-name

where

segment-name is the name of a stack segment.

This declaration allows you to specify stack segments that will be used to initialize the task state segment. The segment selector value and privilege level are extracted from the segment table. The Evaluation Builder checks to determine if the specified segment is a stack segment of privilege level less than or equal to 2. Note that stack privilege should be previously defined using a SEGMENT definition.

Example

```
SEGMENT stack_seg0 (LEVEL = 0),
        stack_seg1 (LEVEL = 1),
        stack_seg2 (LEVEL = 2);
STACK stack_seg0, stack_seg1, stack_seg2;
```

In the above example the segments 'stack__seg0', 'stack__seg1', and 'stack__seg2' are assigned privilege levels 0, 1, and 2 respectively. The STACK definition indicates that these three segments are to be used to initialize the registers pairs (SS0,SP0), (SS1,SP1), and (SS2,SP2) in the initial task state segment.

Keywords and Their Abbreviated Forms

The keywords used in the Evaluation Builder language also have the abbreviated forms listed below:

CALL	CA
CONFORMING	CF
ENTRY	ET
GATE	GA
INTERRUPT	IT
LEVEL	LV
LIMIT	LM
NOCONFORMING	NOCF
SEGMENT	SM
STACK	ST
TABLE	TB
TRAP	TR

Sample Builder Program

The following example illustrates a simple Evaluation Builder program that you might enter.

```
SEGMENT seg1(LEVEL = 0, CONFORMING),
        seg2(LEVEL = 1),
        stack_seg0 (LEVEL = 0),
        stack_seg1 (LEVEL = 1),
        stack_seg2 (LEVEL = 2);
STACK stack_seg0, stack_seg1, stack_seg2;
GATE pub1 (LEVEL = 2),
     pub2,
     pub3(interrupt),
     pub4(interrupt),
     pub5(interrupt);
TABLE GDT(LIMIT = +80, ENTRY = (seg1,seg2,pub1,pub2)),
      IDT(ENTRY = (5:pub3, 10:pub4, 20:pub5))
END
```

CHAPTER 5
X286E SUPPORT PACKAGE

This chapter provides information needed to use the iAPX 286 Evaluation Package Run-Time Support Procedures (X286E). It describes:

- Product environment
- X286E conventions
- X286E initialization
- Descriptor and table management
- Task state segment management
- Segment management
- Free space management

The iAPX 286 Evaluation Package is intended to allow you to experiment with the iAPX 286 architecture, particularly its operating system features. The X286E package provides assistance in your use of these features. The X286E portion of the evaluation package consists of run-time procedures that can be linked with your program and a collection of assembler macro, structure and record definitions that you can include in your program.

X286E provides four classes of service:

- Descriptor and descriptor table management
- Task state segment management
- Segment management
- Free space management

The descriptor and descriptor table management procedures assist you to manage descriptor slots; to create, delete, and change descriptors; and to create new local descriptor tables. Task state segment management is an extension of descriptor management. It allows you to change a data segment into a task state segment. Segment management procedures allow you to create segments and allocate memory space and to free space occupied by segments. The free space management procedures are used to manage the pool of free memory. These procedures are used in the creation and deletion of segments and LDT's.

Product Use Environment

The X286E procedures are executed by the iAPX 286 Evaluation Simulator (SM286E). They are contained in the file BD286E.INC and are automatically linked into your program by the Evaluation Builder if the BD286E.INC file is found. The X286E macro, structure and record definitions are contained in the file X286E.INC, which may be included in your source file when assembling with the evaluation assembler (AS286E). The required hardware for use of the evaluation package is a Series III Microcomputer Development System.

Conventions

X286E applies three categories of conventions:

- Calling conventions
- Descriptor table conventions
- Privilege level conventions

Calling Conventions

The X286E procedures expect parameters to be placed on the stack, with pointer parameters requiring both a segment selector and offset part. With one exception, all of the parameters are either 16 bit words or 32 bit pointers. The 16 bit parameters are either integers or selectors. The one exception is the access rights byte parameter that is padded to 16 bits to keep the stack word aligned. These procedures restore only the CS, IP, DS, SS, SP, and BP registers before they return. All procedures return a status code in the AX register. A status code value of zero indicates a successful completion of the procedure. A non-zero status code value indicates that the procedure has failed and encodes the reason for the failure.

Many of the procedures allow manipulation of the descriptors in the tables that are not immediately accessible. You select such descriptors by using a 32 bit number consisting of a segment selector and a table selector. The table selector should either be null or be the selector of an LDT descriptor. If the table descriptor is null, the descriptor is taken to be either the GDT or the current LDT depending on the TI field of the segment selector.

Descriptor Table Conventions

The first two slots in the GDT (indices 1 and 2) and any LDT (indices 0 and 1) are reserved for use by X286E and must not be altered by your program. The descriptor immediately following an LDT descriptor must be an alias that describes the descriptor table as a writable data segment. Similarly, descriptor number one (the second descriptor in the LDTs and the first useable descriptor in the GDT) in each descriptor table must be a descriptor that describes the descriptor table as a writable data segment. To facilitate access to the IDT, GDT entry number two is dedicated as a data descriptor for the IDT. Note that the Evaluation Builder automatically sets up the proper data segment aliases for the tables (GDT, IDT, initial LDT) it constructs.

As X286E does not do extensive task state segment manipulation, aliases for the task state segment are not required. However, you may find such aliases useful for your own purposes, in which case you must create and manage them yourself.

Privilege Level Conventions

The X286E procedures all reside in one level 0 code segment. The gates that point to them also have privilege level 0 thus restricting access to X286E to level 0. This can be changed at run-time by using the put_descriptor and get_descriptor procedures to change the privilege levels in some or all of the gates.

X286E uses a level 0 stack, which must be provided by the user program. X286E will use at most 100 bytes on this stack per call. This means that the user's level 0 stack should be enlarged by 100 bytes to accommodate X286E.

X286E Initialization

You must call the X286E initialization procedure before any other X286E procedure can be called. This procedure initializes the free space table. The procedure is parameterless. It is called as follows:

```
CALL X286_INITIALIZE
```

The procedure always returns with a status code of zero indicating successful initialization.

Descriptor and Table Management

Descriptor and table management consists of a set of descriptor and table management procedures and a set of descriptor declarations. These provide only primitive services. Higher level services, such as alias management and synchronization of access to shared segments, are not provided.

Descriptor And Table Management Procedures

This section consists of a set of X286E procedures that assist you in managing descriptor table slots*, creating, deleting and changing descriptors, and in creating new local descriptor tables. The following descriptor and table management procedures are presented in this section:

- CREATE_LDT
- ALLOC_SLOTS
- FREE_SLOTS
- COPY_DESCRIPTOR
- MOVE_DESCRIPTOR
- GET_DESCRIPTOR
- PUT_DESCRIPTOR
- INSTALL_GATE
- GET_IDT_DESCRIPTOR
- PUT_IDT_DESCRIPTOR
- INSTALL_IDT_GATE

*The term *descriptor table slot* is used in this chapter to refer to the eight-byte field corresponding to a descriptor table entry. Thus, for example, a single *slot* is required to hold a segment descriptor.

CREATE_LDT

The CREATE_LDT procedure creates a new local descriptor table and allocates space for it.

Procedure Call Syntax

```
PUSH nslots
PUSH SEG copy_list
PUSH OFFSET copy_list
PUSH SEG move_list
PUSH OFFSET move_list
PUSH SEG LDT_sel
PUSH OFFSET LDT_sel
CALL CREATE_LDT
```

Input Parameters

nslots — a numeric value that specifies the number of table slots to be allocated to the new local descriptor table. If *nslots* equals zero, the new LDT is given the same number of descriptor slots as the current LDT. If *nslots* contains a non-zero value, the new LDT is given that number of descriptor slots.

copy_list — an array of selectors that designate the descriptors to be copied from the current LDT to the new LDT. The first word in the array must contain the number of descriptors to be copied. The remainder of the array is composed of the string of selectors to be copied.

move_list — An array of selectors that designate the descriptors to be moved to the new LDT and deleted from the current LDT. The first word in the array must contain the number of descriptors to be moved. The remainder of the array is composed of the string of selectors designating the descriptors to be moved.

Output Parameters

LDT_sel — the word in memory which receives the selector to the descriptor for the new LDT.

status_code — a status code is returned in AX. A zero value indicates a successful completion of the procedure. A non-zero value indicates that the procedure failed and encodes the reason for failure.

Description

This procedure creates a new local descriptor table and allocates memory for it. Two new slots are allocated in the GDT for storing the LDT descriptor and a data segment alias for the table. The number of descriptor slots to be contained in the new LDT is specified in *nslots*. If *nslots* contains a non-zero value, the new LDT will contain the number of slots specified in *nslots*. If *nslots* contains a zero value, the new LDT will contain the same number of descriptor slots as the current LDT.

In addition to creating a new LDT, the procedure can copy and/or move descriptors from the current LDT into the new LDT. The copy and move both cause parallel descriptor transfers. That is, in either case, the descriptor in slot n of the current LDT is transferred to slot n of the new LDT. In the case of *copy*, each specified descriptor is copied into the associated slot in the new LDT leaving the descriptor in

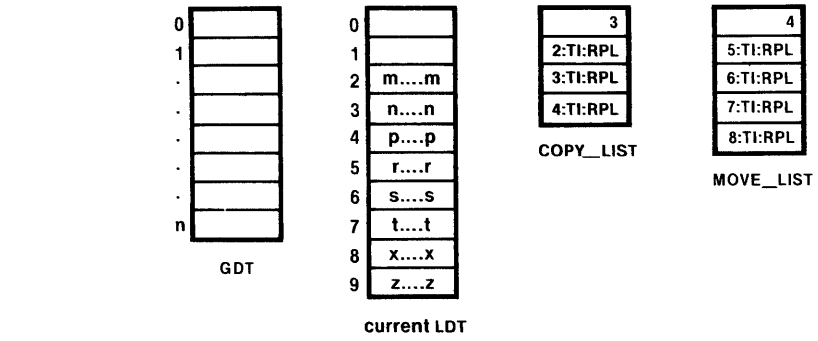
the current LDT unchanged. When descriptors are *moved*, the designated descriptors are moved from the current LDT to parallel slots in the new LDT. However, each of the designated descriptors is then deleted from the current LDT by zeroing its access rights byte. Slots in the new LDT not initialized by copy or move are zeroed out.

