iRMX 86™ CONFIGURATION GUIDE

Order Number: 9803126-03

Copyright © 1981 Intel Corporation
Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

REV.	REVISION HISTORY	PRINT DATE
-01	Original Issue	4/80
-02	Adds information on Nucleus, Application Loader, and Bootstrap Loader configuration; corrects technical and typographical errors; and documents Release 2 of the iRMX 86 Operating System.	11/80
-03	Adds information on Nucleus, Terminal Handler, and Debugger component configuration; adds chapters on Extended I/O System and Human Interface configuration; corrects technical and typographical errors; and documents Release 3 of the iRMX 86 Operating System.	5/81

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
CREDIT
i
ICE
iCS
im
Insite
Intel
Intel
Intelevision
Intellec
iRMX
iSBC
iSBX
Library Manager
MCS
Megachassis
Micromap
Multibus
Multimodule
PROMPT
Promware
RMX/80
System 2000
UPI
μScope

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

A385/681/5K CP

PREFACE

The iRMX 86 Operating System is a software package that provides a realtime, multitasking environment for Intel iAPX 86-based microcomputers, including the iSBC 86/12A single board computer. This manual contains the instructions that you need to configure an iRMX 86 application system using Release 3.0 of the iRMX 86 Operating System. By following the instructions in this manual, you can use an INTELLEC Series II or Series III Microcomputer Development System to build an iRMX 86 system.

READER LEVEL

This manual assumes that you are a system programmer, experienced in dealing with operating systems. In particular, it assumes you are familiar with the following:

- The iRMX 86 Operating System and the iRMX 86 reference manuals
- The 8086/8087/8088 Macro Assembly Language and/or PL/M-86
- The INTELLEC Series II or Series III Microcomputer Development System
- LINK86 and LOC86
- The notions of segments, groups, and classes as they apply to assembly language, PL/M-86, LINK86, and LOC86

NOTATIONAL CONVENTIONS

The following conventions are used to show syntax in this manual:

UPPERCASE Information appearing in uppercase must be entered or coded exactly as shown. This information, however, can actually be entered in uppercase or lowercase.

lowercase Fields appearing in lowercase indicate variable information. The user must enter the appropriate value or symbol for variable fields.

RELATED PUBLICATIONS

The following manuals provide additional information that may be helpful to users of this manual.

Manual	Number
Introduction to the iRMX 86™ Operating System	9803124
iRMX 86™ Installation Guide	9803125
iRMX 86™ Nucleus Reference Manual	9803122
iRMX 86™ Terminal Handler Reference Manual	143324
iRMX 86™ Debugger Reference Manual	143323
iRMX 86™ Basic I/O System Reference Manual	9803123
iRMX 86™ Extended I/O System Reference Manual	143308
iRMX 86™ Loader Reference Manual	143318
iRMX 86™ Human Interface Reference Manual	9803202
iRMX 86™ System Programmer's Reference Manual	142721
iRMX 86™ Programming Techniques Manual	142982
Guide to Writing Device Drivers for the iRMX 86™ I/O System	142926
ISIS-II User's Guide	9800306
INTELLEC Series III Microcomputer Development System Console Operating Instructions	121601
PL/M-86 User's Guide for 8086-Based Development Systems	121636
PL/M-86 Programming Manual for 8080/8085-Based Development Systems	9800466
PL/M-86 Compiler Operating Instructions for 8080/8085-Based Development Systems	9800478
8086/8087/8088 Macro Assembly Language Reference Manual for 8086-Based Development Systems	121627

8086/8087/8088 Macro Assembly Language Reference Manual for 8080/8085-Based Development Systems	121623
8086/8087/8088 Macro Assembler Operating Instructions for 8086-Based Development Systems	121628
8086/8087/8088 Macro Assembler Operating Instructions for 8080/8085-Based Development Systems	121624
iAPX 86,88 Family Utilities User's Guide for 8086-Based Development Systems	121616
8086 Family Utilities User's Guide for 8080/8085-Based Development Systems	9800639
The 8086 Family User's Manual	9800722
The 8086 Family User's Manual - Numerics Supplement	121586
ICE-86 In-circuit Emulator Operating Instructions for ISIS-II Users	9800714
iSBC 957A INTELLEC - iSBC 86/12A Interface and Execution Package User's Guide	142849
Universal PROM Programmer User's Manual	9800819

CONTENTS

PAGE

CHAPTER 1
INTRODUCTION
Configuration Environment.......... 1-3
Tasks, Jobs, and the Initial System.......... 1-3
Types of System Configuration.......... 1-5
Using This Manual.......... 1-5

CHAPTER 2
PROCEDURAL OVERVIEW.......... 2-1

CHAPTER 3
PREPARING JOBS FOR SYSTEM CONFIGURATION
Preparing Application Jobs.......... 3-1
Language Requirements.......... 3-2
INCLUDE Files.......... 3-2
Size Control Considerations.......... 3-3
Initialization.......... 3-3
Preparing the Subsystems.......... 3-5

CHAPTER 4
LOCATING A TEST AND DEVELOPMENT SYSTEM IN RAM
General System Layout.......... 4-1
System Type.......... 4-2
High and Low Location of Modules.......... 4-2
Module Order.......... 4-2
Preparing a Memory Map.......... 4-3
Link and Locate the Subsystems and Application Jobs.......... 4-8
Linking and Locating the Subsystems.......... 4-8
Overview.......... 4-8
Preparing Diskettes.......... 4-9
Placing Diskettes in the Proper Drives.......... 4-10
Using a Series III Development System.......... 4-10
Linking and Locating Application Jobs.......... 4-11
Linking Application Jobs.......... 4-13
Locating Application Jobs.......... 4-14
The Iterative Link and Locate Process.......... 4-15
Build the Configuration File.......... 4-22
%JOB Macro.......... 4-23
Data Segment Allocation.......... 4-27
Stack Allocation.......... 4-27
%SAB Macro.......... 4-28
%SYSTEM Macro.......... 4-30
Macro Parameters for Subsystems.......... 4-33
Creating the Configuration File.......... 4-37
Generate the Root Job.......... 4-38
Load and Test the System.......... 4-39

CONTENTS (continued)

PAGE

CHAPTER 5
CONFIGURING THE FINAL ROM/RAM-BASED SYSTEM
Minimizing the Memory Address Space.. 5-1
Locating the ROM/RAM-based System.. 5-2
Prepare a Memory Map.. 5-3
Locate the Modules.. 5-4
Testing the System in RAM.. 5-5

CHAPTER 6
CONFIGURING THE NUCLEUS
Modifying NTABLE.A86.. 6-2
Selecting Nucleus Internal Features.. 6-3
Parameter Validation.. 6-3
System Default Exception Handler.. 6-6
Selecting Nucleus System Calls.. 6-4
Modifying NDEVCF.A86.. 6-6
Programmable Interrupt Controller (PIC) Configuration............ 6-7
%MASTER_PIC Macro.. 6-7
%SLAVE_PIC Macro.. 6-8
Programmable Interval Timer (PIT) Configuration.................. 6-8
8087 NDP Configuration.. 6-9
Maximal, Default, and Minimal Configuration........................ 6-10
Assembling the Configuration Files, Linking and Locating the Nucleus 6-11
Nucleus Initialization Errors.. 6-11
Nucleus and Memory Initialization Errors......................... 6-12
Root Task Errors.. 6-12

CHAPTER 7
CONFIGURING THE TERMINAL HANDLER
Modifying MCONFG.A86.. 7-1
%TH_19000_BAUD_COUNT Macro.. 7-2
%MTH Macro.. 7-3
%TH_USART Macro.. 7-4
%TH_TIMER Macro.. 7-5
%TH_CHAR_LENGTH Macro.. 7-5
%TH_MAILBOX_NAMES Macro.. 7-6
%TH_INT_LEVELS Macro.. 7-6
Assembling MCONFG.A86, Linking and Locating the Terminal Handler... 7-7
MTH.CSD Modifications.. 7-8
Submitting MTH.CSD.. 7-8
Creating Multiple Versions of the Terminal Handler................ 7-9

CHAPTER 8
CONFIGURING THE DEBUGGER
Modifying DTHCNF.A86.. 8-1
Assembling DTHCNF.A86, Linking and Locating the Debugger........... 8-2
DB.CSD Modifications.. 8-2
Submitting DB.CSD.. 8-2

CONTENTS (continued)

PAGE

CHAPTER 9
CONFIGURING THE BASIC I/O SYSTEM
INCLUDE Files.. 9-3
Selecting Non-File/Connection Interface Features (ITABLE.A86)...... 9-3
Selecting the File/Connection Interface Features (ITABLE.A86)...... 9-5
File Driver Global Data.. 9-5
File Driver Tables... 9-7
Selecting Features (ITABLE.A86).................................. 9-18
Describing the I/O Devices (IDEVCF.A86).......................... 9-20
Device Numbering... 9-20
Device-Unit Information Blocks................................. 9-21
Device And Unit Information Tables............................. 9-27
Common Device Driver Tables.................................. 9-27
Random Access Device Driver Tables........................... 9-29
Device Driver Tables for Intel-supplied Device Drivers......... 9-30
General Device Information....................................... 9-38
Assembling the Configuration Files, Linking and Locating the Basic I/O System... 9-38
Basic I/O System Initialization.................................. 9-39

CHAPTER 10
CONFIGURING THE APPLICATION LOADER
Modifying LDRCNF.P86... 10-1
Compiling LCONFG.P86, Linking and Locating the Loader............. 10-2

CHAPTER 11
CONFIGURING THE BOOTSTRAP LOADER
First Stage Configuration... 11-1
%CONSOLE Macro.. 11-2
%MANUAL Macro... 11-3
%AUTO Macro... 11-3
%DEVICE Macro... 11-3
%END Macro.. 11-4
Driver Configuration.. 11-4
Intel-Supplied Procedures...................................... 11-5
iSBC 204 Device Driver....................................... 11-5
iSBC 206 Device Driver....................................... 11-6
iSBC 215/220 Device Driver................................... 11-7
iSBC 254 Device Driver....................................... 11-8
User-Supplied Procedures....................................... 11-9
Assembling the Configuration Files, Linking and Locating the Bootstrap Loader.. 11-9

CONTENTS (continued)

PAGE

CHAPTER 12
CONFIGURING THE EXTENDED I/O SYSTEM
Selecting System Calls (ETABLE.A86).................................... 12-3
Selecting Logical Devices (EDEVCF.A86)................................ 12-4
Selecting I/O Jobs (EJOBCF.A86)....................................... 12-6
%IO_USER Macro.. 12-7
%IO_JOB Macro... 12-8
Assembling the Configuration Files, Linking and Locating the
Extended I/O System.. 12-10
Extended I/O System Initialization.................................... 12-11

CHAPTER 13
CONFIGURING THE HUMAN INTERFACE
Modifying HCONFG.A86.. 13-1
Compiling HCONFG.A86, Linking and Locating the Human Interface..... 13-4
Human Interface Requirements... 13-5
Terminal Handler or Debugger Requirements............................ 13-5
Basic I/O System Requirements... 13-5
Extended I/O System Requirements...................................... 13-6
Creating Human Interface Volumes...................................... 13-7
Creating Human Interface Commands..................................... 13-8
Using a Series III Development System................................. 13-7
Using a Series II Development System.................................. 13-9
Additional Requirements for Absolute Code............................ 13-10

APPENDIX A
EXAMPLE SYSTEM CONFIGURATION
Prepare a Memory Map.. A-2
Configure the Subsystems and Link and Locate the System............ A-4
Prepare, Link, and Locate the Nucleus................................ A-6
Prepare, Link, and Locate the Debugger............................... A-10
Allow Space for the Root Job.. A-12
Link the Application Job.. A-12
Locate the Application Job.. A-13
Build the Configuration File.. A-15
%JOB Macro Calls.. A-15
Debugger %JOB Call... A-15
Application Job %JOB Call.. A-17
%SAB Macro Calls.. A-19
%SYSTEM Macro Call.. A-21
Create the Actual Configuration File.................................. A-23
Generate the Root Job... A-23
Load the System... A-24

CONTENTS (continued)

PAGE

APPENDIX B
BURNING THE NUCLEUS INTO 2732 PROM
Requirements.. B-1
Locate the Nucleus.. B-1
Burn the Code Into PROM....................................... B-2

APPENDIX C
SYSTEM CALL USAGE... C-1

FIGURES

1-1. iRMX 86™ Application System................................ 1-2
1-2. Initial Job Tree... 1-4
4-1. General System Layout...................................... 4-3
4-2. System Memory Map.. 4-5
4-3. Example Memory Map Worksheet................................ 4-7
4-4. Subsystem SUBMIT File Procedure............................. 4-9
4-5. Application Job Link and Locate Procedure.................. 4-12
4-6. Example Nucleus Locate Map................................. 4-18
4-7. Entering the Nucleus End Address on the Memory Map......... 4-21
4-8. %JOB Macro Worksheet....................................... 4-24
4-9. %SAB Macro Worksheet....................................... 4-29
4-10. %SYSTEM Macro Worksheet................................... 4-31
5-1. Memory Layout of a RAM-based System........................ 5-2
5-2. Memory Layout of a ROM/RAM System.......................... 5-3
6-1. NTABLE.A86 Structure....................................... 6-2
6-2. Feature Configuration Table (NTABLE.A86)................... 6-3
6-3. System Call Configuration Table (NTABLE.A86)............... 6-5
6-4. Component Configuration Table (NDEVCF.A86)................. 6-6
8-1. Debugger Configuration File (DTHCNF.A86)................... 8-1
9-1. ITABLE.A86 and IDEVCF.A86 Structure........................ 9-2
9-2. Non-File/Connection Interface Configuration Values......... 9-4
9-3. File Driver Global Data Parameters......................... 9-6
9-4. Physical File Driver Tables................................ 9-8
9-5. Stream File Driver Tables.................................. 9-10
9-6. Reserved File Driver Tables................................ 9-12
9-7. Named File Driver Tables................................... 9-13
9-8. Basic I/O System Features.................................. 9-18
9-9. Device Numbering... 9-20
9-10. Example DUIB Contained in IDEVCF.A86...................... 9-26
9-11. Device Information Table for iSBC 204 Device.............. 9-32
9-12. Unit Information Table for iSBC 204 Unit.................. 9-33
10-1. Application Loader Configuration File (LCONFG.P86)........ 10-1
11-1. First Stage Configuration File (BS1.A86).................. 11-2
11-2. Driver Configuration File (B204.A86)...................... 11-5
11-3. Driver Configuration File (B206.A86)...................... 11-6
11-4. Driver Configuration File (B215.A86)...................... 11-7

FIGURES (continued)

		PAGE
11-5.	Driver Configuration File (B254.A86)......................	11-8
12-1.	Structure of Extended I/O System Configuration Files.......	12-2
12-2.	System Configuration File (ETABLE.A86)....................	12-3
12-3.	Logical Device Configuration File (EDEVCF.A86)............	12-5
12-4.	I/O Job Configuration File (EJOBCF.A86)...................	12-7
13-1.	Human Interface Configuration File (HCONFG.P86)...........	13-2
A-1.	Preparing the Memory Map..................................	A-3
A-2.	Completed Memory Map......................................	A-5
A-3.	Example Nucleus Configuration File........................	A-6
A-4.	Nucleus Locate Map..	A-8
A-5.	Debugger Locate Map.......................................	A-10
A-6.	Application Job Locate Map................................	A-13
A-7.	Completed Debugger %JOB Macro Worksheet...................	A-16
A-8.	Completed Application Job %JOB Macro Worksheet............	A-18
A-9.	Completed %SAB Macro Worksheet............................	A-20
A-10.	Completed %SYSTEM Macro Worksheet.........................	A-22
B-1.	UPM SUBMIT File to Burn the Nucleus into PROM.............	B-2

TABLES

3-1.	INCLUDE Files...	3-3
4-1.	Interface Libraries as a Function of PL/M-86 Models and Subsystems..	4-14
4-2.	Suggested %JOB Values for Optional Subsystems.............	4-34
4-3.	Suggested %SYSTEM Values for Optional Subsystems..........	4-36
6-1.	System-Level and Job-Level Parameter Validation...........	6-4
9-1.	Physical File Driver System Calls and Procedure Names......	9-15
9-2.	Stream File Driver System Calls and Procedure Names........	9-16
9-3.	Named File Driver System Calls and Procedure Names.........	9-17
9-4.	Common and Random Access Driver DUIB Values................	9-26
C-1.	System Calls Used by the Terminal Handler..................	C-1
C-2.	System Calls Used by the Debugger..........................	C-2
C-3.	System Calls Used by the I/O System........................	C-2
C-4.	System Calls Used by the Extended I/O System...............	C-3
C-5.	System Calls Used by the Application Loader................	C-3
C-6.	System Calls Used by the Human Interface...................	C-4

CHAPTER 1. INTRODUCTION

The Intel iRMX 86 Operating System is a software package designed for use with the Intel iAPX 86-based microcomputers. It is a powerful and flexible system around which you can build your application system.

The iRMX 86 Operating System consists of a number of subsystems, some of which must be included in your application system, and some of which are optional. The subsystems of the iRMX 86 Operating System are:

- Nucleus — This is the core of the iRMX 86 Operating System and is required by every application system. It provides services for the remainder of the software running in the system.

- Terminal Handler — This is an optional subsystem that provides a real-time interface between your terminal and other software running under the supervision of the Nucleus.

- Debugger — This is an optional subsystem that provides a facility for debugging and monitoring software running under the supervision of the Nucleus.

- I/O System — This is an optional subsystem that provides asynchronous file access capabilities for software running under the supervision of the Nucleus.

- Extended I/O System — This is an optional subsystem that provides high level, synchronous file access capabilities for software running under the supervision of the Nucleus.

- Application Loader — This is an optional subsystem that provides the capability to load object files into memory from disk under the control of the Operating System.

- Bootstrap Loader — This is an optional subsystem that provides the capability to load the other subsystems and/or application jobs into memory from disk and begin system execution.

- Human Interface — This is an optional subsystem that provides an interactive interface between a user and software running under the supervision of the Nucleus.

Software that you create runs in an iRMX 86 application system using the facilities of the Nucleus and the other subsystems.

Configuration consists of selecting the subsystems that are appropriate for your system, tailoring them to meet your individual needs, and combining them with your own application software to form a functional application system. Figure 1-1 illustrates this.

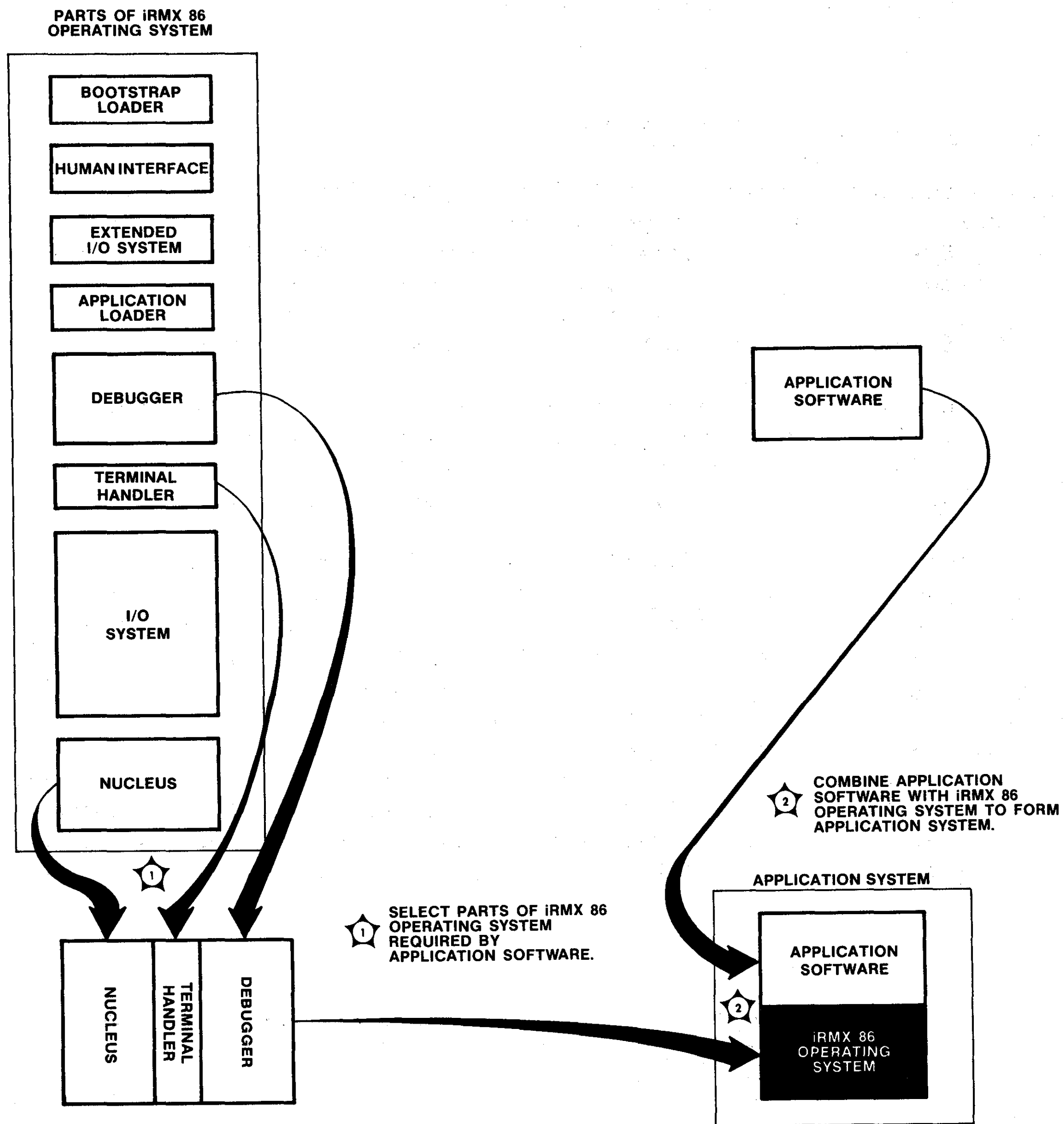


Figure 1-1. iRMX 86™ Application System

CONFIGURATION ENVIRONMENT

The INTELLEC Microcomputer Development System provides the environment in which you create your application system. With the development system you can code and translate your user software, and link and locate the various components of the application system. Upon completion, you can load your iRMX 86 application system from the INTELLEC development system into an iAPX 86-based microcomputer using the ICE-86 in-circuit emulator, or the iSBC 957A package.

The development system can also be used to run the Files Utility (described in the iRMX 86 INSTALLATION GUIDE). The Files Utility can format iRMX 86 disks and copy your system onto disk. Then you can use the Bootstrap Loader to load your system.

TASKS, JOBS, AND THE INITIAL SYSTEM

Tasks are the active parts of an iRMX 86 application system. The subsystems contain tasks that perform some of the functions of the Operating System. Your application software also consists of one or more tasks. Each task is part of a job. A job is the environment in which tasks run; thus a job consists of tasks and the resources that they use. The iRMX 86 NUCLEUS REFERENCE MANUAL describes this in detail.

The jobs in a system form a hierarchy. A task in one job can create other jobs. Tasks in the new jobs can create still other jobs, and so forth. The jobs which contain tasks that create other jobs are called parent jobs, and the jobs they create are their offspring.