Example

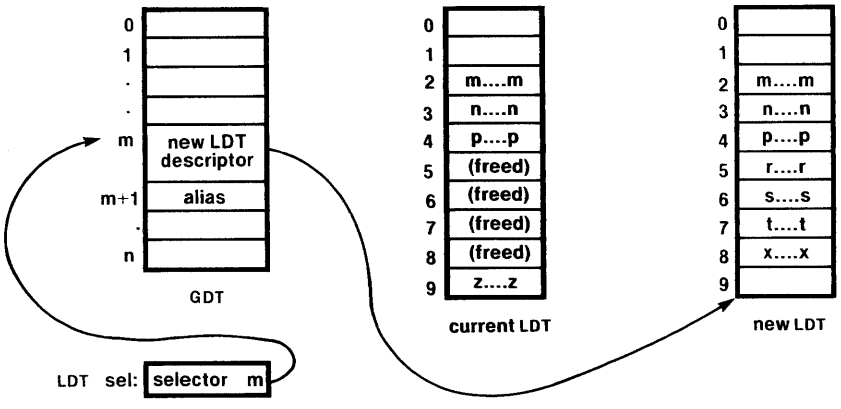
The calling sequence shown below is a typical example of the CREATE_LDT call. Figure 5-1 graphically illustrates call processing.

Call CREATE_LDT:

```
PUSH 0                          ;NEW LDT TO HAVE THE SAME SIZE AS
PUSH SEG COPY_LIST              ;THE CURRENT LDT.
PUSH OFFSET COPY_LIST
PUSH SEG MOVE_LIST
PUSH OFFSET MOVE_LIST
PUSH SEG LDT_SEL
PUSH OFFSET LDT_SEL
CALL CREATE_LDT
```



(a) Initial condition



(b) Final condition

Figure 5-1 CREATE_LDT Processing 121711-01

Status Codes	Meaning
0	Success
1	Insufficient space (for new LDT)
2	Selector from copy or move list is out of range*
3	Insufficient number of adjacent slots free in GDT (2 needed)
10	Invalid LDT size (*nslots* greater than 8K or equal to 1**)

* A status code of 2 indicates partial completion of CREATE_LDT. The LDT and its descriptors (in the GDT) are created and all copies and moves involving valid selectors are done. Copies or moves involving invalid selectors are ignored.

** Because slot 0 in an LDT is reserved, and slot 1 must be a data descriptor for the LDT, each LDT is required to have a minimum of two slots.

The ALLOC_SLOTS procedure allocates one or more consecutive slots in a descriptor table.

Procedure Call Syntax

```
PUSH nslots
PUSH table_sel
PUSH SEG new_sel
PUSH OFFSET new_sel
CALL ALLOC_SLOTS
```

Input Parameters

nslots — a numeric value that specifies the number of consecutive slots that are to be allocated in the designated descriptor table.

table_sel — the selector in the GDT that indexes the descriptor of the table that is to be allocated slots. If this value is null, the GDT itself is indicated. (Note that this is the only case where a null *table_sel* always indicates the GDT.)

Output Parameters

new_sel — the word in memory that receives the selector for the first slot allocated.

status_code — a status code is returned in AX. A zero value indicates a successfull completion of the procedure. A non-zero value indicates that the procedure failed and encodes the reason for failure.

Description

A free slot (available for allocation) is defined to be a table entry having a zero access rights byte. An allocated slot is marked by setting its present bit to 1.

ALLOC_SLOTS allocates one or more consecutive slots in a descriptor table. The parameter *nslots* specifies the number of slots to be allocated. The parameter *table_sel* contains the selector in the GDT that indexes the descriptor for the table to be allocated slots. If *table_sel* is null, the slots are to be allocated to the GDT.

The procedure stores the selector for the first slot allocated into *new_sel* and returns a status code in AX.

Example

The calling sequence shown below is a typical use of the ALLOC_SLOTS procedure. Figure 5-2 illustrates call processing.

```
PUSH 4                          ;ALLOCATE FOUR SLOTS
PUSH LDT_SEL                    ;SELECTOR FOR LDT
PUSH SEG SLOT_SEL               ;SLOT_SEL WILL INDEX
PUSH OFFSET SLOT_SEL            ;FIRST SLOT ALLOCATED.
CALL ALLOC_SLOTS
```

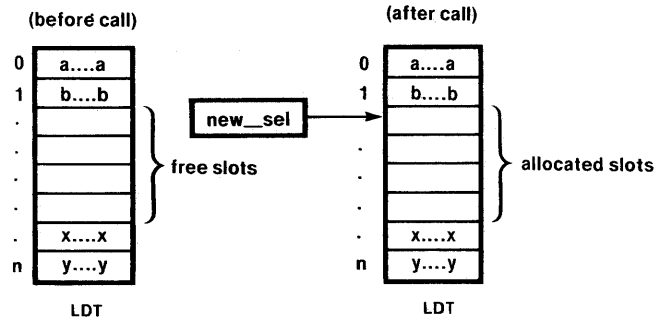


Figure 5-2. ALLOC_SLOTS Processing 121711-02

Status Codes	Meaning
0	Success
3	Insufficient (consecutive) free descriptor slots.
5	Invalid table selector

The FREE_SLOTS procedure frees one or more consecutive slots in a descriptor table.

Procedure Call Syntax

```
PUSH nslots
PUSH table_sel
PUSH sel
CALL FREE_SLOTS
```

Input Parameters

nslots — a numeric value that specifies the number of consecutive slots that are to be freed from the designated descriptor table.

table_sel — the selector in the GDT that indexes the descriptor of the table that contains the slots to be freed. If this value is null, either the GDT or the current LDT is indicated, depending on the TI bit in *sel*.

sel — the selector that indexes the first slot to be freed from the designated descriptor table.

Output Parameter

status_code — a status code is returned in AX. A zero value indicates a successful completion of the procedure. A non-zero value indicates that the procedure failed and encodes the reason for failure.

Description

FREE_SLOTS frees one or more consecutive slots in a descriptor table by zeroing the access rights byte for these slots. Parameter *nslots* specifies the number of slots to be freed from the table. The parameter *table_sel* designates the selector in the GDT that indexes the descriptor of the table containing the slots to be freed. If *table_sel* is null, the slots are freed from the GDT or the current LDT, depending on the TI bit of *sel*. The parameter *sel* is the selector that indexes the first slot in the table to be freed. Attempts to free slots beyond the end of a table are ignored.

Example

The calling sequence shown below is an example of a typical call to this procedure. Figure 5-3 illustrates FREE_SLOTS processing.

```
DEFAULT_TABLE   EQU 0

PUSH 2                  ;FREE TWO SLOTS
PUSH DEFAULT_TABLE      ;USE GDT OR CURRENT LDT
PUSH SLOT_NO            ;SELECTOR (IN GDT) FOR FIRST FREED SLOT
CALL FREE_SLOTS
```

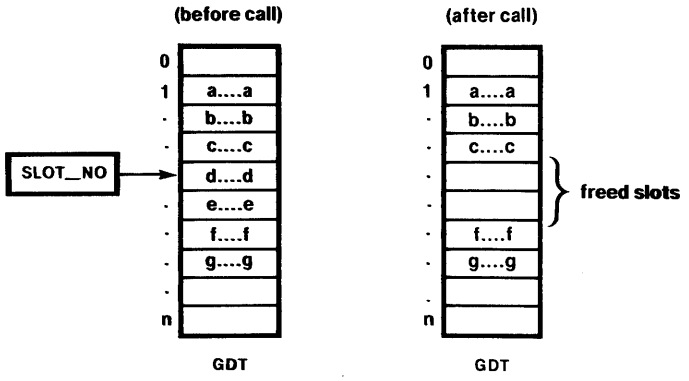


Figure 5-3. FREE_SLOTS Processing 121711-03

Status Codes	Meaning
0	Success
5	Invalid table selector

This procedure copies a descriptor from one descriptor slot to another.

Procedure Call Syntax

```
PUSH from_table
PUSH from_sel
PUSH to_table
PUSH to_sel
CALL COPY_DESCRIPTOR
```

Input Parameters

from_table the selector of the descriptor table containing the descriptor to be copied. If this value is null, then either the GDT or the current LDT is indicated, depending on the TI bit in *from_sel*.

from_sel the selector that indexes the slot containing the descriptor to be copied

to_table the selector of the descriptor table that the descriptor is to be copied to. If this value is null, then either the GDT or the current LDT is indicated, depending on the TI bit in *to_sel*.

to_sel the selector that indexes the slot that is to receive the copied descriptor

Output Parameter

status_code a status code is returned in AX. A zero value indicates a successful completion of the procedure. A non-zero value indicates that the procedure failed and encodes the reason for failure.

Description

The COPY_DESCRIPTOR procedure copies a descriptor from one descriptor slot to another. The descriptor slot in the *from_table* remains unchanged.

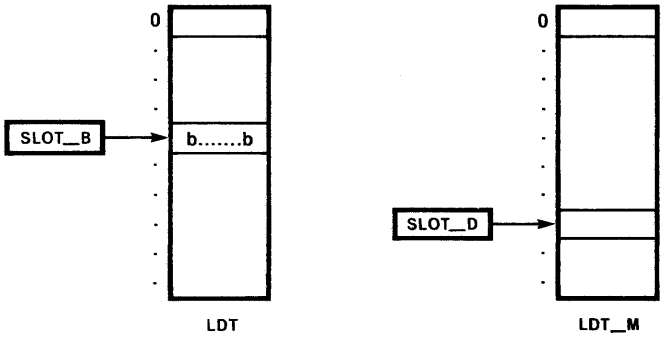
NOTE

No attempt is made to copy or check for aliases, even if the descriptor is an LDT. You must manage aliases (and perform any re-linking required).

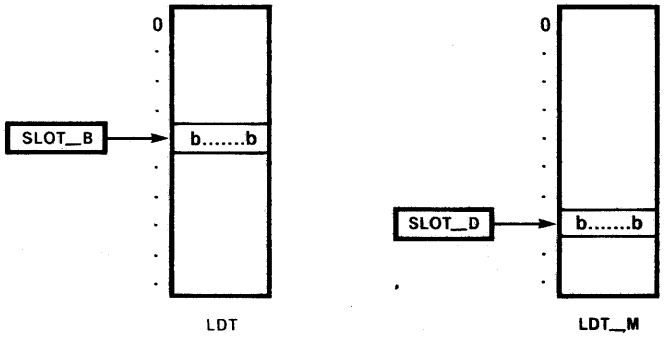
Example

This example is a typical call to COPY_DESCRIPTOR. Figure 5-4 illustrates COPY_DESCRIPTOR processing.

```
PUSH 0
PUSH SLOT_B
PUSH LDT_H
PUSH SLOT_D
CALL COPY_DESCRIPTOR
```



(a) Before CALL COPY_DESCRIPTOR



(b) After CALL COPY_DESCRIPTOR

Figure 5-4. COPY_DESCRIPTOR Processing 121711-04

Status Codes	Meaning
0	Success
2	Selector out of range
5	*from_table* or *to_table* does not select a table

This procedure moves a descriptor from one descriptor slot to another.

Procedure Call Syntax

```
PUSH from_table
PUSH from_sel
PUSH to_table
PUSH to_sel
CALL MOVE_DESCRIPTOR
```

Input Parameters

from_table — the selector of the descriptor table containing the descriptor to be moved. If this value is null, then either the GDT or the current LDT is indicated, depending on the TI bit in *from_sel*.

from_sel — the selector that indexes the slot containing the descriptor to be moved

to_table — the selector of the descriptor table to receive the descriptor. If this value is null, then either the GDT or the current LDT is indicated, depending on the TI bit in *to_sel*.

to_sel — the selector that indexes the slot that is to receive the descriptor.

Output Parameter

status_code — a status code is returned in AX. A zero value indicates a successful completion of the procedure. A non-zero value indicates that the procedure failed and encodes the reason for failure.

Description

The MOVE_DESCRIPTOR procedure moves a descriptor from one descriptor slot to another. The descriptor slot is then freed in the original table by zeroing its access rights byte.

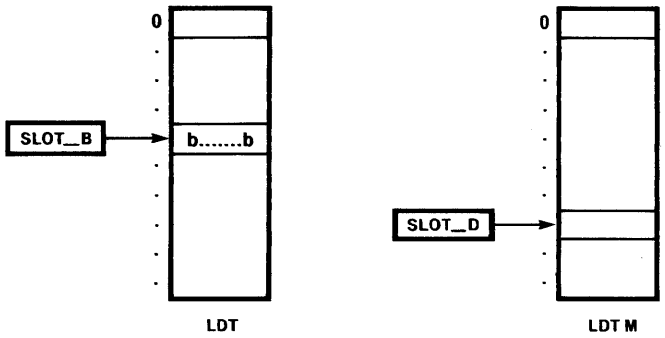
NOTE

No attempt is made to move or check for aliases, even if the descriptor is an LDT. You must manage aliases (and perform any re-linking required).

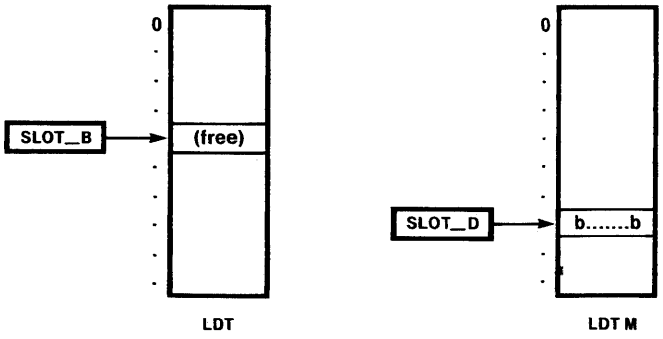
Example

This example is a typical call to MOVE_DESCRIPTOR. Figure 5-5 illustrates MOVE_DESCRIPTOR processing.

```
PUSH 0
PUSH SLOT_B
PUSH LDT_M
PUSH SLOT_D
CALL MOVE_DESCRIPTOR
```



(a) Before CALL MOVE_DESCRIPTOR



(b) After CALL MOVE_DESCRIPTOR

Figure 5-5. MOVE_DESCRIPTOR Processing 121711-05

Status Codes	Meaning
0	Success
2	Selector out of range
5	*from_table* or *to_table* does not select a table

The GET_DESCRIPTOR procedure copies a descriptor from a descriptor table to a data segment.

Procedure Call Syntax

```
PUSH from_table
PUSH from_sel
PUSH SEG buffer
PUSH OFFSET buffer
CALL GET_DESCRIPTOR
```

Input Parameters

from_table — the selector of the descriptor table containing the descriptor to be copied. If this value is null, then either the GDT or the current LDT is indicated, depending on the TI bit in *from_sel*

from_sel — the selector that indexes the descriptor to be copied

buffer — the eight bytes in the data segment to receive the copy of the descriptor

Output Parameter

status_code — a status code is returned in AX. A zero value indicates a successful completion of the procedure. A non-zero value indicates that the procedure failed and encodes the reason for failure.

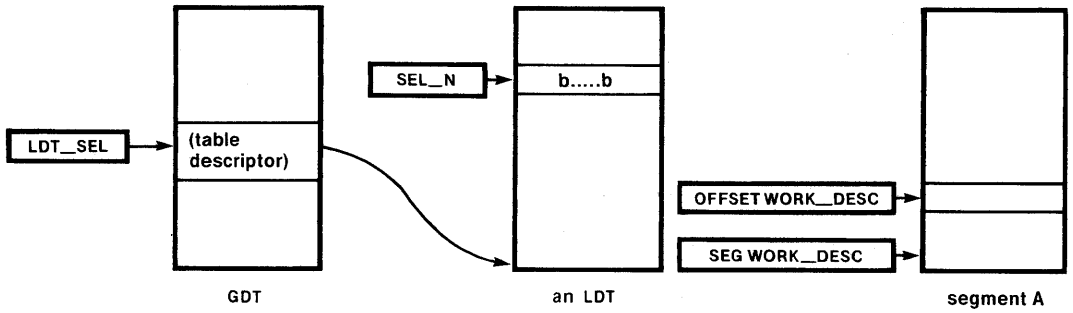
Description

This procedure copies a descriptor from a descriptor table to an eight-byte field in a data segment.

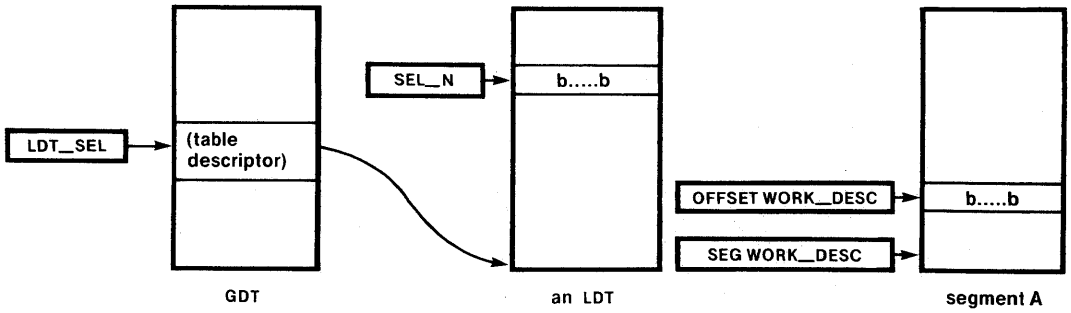
Example

The calling sequence shown below is a typical example of an application of GET_DESCRIPTOR. Figure 5-6 graphically illustrates the processing of this procedure.

```
PUSH LDT_SEL
PUSH SEL_N
PUSH SEG WORK_DESC
PUSH OFFSET WORK_DESC
CALL GET_DESCRIPTOR
```



(a) Before CALL GET_DESCRIPTOR



(b) After CALL GET_DESCRIPTOR

Figure 5-6. GET_DESCRIPTOR Processing 121711-06

Status Codes	Meaning
0	Success
2	Selector out of range
5	*from_table* does not select a table

The PUT_DESCRIPTOR procedure copies a descriptor from a data segment to a descriptor table slot.

Procedure Call Syntax

```
PUSH SEG buffer
PUSH OFFSET buffer
PUSH to_table
PUSH to_sel
CALL PUT_DESCRIPTOR
```

Input Parameters

buffer — eight bytes in the data segment containing the descriptor to be copied.

to_table — the selector of the descriptor table that is to receive the copied descriptor. If this value is null, then either the GDT or the current LDT is indicated, depending on the TI bit in *to_sel*.

to_sel — the selector that indexes the slot where the descriptor is to be copied

Output Parameter

status_code — a status code is returned in AX. A zero value indicates a successful completion of the procedure. A non-zero value indicates that the procedure failed and encodes the reason for failure.