Task and job creation is a dynamic process. However, when you configure a system, you specify an initial system which is created automatically when the system starts executing. The job tree for an initial system consists of an ultimate parent job called the root job and a number of its offspring called first-level jobs. Intel supplies the root job. Some of the first-level jobs are jobs for the subsystems that your application system requires (the Debugger, the Terminal Handler, the I/O System, the Extended I/O System, the Application Loader, and/or the Human Interface). Intel also supplies these jobs as part of the subsystems. The remainder of the first-level jobs are jobs that you provide. Figure 1-2 illustrates an initial job tree.

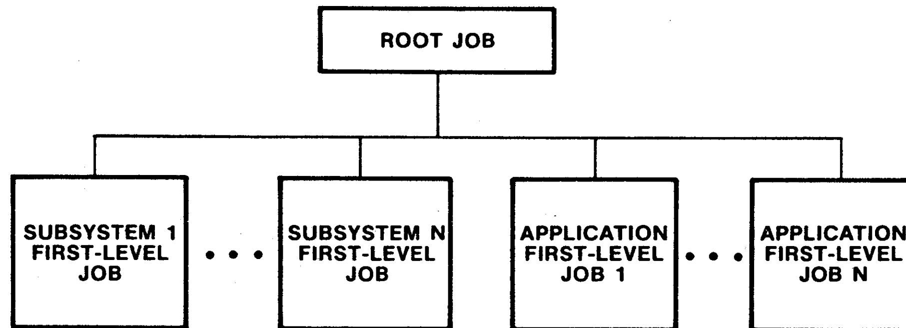


Figure 1-2. Initial Job Tree

First-level jobs can spawn a number of offspring jobs, beginning the dynamic tree structure of the system. The iRMX 86 NUCLEUS REFERENCE MANUAL describes how to create new tasks and jobs. However, in order to create all of its offspring jobs, a first-level job must be able to determine where in memory the code for all of its offspring tasks resides. You provide this ability in one of two ways:

- The easiest and most common way is by linking a first-level job and all of its offspring jobs together in one link module. This allows tasks in the first-level job to use symbolic names to specify the start addresses and data segment bases in calls to CREATE$JOB.

- If the code is too large to link together in one module, you can link and locate the tasks separately. However, this prevents some tasks from referring to other tasks with symbolic names. In this case, when a task in the first-level job creates offspring jobs or tasks, it must specify absolute values for the start address and data segment base parameters of CREATE$JOB or CREATE$TASK.

You must use one of these methods for each first-level job that you create. Chapter 4 contains a further description. When you configure a system, you must supply information to the root job about each first-level job.

TYPES OF SYSTEM CONFIGURATION

When you define an iRMX 86 configuration, you may have one of several goals in mind. You may be putting together your first iRMX 86 system and thus want to test and debug the entire system. You may have gotten portions of the system working to your satisfaction, but now want to correct a few isolated bugs or add a new task or tasks. Or, you may have completely tested and debugged your system and now want to create the final ROM-based version.

When building your first system, you should locate your entire system in RAM only. This saves you the trouble of burning code into PROM, only to have to reburn it later.

When creating your final system, you must perform additional procedures because you are locating the system in two different areas of memory, ROM and RAM. In a ROM/RAM system, you must separate all of the ROM-resident parts of the system and locate them in ROM.

When building an intermediate system, you have debugged and tested portions of your system, but are still developing or debugging others. In this case you have two options. Each time you make a correction you can use the same initial RAM-only configuration, and reload the entire system into RAM for testing. Or, you can follow the procedures for locating a ROM/RAM system for the stable portions of your system only, such as the Nucleus. If you burn the stable portions into PROM, you save the time of loading each time you generate a new system.

USING THIS MANUAL

Chapter 2 of this manual lists the procedure involved in building an iRMX 86-based system in step-by-step instructions. You can read Chapter 2 first as an overview and then use it later as an easy reference.

Chapter 3 and Chapters 6 through 13 discuss creating your application jobs and selecting features of the subsystems that you want to include in your application system. You should read Chapters 3 and 6 and some or all of Chapters 7 through 13, depending on which subsystems you are including in your application system.

Chapter 4 describes locating a test system in RAM. Use it when you are building your first system. It also contains step-by-step instructions, but in much more detail than in Chapter 2.

Chapter 5 describes the modification you must make to your RAM configuration in order to make it a ROM/RAM configuration. Use it in conjunction with Chapter 4 to create either an intermediate or final system.

Appendix A contains a sample configuration session for a RAM-based system.

Appendix B describes the process of burning the Nucleus code into PROM.

Appendix C lists the system call requirements of each of the optional subsystems.

CHAPTER 2. PROCEDURAL OVERVIEW

The process of defining and building an iRMX 86 application system involves a number of steps. The following overview illustrates the main points and refers you to appropriate sections of this manual for detailed descriptions.

1. Build and generate a configuration file for each subsystem that you are going to include in your system. Refer to Chapter 6 for a discussion of the Nucleus, Chapter 7 for the Terminal Handler, Chapter 8 for the Debugger, Chapter 9 for the Basic I/O System, Chapter 10 for the Application Loader, Chapter 11 for the Bootstrap Loader, Chapter 12 for the Extended I/O System, and Chapter 13 for the Human Interface.

2. Write code for and compile all application jobs that you want to be a part of the application system. Refer to Chapter 3 for further information.

3. Prepare the memory map for your system. Refer to the "General System Layout" section of Chapter 4 for further information.

4. Iteratively link and locate each subsystem and first-level application job in your application system. Record the pertinent information on the memory map. Refer to the "Iterative Link and Locate Process" section of Chapter 4 for further information.

5. Build a configuration file containing a %JOB macro for each subsystem and first-level application job, one or more %SAB macros, and one %SYSTEM macro. Refer to the "Build The Configuration File" section of Chapter 4 for further information.

6. Assemble the configuration file and link and locate the root job. Refer to the "Generate The Root Job" section of Chapter 4 for further information.

7. Using the ICE-86 in-circuit emulator or the iSBC 957A package, load the system into RAM. Refer to the "Load and Test The System" section of Chapter 4 for further information.

8. Test and debug the system in RAM. Refer to the "Load And Test The System" section of Chapter 4 for further information.

9. Lay out a ROM/RAM system, but load it into RAM for testing. Refer to Chapter 5 for further information.

10. Load the final system into ROM/RAM. Refer to Chapter 5 for further information.

CHAPTER 3. PREPARING JOBS FOR SYSTEM CONFIGURATION

Building a system involves linking and locating each job in the iRMX 86 application system and providing the root job with information concerning the first-level jobs. Before you begin this process, you must make sure that the pieces of the system, the Intel-supplied jobs and your user jobs, are ready to be combined. This involves the following of two operations.

- Preparing application jobs
- Preparing the subsystems

When you begin the configuration process, you do not need to have all the jobs in your system written, because configuration can be an iterative process. That is, you can build your initial system with only one application job, and test it, before adding more jobs into the system. In this way you can build on a stable system. Or, if all your application jobs are available, you can build your initial system with all of your application jobs present, and test and debug the entire system at once. Regardless of how you build your application system, the information provided in this chapter allows you to integrate jobs easily into the iRMX 86 environment.

PREPARING APPLICATION JOBS

You can write the code for your application tasks in either PL/M-86 or assembly language. This manual assumes that you are using PL/M-86. In order to use assembly language, you must use version 3.0 of the 8086/8087/8088 Macro Assembly Language andadhere to the PL/M-86 calling conventions. These are described in the appropriate 8086/8087/8088 MACRO ASSEMBLER OPERATING INSTRUCTIONS manual. The iRMX 86 PROGRAMMING TECHNIQUES manual also contains information to help you write assembly language tasks.

If you have any problems using the PL/M-86 language or compiling PL/M-86 code, refer to the appropriate PL/M-86 manual, either the PL/M-86 PROGRAMMING MANUAL FOR 8080/8085-BASED DEVELOPMENT SYSTEMS, the PL/M-86 COMPILER OPERATING INSTRUCTIONS FOR 8085-BASED DEVELOPMENT SYSTEMS, or the PL/M-86 USER'S GUIDE FOR 8086-BASED DEVELOPMENT SYSTEMS. However, in order to make use of the features of the iRMX 86 Operating System, you must follow instructions additional to those provided in the PL/M-86 manuals when writing your code. The following sections provide this information.

LANGUAGE REQUIREMENTS

Note the following language requirements when writing your task code:

- Designate all of your tasks as procedures. Do not use main modules in your application system.
- If you are compiling your PL/M-86 code using any model other than large, specify the ROM compiler control. This causes the compiler to place the CONST segment in the CODE class, where it can be more easily loaded into ROM. You do not need to specify the ROM control for those programs compiled using the large model, because the compiler automatically does this for the large model.
- Be careful of using the DATA and INITIAL statements. The DATA statement is valid only if you are using the PL/M-86 large model of computation or if you specify the ROM compiler control. The INITIAL statement cannot be used in a procedure if you are going to place that procedure in ROM. It can be used, however, if you are going to use the Bootstrap Loader or the Application Loader to load the procedure.

INCLUDE FILES

There are a number of files contained on the iRMX 86 release diskettes that can be included with your PL/M-86 procedures at compilation time. You must include some or all of these files, depending on the subsystems your programs make use of. Table 3-1 lists these files according to type and subsystem. You must include the files in the compilation of your procedures if those procedures use the system calls of the associated subsystems. For example, if your procedures make Nucleus and I/O System system calls, then you must include the four files associated with those subsystems in the compilation of your procedures.

The INCLUDE files with the extension EXT contain the external PL/M-86 declarations that procedures need in order to use the system calls of the associated subsystems. You can copy these files, edit them, and eliminate the external declarations for system calls that you do not use in your procedures. This can prevent dynamic storage overflow in the compiler. Refer to the iRMX 86 PROGRAMMING TECHNIQUES MANUAL for further information.

The INCLUDE files with the extension LIT contain the PL/M-86 declarations and assignments for the subsystem condition codes.

To include the necessary files in the compilation of your procedures, use the PL/M-86 $INCLUDE control. Both the PL/M-86 COMPILER OPERATING INSTRUCTIONS FOR 8080/8085-BASED DEVELOPMENT SYSTEMS and the PL/M-86 USER'S GUIDE FOR 8086-BASED DEVELOPMENT SYSTEMS describe this control.

Table 3-1. INCLUDE Files

SUBSYSTEM	EXTERNAL DECLARATIONS FILE	CONDITION CODE FILE
Nucleus	NUCLUS.EXT	NEXCEP.LIT
I/O System	IOS.EXT	IEXCEP.LIT
Application Loader	LOADER.EXT	LEXCEP.LIT
Extended I/O System	EIOS.EXT	EEXCEP.LIT
Human Interface	HI.EXT	

SIZE CONTROL CONSIDERATIONS

Part of the configuration process requires selectively locating various modules of the system (as described in Chapter 4). When you use class names on segment declarations you simplify this procedure. The PL/M-86 compiler provides standard class designators for various segments of a program module. However, when writing your application code, be aware that the assignment of segments, classes, and groups in the PL/M-86 output module varies according to the size control specified with the PL/M-86 compiler call. (Refer to the PL/M-86 COMPILER OPERATING INSTRUCTIONS FOR 8080/8085-BASED DEVELOPMENT SYSTEMS or the PL/M-86 USER'S GUIDE FOR 8086-BASED DEVELOPMENT SYSTEMS for details.)

The size control of the PL/M-86 compiler call also determines how certain registers are initialized. The next chapter discusses this in detail. It is recommended that you use the same PL/M-86 size control for all of your PL/M-86 jobs, and that any assembly language modules be compatible with this control.

INITIALIZATION

When you configure an application system, you specify an initial system consisting of a root job and several first-level jobs. When the system starts executing, the Nucleus creates the root job. A task in the root job, the root task, then creates each of the first-level jobs. Tasks in the first-level jobs create the remainder of the application system.

When created, each first-level job contains only a single task. That single task creates or starts the creation of all other objects required by the first-level job. Thus it is referred to as the <u>initialization task</u> for its job, even though it may perform other functions as well. It is important for you to synchronize the operation of each initialization task with that of the root task to ensure proper functioning of your application system.

The root task is structured so that it creates the first-level jobs one at a time. It contains a programming loop that in general performs the following:

```
Repeat for each first-level job

     1. Create first-level job

     2. Suspend root task (until resumed by a first-level job)

Until finished

End
```

Each time the root task creates a first-level job, the root task suspends itself to allow the initialization task in the new job to perform synchronous initialization. <u>Synchronous initialization</u> consists of functions that must be performed immediately, before some other first-level job is created. Typically, this requires creating objects or making resources available that tasks in first-level jobs not yet created expect to be available when they themselves are created. (For example, the initialization task in the I/O System job must create the entire I/O System before it can allow the root task to create other first-level jobs that might make use of I/O System functions.)

When the initialization task finishes its synchronous initialization, it must inform the root task that it is finished, so that the root task can resume execution and create another first-level job. The initialization task must always inform the root task that it has completed its synchronous initialization process by making the following procedure call:

```
CALL RQ$END$INIT$TASK;
```

This procedure call is not described in any other manual. It requires no parameters. When you call this procedure, the root task resumes execution, allowing it to create the next first-level job. You must include a call to RQENDINIT$TASK in the initialization task of each of your first-level jobs, even if the jobs require no synchronous initialization. If one of the first-level tasks does not include this call, the root job remains suspended and cannot create any of the remaining first-level jobs. File NUCLUS.EXT contains the external declaration for this RQENDINIT$TASK and the Nucleus interface library (described in Chapter 4) contains its code.

The amount of synchronous initialization that an initialization task must do depends on your job structure. You may require some of your initialization tasks to create all of the offspring jobs and a number of other objects before calling RQENDINIT$TASK. Some others may only have to perform one or two functions, call RQENDINIT$TASK, and then resume the process of initialization asynchronously. Still other initialization tasks may not have any synchronous initialization requirements and so can call RQENDINIT$TASK before performing any initialization. You must determine how the pieces of your system interact, and how they must be synchronized.

Another important factor in initialization is the order in which the root job creates the first-level jobs. The amount of processing your initialization tasks must do before calling RQENDINIT$TASK may depend on which jobs the root task has already created and which jobs it has yet to create. The order in which the root task creates first-level jobs depends on the order that you specify these jobs in a configuration file, not on the priority of the tasks in those jobs. (Refer to the description of the %JOB macro in Chapter 4.) Always specify the %JOB calls for the subsystems first, so that they are created and initialized first and are available to all other jobs. The order in which you specify your other first-level jobs depends on your application system.

You should always use RQENDINIT$TASK as described in this section in order to perform your synchronous initialization. Do not attempt to accomplish the same function by temporarily raising and lowering task priorities. The iRMX 86 Operating System does not guarantee that your tasks will execute in the correct order if you use priorities to determine initialization order.

PREPARING THE SUBSYSTEMS

The subsystems have been created in a manner that allows you to choose system calls and features that you want to have available in your system. To prepare them, you must create tables that select or omit features. You must create these tables in a format understandable to the 8086/8087/8088 Macro Assembler, assemble them, and link them to the subsystems. The details of preparing individual subsystems are contained in Chapters 6 through 11.

CHAPTER 4. LOCATING A TEST AND DEVELOPMENT SYSTEM IN RAM

After you have prepared your application jobs and the subsystems, you should locate your first system entirely in RAM to facilitate testing and debugging of your programs. It is much easier to test and debug your programs in RAM than it is to continually reburn your PROMs when you detect errors. After debugging in RAM, you can locate the final system in ROM/RAM or copy it to a secondary storage device and load it with the Bootstrap Loader.

Putting together a RAM-based system consists of the following steps:

1. Laying out the system
2. Linking and locating the Nucleus and application jobs
3. Building the configuration file
4. Generating the root job
5. Loading and testing the system

If you wish, the linking and locating can be separate steps. That is, you can use LINK86 to link any or all of your subsystems and jobs before ever starting the locate process. However, because the release diskettes for each subsystem contain SUBMIT files that link and locate the subsystems in one step, this manual considers the link and locate processes together. The following sections discuss the steps in more detail.

GENERAL SYSTEM LAYOUT

Linking and locating a system is an iterative process. That is, you must link and locate one first-level job and the offspring jobs, examine the locate maps to determine the ending address, and use that information to link and locate the next first-level job and offspring jobs. Therefore, before you use LINK86 to link the pieces of your system together and LOC86 to assign absolute memory addresses, you must decide where in memory to start locating the pieces, and in which order to locate them. The following sections discuss the various factors that you must consider when laying out your system.

NOTE

This manual assumes that you link each first-level job together with its offspring jobs to produce a single link module, referred to as the first-level job, which you then locate at an absolute address. If you do not link your jobs together, you should follow the same locate procedure outlined in this chapter, but locate every job, not just the first-level jobs.

SYSTEM TYPE

At first, when creating an initial test system, you should locate all of your modules in RAM. This allows you to lay out the system on a job-by-job basis. You can locate all segments associated with one job (code segments, data segments, etc.) sequentially in RAM and locate all segments of the next job following the first.

Later, if you locate a final ROM/RAM system, you must locate the system by class, not by job. You must locate the code classes from all of the jobs at ROM addresses, and the data, stack, and memory classes at RAM addresses. Chapter 5 describes locating a ROM/RAM system. For now, however, lay out your system on a job-by-job basis, totally in RAM.

HIGH AND LOW LOCATION OF MODULES

You can locate the system either high or low in memory. When locating high in memory, assign the first module to the numerically largest absolute memory locations and the succeeding modules to numerically smaller locations. When locating low in memory, assign the first module to the numerically smallest memory locations and the succeeding modules to numerically larger locations. This manual assumes low location of modules because it is the easiest method to use with the iterative link and locate process that this chapter describes.

MODULE ORDER

The order in which you lay out your modules in memory should depend on the relative stability of the modules. You must later create a system configuration file that contains addresses of various parts of your system. If you can keep the stable portions of your system located at constant addresses, you can minimize modifications to the system configuration file.

Although you can change the sizes of the Nucleus and the subsystems by reconfiguring them, it is likely that their sizes will remain constant during your development cycle. Therefore, locate these modules first. Locate any of your application jobs that are subject to change later in memory, so that their size fluctuations do not necessitate changing the addresses of the other modules. In general, lay out your system as shown in Figure 4-1.

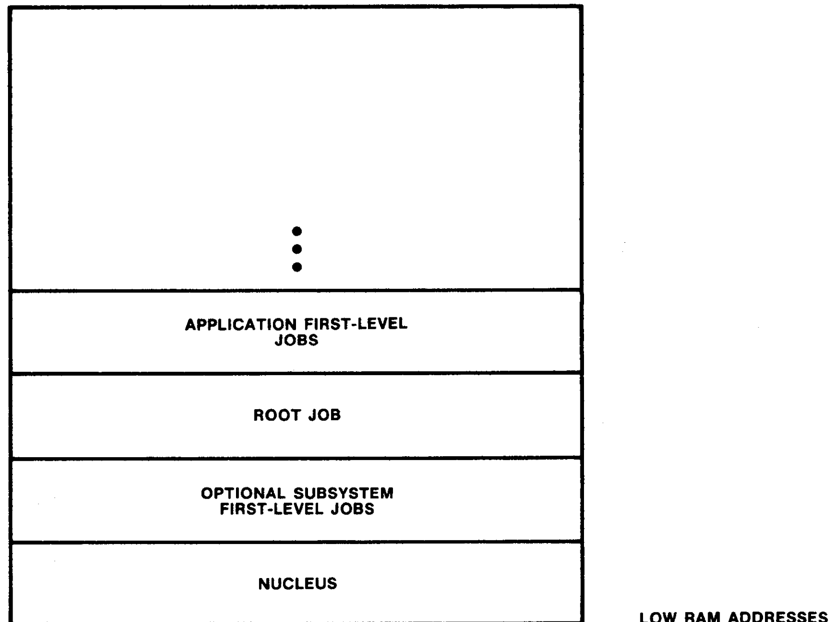


Figure 4-1. General System Layout

Notice that Figure 4-1 illustrates locating the root job. Later sections of this chapter discuss the root job. For now, just note its position in the system layout.

As you continue to test and debug your system, you may wish to create a fairly stable system with just a few user jobs and use that system as a base on which to build, testing and debugging each new job you add before adding others. If so, use the layout shown in Figure 4-1 for your first system and locate any new jobs you add at the end, where there is space available for them to grow during the debugging period.

PREPARING A MEMORY MAP

After you have decided in general how to lay out your system, prepare a memory map to indicate this. Figure 4-2 is a worksheet that you can use for this. To prepare a memory map, do the following:

1. In the right column of the worksheet, record the highest RAM address in your system. Notice that addresses corresponding to the beginning and end of the interrupt vector and reset vector have already been recorded. If you intend to use the iSBC 957A package to load your system into the iSBC 86/12A, reserve the locations 40:0 through 6F:F for the iSBC 957A monitor.

2. In the center column of the worksheet, list the modules in the order you wish to locate them, one to a line, with the first module (the Nucleus) closest to the bottom of the worksheet.

3. In the right column, on the same line as the first module, record the lowest available RAM address. Use this value when locating the first module.

After you have made this map, use it during the link and locate procedure to condense the important information from the locate maps. You can use it to record the starting and ending locations of the modules, as well as other important information, such as entry points and data segment bases.

iRMX 86™ SYSTEM MEMORY MAP WORKSHEET

Configuration file name: ______________________________

Start address/ Data segment base	Module	Length	Absolute Address
			FFFF:F
	(reserved)		
	reset vector		
			FFFF:0
			______
			______
			______
			______
			______
			______
			______
			______
			______
			______
			______
			______
			______
			______
			______
			40:0
	Interrupt vector		
			0:0

Figure 4-2. System Memory Map

Example:

This example shows how to prepare a memory map worksheet for a sample system. The sample system has the following characteristics:

- 128K of contiguous RAM exists in the system. Thus the highest RAM address is 1FFF:F.
- The system consists of four modules: the Nucleus, the Debugger, a first-level application job, and the root job.
- The iSBC 957A package is going to be used to load the system into memory. Therefore, the lowest available address is 70:0. This system starts with 1C0:0 as the first address to allow for future inclusion of the Bootstrap Loader (although if you are going to use the Bootstrap Loader with a device similar to to an iSBC 215 device, you should allow more room and start at 200:0).

Figure 4-3 shows a prepared memory map for this system. When the modules are located, actual addresses can be recorded on this worksheet.

iRMX 86™ SYSTEM MEMORY MAP WORKSHEET

Configuration file name: ______________________________

Start Address/ Data segment base	Module	Length	Absolute Address
			FFFF:F
	(reserved)		
	reset vector		FFFF:0
			1FFF:F
	Highest RAM address		______
			______
			______
	Application Job		______
			______
			______
	Root Job		______
			______
			______
	Debugger		______
			______
			______
	Nucleus		1C0:0
			______
			40:0
	Interrupt vector		0:0

Figure 4-3. Example Memory Map Worksheet

LINK AND LOCATE THE SUBSYSTEMS AND APPLICATION JOBS

After you have laid out your system, you can begin the iterative process of linking and locating your system. The following sections discuss this process. The first two sections discuss the procedures used to link and locate individual subsystems and application jobs. The third section describes how you must combine these individual link and locate processes and put together the entire application system.