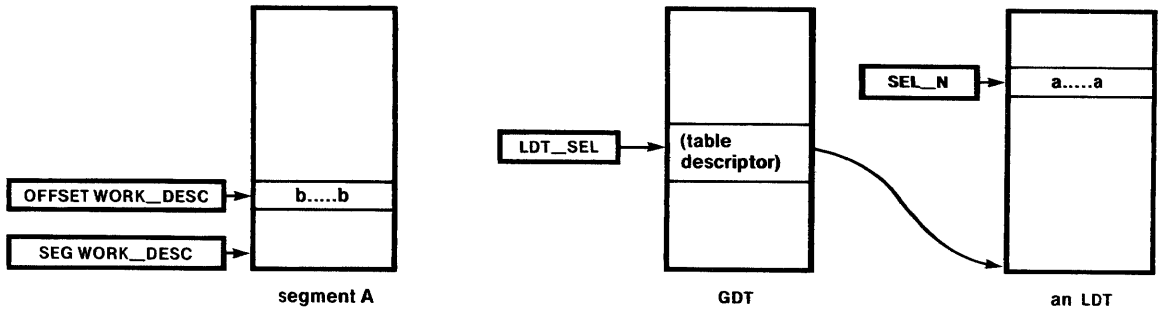
Description

This procedure copies a descriptor from a data segment to a descriptor table. The procedure overwrites whatever descriptor is in the selected slot in the descriptor table.

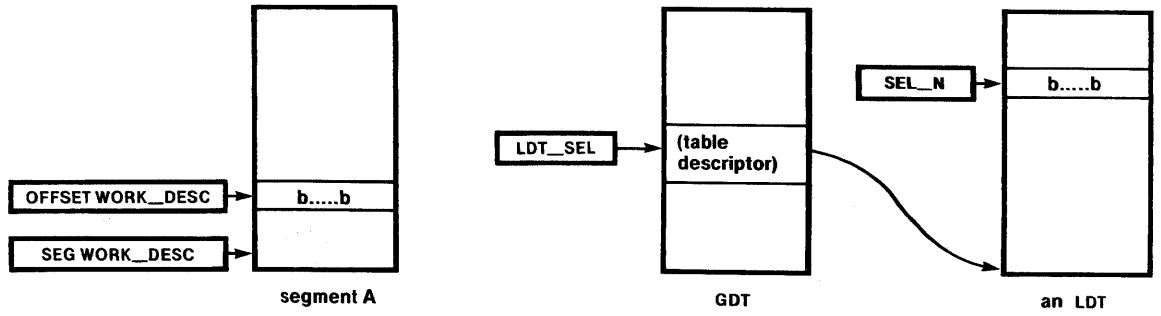
Example

The calling sequence shown below is a typical example of an application of PUT_DESCRIPTOR. Figure 5-7 graphically illustrates the processing of this procedure.

```
PUSH SEG WORK_DESC
PUSH OFFSET WORK_DESC
PUSH LDT_SEL
PUSH SEL_N
CALL PUT_DESCRIPTOR
```



(a) Before CALL PUT_DESCRIPTOR



(b) After CALL PUT_DESCRIPTOR

Figure 5-7. PUT_DESCRIPTOR Processing 121711-07

Status Codes	Meaning
0	Success
2	Selector out of range
5	*to_table* does not select a table

This procedure writes a gate descriptor into the GDT or an LDT.

Procedure Call Syntax

```
PUSH to_table
PUSH to_sel
PUSH WORD PTR gate_AR
PUSH nparm_words
PUSH SEG target
PUSH OFFSET target
CALL INSTALL_GATE
```

Input Parameters

to_table — the selector of the descriptor table that is to receive the gate descriptor. If this value is null, then either the GDT or the current LDT is indicated, depending on the TI bit in *to_sel*.

to_sel — the selector indexing the slot in the descriptor table where the gate descriptor is to be placed

gate_AR — the byte that is to be placed in the access rights byte of the descriptor

nparm_words — the number of parameter words for the procedure in the case of call gates (i.e., the value to be placed in the word count byte of the descriptor)

target — a FAR procedure or task state segment

Output Parameter

status_code — a status code is returned in AX. A zero value indicates a successful completion of the procedure. A non-zero value indicates that the procedure failed and encodes the reason for failure.

Description

The INSTALL_GATE procedure writes a gate descriptor into the GDT or an LDT. *to_table* and *to_sel* specify the table and descriptor slot into which the gate is to be placed. *gate_AR* is the byte that is to be placed in the access rights byte of the descriptor. It can be any value, this procedure does not check it. *nparm_words* is the number of parameter words for the procedure in the case of call gates. It must be specified for both call and task gates although it is ignored for the later case. *target* is a procedure or task state segment. If the gate is to be a task gate the pointer to *target* must still be a 32 bit pointer whose selector denotes a TSS. The offset portion will be written into the descriptor but not used by the iAPX 286.

Example

The calling sequence shown below is a typical example of a CALL INSTALL_GATE. Figure 5-8 graphically illustrates the processing of this procedure.

```
CGATE_A EQU 24H ;(SLOT 4 IN CURRENT LDT)
PCOUNT EQU  3

PUSH 0
PUSH CGATE_A
PUSH WORD PTR AR_BYTE
PUSH PCOUNT
PUSH SEG PROC_A
PUSH OFFSET PROC_A
CALL INSTALL_GATE
```

This call loads the call gate descriptor as shown below:

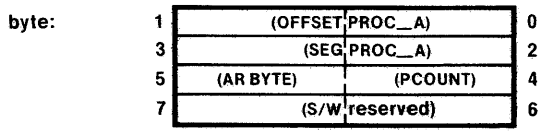


Figure 5-8. Call Gate Descriptor Content 121711-08

PUSH 0 causes the call gate descriptor to be loaded into the LDT in the slot indexed by CGATE_A.

Status Codes	Meaning
0	Success
2	Selector out of range
5	*to_table* does not select a table

GET_IDT_DESCRIPTOR

This procedure copies a descriptor from the IDT to a data segment.

Procedure Call Syntax

```
PUSH int_num
PUSH SEG buffer
PUSH OFFSET buffer
CALL GET_IDT_DESCRIPTOR
```

Input Parameters

int_num — the interrupt number (i.e., the slot index) in the IDT

buffer — eight bytes in memory to receive the copy of the descriptor from the IDT

Output Parameter

status_code — a status code is returned in AX. A zero value indicates a successful completion of the procedure. A non-zero value indicates that the procedure failed and encodes the reason for failure.

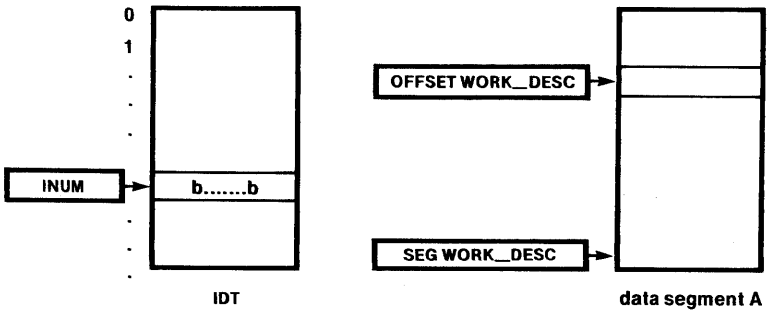
Description

This procedure copies a descriptor from the IDT to an eight-byte field in a data segment.

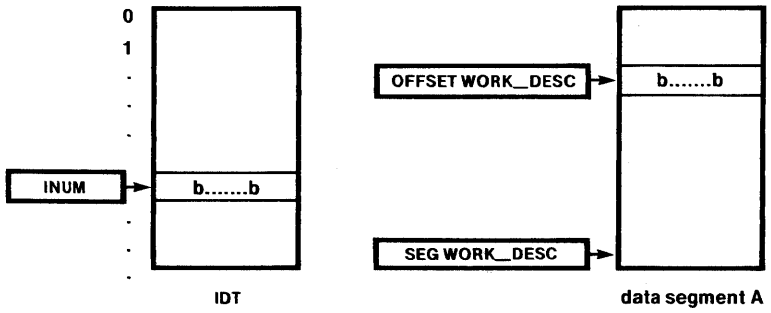
Example

The calling sequence shown below is a typical example of a GET_IDT_DESCRIPTOR call. Figure 5-9 graphically illustrates the results of this procedure.

```
PUSH INUM
PUSH SEG WORK_DESC
PUSH OFFSET WORK_DESC
CALL GET_IDT_DESCRIPTOR
```



(a) Before CALL GET_IDT_DESCRIPTOR



(b) After CALL GET_IDT_DESCRIPTOR

Figure 5-9. GET_IDT_DESCRIPTOR Processing 121711-09

Status Codes	Meaning
0	Success
6	*int_num* out of range

PUT_IDT_DESCRIPTOR

This procedure copies a descriptor from a data segment to the IDT.

Procedure Call Syntax

```
PUSH SEG buffer
PUSH OFFSET buffer
PUSH int_num
CALL PUT_IDT_DESCRIPTOR
```

Input Parameters

buffer — eight bytes in the data segment containing the descriptor to be copied

int_num — the interrupt number (i.e.,the slot index) of the IDT entry where the descriptor is to be placed

Output Parameter

status_code — a status code is returned in AX. A zero value indicates a successful completion of the procedure. A non-zero value indicates that the procedure failed and encodes the reason for failure.