LINKING AND LOCATING THE SUBSYSTEMS

The release diskette for each subsystem contains a SUBMIT file that assembles the subsystem's configuration files, links the necessary modules together, and locates the subsystem at an absolute address. These SUBMIT files can be run on a Series II Microcomputer Development System. Chapters 6 through 11 identify the SUBMIT files for each subsystem and describe their parameters. This section provides an overview of the general process, describes some restrictions that you must adhere to in order to use the SUBMIT files as released, and describes the modifications you must make to the SUBMIT files if you run them on a Series III Microcomputer Development System.

Overview

Figure 4-4 illustrates the procedure that most of the subsystem SUBMIT files follow in order to produce linked and located subsystems. This procedure includes assembling (or compiling) the subsystem configuration file or files, linking the object files together with the subsystem library or libraries (which contain the actual code for the subsystem) and any necessary interface libraries, and locating the resulting link module at absolute addresses. You can examine the individual SUBMIT files to determine the commands used, if you wish. However, after you have modified your subsystem configuration files to reflect your desired system, prepared the diskettes correctly, and placed them in the proper Series II development system disk drives, you need only use the ISIS-II SUBMIT command to run the SUBMIT files. (If you use a Series III development system, you will have to make additional modifications to the SUBMIT files in order to take advantage of the Series III capabilities.)

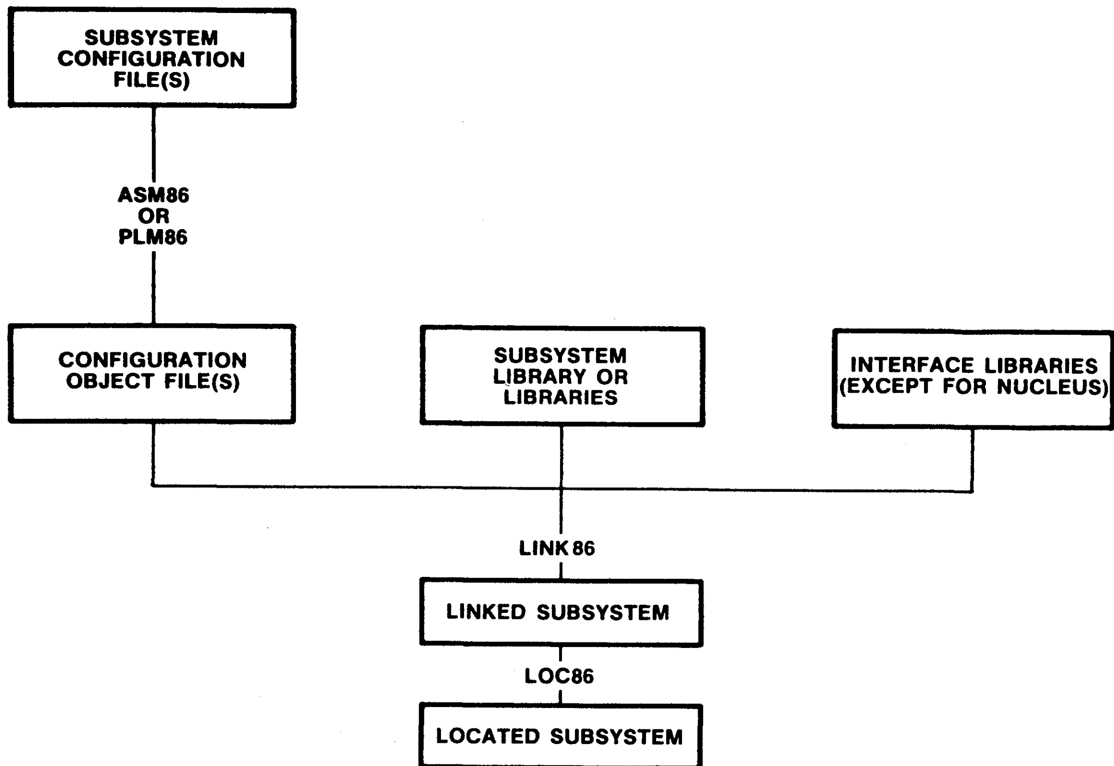


Figure 4-4. Subsystem SUBMIT File Procedure

Preparing Diskettes

When you configure individual subsystems, you should never make modifications to the actual release diskettes. If you want to change any of the files on the release diskettes, such as the subsystem configuration files (named file.A86 or file.P86) or the SUBMIT files (named file.CSD), copy these files to another diskette first.

Placing Diskettes in the Proper Drives

In order to use the subsystem SUBMIT files as they are released, you must place your diskettes in the proper drives of the INTELLEC development system. All subsystem SUBMIT files assume that you have a four drive development system and that you have placed diskettes in the drives as follows:

F0 A system disk containing LINK86, LOC86, ASM86 (version 3.0), and/or PLM86, as well as COPY, DELETE, and SUBMIT.

F1 A diskette on which you should place any modified versions of configuration files and copies of all SUBMIT files. The SUBMIT files also write the located code for the subsystems to this diskette. If you need to make changes to any file on a release diskette, you should first copy it from the release diskette to this diskette and then make the changes. If you copy a configuration file to this diskette and make changes to it, you will also have to make changes to its associated SUBMIT file to correct the drive number for the configuration file. You should copy all SUBMIT files to this diskette before entering the SUBMIT command, so that you can leave your release diskettes in a write-protected state.

F2 The subsystem release diskette. The SUBMIT file reads libraries and INCLUDE files from this diskette, but does not modify the diskette in any way.

F3 A temporary and listing diskette. The SUBMIT file writes all intermediate files (such as link files) to this diskette and deletes them when it no longer needs them. It also writes all listing files, link maps, and locate maps to this diskette.

If you do not have a four-drive development system, or if your drives are set up with different disk mnemonics, you must modify the subsystem SUBMIT files and the subsystem configuration files to accomodate this.

Using a Series III Development System

If you use a Series III Microcomputer Development System in your configuration process, your versions of PLM86, ASM86, LINK86, and LOC86 run under 8086 execution mode. Thus you will have to modify the configuration files and SUBMIT files in order to make them run correctly on your development system. Use the following guidelines when making these modifications.

Guidelines for Configuration Files

- Ensure that each assembly language configuration file (a file whose name is of the form "file.A86") contains the NAME directive. This directive has the form:

  NAME modname

  where modname is the name of your module. The one restriction on modname is that it must be different from all other modname values in the linked system. Place this NAME directive at the beginning of the file. Refer to the 8086/8087/8088 MACRO ASSEMBLY LANGUAGE REFERENCE MANUAL FOR 8086-BASED DEVELOPMENT SYSTEMS for more information concerning the NAME directive.

Guidelines for SUBMIT Files

- For each SUBMIT file (files whose names are of the form "file.CSD"), add RUN before each ASM86, PLM86, LINK86, and LOC86 command or put a RUN/EXIT pair around every set of ASM86, PLM86, LINK86, and LOC86 commands. Refer to the INTELLEC SERIES III MICROCOMPUTER DEVELOPMENT SYSTEM CONSOLE OPERATING INSTRUCTIONS for more information about the RUN and EXIT commands.

- Add the NOINITCODE control to every LOC86 command in your SUBMIT files. Refer to the iAPX 86,88 FAMILY UTILITIES USER'S GUIDE FOR 8086-BASED DEVELOPMENT SYSTEMS for more information about this control.

- Remove the DATE control from every ASM86 and PLM86 statement. In most cases, this means that you will also be removing one of the formal parameters from the SUBMIT file. Therefore, when you invoke the SUBMIT file, do not specify the date parameter, but include the comma (,) as a placeholder.

- Remove the MACRO control from the ASM86 statements in all SUBMIT files.

LINKING AND LOCATING APPLICATION JOBS

The most common method of linking and locating your application jobs is to link the first-level job together with every job ultimately created by that first-level job and one or more interface libraries. You must then locate this module at an absolute address. Figure 4-5 illustrates this link and locate procedure.

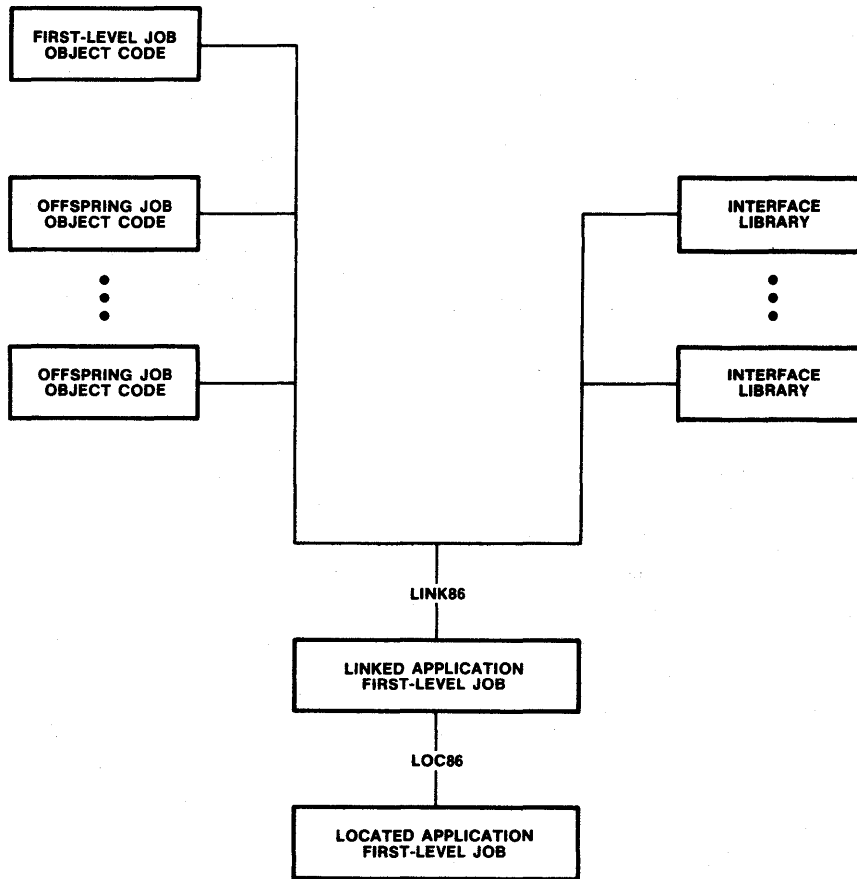


Figure 4-5. Application Job Link and Locate Procedure

If you do not link your first-level jobs together with their offspring jobs, you should link and locate the offspring jobs first. By doing this, you can obtain the absolute starting locations of the offspring tasks from the locate maps and specify these values in the CREATE$JOB and CREATE$TASK calls of their parent tasks before compiling the parents.

The following sections describe the individual link and locate commands in more detail, and describe the interface libraries. The guidelines discussed in the "Using a Series III Development System" section of this chapter also apply to these link and locate commands.

Linking Application Jobs

The LINK86 command is used to link your application jobs. This command is described in detail in the appropriate iAPX 86,88 FAMILY UTILITIES USER'S GUIDE. However, the format of the LINK86 command that you must enter is:

```
    LINK86                            &
        :fx:app_job.obj,              &
        :fx:interface.lib             &
     TO      :fx:app_job.lnk  MAP  PRINT(:fx:app_job.mpl)
```

where:

fx	The appropriate disk mnemonic, indicating where the file resides.
app_job.obj	Object code for your application job. You do not need to provide this code on one file; you can link in several files or libraries at this point.
interface.lib	Interface libraries for the subsystems that your jobs make use of. You may have to link in several libraries at this point. These interface libraries are described in later paragraphs of this section.
app_job.lnk	Name of the file in which LINK86 places the module containing your linked application code. Use this file as the input file when locating your application job.
app_job.mpl	Name of the file in which LINK86 writes the link map for the application job.

During the link process, you must link in a number of interface libraries. These libraries contain the routines that satisfy external references to system calls that you make in your application code. The number and names of the libraries that you must link in with your application code depend on which subsystems your jobs use and which model of PL/M-86 computation the jobs were compiled under. Table 4-1 shows the correlation between subsystems, models of computation, and interface libraries. Specify these libraries as the last modules in the LINK86 input list so that they can satisfy references from all linked modules. Notice that no library exists for the small model of PL/M-86 computation; the iRMX 86 Operating System does not support applications compiled in small.

Table 4-1. Interface Libraries as a Function of PL/M-86 Models and Subsystems

	COMPACT	LARGE OR MEDIUM
Nucleus	RPIFC.LIB	RPIFL.LIB
Basic I/O System	IPIFC.LIB	IPIFL.LIB
Application Loader	LPIFC.LIB	LPIFL.LIB
Extended I/O System	EPIFC.LIB	EPIFL.LIB
Human Interface	HPIFC.LIB	HPIFL.LIB

Locating Application Jobs

After you have used LINK86 to generate a link module for your application job, you must use LOC86 to bind this link module to absolute addresses. The appropriate iAPX 86,88 FAMILY UTILITIES USER'S GUIDE contains specific instructions on the use of the LOC86 command.

Since you are laying out your test system by job rather than by class, use a combination of the ORDER and ADDRESSES controls on LOC86 to simplify the location process. Use the ORDER control to declare the order in which the classes of the job are to be located. Then declare the absolute address of the code class with the ADDRESSES control. LOC86 automatically locates the rest of the classes following the code class. If you do this, a call to LOC86 appears similar to the following:

```
LOC86   input_file  TO  output_file                      &
        ORDER (CLASSES (CODE, DATA, STACK, MEMORY))      &
        SEGSIZE (STACK (stack_size))                     &
        ADDRESSES (CLASSES (CODE (absolute_address)))    &
        MAP  PRINT (map_file)                            &
        OBJECTCONTROLS (NOLINES,NOCOMMENTS,NOPUBLICS,    &
                        NOSYMBOLS)
```

where:

input_file — Name of link file produced previously by LINK86.

output_file — Name of the file in which LOC86 writes the absolute module.

stack_size — Size of this job's stack. Use this control for those jobs requiring a statically allocated stack. A stack is statically allocated if you, and not the Operating System, specify a stack location and size. A minimum value of 200H should be specified, if this control is required; otherwise specify zero. It is recommended that you specify zero for this parameter and let the Nucleus dynamically allocate a stack whenever possible. This depends, however, on the model of PL/M-86 computation that you used when compiling your code. With dynamically allocated stacks, you specify the stack size in the %JOB macro call. Refer to the "%JOB Macro" section of this chapter for further information.

absolute_address — Absolute starting location of the code segment of the job. You can obtain this address by examining the locate map of the previously located module. Refer to the next section of this chapter for further information about determining absolute addresses.

map_file — Name of the file in which LOC86 writes the locate map. Always generate the locate map. You need it in order to determine where to locate the next module. It also contains information that you need in order to generate the configuration file (as described in the "Build the Configuration File" section of this chapter).

Use this form of the LOC86 command to locate each application job.

THE ITERATIVE LINK AND LOCATE PROCESS

As mentioned before, the link and locate process is an iterative process. You must link and locate one job, examine its locate map to determine its ending address, and use that information to link and locate the next job. This process can be broken down into the following steps:

1. Link and locate the Nucleus first by submitting the NUCLUS.CSD SUBMIT file (described in Chapter 6). A parameter to this SUBMIT file is used to assign the Nucleus to absolute memory locations. Specify the lowest available memory locations for the Nucleus (1C00 or 2000, depending on the type of device, if you are using the Bootstrap Loader to load your system into memory; 800 if you are using the iSBC 957A package; or 400 otherwise).

2. Determine the ending address of the Nucleus from the locate map generated by LOC86. Record this value in the memory map.

3. Using the next available address as input to a subsystem SUBMIT file (described in Chapters 7 through 13), link and locate the next subsystem.

4. Determine the starting and ending addresses from the locate map. Record these values in the memory map. Also record the entry point address and data segment base in the leftmost column. You will use these values later when creating the configuration file.

5. Go back to step 3 and continue until you have linked and located all of the subsystems.

6. Use the next available address as the starting address for the root job. You do not have to link or locate the root job at this point. Instead, assume a length of 600H bytes for it and leave that amount of space. (This length is adequate for an application system with approximately 20 first-level jobs. If your system contains more than 20 first-level jobs, you should allow more space for the root job.) Record the starting and ending addresses on the memory map.

7. Using the next available address as input to the ADDRESSES control of the LOC86 command, link and locate the first application job.

8. Determine the starting address, the ending address, the entry point address, and the data segment base from the locate map and record these values in the memory map.

9. Instead of using the next available address as input to the ADDRESSES control, leave some space between application jobs so that these jobs can grow during the debugging process, if needed. Use your own judgement as to how much space to leave, but if you are unsure, leave approximately 1K bytes for growth. Add this padding factor to the ending address of the previously located module and record that figure as the starting address of the next module.

10. Using the starting address recorded on the memory map as input to the ADDRESSES control of LOC86, link and locate the next application job.

11. Go back to step 8 and continue until you have located all application jobs.

After you perform this procedure once, you can create an additional SUBMIT file to locate all of the modules at once. This procedure can contain LINK86 commands for those jobs that may have to be relinked. The padding factors that you included when assigning memory locations allow the modules to grow during the debugging process without affecting the LOC86 commands used to locate them.

NOTE

The locate map for the Nucleus always contains a warning message similar to the one shown in the following example. This is a normal message for the Nucleus. It does not indicate errors in the LOC86 command.

Example:

This example assembles the Nucleus configuration file, links and locates the full Nucleus, and records values from the locate map in the memory map worksheet. It makes the following assumptions about the system:

- Disk drive F0 on the INTELLEC Series II Development System is a system disk containing LOC86.
- Disk drive F1 contains the user's configuration diskette. This diskette contains a copy of the Nucleus SUBMIT file NUCLUS.CSD.
- Disk drive F2 contains the Nucleus release diskette.
- Disk drive F3 contains a diskette on which the SUBMIT file can place temporary and intermediate files.
- The Nucleus is going to be located at address 1C00H.

The following command calls a SUBMIT file to assemble the Nucleus configuration file and link and locate the Nucleus. Refer to Chapter 6 for a complete description of the SUBMIT file.

```
SUBMIT :F1:NUCLUS(04-01-81, 1C00H)
```

The located object module is written to file NUCLUS on drive F1 and the locate map to file NUCLUS.MP2 on drive F3. Figure 4-6 shows a part of this locate map. This map should be viewed as an example only. It may differ from the one generated when you locate the Nucleus.

```
ISIS-II MCS-86 LOCATER, V1.3 INVOKED BY:
LOC86                                        &
      :f3:nuclus.lnk  TO :f1:nuclus          &
      MAP PRINT(:f3:nuclus.mp2)              &
      NOLINES NOCOMMENTS NOSYMBOLS           &
      SEGSIZE(stack(0))                      &
      ORDER(classes(code, data))             &
      ADDRESSES(classes(code(01C00H)))
WARNING 26:  DECREASING SIZE OF SEGMENT
  SEGMENT:  STACK

SYMBOL TABLE OF MODULE NBEGIN
READ FROM FILE :F3:NUCLUS.LNK
WRITTEN TO FILE :F1:NUCLUS

BASE   OFFSET TYPE SYMBOL                    BASE   OFFSET TYPE SYMBOL

01C0H  0000H  PUB  NBEGIN

MEMORY MAP OF MODULE NBEGIN
READ FROM FILE :F3:NUCLUS.LNK
WRITTEN TO FILE :F1:NUCLUS

SEGMENT MAP

START     STOP      LENGTH ALIGN NAME              CLASS

00000H    003FFH     0400H   A   (ABSOLUTE)
01C00H    07A79H     5E7AH   W   CODE              CODE
07A7AH    07A93H     001AH   W   OBJ_SEG           CODE
07A94H    07A9DH     000AH   W   JOB_SEG           CODE
07A9EH    07AB1H     0014H   W   TASK_SEG          CODE
07AB2H    07AB9H     0008H   W   MB_SEG            CODE
07ABAH    07AC1H     0008H   W   SEM_SEG           CODE
07AC2H    07ACBH     000AH   W   REG_SEG           CODE
07ACCH    07AD9H     000EH   W   FS_SEG            CODE
07ADAH    07AF3H     001AH   W   INT_SEG           CODE
07AF4H    07AF9H     0006H   W   EXCEP_SEG         CODE
07AFAH    07B29H     0030H   W   VALID_SEG         CODE
07B2AH    07B2EH     0005H   W   PIC_CNF_SEG       CODE
07B30H    07B41H     0012H   W   _IMR_PORT         CODE
07B42H    07B53H     0012H   W   _EOI_PORT         CODE
07B54H    07B65H     0012H   W   _ISR_READ_PORT    CODE
07B66H    07B6EH     0009H   B   _PIC_INFO         CODE
07B6FH    07B78H     000AH   B   TIMER_CNF_SEG     CODE
07B7AH    07B7AH     0000H   W   CSEG              CODE
07B7AH    07B7BH     0002H   W   SLAVE_SEG         CODE
07B7CH    07DE3H     0268H   W   DATA              DATA
07DF0H    07DFBH     000CH   G   READYLISTROOT_    DATA
                                -DATA
07DFCH    07EFBH     0100H   W   KSTACK            DATA
```

Figure 4-6. Example Nucleus Locate Map

```
07F00H    07F0DH     000EH    G    DELAYLISTROOT_    DATA
                                  -DATA
07F10H    07F61H     0052H    G    IDLETASK_DATA     DATA
07F70H    07F7FH     0010H    G    INTVEC_REG_SEG    DATA
07F80H    07F8FH     0010H    G    EXT_REG_SEG       DATA
07F90H    07F9FH     0010H    G    JOB_REG_SEG       DATA
07FA0H    07FAFH     0010H    G    SEM_REG_SEG       DATA
07FB0H    07FBFH     0010H    G    MAIL_REG_SEG      DATA
07FC0H    07FCFH     0010H    G    OD_REG_SEG        DATA
07FD0H    07FDFH     0010H    G    POOL_REG_SEG      DATA
07FE0H    07FEFH     0010H    G    DELETION_REG_S    DATA
                                  -EG
07FF0H    07FFFH     0010H    G    ??SEG
08000H    08000H     0000H    G    LIB_87_PUB
08000H    08003H     0004H    G    LIB_87_INIT
08004H    08004H     0000H    W    STACK             STACK
08010H    0813BH     012CH    G    ISTACK            STACK
0813CH    0813CH     0000H    W    MEMORY            MEMORY


GROUP MAP

ADDRESS   GROUP OR SEGMENT NAME
07B70H    DGROUP
          DATA
          READYLISTROOT_DATA
          KSTACK
          DELAYLISTROOT_DATA
          IDLETASK_DATA
          ISTACK
          INTVEC_REG_SEG
          EXT_REG_SEG
          JOB_REG_SEG
          SEM_REG_SEG
          MAIL_REG_SEG
          OD_REG_SEG
          POOL_REG_SEG
          DELETION_REG_SEG
01C00H    CGROUP
          CODE
          OBJ_SEG
          JOB_SEG
          TASK_SEG
          MB_SEG
          SEM_SEG
          REG_SEG
          FS_SEG
          INT_SEG
          EXCEP_SEG
          VALID_SEG
```

Figure 4-6. Example Nucleus Locate Map (continued)

```
PIC_CNF_SEG
_IMR_PORT
_EOI_PORT
_ISR_READ_PORT
_PIC_INFO
TIMER_CNF_SEG
CSEG
SLAVE_SEG
```

Figure 4-6. Example Nucleus Locate Map (continued)

Notice the warning message contained on the locate map. This is a normal message that always occurs when you locate the Nucleus. Do not be alarmed by it. It does not indicate an error.

As you can see from arrow A in Figure 4-6, the next available memory location is 8DF:0. The last location used by the Nucleus is 8DE:F. The memory map shown in Figure 4-7 contains these values.

iRMX 86™ SYSTEM MEMORY MAP WORKSHEET

Configuration file name: ____________________

Start address/ Data segment base	Module	Length	Absolute Address
			FFFF:F
	(reserved)		
	reset vector		FFFF:0
			1FFF:F
	Highest RAM address		______
			______
			______
	Application Job		______
			______
			______
	Root Job		______
			______
			______
	Debugger		8DF:0
			______
			8DE:F
	Nucleus		1C0:0
			______
			40:0
	Interrupt vector		0:0

Figure 4-7. Entering the Nucleus End Address on the Memory Map

BUILD THE CONFIGURATION FILE

After you have created the memory map and located the jobs, you are ready to build the system configuration file or modify one of the existing files provided on the subsystem release diskettes. The system configuration file is an assembly language source file which must describe each first-level job, the system address blocks, and the system as a whole. The file must provide this information in the form of macro calls. The macros used in the configuration file are:

```
%JOB
%SAB
%SYSTEM
```

These macros are described individually in the remainder of this section. However, the following instructions apply to the configuration file as a whole and to all of the macros.

- In addition to placing macros in your system configuration file, you must also include two other statements. The first statement in your system configuration file must be an $INCLUDE statement to include the file CTABLE.MAC in the assembly of your configuration file. CTABLE.MAC is on the Nucleus release diskette and contains the definitions of the macros used in the remainder of the configuration file. The format of this statement is:

  ```
  $INCLUDE (:fx:CTABLE.MAC)
  ```

  where fx is the identifier of the drive containing the Nucleus release diskette.

  The last statement in your configuration file must be the END statement. The format of this statement is:

  ```
  END
  ```

- You can include space characters anywhere within your macro calls, in order to provide a more readable configuration file. Space characters consist of blank spaces.

- You can spread your macro calls over several lines (such as placing one parameter on a line) if you specify the comment macro (%') on each line that is continued. An example of this is:

  ```
  %SAB (05000,        %' start base
        0FFFF,        %' end base
        U)
  ```

  You must use these comment macros in order to exclude the carriage return and line feed characters from being processed by the assembler.

- For certain parameters you must specify a value of type addr, offset, or base. These are described as follows:

  addr — An absolute address in the form base:offset.

  base — An absolute paragraph address.

  offset — A value which is offset from the specified base.

  You must supply these values exactly as they appear on the locate map produced by LOC86. The required radix for each of these values is hexadecimal. Therefore, do not include the suffix H when specifying a value. For other parameter types (such as byte or word), you must explicitly specify a radix, with decimal the assumed default.

%JOB MACRO

The %JOB macro is used to specify parameters for first-level jobs. You must specify one %JOB macro for each optional subsystem first-level job and each application first-level job. Figure 4-8 contains the format of the %JOB macro. It is a worksheet that you can use to prepare the macro calls. The values in parentheses are suggested values. Use these suggested values for noncritical parameters.

Macro call: JOB (defines first-level jobs)

Number of calls required: one for each first-level job

CONFIGURATION FILE NAME: ______________________

FORMAT:

parameter	type	suggested default	value
%JOB (directory_size,	word	(0)	________
pool_min,	word		________
pool_max,	word	(0FFFFH)	________
max_objects,	word	(0FFFFH)	________
max_tasks,	word	(0FFFFH)	________
max_task_priority,	byte	(0)	________
exception_handler_entry,	addr	(0:0)	________
exception_handler_mode,	byte	(1)	________
job_flags,	word	(0)	________
init_task_priority,	byte	(0)	________
init_task_entry,	addr		________
data_segment_base,	base	(0)	________
stack_pointer,	addr	(0:0)	________
stack_size,	word	(512)	________
task_flags)	word	(0)	________

NOTES:

1. Type addr is specified as base:offset.

2. Types addr and base must be entered as hexadecimal numbers without the suffix H. Types word and byte default to decimal, but will accept all radix suffixes.

Figure 4-8. %JOB Macro Worksheet

As you can see from Figure 4-8, the format of the %JOB macro is almost the same as the format of the CREATE$JOB system call described in the iRMX 86 NUCLEUS REFERENCE MANUAL. The only difference between the two is that the %JOB macro omits the param$obj parameter and includes the exception_handler_entry and exception_handler_mode parameters in line, rather than as a pointer to a structure containing this data. The other parameters are the same. For completeness, a short description of each of the parameters of the %JOB macro follows. For more detailed information, refer to the description of the CREATE$JOB system call in the iRMX 86 NUCLEUS REFERENCE MANUAL.

directory_size	Maximum allowable number of entries in this job's object directory. A value of zero indicates that no directory is to be created. The maximum value for this parameter is 0FF0H.
pool_min	Minimum allowable size of the job's memory pool, in 16-byte paragraphs.
pool_max	Maximum allowable size of the job's memory pool, in 16-byte paragraphs.
max_objects	Maximum number of objects that can exist simultaneously in the job. A value of 0FFFFH indicates unlimited objects.
max_tasks	Maximum number of tasks that can exist simultaneously in the job. A value of 0FFFFH indicates unlimited tasks.
max_task_priority	Maximum allowable priority of tasks in the job. Specify a value in the range 0 to 255 decimal. A value of zero indicates that the priority of the root task is the maximum allowable.
exception_handler_entry	Pointer to the start address of the job's exception handler. A value of 0:0 indicates that the job uses the exception handler specified in the %SYSTEM macro (described later in this chapter).
exception_handler_mode	Encoded indication of the job's exception mode. Values are interpreted as follows:

value	Pass control to exception handler
0	Never
1	On programming error conditions only
2	On environmental conditions only
3	On all exceptional conditions

Parameter	Description
job_flags	Information that the Nucleus needs to create and maintain the job. Bits in this word are interpreted as follows: bit / meaning 15-2 — Reserved. 1 — If set to 0, the Nucleus validates parameters for all system calls made by tasks in this job or its offspring. Refer to the "Parameter Validation" section of Chapter 6 for a further discussion of parameter validation. If set to 1, the Nucleus does not validate parameters for tasks in this job. 0 — Reserved.
init_task_priority	Priority of this job's initialization task. A value of zero assigns the initialization task a priority equal to the max_job_priority parameter.
init_task_entry	Entry point of this job's initialization task.
data_segment_base	Base value of the initialization task's data segment. A value of zero indicates that the task itself assigns the data segment.
stack_pointer	Address of the initialization task's stack. A value of 0:0 causes the Nucleus to allocate a stack segment to the task and initialize the SS register to the base address of this segment and the SP register to the value of the stack_size parameter. It is recommended that you specify 0:0 for this parameter. This permits dynamic stack allocation and deallocation.
stack_size	Size in bytes of the initialization task's stack segment. Specify 200H as a minimum value for this parameter. You must enter a value for this parameter even if the job uses dynamically allocated stacks.

task_flags — Information that the Nucleus needs to create and maintain the job's initial task. Bits in this word are interpreted as follows:

bit	meaning
15-1	Reserved bits which should be set to zero.
0	If one, the initial task contains floating-point instructions which require the 8087 NDP for execution. If zero, the initial task contains no floating point instructions.

For your PL/M-86 jobs, the parameters that you must specify for each %JOB macro depend on the PL/M-86 size control. The following sections outline the differences with references to the appropriate %JOB macro parameters.

Data Segment Allocation

PL/M-86 large model procedures. Large model procedures have statically allocated data segments. However, you do not have to specify data segment values because the PL/M-86 compiler generates code that automatically initializes the data segment (DS) register for each procedure when that procedure begins executing. Therefore, for large model procedures, set the data_segment_base parameter to zero.

PL/M-86 medium and compact model procedures. These procedures also have statically allocated data segments. However, the compiler does not automatically initialize the data segment register for medium and compact models of computation. Therefore, if you compile a procedure using either of these models, you must specify the base address of that module's DGROUP as the data_segment parameter of the %JOB macro. Obtain the base address from the locate map produced by LOC86. (DGROUP includes the data, stack, and memory segments/classes for the medium model and the data segment/class for the compact model. The constant segment/class is included in CGROUP if the ROM compiler control is used.)

Stack Allocation

PL/M-86 large and compact models. The Operating System must dynamically allocate stacks for procedures compiled in these models. Thus, you must specify 0:0 for the stack_pointer parameter of the %JOB macro. The Operating System allocates a stack to the job with a size that you specify in the stack_size parameter of the %JOB macro.

When you use LOC86 to locate these procedures, you must use the SEGSIZE(STACK(0)) control. Refer to "The Locating Application Jobs" section of this chapter for further information about this control.

PL/M-86 medium models. Procedures compiled using the medium model require statically allocated stacks. Thus, for these procedures, you must specify the address of the stack in the stack_pointer parameter of the %JOB macro. Use the value indicated in the locate map when specifying this parameter. You should also specify the size of the stack in the stack_size parameter of the %JOB macro. Use the same size as you specified in the SEGSIZE(STACK(...) control of the LOC86 command. Refer to "The Locating Application Jobs" section of this chapter for further information concerning the LOC86 command.

%SAB MACRO

The %SAB macro declares the size and location of each system address block. A system address block is an area of addressable memory not available as dynamically reusable memory. This includes ROM, nonexistent memory, and RAM reserved for jobs. The Nucleus needs to know where these reserved areas are so that it does not reassign them. Look at the memory map worksheet that you filled out for your system and use %SAB macro calls to reserve all memory needed for the Nucleus, subsystems, application jobs, interrupt vector, reset vector, iSBC 957A workspace, and ROM.

The format of the %SAB macro is shown in Figure 4-9. This figure is a worksheet that you can use to prepare the macro calls.

Macro call: SAB (for system address blocks)

Number of calls required: one or more

CONFIGURATION FILE NAME: ______________________

FORMAT:

	parameter	type	suggested default	value
%SAB	(start_base,	base		______
	end_base,	base		______
	type)	see note 1	U	______
%SAB	(start_base,	base		______
	end_base,	base		______
	type)	see note 1	U	______
%SAB	(start_base,	base		______
	end_base,	base		______
	type)	see note 1	U	______

NOTES:

1. The type parameter is reserved for future use. Enter the character U for this parameter.

2. A SAB is declared between start_base:0 and end_base:F, inclusive.

3. Types addr and base must be entered as hexadecimal numbers without the suffix H. Types word and byte default to decimal but will accept all radix suffixes.

Figure 4-9. %SAB Macro Worksheet

The parameter of the %SAB macro are described as follows:

start_base	A base value that defines the beginning of the SAB. The starting address of the SAB is interpreted as start_base:0.
end_base	A base value that defines the end of the SAB. The end address of the SAB is interpreted as end_base:F.
type	Reserved for future use. Enter the character U for this parameter.

When placing %SAB macro calls in the system configuration file, you must observe the following guidelines:

- You must order your %SAB macro calls by the start addresses of the memory that they declare (from smallest to largest).
- You must declare the interrupt vector as a system address block with a %SAB call.
- You must make %SAB calls for all ROM in your system.
- You must not overlap system address blocks (that is, %SAB macro calls must declare discrete areas of memory).
- Each block of memory not declared with a %SAB macro call must be at least (minimum transfer size +48 decimal bytes) in length. Refer to the "%SYSTEM Macro" section of this chapter for a description of the minimum transfer size.
- The first block of memory not declared with a %SAB macro call must be at least 160 decimal bytes long.

%SYSTEM MACRO

The %SYSTEM macro is used to declare parameters that affect the system as a whole. You must declare exactly one %SYSTEM macro for your entire system and place it immediately prior to the END statement in the system configuration file. Figure 4-10 contains the format of the %SYSTEM macro. This figure is a worksheet that you can use to prepare the macro call. The values in parentheses are suggested values. Use these suggested values for noncritical parameters.

Macro call: SYSTEM (system parameters)

Number of calls required: exactly one

CONFIGURATION FILE NAME ______________________

FORMAT:

	parameter	type	suggested default	value
%SYSTEM	(nucleus_entry,	base		______
	rod_size,	word	(30)	______
	min_trans_size,	word	(64)	______
	debugger,	see note 1	(A)	______
	default_e_h_provided,	see note 2	(N)	______
	exception_mode)	word		______

NOTES:

1. Valid entries for the debugger parameter include:

   A Debugger available
   N No debugger available

2. Valid entries for the default_e_h_provided parameter include:

   Y Yes
   D Debugger
   N No

3. Types addr and base must be entered as hexadecimal numbers without the suffix H. Types word and byte default to decimal, but will accept all radix suffixes.

Figure 4-10. %SYSTEM Macro Worksheet

The parameters of the %SYSTEM macro are described as follows:

nucleus_entry — Base value of the code segment of the Nucleus. For this parameter, specify the base portion of the value shown on the Nucleus locate map.

rod_size — Maximum number of objects that can be cataloged in the root object directory.

min_trans_size — Minimum amount of memory, in 16-byte paragraphs, that the Nucleus allows to be transferred between jobs. If your application programs consistently request memory in larger than 64-paragraph blocks, you should adjust this parameter to reflect this, in order to cut down on system overhead involved with transferring memory. However, do not specify a value much larger than the amount of memory your programs ordinarily request or memory fragmentation will occur, and the additional memory will be wasted.

debugger — Letter that indicates whether or not the Debugger is available. Possible values include:

- A — The Debugger is available.
- N — No Debugger is available.

default_e_h_provided — Letter that indicates the system default exception handler. Possible values include:

- Y — A user-supplied exception handler is the system default.
- D — The Debugger is the system default exception handler.
- N — No default exception handler is specified.

If you specify Y for this parameter, you must create your own exception handler, designate it to be a public procedure having name RQSYSEX, and link it to the root job. If you specify D for this parameter, make sure to include the Debugger in your system. Even if you include the Debugger in your system, you do not have to specify it as the system default exception handler.

mode — Encoded indication of the exception handler mode. Values are interpreted as follows:

value	Pass control to exception handler
0	Never
1	On programming error conditions only
2	On environmental conditions only
3	On all exceptional conditions

MACRO PARAMETERS FOR SUBSYSTEMS

Tables 4-2 and 4-3 list parameter values for the %JOB and %SYSTEM calls which you should use, depending on the subsystems in your application system. These tables list recommended values. A blank entry in either of the tables implies that you must determine this value. Notes for these tables follow the tables.

Notice that Tables 4-2 and 4-3 contain no entries for the Human Interface. You do not specify a %JOB macro for the Human Interface in order to include it in your application system. Instead, you must include the Human Interface as an I/O job during the configuration of the Extended I/O System. Refer to Chapters 12 and 13 for further information.

Table 4-2. Suggested %JOB Values for Optional Subsystems

Macro	Parameter	Debugger Value	Terminal Handler Value	I/O System Value
%JOB	directory_size	0	0	0
	pool_min	170H (without NDP support) 178H (with NDP support)	85H	500H
	pool_max	0FFFFH	0FFFFH	500H
	max_objects	0FFFFH	0FFFFH	0FFFFH
	max_tasks	0FFFFH	0FFFFH	0FFFFH
	max_job_priority	0	0	0
	exception_hand-ler_entry	0:0	0:0	0:0
	exception_hand-ler_mode	0	0	0
	job_flags	0	0	0
	init_task_pri-ority	0	0	128
	init_task_entry	note 1	note 1	note 1
	data_segment_base	0	0	0
	stack_pointer	0:0	0:0	0:0
	stack_size	300H	300H	200H
	task_flags	0 (without NDP support) 1 (with NDP support)	0	0

Table 4-2. Suggested %JOB Values for Optional Subsystems (continued)

Macro	Parameter	Application Loader Value	Extended I/O System Value
%JOB	directory_size	0	0
	pool_min	20H	180H
	pool_max	20H	180H
	max_objects	50	0FFFFH
	max_tasks	5	0FFFFH
	max_job_priority	0	0
	exception_hand-ler_entry	0:0	0:0
	exception_hand-ler_mode	0	0
	job_flags	0	0
	init_task_pri-ority	130	140
	init_task_entry	note 1	note 1
	data_segment_ base	0	0
	stack_pointer	0:0	0:0
	stack_size	160	300H
	task_flags	0	0

Table 4-3. Suggested %SYSTEM Values for Optional Subsystems

Macro	Parameter	Debugger Value	Terminal Handler Value	I/O System Value
%SYSTEM	nucleus_entry	note 2	note 2	note 2
	rod_size	30	30	30
	min_trans_size	40H	40H	40H
	debugger	A	note 3	note 3
	default_e_h_provided	D	note 3	note 3
	mode	3	note 4	note 4

Macro	Parameter	Application Loader Value	Extended I/O System Value
%SYSTEM	nucleus_entry	note 2	note 2
	rod_size	30	30
	min_trans_size	40H	40H
	debugger	note 3	note 3
	default_e_h_provided	note 3	note 3
	mode	note 4	note 4

NOTES FOR TABLES 4-2 AND 4-3:

1. Determine the values of the initialization task entry points from the absolute address specified as input to LOC86 (or the SUBMIT file used to locate the subsystem). The base portion of this address is the base of the entry point. The offset portion of the entry point is 0. Thus the entry points for all subsystems are of the form "base:0".

2. Determine this value from the Nucleus locate map. Use the base portion of the code class start address.

3. These values vary depending on whether you include the Debugger in your application system.

4. These values vary depending on which exceptions are to be handled by the exception handler.

CREATING THE CONFIGURATION FILE

After you have filled out all of the necessary %JOB, %SAB, and %SYSTEM worksheets, use a text editor to build one file containing all of these macro calls. You can create an entirely new file or use one of the files available on the release diskettes. Each optional subsystem release diskette contains an example system configuration file. This file is named xROOT.A86, where x indicates the subsystem with which it is associated (for example, IROOT.A86 is contained on the I/O System release diskette and LROOT.A86 on the Application Loader release diskette). Each one of these files contains %JOB calls for that particular subsystem and all other required subsystems, %SAB calls, and a %SYSTEM call. You can use any of these example system configuration files as your system configuration file by filling in the absolute addresses, adding %JOB calls for your application jobs, and modifying the %SAB calls to reflect your hardware environment.

After you create or modify your system configuration file, write its name on the macro worksheets, because you must specify this name later, during the root job generation process.

When creating or modifying your system configuration file, remember the following things:

- Place an $INCLUDE statement for the file CTABLE.MAC as the first statement of the file and the END statement as the last statement of the file.

- Enter all of the %JOB, %SAB, and %SYSTEM macro calls into this file. You can enter them in the same format as shown on the worksheets if you place comment macros (%') at the end of each continued line, or you can place the parameters for each macro call on a single line. Enter the required %SAB calls for your system, a %JOB call for each first-level job, and one %SYSTEM call. Place the %SYSTEM macro call immediately before the END statement. It must be the last macro call.

- It does not matter whether you place %JOB or %SAB calls first in the file. However, the order in which you enter the individual %JOB and %SAB calls is important. The %SAB calls must abide by the order restrictions described in the "%SAB Macro" section of this chapter. The %JOB call order is important because the root job initializes jobs in the order that their %JOB calls appear in the system configuration file. Always place the %JOB calls in the following order:

    1. Subsystem first-level jobs, in the following order:

        Terminal Handler and/or Debugger (in any order)
        Basic I/O System
        Application Loader
        Extended I/O System

        You can omit the Application Loader and still include the Extended I/O System. However, if you include the Human Interface as an I/O job during Extended I/O System configuration (refer to Chapters 12 and 13), you must include the Application Loader and place its %JOB macro in the indicated position.

    2. Application first-level jobs

    Place the subsystem first-level %JOB calls first so that the services of the subsystems are available to the remainder of the first-level jobs when the first-level jobs are initialized.

    The order in which you place %JOB calls for your application jobs depends on the content of these jobs. Any job whose services are immediately used by other jobs should be initialized before the other jobs; thus you should place its %JOB call earlier in the file.

After you have created the configuration file or modified one of the existing ones, you can go on to the next section and generate the root job.

GENERATE THE ROOT JOB

In order to generate the root job, you must do three things:

- Assemble the configuration file
- Link the root job and its associated modules
- Locate the root job

The Nucleus release diskette contains a SUBMIT file, CROOT.CSD, which you can use to perform all three of these functions. In order to use this SUBMIT file, you must first prepare your diskettes and place them in the proper drives as explained in the "Linking and Locating the Subsystems" section of this chapter. Then you can enter the following command:

```
SUBMIT :fx:CROOT( file, date, loc_addr)
```

where:

fx — The appropriate disk identifier, indicating the drive containing CROOT.CSD.

file — Name of the system configuration file. You should not include a disk identifier or an extension with this file name. The SUBMIT file assumes that this file resides on drive F1 and has the extension "A86". The SUBMIT file places the located root job on drive F1 in a file of the same name but without the extension and the link and locate maps on drive F3 in files of the same name but with extensions "MP1" and "MP2" respectively. The SUBMIT file also expects file CROOT.LIB to be available on drive F2.

date — Date on which this configuration takes place.

loc_addr — Address at which to locate the root job. Examine the memory map that you made earlier to determine the ending address of the last module that you located. Add a padding factor to that value, if necessary, and use the sum for the starting address of the root job. If you want to specify this value as a hexidecimal number, you must include the suffix H.

NOTE

If you are providing your own system exception handler, you must assemble it and modify CROOT.CSD in order to link the exception handler in with the root job. Its declaration must occur just prior to CROOT.LIB in the LINK86 portion of CROOT.CSD.

LOAD AND TEST THE SYSTEM

After you have located all of your jobs (Nucleus, subsystem jobs, application jobs, and root job), you are ready to load the system into RAM and test it. Use the ICE-86 in-circuit emulator or the iSBC 957A package to load your system from disk into RAM. The procedure for using ICE-86 is available in the ICE-86 IN-CIRCUIT EMULATOR OPERATING INSTRUCTIONS FOR ISIS-II USERS. The procedure for using the iSBC 957A package is described in the iSBC 957A INTELLEC - iSBC 86/12A INTERFACE AND EXECUTION PACKAGE USER'S MANUAL. When loading into RAM, be sure to load the root job last, so that its starting address gets correctly loaded into the code segment (CS) and instruction pointer (IP) registers.

When you have loaded your system, test it, correct any errors, reassemble or recompile any appropriate program code, re-link and relocate the necessary modules, and load the system again with the ICE-86 in-circuit emulator or the iSBC 957A package. You can continue this procedure (essentially a subset of the procedures described in this chapter) until you have created an error-free system. Then you can copy your final system to iRMX 86-formatted disks and use the Bootstrap Loader to load your system or you can build a final ROM/RAM-based system.

If you are going to use the Bootstrap Loader, refer to Chapter 11 of this manual for configuration information. Also refer to the iRMX 86 LOADER REFERENCE MANUAL for information on how to use the Bootstrap Loader.

If you are going to build a final ROM/RAM-based system, you can, in order to shorten the load time, burn your fully tested and completely debugged jobs into PROM while still testing and developing other jobs in RAM. Then, each time you reload your system, you need only load the jobs you are still working on.