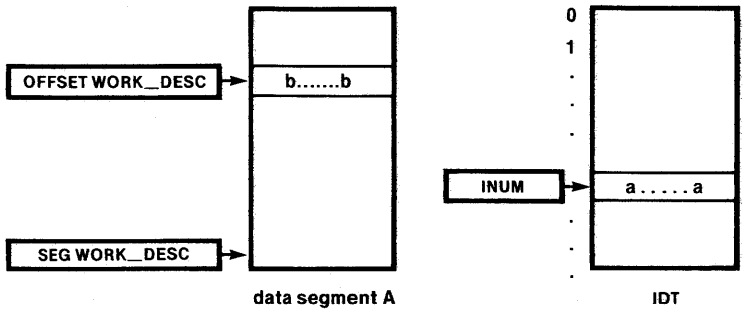
Description

This procedure copies a descriptor from an eight-byte field in a data segment to the IDT, overwriting the previous content of the slot specified.

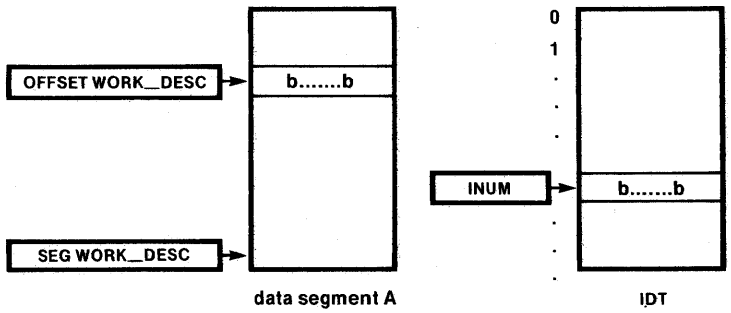
Example

The calling sequence shown below is a typical example of a PUT_IDT_DESCRIPTOR call. Figure 5-10 graphically illustrates the results of of this procedure.

```
PUSH SEG WORK_DESC
PUSH OFFSET WORK_DESC
PUSH INUM
CALL PUT_IDT_DESCRIPTOR
```



(a) Before CALL PUT_IDT_DESCRIPTOR



(b) After CALL PUT_IDT_DESCRIPTOR

Figure 5-10. PUT_IDT_DESCRIPTOR Processing 121711-10

Status Codes	Meaning
0	Success
6	*int_num* out of range

The INSTALL_IDT_GATE procedure writes a gate descriptor into the IDT.

Procedure Call Syntax

```
PUSH int_num
PUSH WORD PTR gate_AR
PUSH SEG interrupt_handler
PUSH OFFSET interrupt_handler
CALL INSTALL_IDT_GATE
```

Input Parameters

int_num	the interrupt number (i.e., the slot index) in the IDT
gate_AR	the byte to be placed in the access rights byte of the gate descriptor
interrupt_handler	an interrupt procedure or task

Output Parameter

status_code	a status code is returned in AX. A zero value indicates a successful completion of the procedure. A non-zero value indicates that the procedure failed and encodes the reason for failure.

Description

This procedure writes a gate descriptor into the IDT. *int_num* is the interrupt number which is also the index of the IDT slot that is to contain the gate descriptor. *gate_AR* is the byte that is to be placed in the access rights byte of the descriptor and defines the gate's access rights. This procedure does not check the access rights value. *interrupt_handler* is an interrupt procedure or task. If the gate is to be a task gate, the pointer to the interrupt_handler must still be a 32 bit pointer. The offset portion will be written into the gate descriptor but will not be used by the iAPX 286.

Example

The calling sequence shown below is a typical example of a INSTALL_IDT_GATE call. Figure 5-11 graphically illustrates the results of this procedure.

```
PUSH INUM
PUSH WORD PTR ARBYTE
PUSH SEG IH_PROC
PUSH OFFSET IH_PROC
CALL INSTALL_IDT_GATE
```

This call loads the interrupt call gates as shown below.

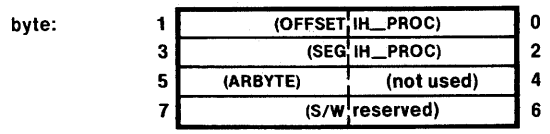


Figure 5-11. Interrupt Gate Descriptor Content 121711-11

PUSH INUM causes the above interrupt gate descriptor to be loaded in the IDT in the slot corresponding to interrupt number INUM.

Status Codes	Meaning
0	Success
6	*int_num* out of range

Descriptor Declarations

In addition to the descriptor and table management procedures, assembly language structure and record definitions are provided to assist you in coding routines that deal with descriptors and tables.

The DATA_AR record defines the access rights byte of a data segment. The record fields are identified by the following field names:

PRES	the present bit
DPL	the descriptor privilege level field
S	the segment bit (initialized to 1)
EXEC	the executable bit (initialized to 0)
EXPAND_DOWN	the expand down bit
WRITABLE	the writable bit
ACCESSED	the accessed bit

The CODE_AR record defines the access rights byte of a code segment. The record fields are identified by the following field names:

PRES	the present bit
DPL	the descriptor privilege level bit
S	the segment bit (initialized to 1)
EXEC	the executable bit (initialized to 1)
CONF	the conforming bit
READABLE	the readable bit
ACCESSED	the accessed bit

The CNTRL_AR record defines the access rights of control descriptors. The record fields are identified by the following field names:

PRES	the present bit
DPL	the descriptor privilege level field
S	the segment bit (initialized to 0)
CTYPE	the control descriptor type

Constants are defined for the control descriptor types. The names of the constants are:

TS_TYPE	the type number of a task state segment(1)
BUSY_TS_TYPE	the type number of a busy task state segment(3)
LDT_TYPE	the type number of a local descriptor table(2)
CALL_GATE_TYPE	the type number of a call gate(4)
INT_GATE_TYPE	the type number of an interrupt gate(6)
TRAP_GATE_TYPE	the type number of a trap gate(7)
TASK_GATE_TYPE	the type number of a task gate(5)

Structures are defined for segment descriptors and for gate descriptors.

The segment descriptor structure is named SEG_DESCR. The descriptor fields are identified by the following field names:

LIMIT	the segment limit field
ADDR_LOW	the least significant word of the base address
ADDR_HIGH	the most significant byte of the base address
AR	the access rights byte
SW_WORD	the software reserved word

The gate descriptor structure is named GATE_DESCR. The structure fields are identified by the following field names:

TARGET	the selector and offset of the gate target
NPARM_WORDS	the number of parameter words (for call gates)
AR	the access rights byte
SW_WORD	the software reserved word

A record named SELECTOR that describes a selector is also provided. The record fields are identified by the following field names:

INDEX	the index field of a selector
TI	the table indicator bit
RPL	the requested privilege level field

Task State Segment Management

Task state segment management is an extension of descriptor and descriptor table management. It allows you to change a data segment into a task state segment. This capability is provided through a procedure to turn a data segment into a task state segment, a macro that defines a task state segment, and a declaration for the task state segment structure.

The CREATE_TASK procedure enables you to create a task. This procedure creates the task by turning a data segment into a task state segment.

You can delete the task state segment with the DELETE_SPACE procedure.

Task state segment management is also supported at assembly time with a macro %DEFINE_TASK for creating a task state segment as a data segment and with a definition for a task state segment structure.

CREATE_TASK

The CREATE_TASK procedure changes a data segment into a task state segment.

Procedure Call Syntax

```
PUSH TS_sel
CALL CREATE_TASK
```

Input Parameter

TS_sel — the selector in the GDT for the data segment descriptor that is to be changed into a task state segment

Output Parameter

status_code — a status code is returned in AX. A zero value indicates a successful completion of the procedure. A non-zero value indicates that the procedure failed and encodes the reason for failure.

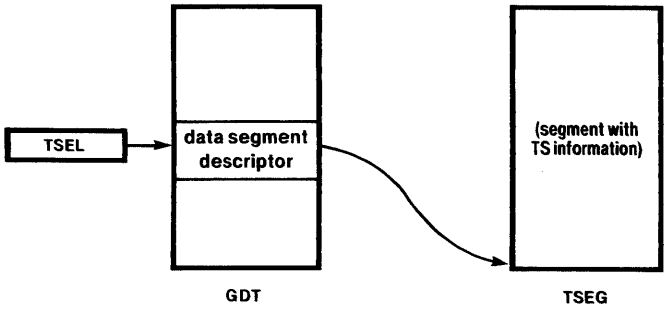
Description

This procedure changes a data segment into a task state segment by changing its descriptor to a task state segment descriptor. The data segment must have sufficient space for a task segment(at least 44 bytes).

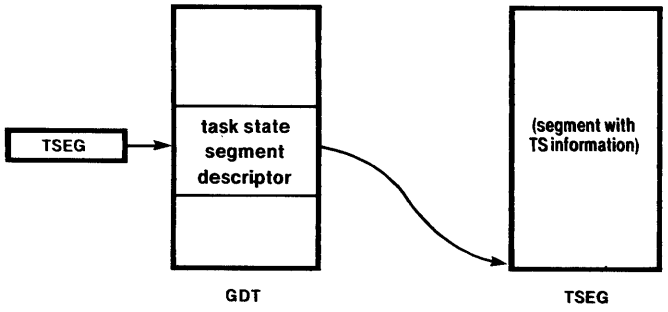
Example

```
PUSH TSEL
CALL CREATE_TASK
```

This call transforms data segment TSEG into a task state segment.



(a) Before CALL CREATE_TASK



(b) After CALL CREATE_TASK

Figure 5-12. CREATE_TASK Processing 121711-12

Status Codes	Meaning
0	Success
2	*TS_sel* is out of range
7	*TS_sel* does not select a data segment in the GDT, or the data segment is too small.