Chapter 5 describes the procedures necessary to configure a ROM/RAM system. In general, it describes how to turn a completely debugged RAM system into a ROM/RAM system. If you want to burn your jobs into PROM as you finish testing and debugging them, make sure that all the fully tested and debugged jobs are configured as described in Chapter 5. The remaining jobs can be tested in RAM and burned into PROM as they are completed.

CHAPTER 5. CONFIGURING THE FINAL ROM/RAM BASED SYSTEM

If you have followed the procedures outlined in Chapters 3 and 4 of this manual, you should have a fully tested RAM-based iRMX 86 application system. In order to create the final ROM/RAM system, you should do the following:

- Minimize the memory address space requirements of your system by eliminating the padding factor you used originally when locating your jobs.
- Locate your system so that all the ROM-resident segments are contiguous.
- Test your final system in RAM first, locate it into ROM/RAM, and burn the appropriate parts into PROM.

The remainder of this chapter discusses these procedures in more detail. You should read the entire chapter, however, before modifying your system. The order in which you perform these procedures depends on your individual system requirements.

MINIMIZING THE MEMORY ADDRESS SPACE

When you originally located your first-level application jobs, you included padding factors in the calculations used to determine starting addresses of succeeding jobs. The additional space allocated with the padding factors allowed you to make small changes in your programs that increased their sizes without changing the LOC86 commands used to locate them. The modules, despite increasing in size, did not overlap each other. In your final ROM/RAM system, you have already debugged all of your programs; their sizes are fixed. So now you can eliminate any extra space existing between modules, if you desire. You also estimated the size of the root job and included this estimate in the %SAB macro call. You can now make a much better estimate of the size of the root job and modify your %SAB macro call to indicate this.

Follow the procedures outlined in the "Locate the Jobs" section of Chapter 4 to locate your application first-level jobs again. This time, however, leave out the padding factors between jobs. Then modify the configuration file by changing the %SAB and %JOB macro calls as follows:

- Change the %JOB macro call for each application first-level job to reflect the new location of the job.
- Change the %SAB macro calls to reflect the smaller size of reserved memory.

- Also change the %SAB macro call to more accurately reflect the size of the root job. You can make a better estimate of its size because you have already located it during the testing phase. Use the locate maps generated for it to make a size estimate.

If the starting address of the root job has changed, specify this new address in the CROOT.CSD file. Then, re-submit CROOT.CSD. (The section in Chapter 4 entitled "Generating the Root Job" describes this procedures.)

After you have located your system, load it into RAM and test it again to make sure that it functions correctly.

You can perform this procedure in conjunction with the one described in the next section. However, it might be wise to perform them separately in order to localize any possible errors.

LOCATING THE ROM/RAM BASED SYSTEM

When you located your initial test and development system, as described in Chapter 4, you located it by job. That is, if you had three jobs, they were laid out as shown in Figure 5-1.

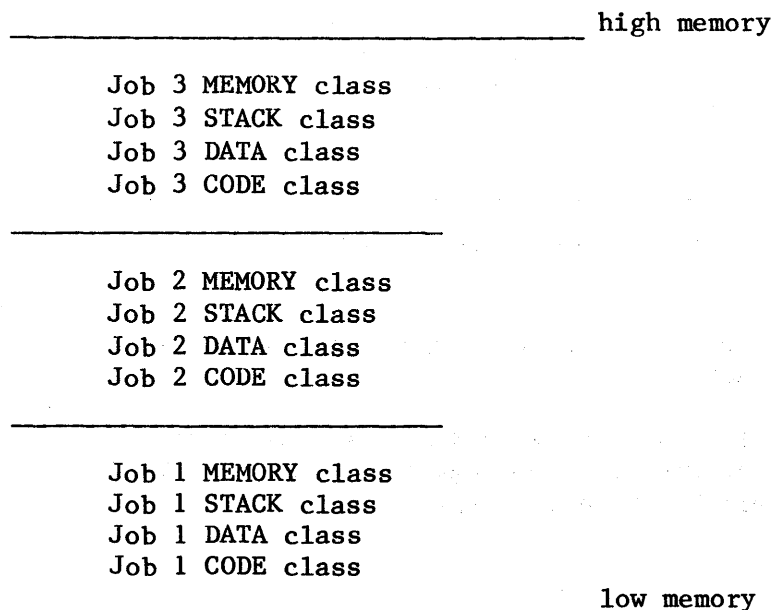


Figure 5-1. Memory Layout of a RAM-based System

This was relatively easy; it allowed you to use the ORDER control in LOC86 and specify only one address for each job with the ADDRESSES control. The SUBMIT files used to link and locate each of the subsystems used this method also. However, when configuring a ROM/RAM system, you should lay out the system by class, not by job. All of the ROM-resident segments from all of the jobs should be positioned together. Likewise, all of the RAM-resident segments should be positioned together. Thus, if you had the same three jobs and were laying out a ROM/RAM system, you should structure your memory as shown in Figure 5-2.

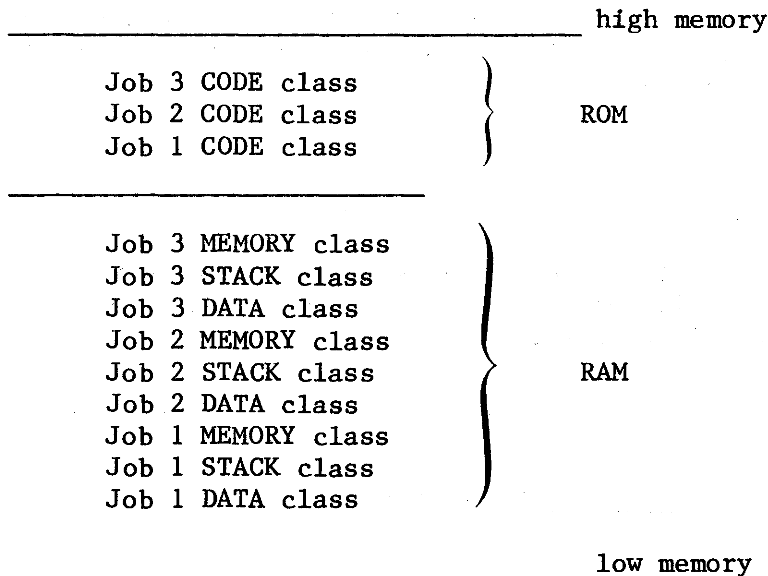


Figure 5-2. Memory Layout of a ROM/RAM System

All of the code classes are located in the upper memory, or ROM, and the remainder are located in RAM.

As you can see, in order to transform your RAM-based system into a ROM/RAM system, you must locate your jobs again. Before you do that, however, you should prepare a new memory map.

PREPARE A NEW MEMORY MAP

To prepare a new memory map, follow the procedures outlined in the "Preparing a Memory Map" section of Chapter 4, with one exception. In this map record not only the first available RAM address and the last

available RAM address, but also the first available ROM address and the last available ROM address. You need this information on your memory map because for ROM/RAM systems you must specify a location for both the ROM-resident code classes and RAM-resident classes.

LOCATE THE MODULES

The procedure for locating the modules of a ROM/RAM system, like that for a RAM-based system, is an iterative procedure. You locate one module, record its addresses in the memory map, and use those values to determine where to locate the next module. The format of the LOC86 command used to locate these modules is slightly different from the one used to locate the RAM-resident system. The format of this command when using a Series II development system is as follows:

```
LOC86   input_file TO output_file                       &
        ORDER (CLASSES (DATA, STACK, MEMORY))           &
        SEGSIZE (STACK (stack_size))                    &
        ADDRESSES (CLASSES (CODE (rom_address),         &
                            DATA (ram_address)))        &
        MAP PRINT (map_file)                            &
        OBJECTCONTROLS(NOLINES, NOCOMMENTS, NOPUBLICS,  &
                       NOSYMBOLS)
```

where:

input_file	Name of the link file produced previously by LINK86.
output_file	Name of the file in which LOC86 writes the absolute module.
stack_size	Size of this job's stack. Use this control for those jobs requiring a statically allocated stack. If this control is required, specify a minimum value of 200H; otherwise specify zero.
rom_address	Absolute starting location of the ROM-resident class (code class) of the module.
ram_address	Absolute starting location of the RAM-resident classes of the module.
map_file	Name of the file in which LOC86 writes the locate map.

If you have a Series III development system, you must also follow the additional guidelines listed in the "Using a Series III Development System" section of Chapter 4.

Use this form of the LOC86 command to locate the Nucleus, each optional subsystem first-level job, the root job, and each application first-level job. The ORDER and ADDRESSES controls of this command differ from those of the RAM-based LOC86 command (refer to the "Locating Application Jobs" section of Chapter 4). In this command, the ORDER control does not mention the code class. The ADDRESSES control requires that you enter two absolute addresses; one to locate the code class in ROM and one to locate the remaining classes in RAM.

The SUBMIT files contained on the subsystem release diskettes that link and locate the subsystems and the root job do not use this form of the LOC86 command. In order to use these SUBMIT files to create a ROM/RAM-based system, you must modify the LOC86 commands contained in these files so that they conform to the methods just described.

One method of locating your ROM/RAM system is as follows:

1. Locate the Nucleus first. Assign its data class to the lowest available RAM address and its code class to the lowest ROM address.

2. Determine the ending addresses of the code class and the memory class from the locate map generated by LOC86. Record these addresses on the memory map.

3. Using the next available ROM and RAM addresses as input to LOC86, locate the first optional subsystem.

4. Determine the ending addresses of the code class and the memory class from the locate map generated by LOC86. Record these addresses on the memory map. Also record the entry point address on the memory map. You need to know this address in order to specify it in the %JOB macro call.

5. Go back to step 3 and continue until you have located all of the subsystems and all of the application jobs.

After you have performed this procedure, follow the procedures outlined in the "Build the Configuration File" section of Chapter 4 in order to modify the configuration file and locate the root job. Note that you must modify the CROOT.CSD file in order to locate the root job as described in this chapter. You must also reserve all areas of RAM needed by the located modules.

TESTING THE SYSTEM IN RAM

Before you actually locate a ROM/RAM system, it is recommended that you follow the procedures outlined in the previous section, but specify RAM addresses for all classes. Then you can load the system into RAM and test it before burning code into PROM. After doing this, you can adjust the addresses to reflect a ROM/RAM system and build your final system.

CHAPTER 6. CONFIGURING THE NUCLEUS

The Nucleus provides system calls and features to support a wide variety of application software activities in a flexible hardware environment. Its structure allows you to take advantage of a breadth of support without sacrificing memory size or performance. If, after writing the code for your application system, you discover that you never make certain system calls or never make use of certain Nucleus features, you can exclude these system calls and features from the Nucleus of your application system.

The Nucleus also supports a variety of hardware environments. You can specify several options for your interrupt controllers and timer. You can also include an 8087 Numeric Data Processor in your system.

The process of including or excluding system calls and features and specifying the component environment is called Nucleus configuration. Nucleus configuration involves the following operations:

- Selecting the internal Nucleus features that you want to include in your application system and omitting the rest.

- Selecting the Nucleus system calls that you want to include in your application system and discarding the rest.

- Identifying certain hardware components that make up your system, and selecting the attributes of these components.

You perform these operations by making modifications to two Intel-supplied Nucleus configuration files: NTABLE.A86 and NDEVCF.A86. NTABLE.A86 defines the system call and feature configuration; NDEVCF.A86 defines the component configuration. These files are assembly language source files which are contained on the Nucleus release diskette. Figure 6-1 illustrates the structure of these files. After modifying the files, you must assemble them and link them with the rest of the Nucleus object files and libraries. The following sections describe this configuration process in detail.

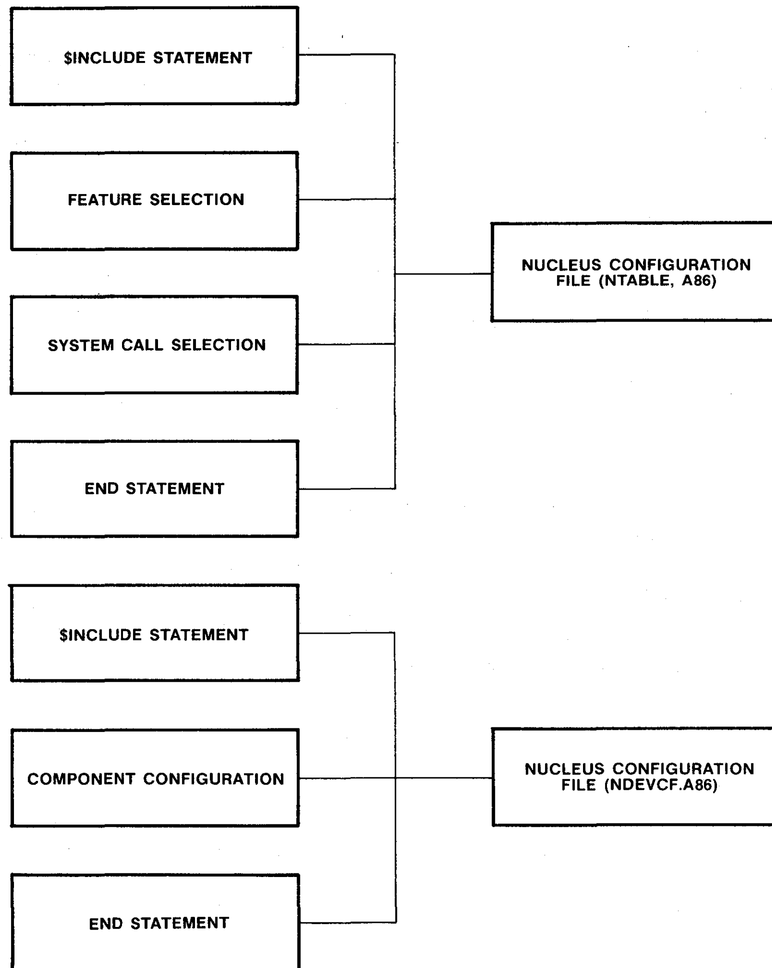


Figure 6-1. NTABLE.A86 and NDEVCF.A86 Structure

MODIFYING NTABLE.A86

As released, NTABLE.A86 defines the full complement of Nucleus system calls and internal features. To eliminate system calls or features, you must modify this file.

NTABLE.A86 consists of a series of macro calls. A macro, which corresponds in name to a system call or Nucleus internal feature, gives directions to the assembler to include the code for that system call or feature in the Nucleus. In order to exclude a system call or feature from your system, delete the metacharacter of the associated macro call (%), and replace it with the comment character (;). By doing this, you change the macro call into a comment and prevent the assembler from evaluating it.

The file NTABLE.MAC, which is available on the Nucleus release diskette, contains the definitions of all macros called in NTABLE.A86. NTABLE.A86 contains an $INCLUDE statement for NTABLE.MAC, which includes it in the assembly of NTABLE.A86.

The following sections describe modifying NTABLE.A86 to select features and system calls.

SELECTING NUCLEUS INTERNAL FEATURES

If you do not modify NTABLE.A86, the following internal features are included with the Nucleus:

- parameter validation
- system default exception handler

You can exclude any or all of these features in order to reduce code size and/or increase performance by modifying the feature configuration table portion of NTABLE.A86. Figure 6-2 illustrates this table. In order to exclude a feature, replace the percent-sign (%) at the beginning of the corresponding macro call with a semicolon (;). The following sections describe each of these Nucleus internal features.

```
$INCLUDE(:F2:NTABLE.MAC)
$EJECT
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;       NUCLEUS FEATURE CONFIGURATION TABLE
;
;       TO LEAVE OUT A FEATURE, CHANGE THE '%' TO THE COMMENT
;       CHARACTER ';' .
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

                %PARAMETER_VALIDATION
                %SYSTEM_EXCEPTION_HANDLER
```

Figure 6-2. Feature Configuration Table (NTABLE.A86)

Parameter Validation

A system call validates input parameters by checking for the existence of objects and by verifying that the objects are of the proper types. You can exclude all parameter validation by Nucleus system calls from your application system by modifying the feature configuration table. To do this, replace the percent-sign (%) in the %PARAMETER_VALIDATION macro with a semicolon (;).

You can include or exclude the support for parameter validation at two levels, the system level and the job level. Modifications to NTABLE.A86 include or exclude the system-level support. You can also include or exclude parameter validation on a job-to-job basis with a parameter to the CREATE$JOB system call (refer to the iRMX 86 NUCLEUS REFERENCE MANUAL for details). If you have included the system-level support, the CREATE$JOB system call allows you include or exclude parameter validation support on an individual job basis. Table 6-1 shows the relationship between system-level and job-level parameter validation support in terms of code savings and performance.

Table 6-1. System-level and Job-level Parameter Validation

System-level parameter validation	Included		Excluded	
Job-level parameter validation	Included	Excluded	Included	Excluded
Is parameter validation performed for this job?	yes	no	no	no
Does the system realize a code savings?	no	no	yes	yes
Does the job realize a performance improvement?	no	yes	yes	yes

System Default Exception Handler

If you do not modify NTABLE.A86, a system default exception handler is included in your system automatically. This exception handler deletes any task that causes an exceptional condition to occur. However, if you remove the %SYSTEM_EXCEPTION_HANDLER macro from NTABLE.A86 by replacing the percent-sign (%) with a semicolon (;), an alternate system exception handler is included in your system. This alternate handler suspends, rather than deletes, a task that causes an exceptional condition. Including this alternate system default exception handler could result in a significant code savings, if you are not otherwise using the DELETE$TASK system call.

SELECTING NUCLEUS SYSTEM CALLS

Figure 6-3 shows the system call configuration table portion of NTABLE.A86. In order to exclude a system call from your application system, replace the percent-sign (%) at the beginning of the corresponding macro with a semicolon (;).

```
%RQGETTYPE
%RQDISABLEDELETION
%RQENABLEDELETION
%RQCATALOGOBJECT
%RQUNCATALOGOBJECT
%RQLOOKUPOBJECT
%RQCREATEEXTENSION
%RQDELETEEXTENSION
%RQCREATECOMPOSITE
%RQDELETECOMPOSITE
%RQINSPECTCOMPOSITE
%RQALTERCOMPOSITE
%RQFORCEDELETE
%RQCREATEJOB
%RQDELETEJOB
%RQOFFSPRING
%RQCREATETASK
%RQDELETETASK
%RQSUSPENDTASK
%RQRESUMETASK
%RQSLEEP
%RQGETTASKTOKENS
%RQGETPRIORITY
%RQSETPRIORITY
%RQCREATEMAILBOX
%RQDELETEMAILBOX
%RQSENDMESSAGE
%RQRECEIVEMESSAGE
%RQCREATESEMAPHORE
%RQDELETESEMAPHORE
%RQSENDUNITS
%RQRECEIVEUNITS
%RQCREATEREGION
%RQDELETEREGION
%RQSENDCONTROL
%RQRECEIVECONTROL
%RQACCEPTCONTROL
%RQCREATESEGMENT
%RQDELETESEGMENT
%RQGETSIZE
%RQGETPOOLATTRIB
%RQSETPOOLMIN
%RQSETOSEXTENSION
%RQSETINTERRUPT
%RQENTERINTERRUPT
%RQENABLE
%RQDISABLE
%RQRESETINTERRUPT
```

Figure 6-3. System Call Configuration Table (NTABLE.A86)

```
                    %RQGETLEVEL
                    %RQEXITINTERRUPT
                    %RQSIGNALINTERRUPT
                    %RQWAITINTERRUPT
                    %RQGETEXCEPTIONHANDLER
                    %RQSETEXCEPTIONHANDLER
                    %RQSIGNALEXCEPTION

    END
```

Figure 6-3. System Call Configuration Table (NTABLE.A86) (continued)

MODIFYING NDEVCF.A86

NDEVCF.A86 consists of a series of macro calls that specify information about the following components:

- Programmable Interrupt Controller (PIC)
- Programmable Interval Timer (PIT)
- 8087 Numeric Data Processor (NDP)

As released, NDEVCF.A86 describes a standard system consisting of a master 8259A PIC and an 8253 PIT. If your system varies from this configuration, or if you want to change the attributes of any of the components, you must modify NDEVCF.A86. Figure 6-4 illustrates the component configuration portion of NDEVCF.A86.

```
$INCLUDE(:F2:NDEVCF.MAC)

        %MASTER_PIC(8259A,0C0H,0,0)

        ;SLAVE_PIC( SLAVE_TYPE, BASE_PORT, EDGE_VS_LEVEL, MASTER_LEVE

        %TIMER(8253,0D0H,28H,12288)

        ;NDP_SUPPORT( ENCODED_LEVEL )

END
```

Figure 6-4. Component Configuration Table (NDEVCF.A86)

The file NDEVCF.MAC, which is available on the Nucleus release diskette, contains the definitions of all macros called in NDEVCF.A86. NDEVCF.A86 contains an $INCLUDE statement which includes NDEVCF.MAC in the assembly of NDEVCF.A86.

The following sections describe modifying NDEVCF.A86 to include information about individual components.

PROGRAMMABLE INTERRUPT CONTROLLER (PIC) CONFIGURATION

The iRMX 86 Operating System supports a hardware environment with either a single PIC (non-cascaded mode) or several PICs (cascaded mode). Two macros are available to define the environment and the attributes of each PIC. These macros are:

```
%MASTER_PIC
%SLAVE_PIC
```

The %MASTER_PIC macro defines the attributes of the master PIC. This macro is required for both cascaded and non-cascaded mode. The %SLAVE_PIC macro is required only in cascade mode and defines the attributes of a slave PIC. One %SLAVE_PIC macro is required for each slave PIC in the system. All %SLAVE_PIC macros must follow the %MASTER_PIC macro in NDEVCF.A86. The following sections describe the formats of the two macros.

%MASTER_PIC Macro

The %MASTER_PIC macro defines the attributes of the single PIC, when in a non-cascaded environment, or the master PIC, when in a cascaded environment. The format of this macro call is as follows:

```
%MASTER_PIC(8259A, base_port, 0, 0)
```

Where:

base_port
Port address of the master PIC. When using Intel processor boards such as the iSBC 86/12A and iSBC 86/05, you must specify a value of 0C0H for this parameter. The Nucleus assumes that all ports are at even port addresses (base port + 0, base port + 2, base port + 4, and so on).

Because the Operating System currently supports only the 8259A PIC, you must specify the remaining parameters as shown. These remaining parameters are reserved for future support upgrades. For further information about the 8259A PIC, refer to THE 8086 FAMILY USER'S MANUAL.

As released, NDEVCF.A86 contains a default %MASTER_PIC call that defines an 8259A PIC with a port address of 0C0H. If your master PIC requires a different value, you must modify this call.

%SLAVE_PIC Macro

The %SLAVE_PIC macro defines the attributes of a slave PIC in a cascaded environment. You must include one %SLAVE_PIC macro for each slave PIC in your system. All of these %SLAVE_PIC calls must follow the %MASTER_PIC call. The format of the %SLAVE_PIC call is as follows:

```
%SLAVE_PIC(8259A, base_port, edge_vs_level, master_level)
```

Where:

base_port — Port address of the slave PIC. The Nucleus assumes that all ports are at even port addresses (base port + 0, base port + 2, base port + 4, and so on).

edge_vs_level — Triggering mode for the PIC. Specify this parameter as follows:

value	description
0	Edge triggering mode
nonzero	Level triggering mode

master_level — Interrupt level on the master PIC which connects to the slave PIC. You must specify a value in the range 0 through 7 for this parameter.

Because the Operating System currently supports only the 8259A PIC, you must specify the remaining parameter as shown. This remaining parameter is reserved for future support upgrades.

As released, NDEVCF.A86 does not include a %SLAVE_PIC call. If your system includes multiple interrupt controllers in a cascaded environment, you must modify NDEVCF.A86 to include a %SLAVE_PIC call for each slave PIC.

PROGRAMMABLE INTERVAL TIMER (PIT) CONFIGURATION

You can specify the attributes of the PIT by calling the %TIMER macro. The format of this macro call is as follows:

```
%TIMER(8253, base_port, level, count)
```

Where:

base_port — Port address of the PIT. When using Intel processor boards such as the iSBC 86/12A and iSBC 86/05, you must specify a value of ODOH for this parameter. The Nucleus assumes that all ports are at even port addresses (base port + 0, base port + 2, base port + 4, and so on).

level — Encoded value specifying the interrupt level of the master PIC to which this timer is connected. This value corresponds to the interrupt levels as follows:

value	level
x8H ($0 \geq x \geq 7$)	Master levels M0 through M7

count — Down count value that is loaded into the timer register. You should use the following formula to determine this value:

count = tick X clock_frequency

where:

tick — The period of time, in milliseconds, that you wish to specify as a clock interval.

clock_frequency — The frequency, in kilohertz, of the clock input to the timer.

Because the Operating System currently supports only the 8253 PIT, the remaining parameter must be specified as shown. This remaining parameter is reserved for future support upgrades. For further information concerning the 8253 PIT, refer to THE 8086 FAMILY USER'S MANUAL.

As released, NDEVCF.A86 contains a default %TIMER call which specifies a port address of 0D0H, an interrupt level of 2, and a 1.288 megahertz clock with 10 millisecond clock interval. If your system requires a different specification, you must modify NDEVCF.A86.

8087 NDP CONFIGURATION

If your system contains an 8087 NDP, you must call the %NDP_SUPPORT macro. This macro sets up a system interrupt handler for the NDP and associates it with a specified Programmable Interrupt Controller. The format of the macro call is as follows:

```
%NDP_SUPPORT(level)
```

where:

level — Encoded value specifying the interrupt level connected to the 8087 NDP interrupt pin. This value corresponds to the interrupt level as follows:

value	level
x8H $(0 \geq x \geq 7)$	Master interrupt levels M0 through M7.
yzH $(0 \geq y \geq 7)$ $(0 \geq z \geq 7)$	Slave interrupt levels 00 through 77.

No other application code can make use of this interrupt level. Also, any task which uses the 8087 NDP must not have a priority high enough to mask this interrupt level. Refer to the iRMX 86 NUCLEUS REFERENCE MANUAL for information concerning interrupt levels and priorities.

As released, NDEVCF.A86 does not include the %NDP_SUPPORT macro call. If your system contains an 8087 NDP, you must modify NDEVCF.A86 to include the %NDP_SUPPORT call.

For further information about the 8087 NDP, refer to THE 8086 FAMILY USER'S MANUAL, NUMERICS SUPPLEMENT.

MAXIMAL, DEFAULT, AND MINIMAL CONFIGURATION

The maximal Nucleus configuration (for features and system calls) consists of all supported Nucleus system calls, parameter validation, and the system default exception handler. This maximal configuration is the same as the default configuration. You do not need to modify NTABLE.A86 in order to obtain this maximal configuration.

The default component configuration defines the attributes of the components as they exist on the iSBC 86/12A single board computer.

The minimal Nucleus configuration for a Nucleus-only application system consists of no configurable internal features and only the following system calls:

CREATE$JOB
SUSPEND$TASK
RESUME$TASK
GET$TASK$TOKENS
SIGNAL$EXCEPTION

You must always include these system calls in your application system. You can, of course, include any or all other system calls and internal features that your application system requires.

ASSEMBLING THE CONFIGURATION FILES, LINKING AND LOCATING THE NUCLEUS

After you have made any necessary modifications to the Nucleus configuration files, NTABLE.A86 and NDEVCF.A86, you must assemble them and link and locate the Nucleus. NUCLUS.CSD, a SUBMIT file contained on the Nucleus release diskette, can be used to perform these functions. In order to use this SUBMIT file, you must first prepare your diskettes and place them in the proper drives of your development system as explained in the "Linking and Locating the Subsystems" section of Chapter 4. You should also examine NTABLE.A86 and NDEVCF.A86 to make sure that the $INCLUDE statements contain the proper disk identifiers. Then you can enter the following command:

```
SUBMIT :fx:NUCLUS(date, loc_adr)
```

where:

fx	The appropriate disk identifier, indicating the drive containing NUCLUS.CSD.
date	The date on which you submit the file (maximum of nine characters).
loc_adr	The address at which to locate the Nucleus. If you want to enter this value as a hexidecimal number, you must include the suffix H.

This command assembles NTABLE.A86 and NDEVCF.A86, links them together with other libraries that contain the Nucleus code and locates the Nucleus at the specified address. It places the located Nucleus in file NUCLUS on drive F1. It also places the assembly listing, link map, and locate map on drive F3 in files NTABLE.LST, NUCLUS.MP1, and NUCLUS.MP2, respectively.

NOTE

> The link map for the Nucleus always contains a warning message indicating a possible overlap. This is a normal message for the Nucleus. It does not indicate an error in the LINK86 command.

NUCLEUS INITIALIZATION ERRORS

If the Nucleus encounters an error during the initialization process, it places diagnostic information in the processor registers and halts the processor. Errors can occur during two operations:

Nucleus and memory initialization

Job creation by the root task

The value placed in the AX register determines which type of error occurred. The following sections outline these errors.

NUCLEUS AND MEMORY INITIALIZATION ERRORS

If an error occurs during the Nucleus and memory initialization process, the Nucleus sets the processor registers as follows:

register	value	description
AX	11H	A Nucleus or memory initialization error occured. The BX register contains a description of the error.
BX	0D001H	There are no SABs defined. There must be at least one.
	0D002H	The interrupt vector is not contained in a SAB.
	0D003H	Reserved.
	0D004H	There is not enough contiguous RAM available for the root job's memory pool.
	0D005H	The SABs are out of order or overlap.
	0D006H	There is not enough RAM available for the system resources of the Nucleus.
	0D007H	An invalid minimum transfer size was specified in the %SYSTEM macro. Refer to the "%SYSTEM Macro" section of Chapter 4 for a description of the minimum transfer size.

ROOT TASK ERRORS

If the root task encounters an error while it is creating the first-level jobs of your application system, it sets the processor registers as follows:

register	value	description
AX	21H	A root task error occurred. The BX, CX, and DL registers contain a description of the error.

register	value	description
BX	varies	BX is set as an index to indicate which %JOB call in the system configuration file caused the error. For example, 1 implies the first %JOB call in the file, 2 the second, and so forth.
CX	varies	CX contains the exception code returned from the CREATE$JOB system call that was called to create the first-level job.
DL	varies	DL contains the number of the parameter in the %JOB call that caused the error. If DL is greater than 8, the parameter number is DL +1. Otherwise, the parameter number is DL.

CHAPTER 7. CONFIGURING THE TERMINAL HANDLER

The Terminal Handler provides real-time, asynchronous I/O between an operator terminal and tasks running under the iRMX 86 Operating System. Terminal Handler configuration involves selecting characteristics of the Terminal Handler and specifying information about the processor board and the terminal. You perform these operations by making modifications to an Intel-supplied Terminal Handler configuration file. This file, MCONFG.A86, is an assembly language source file which is contained on the Terminal Handler release diskette.

As released, MCONFG.A86 defines a Terminal Handler that communicates with a 9600 baud terminal and runs on an iSBC 86/12A single board computer. If you want the Terminal Handler to run on a different hardware configuration, or if you want to change some of the characteristics of the Terminal Handler, you must modify MCONFG.A86, assemble it, link it with the rest of the Terminal Handler object files and libraries, and locate the Terminal Handler at an absolute address. The following sections discuss this configuration process in detail.

MODIFYING MCONFG.A86

MCONFG.A86 can consist of a series of macro calls which identify the characteristics of the Terminal Handler, the terminal, and the processor board. Figure 7-1 illustrates the released MCONFG.A86. As you can see by this figure, the released file contains only an $INCLUDE statement. This $INCLUDE statement causes the Terminal Handler to be assembled with the default configuration parameters. To change the configuration, you must add macro calls to MCONFG.A86. MCONFG.A86 can contain calls to the following macros:

```
%TH_19200_BAUD_COUNT
%MTH
%TH_INT_LEVELS
%TH_USART
%TH_TIMER
%TH_CHAR_LENGTH
%TH_MAILBOX_NAMES
```

The file MTHCNF.MAC, which is available on the Terminal Handler release diskette, contains the definitions of all these macros. MCONFG.A86 contains an $INCLUDE statement for MTHCNF.MAC, which includes it in the assembly of MCONFG.A86.

The following sections describe the macro calls in detail.

```
$include(:f2:mtncnf.mac)

end
```

Figure 7-1. Terminal Handler Configuration File (MCONFG.A86)

%TH_19200_BAUD_COUNT MACRO

If your system's programmable interval timer (PIT) has a clock input frequency other than 1.2288 megahertz, you must call this macro to set the limits on the baud rate attributes of the Terminal Handler. The format of the call to this macro is as follows:

```
%TH_19200_BAUD_COUNT(count)
```

where:

count — Value that when loaded into the timer register generates a maximum baud rate of 19200. The value that you enter for this parameter depends on the clock input frequency to your system's PIT (refer to the following paragraphs). If you do not include the macro call, a default value of 4 is assumed, which corresponds to the default frequency of the clock input to the 8253 PIT on the iSBC 86/12A board.

To derive the value to use for the count parameter, you must first determine the clock input frequency to the PIT (in hertz). Then substitute this frequency into the following equation:

(1) result = (clock frequency in hertz) / (19200 X 16)

Then substitute "result" from equation 1 into the following equation:

(2) fraction = result - INT(result)

where INT(result) is an integer obtained by truncating the fractional portion of "result". If "fraction" from equation 2 is greater than or equal to 0.5, then:

(3) count = INT(result) + 1

error_fraction = 1.0 - fraction

If "fraction" is less than 0.5, then:

(4) count = INT(result)

error_fraction = fraction

Before placing "count" from equation 3 or 4 into the %TH_19200_BAUD_COUNT call, you should first determine the percentage of error in this value. You do this by solving the following equation:

(5) % error = (error_fraction / count) X 100

If the percentage of error is less than 3%, you can use any Terminal Handler-supported baud rate (which you later specify in the %MTH macro, described later in this chapter). However, if the percentage of error is 3% or greater, you will have to perform the following additional computations.

First, determine the desired baud rate of the terminal. Substitute this value for the 19200 in equation 1 and recompute the value of "count" (equations 1 through 4). Again determine the percentage of error (equation 5). If the error is less than 3% with the new baud rate, you can use the Terminal Handler with that new baud rate (and specify it in the %MTH macro). However, if the percentage of error is still 3% or greater, the combination of desired baud rate and clock frequency is unacceptable to the Terminal Handler. You will have to change one or the other. After doing this, recompute the error to verify that it falls below the 3% level.

Regardless of the baud rate you eventually choose, use the "count" value as originally computated (with the 19200 value) as input to the %TH_19200_BAUD_COUNT macro call.

If MCONFG.A86 does not contain a call to the %TH_19200_BAUD_COUNT macro, the Terminal Handler will operate as if you had specified this macro call with a value of 4 for the count parameter. This is an appropriate value for the default input frequency to the 8253 PIT on the iSBC 86/12A board (or any timer with an input frequency of 1.2288 megahertz).

If you specify this macro call, you must place it as the first macro call in MCONFG.A86.

%MTH MACRO

This macro allows you to designate the baud rate and rubout charateristics of your terminal. The format of this macro is as follows:

```
%MTH (baud_rate, rubout_mode, blank_char)
```

where:

baud_rate — Baud rate of the terminal being used with the Terminal Handler. Specify one of the following rates:

19200
9600
4800
2400
1200
600
300
150
110

Refer to the "%TH_19200_BAUD_COUNT Macro" section of this chapter to ensure that the value you enter for this parameter will cause the Terminal Handler to operate correctly. If you omit the macro call, a default value of 9600 is assumed.

rubout_mode — Terminal Handler rubout mode. Enter one of the following:

1 The Terminal Handler echoes the deleted character back to the terminal.

2 The Terminal Handler replaces the deleted character with the blank character.

If you omit the macro call, a default value of 2 is assumed.

blank_char — Blanking character for use with option 2 of rubout_mode. If you omit the macro call, the default blanking character is assumed to be the ASCII space (020H).

If MCONFG.A86 does not contain the %MTH call, the Terminal Handler assumes a 9600 baud terminal with blanking mode 2 and a blanking character of ASCII space (020H).

%TH_USART MACRO

This macro allows you to designate the port address of the USART. The format of this macro call is as follows:

```
%TH_USART(base_port)
```

where:

base_port — Hexadecimal number specifying the base port address of the USART. The Terminal Handler assumes that all ports are at even port addresses (base port + 0, base port + 2, base port + 4, and so on). If you omit the macro call, a value of 0D8H is assumed.

If MCONFG.A86 does not include a call to %TH_USART, the Terminal Handler assumes a USART port address of 0D8H. This value must be used for Intel processor boards, such as the iSBC 86/12A and iSBC 86/05 single board computers.

%TH_TIMER MACRO

This macro allows you to specify information about the programmable interval timer (PIT). The format of the macro call is as follows:

```
%TH_TIMER(base_port, baud_counter)
```

where:

base_port
: Port address of the PIT. The Terminal Handler assumes that all ports are at even port addresses (base port + 0, base port + 2, base port + 4, and so on). If you omit the macro call, a value of 0D0H is assumed.

baud_counter
: Number of the PIT counter connected to the USART clock input. The output of this counter generates the Terminal Handler baud rate. You must specify a value from 0 to 2 for this parameter. If you omit the macro call, a value of 2 is assumed. You must ensure that the counter you select is not used by the Nucleus or any other module. The Nucleus uses counters 0 and 1 of the timer to which it is connected. Therefore, if your system does not contain an off-board timer, you must use counter 2 for the Terminal Handler.

If MCONFG.A86 does not contain the %TH_TIMER macro call, the Terminal Handler operates as if you had specified this macro call with a value of 0D0H for the base_port parameter and a value of 2 for the baud_counter parameter. These values must be used for Intel processor boards, such as the iSBC 86/12A and iSBC 86/05 single board computers.

%TH_CHAR_LENGTH MACRO

This macro allows you to specify the number of bits of valid data per character sent from the USART. The format of this macro call is as follows:

```
%TH_CHAR_LENGTH(length)
```

where:

length
: Number of bits of valid data per character sent from the USART. The only acceptable values for this parameter are 7 and 8. If you omit the macro call, a value of 7 is assumed.

If MCONFG.A86 does not contain the %TH_CHAR_LENGTH macro, the Terminal Handler assumes 7 bit characters, which is appropriate for systems using the ASCII character set.

%TH_MAILBOX_NAMES MACRO

This macro allows you to specify names for the Terminal Handler's input and output mailboxes. The format for this macro call is as follows:

```
%TH_MAILBOX_NAMES(input_mailbox, output_mailbox)
```

where:

input_mailbox	Name of the mailbox used for input to the Terminal Handler. Legitimate names consist of 12 or less alphanumeric characters. If you omit the macro call, the name RQTHNORMIN is assumed.
output_mailbox	Name of the mailbox used for output by the Terminal Handler. Legitimate names consist of 12 or less alphanumeric characters. If you omit the macro call, the name RQTHNORMOUT is assumed.

If MCONFG.A86 does not contain the %TH_MAILBOX_NAMES macro call, the Terminal Handler uses the names RQTHNORMIN and RQTHNORMOUT for its input and output mailboxes.

If you intend to use the Basic I/O System's On Board USART driver to communicate with the Terminal Handler, you must provide a Terminal Handler whose input and output mailboxes have the names RQTHNORMIN and RQTHNORMOUT, respectively. The Basic I/O System will communicate only with a Terminal Handler that uses these mailbox names.

%TH_INT_LEVELS MACRO

This macro allows you to specify the interrupt levels used by the Terminal Handler for input and output. The format of the call to this macro is as follows:

```
%TH_INT_LEVELS(input_level, output_level)
```

where:

input_level — Encoded value specifying the interrupt level used for input to the Terminal Handler. This value corresponds to the interrupt level as follows:

value	level
x8H ($0 \geq x \geq 7$)	Master interrupt levels M0 through M7.

value	level
yzH $(0 \geq y \geq 7)$ $(0 \geq z \geq 7)$	Slave interrupt levels 00 through 77.

If you omit the macro call, a value of 68H is assumed.

output_level — Encoded value specifying the interrupt level used for output by the Terminal Handler. This value corresponds to the interrupt level as follows:

value	level
x8H $(0 \geq x \geq 7)$	Master interrupt levels M0 through M7.
yzH $(0 \geq y \geq 7)$ $(0 \geq z \geq 7)$	Slave interrupt levels 00 through 77.

If you omit the macro call, a value of 78H is assumed.

The input interrupt level must be a higher priority level than the output interrupt level. The iRMX 86 NUCLEUS REFERENCE MANUAL describes the relationship between interrupt levels and priorities.

The maximum priority of user tasks in an application system containing the Terminal Handler depends on the interrupt levels assigned to the Terminal Handler with the %TH_INT_LEVELS macro. The priorities of all user tasks must be lower (numerically higher) than the lowest priority interrupt task in the Terminal Handler. In the default configuration, the Terminal Handler's output interrupt level is set to M7, which corresponds to a priority of 130 for the output interrupt task. Thus, with the default Terminal Handler configuration, all user tasks must have a priority lower (numerically higher) than 130.

If MCONFG.A86 does not contain the %TH_INT_LEVELS macro call, the Terminal Handler assumes master level M6 (68H) for input and master level M7 (78H) for output.

ASSEMBLING MCONFG.A86, LINKING AND LOCATING THE TERMINAL HANDLER

MTH.CSD, a SUBMIT file contained on the Terminal Handler release diskette, can be used to assemble MCONFG.A86 and link and locate the Terminal Handler. In order to use this SUBMIT file, you must first prepare your diskettes and place them in the proper drives of your development system as explained in the "Linking and Locating the Subsystems" section of Chapter 4. You may also have to make modifications to MTH.CSD before submitting it, depending on your Terminal Handler requirements. This section describes MTH.CSD modifications and the format of the command to submit this file.

MTH.CSD MODIFICATIONS

If you are providing code to implement control-C semantics, you must place this code in a procedure named RQABORTAP, which uses only near calls. Since this procedure runs as part of the interrupt task for the Terminal Handler, which does not have an exception handler, it should not make any system calls to delete its task, delete its job, suspend its task, or change its priority. The environmental condition codes generated as a result of making these system calls are returned in-line. Assemble this procedure and modify MTH.CSD to place its object file name in the LINK86 input list immediately after MCONFG.OBJ. Refer to the iRMX 86 TERMINAL HANDLER REFERENCE MANUAL for further information on the default control-C semantics.

The Human Interface release diskette includes a library HI.LIB which contains a module HCONTC that implements control-C semantics for the Human Interface. If you are planning to include the Human Interface in your application system and wish include the control-C features of the Human Interface, you must link this module in with the Terminal Handler through which the Human Interface communicates. Include the following line in the LINK86 input list immediately after MCONFG.OBJ:

```
:fx:HI.LIB(HCONTC),     &
```

This module should replace any control-C semantics files that you would otherwise include.

SUBMITTING MTH.CSD

Enter the following command to assemble MCONFG.A86 and link and locate the Terminal Handler:

```
SUBMIT :fx:MTH(date, loc_adr, type)
```

where:

fx	The appropriate disk identifier, indicating the drive containing MTH.CSD.
date	The date on which you submit the file (maximum of nine characters).
loc_adr	The address at which to locate the Terminal Handler. If you want to enter this value as a hexadecimal number, you must include the suffix H. The base portion of this value is the base portion of the Terminal Handler's entry point. The offset portion of the entry point is 0. You must specify this entry point in the %JOB macro call for the Terminal Handler.

type — Type of Terminal Handler you wish to create. Enter one of the following:

RQOUTPUT	An output-only version of the Terminal Handler is generated.
RQINPUT	An input and output version of the Terminal Handler is generated.

This command assembles MCONFG.A86, links it together with other modules that contain Terminal Handler code, and locates the Teminal Handler at the specified address. It places the located Terminal Handler in file MTH on drive F1. It also places link and locate maps on drive F3 in files MTH.MP1 and MTH.MP2 respectively.

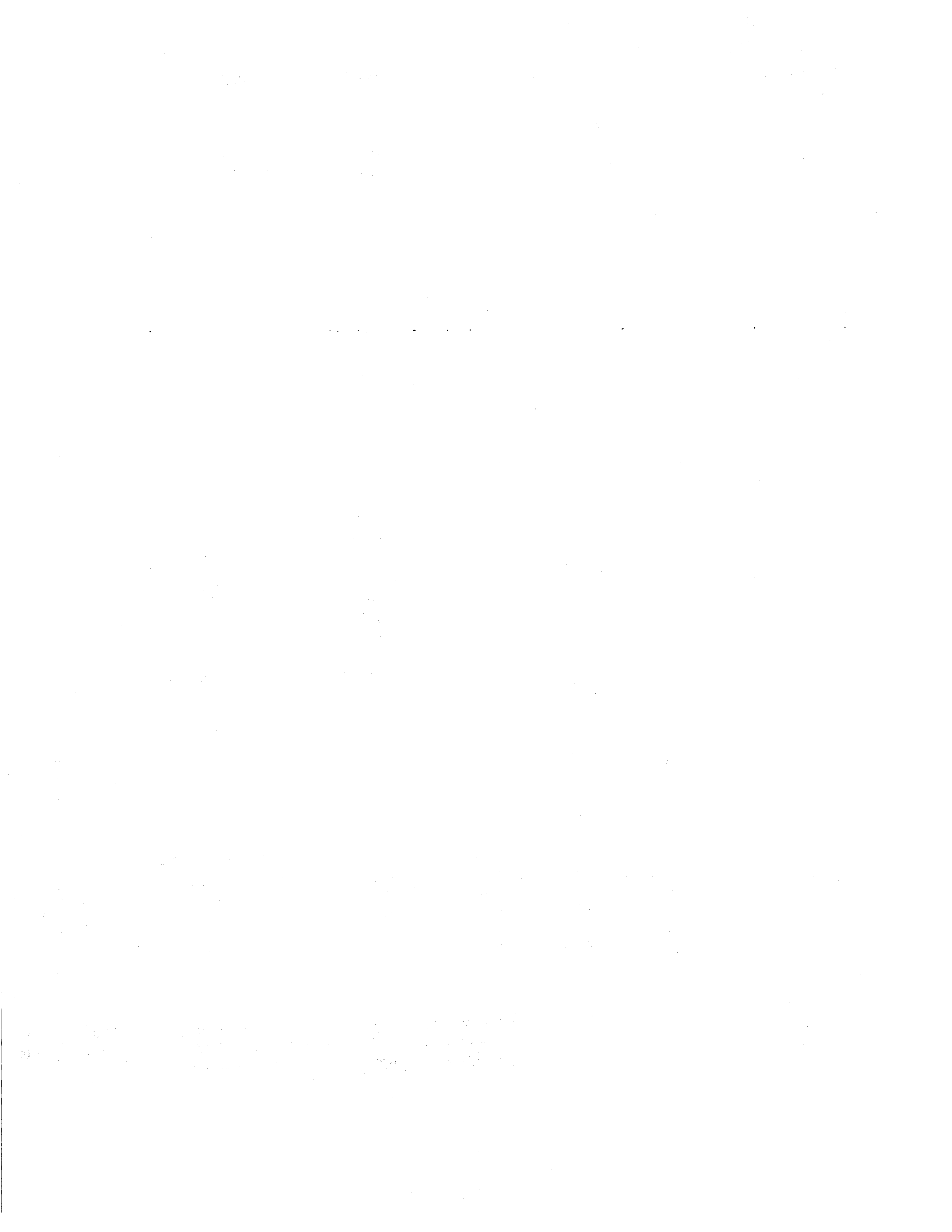
You must specify a %JOB macro in the system configuration file for the Terminal Handler (refer to Chapter 4). In this macro, the entry point depends on the address at which you locate the Terminal Handler (CS:0). The data segment base should be specified as 0 (the Terminal Handler assigns its own data segment).