%DEFINE_TASK

%DEFINE_TASK is an AS286E macro for creating a data segment containing task state segment information.

Macro Invocation Syntax

%DEFINE_TASK(*tname*, *start*, *SSI*, *DSI*, *SS0*, *SS1*, *SS2*, *ldt_sel*, *back_link*)

Input Parameters

tname	the name of the task
start	the starting location (initial CS and IP) of the task, a far label
SSI	a segment name that defines the initial SS for the task
DSI	the segment selector to which DS and ES are to be initialized in the task
SS0	the segment selector for the initial stack segment for level 0
SS1	the segment selector for the initial stack segment for level 1
SS2	the segment selector for the initial stack segment for level 2
ldt_sel	the selector for the task's local descriptor table
back_link	the value to be placed in the back link field of the task state segment

Description

%DEFINE_TASK is not an X286E run-time procedure, but rather an assembly time macro designed to create a data segment to be converted to a task state segment at run time using the procedure CREATE_TASK. The name of the task is provided by the parameter *'tname'*. The task state segment will be defined as a data segment with that name. Task execution will start at the memory location indicated by the parameter *'start'*. This parameter must be a far label. SS is initialized by *'SSI'*. SP will be initialized to 0. DS and ES are initialized by the argument *'DSI'*. The arguments *'SS0'*, *'SS1'*, and *'SS2'* are the segment selectors (names) that provide the initial stack segments for the level 0, 1, and 2 stacks respectively. The initial SP value for the privileged stacks is 0. Argument *ldt_sel* is the selector for the task's local descriptor table. It may be initialized at run time when the task state segment is created. Argument *'back_link'* provides the value to be placed in the back link field.

NOTE

The stack segments specified are expected to be expand-down data segments, thus the initial SP values of zero.

The declaration for the task state segment structure is named TS_STRUC. The task state segment fields are defined by the following field names:

BACK_LINK	the back link field
PRIV_STACK	the array of three pointers to level 0, 1, and 2 stacks

IP_SAVE	the saved instruction pointer
FLAGS_SAVE	the saved flags
AX_SAVE	the saved AX register
CX_SAVE	the saved CX register
DX_SAVE	the saved DX register
BX_SAVE	the saved BX register
SP_SAVE	the saved SP register
BP_SAVE	the saved BP register
SI_SAVE	the saved SI register
DI_SAVE	the saved DI register
ES_SAVE	the saved ES selector
CS_SAVE	the saved CS selector
SS_SAVE	the saved SS selector
DS_SAVE	the saved DS selector
LDT_SAVE	the saved LDT selector

Segment Management

Two procedures are provided for segment management. CREATE_SEG creates new segments and allocates memory space to segments and DELETE_SPACE frees the space occupied by a segment and marks the segment as not present.

The CREATE_SEG procedure creates a new segment and allocates memory space for the segment.

Procedure Call Syntax

```
PUSH WORD PTR AR
PUSH limit
PUSH table_sel
PUSH sel
CALL CREATE_SEG
```

Input Parameters

AR — the access rights byte for the segment

limit — the limit value of the segment

table_sel — the selector of the descriptor table into which the segment descriptor is to be placed. If this value is null, either the GDT or the current LDT is indicated, depending on the TI bit in *sel*.

sel — the selector that indexes the slot allocated in the descriptor table to receive the segment descriptor.

Output Parameter

status_code — a status code is returned in AX. A zero value indicates a successful completion of the procedure. A non-zero value indicates that the procedure failed and encodes the reason for failure

Description

This procedure creates a new segment and allocates memory space for the segment. The parameters *table_sel* and *sel* specify the descriptor table and slot respectively where the segment descriptor is to be placed. The parameters *AR* and *limit* define the access rights and limit values for the segment and are stored in the appropriate segment descriptor fields. The descriptor slot to contain the segment descriptor should have been allocated prior to this call.

Example

The following calling sequence illustrates a typical CREATE_SEG call. Figure 5-13 illustrates CREATE_SEG processing.

```
PUSH WORD PTR AR          ;ACCESS BYTE VALUE
PUSH SEG_LIM              ;SEGMENT LIMIT VALUE
PUSH LDT_SEL              ;SELECTED DESCRIPTOR TABLE
PUSH SEL_N                ;SELECTED SLOT IN TABLE
CALL CREATE_SEG
```

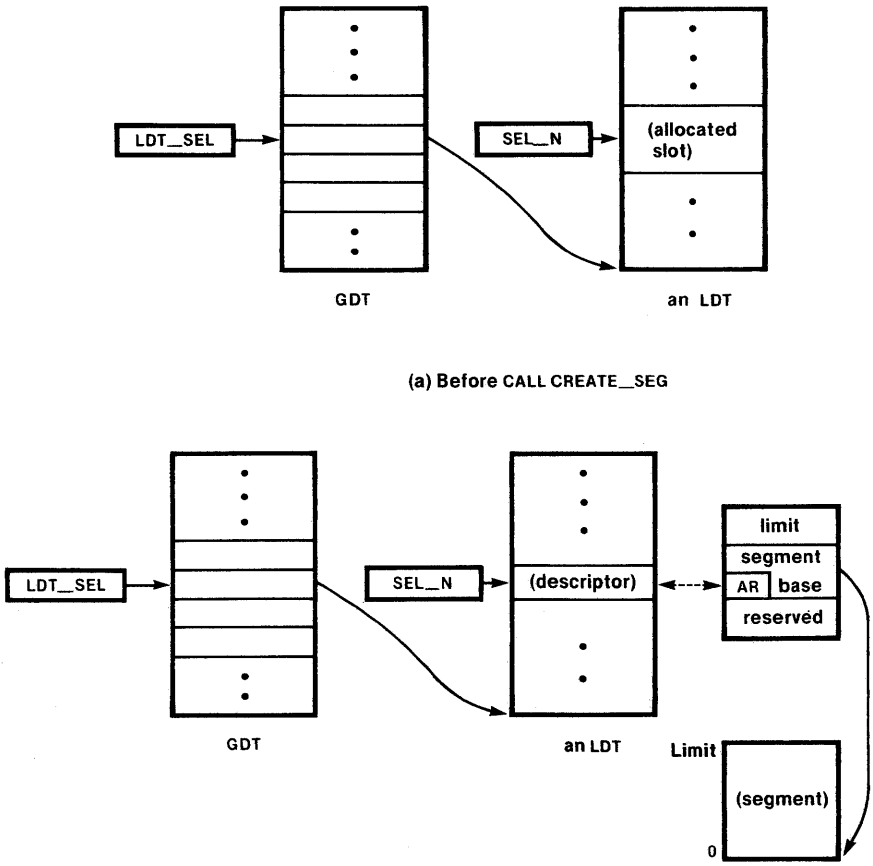


Figure 5-13. CREATE_SEG Processing 121711-13

Status Codes	Meaning
0	Success
1	Insufficient space for the new segment
5	Invalid table selector

The DELETE_SPACE procedure frees space occupied by a segment and marks the segment as not present.

Procedure Call Syntax

```
PUSH table_sel
PUSH sel
CALL DELETE_SPACE
```

Input Parameters

table_sel — the selector of the descriptor table in which the segment descriptor is located. If this value is null, then either the GDT or the current LDT is indicated, depending on the TI bit in *sel*.

sel — the selector that indexes the segment descriptor in the descriptor table

Output Parameter

status_code — a status code is returned in AX. A zero value indicates a successful completion of the procedure. A non-zero value indicates that the procedure failed and encodes the reason for failure.

Description

DELETE_SPACE frees the memory space occupied by a segment and marks the the segment as not present. If the segment is an LDT, its alias is also marked as not present.

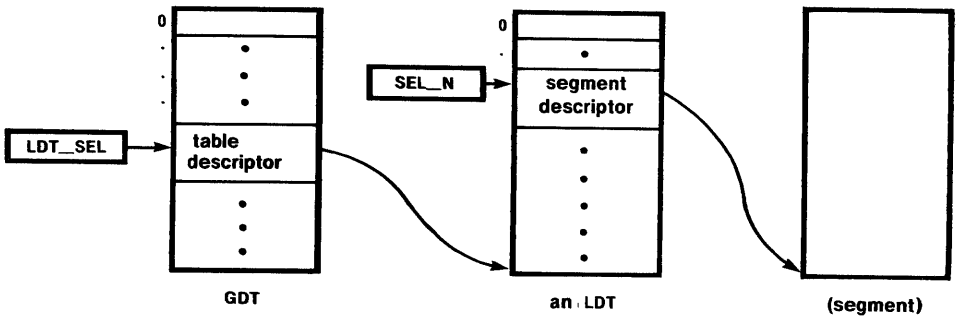
NOTE

There are no other attempts to manage aliases. You must ensure that all other aliases set up by your program are dealt with properly.