CREATING MULTIPLE VERSIONS OF THE TERMINAL HANDLER

If desired, your iRMX 86 system can contain multiple versions of the Terminal Handler. This may be desirable if, for example, you have two tasks that use the Terminal Handler and you want to communicate with these tasks from separate terminals. In order to create multiple versions of the Terminal Handler, you must obey the following rules:

- Each Terminal Handler must use different input and output mailbox names. That is, the %TH_MAILBOX_NAMES calls must be different.
- Each Terminal Handler must use a unique USART. This also means that the %TH_USART calls must be different.
- Each Terminal Handler must use a unique timer. This also means that the %TH_TIMER calls must be different.
- Each Terminal Handler must use different interrupt levels. This also means that the %TH_INT_LEVELS calls must be different.
- The code for the Terminal Handlers must be located in different, non-overlapping areas; each Terminal Handler must have its own data area.
- Each Terminal Handler must have its own %JOB macro in the system configuration file.

If you adhere to these rules, you can create multiple versions of the Terminal Handler in your application system.



CHAPTER 8. CONFIGURING THE DEBUGGER

Because the Debugger contains a copy of the Terminal Handler, Debugger configuration is almost identical to Terminal Handler configuration (except that only one Debugger can be present in the application system). Debugger configuration involves selecting characteristics of the Debugger's Terminal Handler and specifying information about the processor board and the terminal. You perform these operations by making modifications to an Intel-supplied Debugger configuration file. This file, DTHCNF.A86, is an assembly language source file which is contained on the Debugger release diskette.

As released, DTHCNF.A86 defines a Terminal Handler for the Debugger that communicates with a 9600 baud terminal and runs on a system that uses an iSBC 86/12A single board computer. If you want the Debugger's Terminal Handler to run on a different hardware configuration, or if you want to change some of the characteristics of that Terminal Handler, you must modify DTHCNF.A86, assemble it, link it with the rest of the Debugger object files and libraries, and locate the Debugger at an absolute address. The following sections discuss this configuration process in detail.

MODIFYING DTHCNF.A86

DTHCNF.A86 can consist of a series of macro calls which identify the characteristics of the Debugger's Terminal Handler, the terminal, and the processor board. Figure 8-1 illustrates the released DTHCNF.A86, which causes the Debugger to be assembled with default configuration parameters. To modify this file, refer to the "Modifying MCONFG.A86" section of Chapter 7. The macro calls that you can place in DTHCNF.A86 are exactly the same as those described in Chapter 7.

```
$include(:f1:dthcnf.mac)

end
```

Figure 8-1.Debugger Configuration File (DTHCNF.A86)

ASSEMBLING DTHCNF.A86, LINKING AND LOCATING THE DEBUGGER

DB.CSD, a SUBMIT file contained on the Debugger release diskette, can be used to assemble DTHCNF.A86 and link and locate the Debugger. In order to use this SUBMIT file, you must first prepare your diskettes and place them in the proper drives of your development system, as explained in the "Linking and Locating the Subsystems" section of Chapter 4. You may also have to make modifications to DB.CSD before submitting it, depending on your Debugger requirements. This section discusses DB.CSD modifications and the format of the command to submit this file.

DB.CSD MODIFICATIONS

If you are providing code to implement control-C semantics, you must place this code in a procedure named RQABORTAP, which uses only near calls. Since this procedure runs as part of the interrupt task for the Terminal Handler, which does not have an exception handler, it should not make any system calls to delete its task, delete its job, suspend its task, or change its priority. The environmental condition codes generated as a result of making these system calls are returned in-line. Assemble this procedure and modify DB.CSD to place its object file name in the LINK86 input list immediately after DTHCNF.OBJ. Refer to the iRMX 86 TERMINAL HANDLER REFERENCE MANUAL for further information on the default control-C semantics.

The Human Interface release diskette includes a library HI.LIB which contains a module HCONTC that implements control-C semantics for the Human Interface. If you are planning to include the Human Interface in your application system and wish include the control-C features of the Human Interface, you must link this module in with the Terminal Handler with which the Human Interface communicates. If the Human Interface communicates with the Debugger's Terminal Handler, you must link this module in with the Debugger. To do this, include the following line in the LINK86 input list immediately after MCONFG.OBJ:

```
:fx:HI.LIB(HCONTC),     &
```

This module should replace any control-C semantics files that you would otherwise include.

SUBMITTING DB.CSD

Enter the following command to link and locate the Debugger:

```
SUBMIT  :fx:DB(date, loc_adr)
```

where:

fx	The appropriate disk identifier, indicating the drive containing DB.CSD.

date — The date on which you submit the file (maximum of nine characters).

loc_adr — The address at which to locate the Debugger. If you want to enter this value as a hexadecimal number, you must include the suffix H. The base portion of this value is the base portion of the Debugger's entry point. The offset portion of the entry point is 0. You must specify the entry point in the %JOB macro call for the Debugger.

This command links together the modules that make up the Debugger and locates the Debugger at the specified address. It places the located Debugger in file DB on drive F1. It also places the link and locate maps on drive F3 in files DB.MP1 and DB.MP2 respectively.

You must specify a %JOB macro in the system configuration file for the Debugger (refer to Chapter 4). In this macro, the entry point depends on the address at which you locate the Debugger (CS:0). The data segment base should be specified as 0 (the Debugger assigns its own data segment).

CHAPTER 9. CONFIGURING THE BASIC I/O SYSTEM

Basic I/O System configuration involves the following two operations:

- Selecting the features and system calls of the Basic I/O System that you want to include in your application system and discarding those that you do not want.
- Supplying the Basic I/O System with information about the I/O devices on your system.

You perform both of these operations by making modifications to two Intel-supplied Basic I/O System configuration file: ITABLE.A86 and IDEVCF.A86. These files, which are contained on the Basic I/O System release diskette, are assembly language source files. They contain the following information:

ITABLE.A86	This file contains information about the interfaces available with the Basic I/O System, the individual system calls associated with each interface, and the internal features of the Basic I/O System. As released, ITABLE.A86 defines the full complement of system calls and internal features.
IDEVCF.A86	This file contains a description of the devices supported, along with device and unit information for each supported device. As released, IDEVCF.A86 describes a number of commonly available devices.

Figure 9-1 illustrates the structure of these two files.

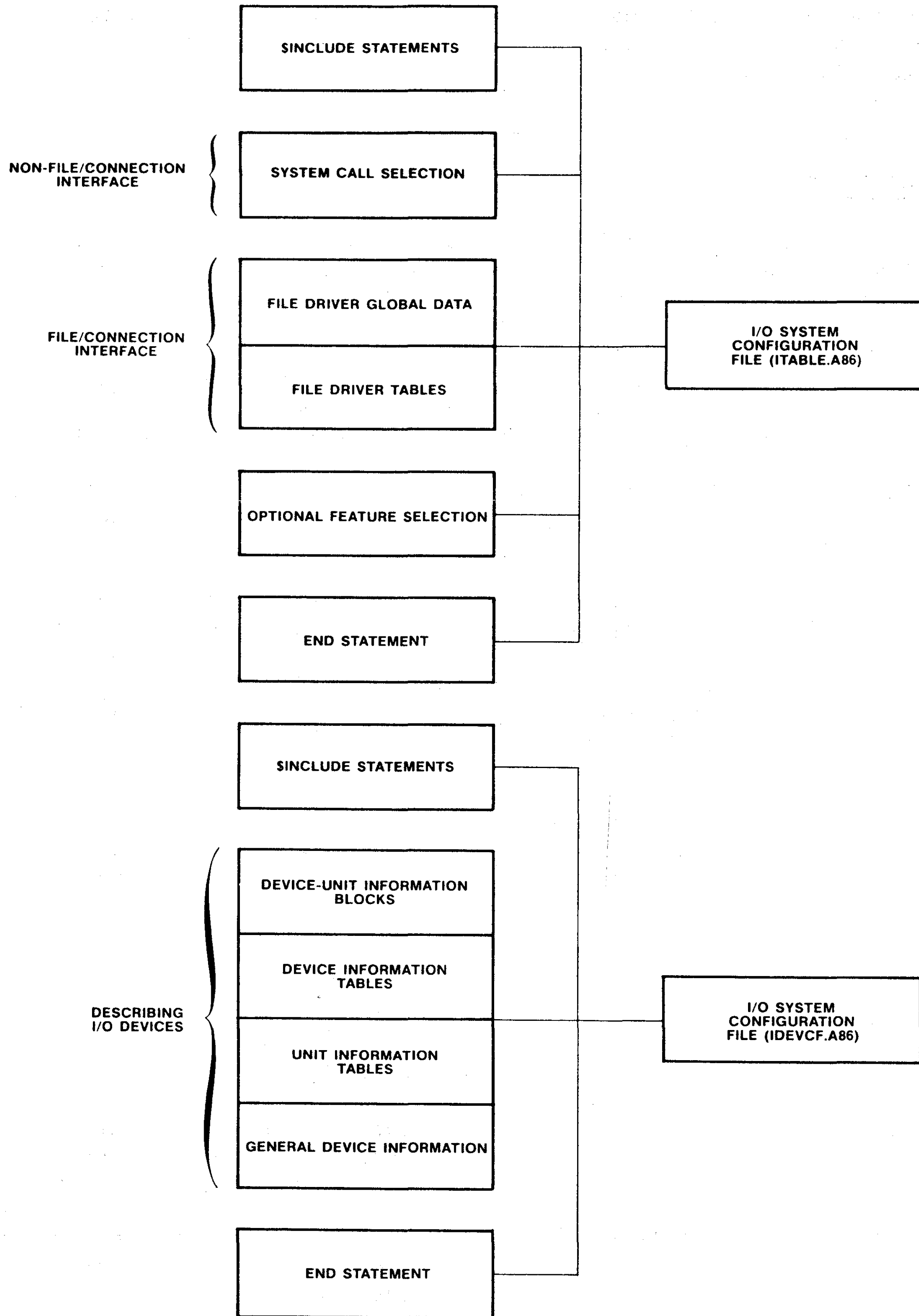


Figure 9-1. ITABLE.A86 and IDEVCF.A86 Structure

The following sections of this chapter show how to modify ITABLE.A86 and IDEVCF.A86 in order to produce a Basic I/O System that supports your individual needs. They also show how to assemble this file and link and locate the Basic I/O System. Some of the sections in this chapter list selected portions of the configuration file in order to aid you in the configuration process.

INCLUDE FILES

ITABLE.A86 must contain an $INCLUDE statement for the following file as the first statement (other than comments or general controls such as $TITLE) in the configuration file.

ITABLE.INC — This file contains segment, structure, macro, and miscellaneous definitions for the non-file/connection interfaces, the file/connection interface, file driver global data, and internal feature configuration.

IDEVCF.A86 must contain an $INCLUDE statement for the following file as the first statement (other than comments or general controls such as $TITLE) in the configuration file.

IDEVCF.INC — This file contains segment, structure, and macro definitions for device driver configuration; structure definitions for device configuration; and the defination of the %DEVICE_TABLES macro. This macro is described in the "General Device Information" section of this chapter.

These files are contained on the Basic I/O System release diskette. As released, ITABLE.A86 and IDEVCF.A86 contain $INCLUDE statements for these files. However, you should examine ITABLE.A86 and IDEVCF.A86 to ensure that the $INCLUDE statements contain the correct disk identifiers.

SELECTING NON-FILE/CONNECTION INTERFACE FEATURES (ITABLE.A86)

The non-file/connection interfaces consist of the following:

parameter interface	This interface supplies local parameters which the Basic I/O System uses each time a task makes a Basic I/O System call.
configuration interface	This interface is used by Operating System extensions to dynamically configure the Basic I/O System.
power-fail interface	This interface informs the Basic I/O System of impending power failure or power available conditions.

date/time interface	This interface supplies date and time information.

In ITABLE.A86, each non-file/connection interface consists of a group of related system calls. Figure 9-2 contains the portion of ITABLE.A86 that defines the non-file/connection interfaces. This code consists of macro calls which correspond in name to system calls. Each macro gives directions to the assembler to include the code for the corresponding system call in the Basic I/O System.

If you do not modify this portion of ITABLE.A86, all of the system calls associated with the non-file/connection interfaces will be included in your application system. In order to exclude a system call from one of the non-file/connection interfaces, delete the metacharacter of the associated macro call (%), and replace it with the comment character (;). By doing this, you change the macro call into a comment and prevent the assembler from evaluating it. Any or all of the system calls shown in Figure 9-2 can be excluded in this manner.

```
                name    itable

$include(:f1:itable.inc)
$eject
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Non-File-Connection Interfaces
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Parameter Interface:
;
        %rq_create_user
        %rq_inspect_user
        %rq_delete_user
        %rq_set_default_user
        %rq_get_default_user
        %rq_set_default_prefix
        %rq_get_default_prefix
;
; Configuration Interface:
;
        %rq_a_physical_attach_device
        %rq_a_physical_detach_device
```

Figure 9-2. Non-File/Connection Interface Configuration Values

```
;
; Power-Fail Interface:
;
        %rq_power_down
        %rq_power_up
;
; Time Interface:
;
        %rq_set_time
        %rq_get_time
```

Figure 9-2. Non-File Connection Interface Configuration Values (continued)

SELECTING THE FILE/CONNECTION INTERFACE FEATURES (ITABLE.A86)

The file/connection interface is the primary programmatic interface to the Basic I/O System, through which jobs manipulate connections and perform I/O. It provides the support for the file types available to the Basic I/O System user: named files, stream files, and physical files. This support is in the form of a file driver for each of these file types.

ITABLE.A86 contains information about all of the file drivers supplied with the Basic I/O System. If you do not modify ITABLE.A86, all of the system calls associated with the file/connection interface will be included in your application system. By modifying this file, you can eliminate entire file drivers or change the number of system calls supported by each driver. ITABLE.A86 contains two types of data pertaining to file drivers.

- File driver global data
- File driver tables

The following sections discuss how to modify this data in order to provide appropriate file driver support for your system.

FILE DRIVER GLOBAL DATA

The file driver global data consists of a group of macro calls which provide parameters used by all file drivers in your Basic I/O System. Figure 9-3 illustrates the portion of ITABLE.A86 which contains this global data. The values shown in this figure are contained in the released version of ITABLE.A86 and are the suggested defaults.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Define file-driver global data
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        %num_file_drivers(4)

        %attach_device_task_prio(129)

        %timer_task_prio(129)
```

Figure 9-3. File Driver Global Data Parameters

The following paragraphs discuss each of the macros shown in Figure 9-3. You can change the parameters of these macro calls from their default settings in order to reflect your individual Basic I/O System requirements.

%NUM_FILE_DRIVERS — This macro declares the number of file drivers in your Basic I/O System. Associated with each file driver is a file driver number. Use the largest file driver number in your system as the value for this parameter. Intel-supplied file drivers are numbered as follows:

driver	number
Physical files	1
Stream files	2
Named files	4

%ATTACH_DEVICE_TASK_PRIO — This macro declares the priority of the attach-device task. This task receives all requests to attach devices (via PHYSICAL$ATTACH$DEVICE). When the attach-device task receives such a request, it creates another task which actually handles the request. The second task's priority is one less than the priority of the task which called PHYSICAL$ATTACH$DEVICE. Thus the second task has a slightly higher priority.

%TIMER_TASK_PRIO	This macro declares the priority of the timer task. This task manages the time-of-day clock for the Basic I/O System. Its priority can impact its performance and the Basic I/O System behavior. If the priority is set too low, the timer task may not get to run as often as it needs and the clock will slip. If the priority is set too high, the timer task may take machine cycles away from high priority tasks.

FILE DRIVER TABLES

Associated with each file driver are two tables of procedure entry points and a macro call, which define the contents of the file driver. ITABLE.A86 contains this information for each of the Intel-supplied file drivers: the physical, stream, and named file drivers.

The macro, named %FILE_DRIVER_INFO, supplies parameters that are of use to the particular file driver. This manual does not define these parameters; therefore you should not modify any of the macro calls unless you are excluding an entire file driver (discussed later in this section).

One of the tables, the request table, lists the entry points of the procedures directly associated with the system calls of that file driver (for example, the REQREAD procedure is associated with the A$READ system call). These routines are called by user tasks.

The other table, the I/O service (or ios) table, lists the entry points of procedures that are ultimately called to service requests generated by procedures in the request table. The procedures in an ios table are called by Basic I/O System routines.

ITABLE.A86 provides the request tables and the ios tables in the form of two assembly language structures, REQ_FILE_DRIVER and IOS_FILE_DRIVER. The REQ_FILE_DRIVER structure defines the request table and the IOS_FILE_DRIVER table defines the ios table. The definitions for these structures are contained in the file ITABLE.INC, which is available on the Basic I/O System release diskette. The configuration file contains an $INCLUDE statement for this file, which includes it in the assembly of the configuration file.

Figure 9-4 shows the portion of ITABLE.A86 that contains the %FILE_DRIVER_INFO call, the request table, and the ios table for the physical file driver.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Physical files:  File-Driver Number 1
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
%file_driver_info(false, 14, 512, 20)
;
; Request part:
;
req_table         segment
req_file_driver   <
&                 reqcreatefile,          ; rq$a$create$file
&                 ,                       ; Reserved
&                 reqattachfile,          ; rq$a$attach$file
&                 ,                       ; Reserved
&                 reqdetachfile,          ; rq$a$delete$connection
&                 ,                       ; rq$a$create$directory
&                 ,                       ; Reserved
&                 ,                       ; rq$a$delete$file
&                 ,                       ; Reserved
&                 ,                       ; rq$a$rename$file
&                 ,                       ; rq$a$change$access
&                 reqopen,                ; rq$a$open
&                 reqclose,               ; rq$a$close
&                 reqread,                ; rq$a$read
&                 reqwrite,               ; rq$a$write
&                 reqseek,                ; rq$a$seek
&                 ,                       ; rq$a$truncate
&                 reqphysspecial,         ; rq$a$special
&                 reqconnectionstatus,    ; rq$a$get$connection$status
&                 reqfilestatus,          ; rq$a$get$file$status
&                 reqgetpathcomponent,    ; rq$a$get$path$component
&                 ,                       ; rq$a$get$directory$entry
&                 ,                       ; rq$a$get$extension$data
&                                         ; rq$a$set$extension$data
&>
req_table         ends
```

Figure 9-4. Physical File Driver Tables

```
;
; I/O System part:
;
ios_table          segment
ios_file_driver <
&                  ,                        ; File-driver init
&                  commoniotask,            ; I/O (connection) Task
&                  physupdate,              ; Update
&                  attachphysicalfile,      ; Attach File
&                  attachphysicalfile,      ; Create File
&                  ,                        ; Change Access (non-null path)
&                  ,                        ; Delete (non-null path)
&                  physread,                ; Read
&                  physwrite,               ; Write
&                  physseek,                ; Seek
&                  physspecial,             ; Special
&                  attachphysicaldevice,    ; Attach Device
&                  commondetachdevice,      ; Detach Device
&                  physopen,                ; Open
&                  physclose,               ; Close
&                  commongetconnst,         ; Get Connection Status
&                  physgetfilest,           ; Get File Status
&                  ,                        ; Get Extension Data
&                  ,                        ; Set Extension Data
&                  ,                        ; Change Access (null path)
&                  ,                        ; Delete (null path)
&                  ,                        ; Rename
&                  physgetpath,             ; Get Path Component
&                  ,                        ; Get Directory Entry
&                  ,                        ; Truncate
&                  physdetachfile           ; Detach File
&>
ios_table          ends
```

Figure 9-4. Physical File Driver Tables (continued)

Figure 9-5 shows the portion of ITABLE.A86 that contains the %FILE_DRIVER_INFO call, the request table, and the ios table for the stream file driver.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Stream-Files:  File-Driver Number 2.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
%file_driver_info(false, 16, 512, 20)
;
; Request part:
;
req_table           segment
req_file_driver     <
&                   reqcreatefile,              ; rq$a$create$file
&                   ,                           ; Reserved
&                   reqattachfile,              ; rq$a$attach$file
&                   ,                           ; Reserved
&                   reqdetachfile,              ; rq$a$delete$connection
&                   ,                           ; rq$a$create$directory
&                   ,                           ; Reserved
&                   reqdeletestrfile,           ; rq$a$delete$file
&                   ,                           ; Reserved
&                   ,                           ; rq$a$rename$file
&                   ,                           ; rq$a$change$access
&                   reqopen,                    ; rq$a$open
&                   reqclose,                   ; rq$a$close
&                   reqread,                    ; rq$a$read
&                   reqwrite,                   ; rq$a$write
&                   ,                           ; rq$a$seek
&                   ,                           ; rq$a$truncate
&                   reqstrspecial,              ; rq$a$special
&                   reqconnectionstatus,        ; rq$a$get$connection$status
&                   reqfilestatus,              ; rq$a$get$file$status
&                   reqgetpathcomponent,        ; rq$a$get$path$component
&                   ,                           ; rq$a$get$directory$entry
&                   ,                           ; rq$a$get$extension$data
&                                               ; rq$a$set$extension$data
&>
req_table           ends
```

Figure 9-5. Stream File Driver Tables