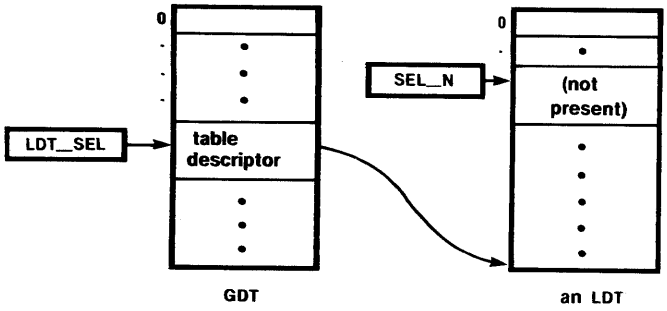
Example

The following example illustrates a typical call of DELETE_SPACE. Figure 5-14 illustrates DELETE_SPACE processing.

```
PUSH LDT_SEL
PUSH SEL_N
CALL DELETE_SPACE
```



(a) Before DELETE_SPACE CALL



(b) After DELETE_SPACE CALL

Figure 5-14. DELETE_SPACE Processing 121711-14

Status Codes	Meaning
0	Successful
4	Not a segment descriptor or space is already freed
5	Invalid table selector

Free Space Management

The free space management procedures allow dynamic allocation and de-allocation of the memory space above the loaded program. All but one paragraph (16 bytes) of the available memory beyond the top of the program loaded is accessible through these procedures for use at runtime. (X286E reserves the first available paragraph for use in managing the free memory space). The procedures deal with 24 bit real addresses. However, due to a restriction of the iAPX 286 Evaluation Simulator, the addresses must be paragraph aligned. That is, the four least significant bits of the address must be 0. The free space management procedures may be accessed indirectly through the CREATE_SEG, CREATE_LDT, and DELETE_SPACE procedures. The remainder of this section describes the free space procedures.

ALLOC_SPACE

The ALLOC_SPACE procedure allocates memory space.

Procedure Call Syntax

```
PUSH nbytes_minus_one
PUSH SEG address
PUSH OFFSET address
CALL ALLOC_SPACE
```

Input Parameter

nbytes_minus_one	the number of bytes, minus one, of memory space to be allocated

Output Parameters

address	the three bytes in memory where the 24-bit real address of the allocated space is placed upon return from the procedure
status_code	a status code is returned in AX. A zero value indicates a successful completion of the procedure. A non-zero value indicates that the procedure failed and encodes the reason for failure.

Description

This procedure allocates from one to 64K bytes of memory space. The first parameter is the number of bytes, minus one, of the amount of memory space that is to be allocated. ALLOC_SPACE returns the 24-bit real address of the allocated space in the three byte location *'address'*.

Example

```
SAVE_ADDR DD 0
NO_BYTES DW 0FFFH

PUSH NO_BYTES
PUSH SEG SAVE_ADDR
PUSH OFFSET SAVE_ADDR
CALL ALLOC_SPACE
```

The above call allocates 1000H bytes of memory space, placing the 24-bit real address of the start of the memory block in the low three bytes of SAVE_ADDR.

Status Codes	Meaning
0	Success
1	Insufficient free space

The FREE_SPACE procedure frees memory space.

Procedure Call Syntax

```
PUSH nbytes_minus_one
PUSH SEG address
PUSH OFFSET address
CALL FREE_SPACE
```

Input Parameters

nbytes_minus_one	the number of bytes, minus one, of memory space to be freed
address	the three bytes holding the 24-bit real address of the space to be freed

Output Parameter

status_code	a status code is returned in AX. A zero value indicates a successful completion of the procedure. A non-zero value indicates that the procedure failed and encodes the reason for failure.

Description

This procedure frees from one to 64K bytes of memory space. The first parameter is the number of bytes, minus one, of memory to be freed. The procedure rounds up the number of bytes actually freed to the nearest multiple of 16. The second parameter points to the three byte location that contains the 24-bit starting address of the memory space to be de-allocated.

Example

```
SAVE_ADDR DD 0
NO_BYTES DW 0FFFH

PUSH NO_BYTES
PUSH SEG SAVE_ADDR
PUSH OFFSET SAVE_ADDR
CALL FREE_SPACE
```

The above example deallocates 1000H bytes of memory starting at the memory location whose 24-bit real address is contained in the three bytes referenced by SAVE_ADDR.

Status Codes	Meaning
0	Success
8	the address is not aligned on a paragraph boundary
9	the space being freed overlaps with space already marked as free

NOTE

Because ALLOC_SPACE and FREE_SPACE deal with the memory available between the top of the program loaded and the top of the useable memory (as reported by the Simulator), attempts to free memory above or below these points will result in "overlap errors" (Status code 9). In particular, it is not possible to use FREE_SPACE to free-up a segment contained in the initially loaded program.

APPENDIX A
ERROR MESSAGES

This appendix lists the error and warning messages issued by the Evaluation Builder.

The Evaluation Builder issues error messages if it finds an invalid invocation line or an illegal input object file. Also, if during the processing of the command file, a syntax error or a semantic error is found, the Evaluation Builder issues diagnostic messages.

Each message put out by the builder has a unique number and may be either a warning or an error (errors are fatal, warnings are non-fatal). Warnings are directed to the list file (if any), and fatal error messages are directed to the console file, and to the list file (if any).

Each message has the following attributes:

- a unique number
- a fatal or non-fatal attribute (error/warning)
- a textual description
- a set (possibly empty) of items that help to identify the nature of the problem with respect to the user's program. An item could be a module name, a segment name, a symbol name, a file name, an extra message, etc. . .

Message numbers are assigned, depending on where they occur, as follows:

- 1xx—Invocation line or object file errors.
- 2xx—Command file errors.
- 3xx—Internal processing errors.

Besides the type of errors mentioned above, there are also errors that may happen at a system level. These errors are all fatal errors and are generated as described in the following paragraph.

Errors at System Interface Level

If an error in a call to the Host Operating System is detected, the Evaluation Builder will issue a fatal error message to the console file and the list file (if any), and will return control to the Host Operating System. The error can be an I/O error, an invalid parameter, an insufficient memory, etc. . .

The error message has the following format:

```
SYSTEM INTERFACE ERROR
error text
FILE: filename
```

The error text, which is Operating System dependent, may include an exception number and a description of the error. The file name will be present only if the error is an I/O error.

Errors in Invocation Line and Input Object File

If an error is found in the invocation line or in the input object file, then one of the following warning/error messages will be issued:

ERROR 100: MISSING INPUT FILE IN COMMAND TAIL
An input file in the command tail is required by the builder. This is a fatal error, the builder immediately terminates processing and returns control to the Host Operating System.

ERROR 101: INVALID TOKEN IN COMMAND TAIL
TOKEN: token string
An unknown control, indicated by the token string, was found in the command tail. This is a fatal error, the builder immediately terminates processing and returns control to the Host Operating System.

ERROR 102: INVALID DELIMITER IN COMMAND TAIL
ERROR OCCURS AFTER TOKEN
TOKEN: token string
An invalid delimiter, whose location is indicated by the token string, was found in the command tail. This is a fatal error, the builder immediately terminates processing and returns control to the Host Operating System.

ERROR 103: INVALID FILE NAME IN COMMAND TAIL
TOKEN: token string
An invalid file name, indicated by the token string, was found in the command tail. This is a fatal error, the builder immediately terminates processing and returns control to the Host Operating System.

WARNING 104: MODULE IS NOT A MAIN MODULE
The register initialization record is not present in the input object module. This is only a warning. No task state segment will be built for the module. The user may specify register initialization using the END directive in the corresponding assembly language module.

WARNING 105: MODULE CONTAINS UNSATISFIED EXTERNALS
SYMBOL: symbol name
.
.
SYMBOL: symbol name
There are no matching public symbols for the listed external symbols. The only external symbols allowed are the DQ_SIM symbol for I/O purpose and the public gates that are defined in the X286E file, which must exist on the same disk as for the builder.

WARNING 106: INVALID TASK STATE SEGMENT
error text
An entry for the task state segment is illegal (invalid). The error text indicates why. It may be a missing definition for a base register/register pair such as CS:IP or SS:SP or DS/ES, the user can provide this information through the END directive of the assembly language module. Alternately, it may be a missing stack entry, the user may optionally provide this information by using the STACK definition in the command file. The error may also be caused by an illegal privilege level assignment; the user should make sure that the code segment (CS) privilege level is equal to the stack segment (SS) privilege level and numerically less than or equal to the data segment (DS) privilege level.

WARNING 107: FIXUP REFERENCE TO LESS PRIVILEGED LEVEL
SEGMENT IS REFERENCING SEGMENT
SEGMENT: segment name
REFERENCED OBJECT: external name or segment name

An illegal inter-level reference was found. The referencing segment is indicated by the SEGMENT item and the referenced object is indicated by the REFERENCED OBJECT item. The user should check the segment privilege specification in the SEGMENT definition.

WARNING 108: FIXUP REFERENCE TO MORE PRIVILEGED LEVEL
SEGMENT IS REFERENCING SEGMENT
SEGMENT: segment name
REFERENCED OBJECT: external name or segment name

An illegal inter-level reference was found. The referencing segment is indicated by the SEGMENT item and the referenced object is indicated by the REFERENCED OBJECT item. The user should check the segment privilege specification in the SEGMENT definition.

WARNING 109: REFERENCE TO UNSATISFIED EXTERNAL
SYMBOL: symbol name

There is a reference to an unresolved external symbol indicated by its name. Processing continues with no modification to the unresolved reference.

WARNING 110: REFERENCE TO UNGATED PUBLIC
SYMBOL: symbol name

There is a reference to a more privileged executable segment through a public symbol, but there is no gate defined for the public entry. The user should define a gate for the corresponding public using the GATE definition in the command file.

WARNING 111: INTERRUPT OR TRAP GATE PLACED IN LDT
SYMBOL: symbol name

The builder attempts to put all segment or gate descriptors not explicitly assigned by the user into the LDT table, and [illegible] is an interrupt gate or a trap gate among them.

ERROR 112: INVALID OBJECT MODULE INPUT
FILE: filename

The input object module is not a valid module. This could mean that the file does not contain an object file output by the evaluation assembler.

ERROR 113: INVALID STACK SEGMENT SIZE
extra message
SEGMENT: segment name

The size of the specified stack segment is either one byte too small or one byte too big. The extra message will indicate which case. The user should correct the size declaration for the corresponding stack segment in the assembly language module.

WARNING 114: DUPLICATE PUBLIC ENCOUNTERED
SYMBOL: public name

The specified public name in the user program is already used as public symbol in the X286E file. The user should change the public name in the assembly language module.

Errors in Command File

If errors are found on a line in the command file, diagnostic messages will be inserted into the listing after the line on which they were detected.

The messages have the following format:

*** WARNING 2xx: LINE n, NEAR 'token', message

Where 2xx is the warning number, n is the number of the line on which the error occurred, 'token' indicates the location of the error, and message is the warning message corresponding to the warning number. The warning messages are listed below along with their assigned numbers.

First are messages to indicate syntax errors found in a command file. The error recovery involves the insertion and/or the deletion of tokens until the current token is acceptable. The messages indicate which tokens have been inserted or skipped.

WARNING 200: ILLEGAL TOKEN(S) SKIPPED UNTIL 'token'
WARNING 201: MISSING ';' INSERTED
WARNING 202: MISSING ',' INSERTED
WARNING 203: MISSING '(' INSERTED
WARNING 204: MISSING ')' INSERTED
WARNING 205: MISSING '=' INSERTED
WARNING 206: MISSING ':' INSERTED
WARNING 207: MISSING SEGMENT ATTRIBUTE, 'NOCONFORMING' INSERTED
WARNING 208: MISSING GATE TYPE, 'CALL' INSERTED
WARNING 209: MISSING TABLE NAME, 'GDT' INSERTED
WARNING 210: MISSING PARAMETER KEYWORD
WARNING 211: MISSING NUMBER
WARNING 212: MISSING IDENTIFIER

Following are semantic errors:

WARNING 213: SPECIFIED SEGMENT OR GATE NOT FOUND
A symbolic name was used in the specification of an entry for a descriptor table, but there is no matching segment or gate name. This is usually the result of a typo in the command line or a corresponding gate was not defined using the GATE definition. The table entry will be marked as invalid entry.

WARNING 214: SPECIFIED SEGMENT NOT FOUND IN INPUT MODULE
A symbolic name was used in a SEGMENT definition or a STACK definition, but no such segment was found in the input module. Probably there is a typo in the command line. The definition for the segment will be ignored.

WARNING 215: PUBLIC SYMBOL NOT FOUND IN INPUT MODULE
A symbolic name was used to define a gate in a GATE definition, but there is no such public symbol in the input module. This is usually the result of a typo in the command line or the user forgot to declare the symbol as a public symbol in the corresponding assembly language module.

WARNING 216: STACK ALREADY SPECIFIED
The stack segment for the same privilege level was specified more than once in a STACK definition. The first one specified is used.

WARNING 217: TABLE ENTRY ALREADY SPECIFIED
The ENTRY keyword was specified more than once in a TABLE definition. The new entries will be added to the corresponding table.

WARNING 218: TABLE LIMIT ALREADY SPECIFIED
A TABLE limit was specified more than once. The first one specified is used.

WARNING 219: PRIVILEGE LEVEL ALREADY SPECIFIED
Privilege level for a gate or a segment was specified more than once. The first value specified is used.

WARNING 220: GATE TYPE ALREADY SPECIFIED
The gate type (CALL or INTERRUPT or TRAP) was specified more than once for the same gate. The firs type specified is used.

WARNING 221: SEGMENT ATTRIBUTE ALREADY SPECIFIED
The segment attribute (CONFORMING or NOCONFORMING) was specified more than once for the same segment. The first attribute specified is used.

WARNING 222: ILLEGAL VALUE FOR PRIVILEGE LEVEL
An illegal number was used to specify a segment or a gate privilege level. This may be an invalid number or a valid number but greater than 3, the default value (3) is used instead. This may also be the case where the specified privilege level for a gate is less than the privilege level of the public entry it points to, this is only a warning and the specified value is used.

WARNING 223: ILLEGAL VALUE FOR TABLE LIMIT
An illegal number was used to specify a descriptor table limit. This may be an invalid number or a valid number but greater than the maximum value allowed (8191), the default value (+16) is used instead. This may also be the case where the specified limit is too small for the corresponding table, then the limit will be set equal to the actual number of table entries. The message is also issued if the table limit (specified by the user or not) becomes larger than 8191 because of an entry index assignment, in this case the table limit will be set equal to 8191.

WARNING 224: ILLEGAL VALUE FOR ENTRY INDEX
An illegal number was used to specify an index for a descriptor table entry. This may be an invalid number or a valid number but greater than the maximum value allowed (8191), the specified number is ignored. This may also be a valid number but the entry is reserved for Intel use or entry indices are not assigned in an increasing order, this is only a warning and the specified value is used.

WARNING 225: ILLEGAL IDT ENTRY
A segment or a call gate was specified as an entry for the Interrupt Descriptor Table. This is only a warning, and the specified entry is added to the IDT.

WARNING 226: ILLEGAL GDT OR LDT ENTRY
A trap gate or an interrupt gate was specified as an entry to the Global Descriptor Table or the Local Descriptor Table. This is only a warning, and the specified entry is added to the specified table.

WARNING 227: ILLEGAL STACK ENTRY
In the definition of a stack entry for the Task State Segment a non-stack segment was used or a stack segment of privilege greater than 2 was used. The entry will be marked as invalid entry.

WARNING 228: IDENTIFIER TOO LONG
A token of more than 40 bytes was found. The token is rejected.

WARNING 229: NUMBER GREATER THAN 64K
A numeric constant greater than 64K was found at a location where a 16 bit constant was expected such as values for privilege level, limit, or table entry index. The value is set equal to the specified number modulo 64K.

WARNING 230: ILLEGAL ATTRIBUTE FOR NON-EXECUTABLE SEGMENT
The CONFORMING/NOCONFORMING attribute was assigned to a non-executable (data) segment. The specification is ignored.

WARNING 231: UNGATABLE PUBLIC

A CALL gate was defined using a public symbol, but either the public entry does not have a word count or its word count is greater than 31. The user should go back to the assembly language program and specify a correct word count for the corresponding public procedure.

Internal Processing Errors

The fatal internal errors should never occur. The format of the error message is as follows:

*** ERROR 3xx: INTERNAL PROCESSING ERROR, message

Following is the list of error messages of this class, along with their assigned numbers:

ERROR 300: INTERNAL NAMES OUT OF SYNCH
ERROR 301: ERROR IN WORKFILE PROCESSING
ERROR 302: INVALID INPUT FORMAT
ERROR 303: IMPROPER USE OF PROCEDURAL INTERFACE
ERROR 304: TOO MANY INTERNAL NAMES
ERROR 305: INSUFFICIENT WORK AREA
ERROR 306: NAME TOO LONG
ERROR 307: ILLEGAL INTERNAL NAME
ERROR 308: MISPLACED DEFINITION
ERROR 309: INVALID FILE TYPE
ERROR 310: READ PAST EOF
ERROR 311: SECTION TOO SHORT
ERROR 312: INVALID INPUT FILE
ERROR 313: ATTEMPT TO USE UNALLOCATED STRUCTURE
ERROR 314: INTERNAL NAME FOR TEXT NOT A SEGMENT
ERROR 315: UNRECOGNIZED FIXUP FORMAT
ERROR 316: INCLUDE FILE SIZE ERROR
ERROR 317: NAME TABLE OVERFLOW
ERROR 318: PARSER STACK OVERFLOW
ERROR 319: MULTIPLE ARGUMENTS DISCOVERED DURING A REDUCTION
ERROR 320: PARSER ARGUMENT STACK UNDERFLOW
ERROR 321: PARSER STATE STACK UNDERFLOW
ERROR 322: NO TOKEN AVAILABLE FOR ERROR RECOVERY
ERROR 323: INVALID CONTROL PARAMETER

APPENDIX B
STATUS CODES

This appendix lists the status codes issued by the X286E run-time procedures.

Each X286E run-time procedure will return a status code in AX. The value of the status code indicates the success or failure of the procedure. A non-zero value indicates that the procedure has failed and encodes the reason for failure.

The following list contains the code value and meaning of each status code.

Status Code	Meaning
0	Successful completion of the procedure
1	Insufficient space
2	Selector out of range
3	Insufficient number of consecutive free descriptor slots
4	Not a segment descriptor or space is already freed
5	Invalid table selector or no table selected
6	Interrupt number out of range
7	Selector does not select a data segment in the GDT or the data segment is too small
8	Address is not aligned on a paragraph boundary
9	The space being freed overlaps with space already marked as free
10	Invalid LDT size

INDEX

REQUEST FOR READER'S COMMENTS

Intel's Technical Publications Departments attempt to provide documents that meet the needs of all Intel product users. This form lets you participate directly in the documentation process. Your comments will help us correct and improve our manuals. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this document. If you have any comments on the equipment software itself, please contact your Intel representative. If you wish to order manuals contact the Intel Literature Department (see page ii of this manual).

1. Please describe any errors you found in this manual (include page number).

2. Does the document cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of document for your needs? Is it at the right level? What other types of documents are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating). _____

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

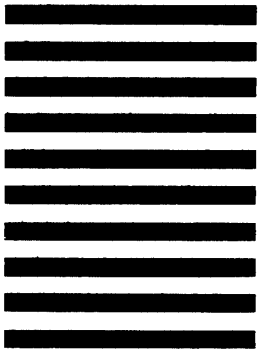
CITY _____ STATE (COUNTRY) _____ ZIP CODE _____

Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS ...

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY
IF MAILED
IN U.S.A.



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1040 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
Attn: Technical Publications M/S 6-2000
3065 Bowers Avenue
Santa Clara, CA 95051

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) **987-8080**

Printed in U.S.A.