```
;
; I/O System part:
;
ios_table          segment
ios_file_driver    <
&                   nullfdinit,                  ; File-driver init
&                   commoniotask,                ; I/O (connection) Task
&                   ,                            ; Update
&                   attachstreamfile,            ; Attach File
&                   createstreamfile,            ; Create File
&                   ,                            ; Change Access (non-null path)
&                   ,                            ; Delete (non-null path)
&                   strread,                     ; Read
&                   strwrite,                    ; Write
&                   ,                            ; Seek
&                   strspecial,                  ; Special
&                   attachstreamdevice,          ; Attach Device
&                   commondetachdevice,          ; Detach Device
&                   stropen,                     ; Open
&                   strclose,                    ; Close
&                   commongetconnst,             ; Get Connection Status
&                   strgetfilest,                ; Get File Status
&                   ,                            ; Get Extension Data
&                   ,                            ; Set Extension Data
&                   ,                            ; Change Access (null path)
&                   strdelete,                   ; Delete (null path)
&                   ,                            ; Rename
&                   strgetpath,                  ; Get Path Component
&                   ,                            ; Get Directory Entry
&                   ,                            ; Truncate
&                   strdetachfile                ; Detach File
&>
ios_table          ends
```

Figure 9-5. Stream File Driver (continued)

Figure 9-6 shows the portion of ITABLE.A86 that contains null structures for the reserved file driver, driver 3. You can use structures like those contained in Figure 9-6 to exclude any other file driver from your application system. To do this, replace the %FILE_DRIVER_INFO call and the REQ_FILE_DRIVER and IOS_FILE_DRIVER structures associated with that driver with null structures of the following form:

```
%FILE_DRIVER_INFO(0,0,0,0)

REQ_TABLE      SEGMENT
REQ_FILE_DRIVER  < >
REQ_TABLE      ENDS

IOS_TABLE      SEGMENT
IOS_FILE_DRIVER  < >
IOS_TABLE      ENDS
```

Even if you make changes to the structures, you must maintain them in the order that they appear in the released Basic I/O System configuration file.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; File-Driver Number 3:  Reserved.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
%file_driver_info(0,0,0,0)
;
; Request part:
;
req_table          segment
req_file_driver    <>
req_table          ends
;
; I/O System part:
;
ios_table          segment
ios_file_driver    <>
ios_table          ends
$eject
```

Figure 9-6. Reserved File Driver Tables

Figure 9-7 shows the portion of ITABLE.A86 that contains the %FILE_DRIVER_INFO call, the request table, and the ios table for the named file driver.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Named Files:  File-Driver Number 4.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
%file_driver_info(true, 26, 1024, 54)
;
; Request part:
;
req_table        segment
req_file_driver  <
&                reqcreatefile,          ; rq$a$create$file
&                ,                       ; Reserved
&                reqattachfile,          ; rq$a$attach$file
&                ,                       ; Reserved
&                reqdetachfile,          ; rq$a$delete$connection
&                reqcreatedirectory,     ; rq$a$create$directory
&                ,                       ; Reserved
&                reqdeletefile,          ; rq$a$delete$file
&                ,                       ; Reserved
&                reqrenamefile,          ; rq$a$rename$file
&                reqchangeaccess,        ; rq$a$change$access
&                reqopen,                ; rq$a$open
&                reqclose,               ; rq$a$close
&                reqread,                ; rq$a$read
&                reqwrite,               ; rq$a$write
&                reqseek,                ; rq$a$seek
&                reqtrunc,               ; rq$a$truncate
&                reqnumspecial,          ; rq$a$special
&                reqconnectionstatus,    ; rq$a$get$connection$status
&                reqfilestatus,          ; rq$a$get$file$status
&                reqgetpathcomponent,    ; rq$a$get$path$component
&                reqgetdirectoryentry,   ; rq$a$get$directory$entry
&                reqnumgetextensiondata, ; rq$a$get$extension$data
&                reqnumsetextensiondata  ; rq$a$set$extension$data
&>
req_table        ends
```

Figure 9-7. Named File Driver Tables

```
;
; I/O System part:
;
ios_table          segment
ios_file_driver    <
&                  ,                        ; File-driver init
&                  commoniotask,            ; I/O (connection) Task
&                  numupdate,               ; Update
&                  attachnamedfile,         ; Attach File
&                  createnamedfile,         ; Create File
&                  namedchangeaccess,       ; Change Access (non-null pat
&                  nameddelete,             ; Delete (non-null path)
&                  numread,                 ; Read
&                  numwrite,                ; Write
&                  numseek,                 ; Seek
&                  numspecial,              ; Special
&                  attachnameddevice,       ; Attach Device
&                  commondetachdevice,      ; Detach Device
&                  numopen,                 ; Open
&                  numclose,                ; Close
&                  commongetconnst,         ; Get Connection Status
&                  numgetfilest,            ; Get File Status
&                  namgetextdata,           ; Get Extension Data
&                  namsetextdata,           ; Set Extension Data
&                  namchaccess,             ; Change Access (null path)
&                  namdelete,               ; Delete (null path)
&                  namrename,               ; Rename
&                  namgetpath,              ; Get Path Component
&                  namdirentry,             ; Get Directory Entry
&                  numtrunc,                ; Truncate
&                  numdetachfile            ; Detach File
&>
ios_table          ends
```

Figure 9-7. Named File Driver Tables (continued)

You can modify the file driver tables in one of two ways. If you want to exclude an entire driver from your Basic I/O System, substitute null structures for its %FILE_DRIVER_INFO call and its REQ_FILE_DRIVER and IOS_FILE_DRIVER structures in ITABLE.A86. However, if you want to eliminate one or more system calls from a file driver but still include the file driver as part of your Basic I/O System, delete the procedure names associated with the affected system calls from both the REQ_FILE_DRIVER structure and the IOS_FILE_DRIVER structure.

Replace the procedure names in the REQ_FILE_DRIVER structure with the name NOTCONFIGURED. Replace the procedure names in the IOS_FILE_DRIVER structure with commas. This causes the Basic I/O System to return the ENOTCONFIGURED exception code to any task that attempts to invoke one of the eliminated system calls. Table 9-1 lists the system calls and associated procedure names for the physical file driver. Table 9-2 lists the system calls and associated procedure names for the stream file driver. Table 9-3 lists the system calls and the associated procedure names for the named file driver.

Table 9-1. Physical File Driver System Calls and Procedure Names

SYSTEM CALL	REQ_FILE_DRIVER NAME	IOS_FILE_DRIVER NAME
A$CREATE$FILE	REQCREATEFILE	ATTACHPHYSICALFILE (the second one)
A$ATTACH$FILE	REQATTACHFILE	ATTACHPHYSICALFILE (the first one)
A$OPEN	REQOPEN	PHYSOPEN
A$SEEK	REQSEEK	PHYSSEEK
A$READ	REQREAD	PHYSREAD
A$WRITE	REQWRITE	PHYSWRITE
A$SPECIAL	REQPHYSSPECIAL	PHYSSPECIAL
A$CLOSE	REQCLOSE	PHYSCLOSE
AGETCON- NECTION$STATUS	REQCONNECTIONSTATUS	PHYSGETCONNST
AGETFILE$STATUS	REQFILESTATUS	PHYSGETFILEST
AGETPATH$COM- PONENT	REQGETPATHCOMPONENT	PHYSGETPATH
A$DELETE$CON- NECTION	REQDETACHFILE	PHYSDETACHFILE

Table 9-2. Stream File Driver System Calls and Procedure Names

SYSTEM CALL	REQ_FILE_DRIVER NAME	IOS_FILE_DRIVER NAME
A$CREATE$FILE	REQCREATEFILE	CREATESTREAMFILE
A$ATTACH$FILE	REQATTACHFILE	ATTACHSTREAMFILE
A$OPEN	REQOPEN	STROPEN
A$READ	REQREAD	STRREAD
A$WRITE	REQWRITE	STRWRITE
A$SPECIAL	REQSTRSPECIAL	STRSPECIAL
A$CLOSE	REQCLOSE	STRCLOSE
AGETCON-NECTION$STATUS	REQCONNECTIONSTATUS	STRGETCONNST
AGETFILE$STATUS	REQFILESTATUS	STRGETFILEST
AGETPATH$COMPONENT	REQGETPATHCOMPONENT	STRGETPATH
A$DELETE$CONNECTION	REQDETACHFILE	STRDETACHFILE
A$DELETE$FILE	REQDELETESTRFILE	STRDELETE

Table 9-3. Named File Driver System Calls and Procedure Names

SYSTEM CALL	REQ_FILE_DRIVER NAME	IOS_FILE_DRIVER NAME
A$CREATE$FILE	REQCREATEFILE	CREATENAMEDFILE
A$ATTACH$FILE	REQATTACHFILE	ATTACHNAMEDFILE
A$CREATE$DIRECTORY	REQCREATEDIRECTORY	CREATENAMEDFILE
A$CHANGE$ACCESS	REQCHANGEACCESS	NAMEDCHANGEACCESS NAMCHACCESS
A$RENAME$FILE	REQRENAMEFILE	NAMRENAME
A$OPEN	REQOPEN	NUMOPEN
A$SEEK	REQSEEK	NUMSEEK
A$READ	REQREAD	NUMREAD
A$WRITE	REQWRITE	NUMWRITE
A$SPECIAL	REQNUMSPECIAL	NUMSPECIAL
A$CLOSE	REQCLOSE	NUMCLOSE
AGETCON- NECTION$STATUS	REQCONNECTIONSTATUS	NUMGETCONNST
AGETFILE$STATUS	REQFILESTATUS	NUMGETFILEST
AGETDI- RECTORY$ENTRY	REQGETDIRECTORYENTRY	NAMDIRENTRY
AGETPATH$COMPONENT	REQGETPATHCOMPONENT	NAMGETPATH
A$DELETE$CONNECTION	REQDETACHFILE	NUMDETACHFILE
A$TRUNCATE	REQTRUNC	NUMTRUNC
A$DELETE$FILE	REQDELETEFILE	NAMEDDELETE NAMDELETE
AGETEXTENSION$DATA	REQNUMGETEXTENSIONDATA	NAMGETEXTDATA
ASETEXTENSION$DATA	REQNUMSETEXTENSIONDATA	NAMSETEXTDATA

In addition to the file driver tables shown in Figures 9-4, 9-5, 9-6, and 9-7, ITABLE.A86 also contains external declarations for all the symbols referenced by the REQ_FILE_DRIVER and the IOS_FILE_DRIVER structures. If you modify these structures to exclude system calls from your Basic I/O System, you can eliminate the EXTRN statements for the excluded procedure names, as long as these procedure names are not referenced by other file driver structures. However, make sure that all references to a procedure name have been eliminated before removing its EXTRN statement.

Example:

To remove the A$RENAME$FILE system call from the named file driver, replace the REQRENAMEFILE procedure name in the named file driver REQ_FILE_DRIVER structure with RQNOTCONFIGURED. Also replace the NAMRENAME procedure name in the named file driver IOS_FILE_DRIVER structure with a comma. Then remove the EXTRN statements for these names from the configuration file.

SELECTING FEATURES (ITABLE.A86)

Figure 9-8 shows the part of ITABLE.A86 that selects features of the Basic I/O System. These features are selected with macro calls, much in the same way as the non-file/connection interfaces. However, unlike the non-file/connection interface, features are selected by excluding the corresponding macro call from ITABLE.A86. In order to include a macro call from ITABLE.A86 (and thus exclude the feature), replace the semicolon (;) at the beginning of the corresponding macro call with a percent-sign (%).

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Define any features to be configured.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        ;dummy_timer
        ;no_create_false
        ;no_truncate
        ;no_allocate
```

Figure 9-8. Basic I/O System Features

The following paragraphs discuss each of the macros shown in Figure 9-8.

%DUMMY_TIMER — The presence of this macro call causes the Basic I/O System to be assembled without timing facilities. Without timing facilities, the Basic I/O System fills in all time fields with a zero value and saves the overhead of maintaining a timer. Also, without timing facilities you can debug your application system in single-step mode using the iSBC 957A monitor, because the system is not constantly servicing clock interrupts. However, if you include the %DUMMY_TIMER macro call, you should exclude the GET$TIME and SET$TIME system calls from your Basic I/O System.

If you exclude the %DUMMY_TIMER call from ITABLE.A86, the timing facilities of the Basic I/O System are included in your application system.

%NO_CREATE_-FALSE — The presence of this macro call causes the Basic I/O System to be assembled without the ability to create connections to existing files with the CREATE$FILE system call. In particular, if this macro is present, the user cannot call CREATE$FILE with the must$create parameter set to false. If the Basic I/O System encounters such a call, it returns the E$SUPPORT exception code. This option implies that CREATE$FILE can only be used to create connections to nonexistent files. Refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL for detailed information concerning the CREATE$FILE system call.

%NO_TRUNCATE — The presence of this macro call causes the Basic I/O System to be assembled without the TRUNCATE system call and its associated modules. The presence of this macro call also requires the presence of the %NO_CREATE_FALSE call, since setting the must$create parameter of CREATE$FILE to false requires the ability to truncate the file (refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL). This macro call also requires that you omit the TRUNCATE and DELETE$FILE system calls from your application system by removing their entries from the file driver tables (refer to the "File Driver Tables" section of this chapter).

%NO_ALLOCATE — The presence of this macro call causes the Basic I/O System to be assembled without the ability to extend files beyond their current end-of-file boundaries. This macro call requires that you omit the CREATE$FILE and CREATE$DIRECTORY system calls from your application system by removing their entries from the file driver tables (refer to the "File Driver Tables" section of this chapter). You should not include this macro call in ITABLE.A86 unless the users of the Basic I/O System only read from existing files on named-file volumes. However, inclusion of this macro call reduces the size of the Basic I/O System.

DESCRIBING THE I/O DEVICES (IDEVCF.A86)

An I/O device consists of a controller and one or more units. A device driver services each I/O device. In order to include I/O devices in your system, you must provide specific information about devices, units, and device drivers as well as general information about all devices in the system. The specific information is provided in the form of device-unit information blocks (DUIBs). The general information is provided in the form of a macro call. The Basic I/O System configuration file, IDEVCF.A86, contains several DUIBs for standard devices as well as the general information for these devices.

Using the information presented in this chapter as a guide, you must modify IDEVCF.A86 so that it describes the devices attached to your system.

Before presenting the specific information that you need in order to modify the configuration file, this chapter describes the terms device number, unit number, and device-unit number, since you must specify each.

DEVICE NUMBERING

Figure 9-9 contains a simplified drawing of three I/O devices in a system. The device numbers of these three devices are 0, 1, and 2, as shown. The device number represents the device as a whole. A unit number uniquely identifies a unit within a device (such as a single disk drive of a multi-drive device). Notice that the unit numbers of one device can duplicate the unit numbers of another. A device-unit number uniquely identifies a unit of a device among all the units of all the devices. Device-unit numbers are not duplicated.

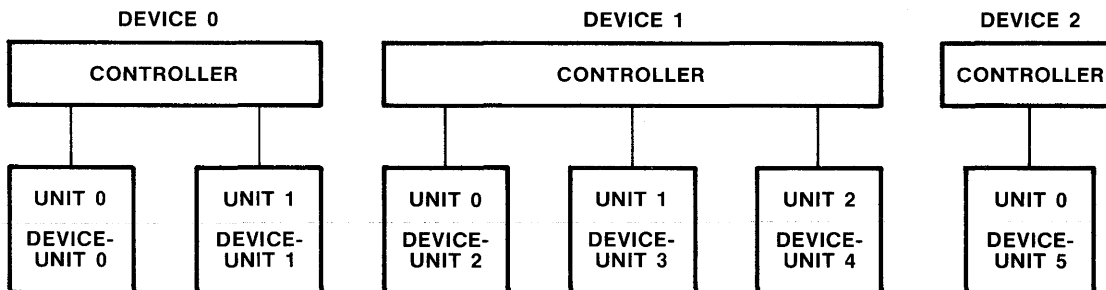


Figure 9-9. Device Numbering

Before creating your Basic I/O System configuration file, assign each device in your system a device number. Likewise, assign each unit a unit number and a device-unit number. The order of assignment is not important (with the exception of the unit number), as long as it is consistent with the definitions supplied previously, and the numbering of devices, units, and device-units begins with 0. The device driver uses the unit number to select the correct unit on the device. Make sure that if you have two of the same type of controller (such as two iSBC 204 controllers), assign each of them a separate device number.

DEVICE-UNIT INFORMATION BLOCKS

A device-unit information block (DUIB) is a block of information that you must supply for each device-unit in your system. You can provide this information by entering DEFINE_DUIB assembly language structures into the Basic I/O System configuration file. The definition of this structure is contained in file IDEVCF.INC, which is available on the Basic I/O System release diskette. IDEVCF.A86 contains an $INCLUDE statement for this file, which includes it in the assembly of the configuration file. The format of the DEFINE_DUIB structure is as follows:

```
DEFINE_DUIB  <
&   dev_name,
&   file_drivers,
&   functions,
&   flags,
&   dev_gran,
&   low_dev_size,
&   high_dev_size,
&   device_number,
&   unit_number,
&   device_unit_number,
&   init_io,
&   finish_io,
&   queue_io,
&   cancel_io,
&   device_info,
&   unit_info,
&   update_timeout,
&   num_buffers,
&   priority
&   >
```

where:

dev_name	Name of the device-unit. Specify this name as a string of 14 characters or less, surrounded by single quotes. This name is supplied to the PHYSICAL$ATTACH$DEVICE system call in order to identify the device to be attached. For example, the name 'F0' could be used as the name of an iSBC 204 unit.

file_drivers

WORD specifying file driver validity. Setting bit number i of this word implies that file driver number i+1 can attach this device-unit. Clearing bit number i implies that file driver number i+1 cannot attach this device-unit. Bits are numbered from right to left, starting with bit 0. Legitimate file drivers and their associated bit numbers include:

File Driver	Bit Number
physical	0
stream	1
named	3

The remainder of the word must be set to zero.

functions

WORD specifying I/O function validity. Setting bit number i of this word implies that this device-unit supports function number i. Clearing bit number i implies that the device-unit does not support function number i. Bits are numbered from right to left, starting with bit 0. Legitimate functions and their numbers include:

Function	Bit Number
F$READ	0
F$WRITE	1
F$SEEK	2
F$SPECIAL	3
F$ATTACH$DEV	4
F$DETACH$DEV	5
F$OPEN	6
F$CLOSE	7

Bits 4 and 5 must always be set. Every device driver requires these functions. Bits 8-15 of this word must be set to zero.

flags

BYTE specifying characteristics of diskette devices. The significance of the bits is as follows:

bit	meaning
0	Reserved
1	0 = single density; 1 = double density
2	0 = single sided; 1 = double sided
3-7	Reserved

dev_gran	WORD specifying the device granularity in bytes. This parameter is generally used for random access devices (described later in this section). It specifies the minimum number of bytes of information that the device reads or writes in one operation, or the sector size of the device. You should set this value equal to the volume granularity specified when the volume was formatted. For example, for an iSBC 204 or iSBC 206 unit you could specify 128 or 512.
low_dev_size	Low-order WORD of the device storage capacity, in bytes.
high_dev_size	High-order WORD of the device storage capacity, in bytes.
device_number	BYTE specifying the number of the device with which this DUIB is associated. Refer to the "Device Numbering" section of this chapter for more specific information.
unit_number	BYTE specifying the unit number of the device-unit associated with this DUIB. This number identifies one out of a possible several units of a device. Refer to the "Device Numbering" section of this chapter for more specific information.
device_unit_number	BYTE specifying the number of the device-unit associated with this DUIB. Refer to the "Device Numbering" section of this chapter for more specific information.
init_io	WORD specifying the offset in the code segment of this unit's Initialize I/O device driver procedure. Refer to Table 9-4 for special information concerning common and random access drivers. The GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 I/O SYSTEM contains additional information about this procedure. You must also place an EXTRN statement for the init_io parameter in the configuration file.
finish_io	WORD specifying the offset in the code segment of this unit's Finish I/O device driver procedure. Refer to Table 9-4 for special information concerning common and random access drivers. The GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 I/O SYSTEM contains additional information about this procedure. You must also place an EXTRN statement for the finish_io parameter in the configuration file.

queue_io	WORD specifying the offset in the code segment of this unit's Queue I/O procedure. Refer to Table 9-4 for special information concerning common and random access drivers. The GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 I/O SYSTEM contains additional information about this procedure. You must also place an EXTRN statement for the queue_io parameter in the configuration file.
cancel_io	WORD specifying the offset in the code segment of this unit's Cancel I/O procedure. Refer to Table 9-4 for special information concerning common and random access drivers. The GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 I/O SYSTEM contains additional information about this procedure. You must also place an EXTRN statement for the cancel_io parameter in the configuration file.
device_info	POINTER to a device information table for this device. Specify 0 for this parameter if the driver for the associated device does not need this field. Refer to the "Device and Unit Information" section of this chapter for specific information concerning the device-information table.
unit_info	POINTER to a unit information table for this unit. Specify 0 for this parameter if the driver for the associated unit does not need this field. Refer to the "Device and Unit Information" section of this chapter for specific information concerning the unit-information table.
update_timeout	WORD specifying the number of clock intervals (defined during Nucleus configuration; see Chapter 6) that the I/O system waits after completing an I/O request before updating file data structures on the volume. After a request on a connection has been completed, the Basic I/O System updates the data structures on the volume unless a new request is made before this timeout value expires. If you specify a zero value for this parameter, the Basic I/O System updates the structures during each request. A value of 0FFFFH indicates that updates occur only when the device is detached. When your Basic I/O System is in the debugging stages, you should specify a zero timeout value. Otherwise it is recommended that you specify a value for this parameter that when multiplied by the length of a clock interval

yields an update timeout value of about 1 second, unless your system can maintain disk integrity with the OFFFFH value. You can adjust the update_timeout value in conjunction with the num_buffers parameter to increase the performance of the Basic I/O System when dealing with this unit.

num_buffers — WORD which specifies whether a device is a random access device and, if it is, specifies how many buffers the device uses. If this parameter is nonzero, it specifies that the device is of the random access variety and indicates the number of buffers this unit will have for blocking and deblocking I/O requests. Each sector is one buffer plus 16 bytes long. The Basic I/O System uses these buffers to store partial blocks of data when I/O requests do not start on sector boundaries or transfer counts are not multiples of the device granularity. In most application systems, 4 is an optimal value for this parameter. A value of 0 for this parameter indicates that the device is not a random access device.

priority — BYTE specifying the priority of the Basic I/O System service task for the device.

You must specify a DEFINE_DUIB structure for each device-unit in the system. However, you can specify more DUIB structures than device-units. Each DUIB must have a unique name (specified with the dev_name parameter) but more than one DUIB can apply to the same device-unit. Different DUIBs can be used to specify different characteristics for the same device-unit. Therefore, different device or unit characteristics can be selected at run-time when the DUIB is associated with the device-unit (such as choosing between single- and double-density diskettes, for example). This is done with the A$PHYSICAL$ATTACH$DEVICE system call. You must arrange all of the DUIBs contiguously in your configuration file. Do not place any other code between the DEFINE_DUIB structures.

The Basic I/O System supports the notion of common and random access drivers, providing a set of support routines for these drivers. When you use common and random access drivers you must reference these Intel-supplied procedures in the DEFINE_DUIB structure. The DEFINE_DUIB parameters and the values which you must enter are listed in Table 9-4. The GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 I/O SYSTEM discusses common and random access device drivers in more detail.

Table 9-4. Common and Random Access Driver DUIB Values

DEFINE_DUIB Parameter	Common and Random Access Driver Parameter
init_io	INITIO
finish_io	FINISHIO
queue_io	QUEUEIO
cancel_io	CANCELIO

Former releases of the Basic I/O System provided two versions of the procedures listed in Table 9-4, one version for common drivers and one version for random access drivers. The random access procedures had the names listed in Table 9-4, but with the characters "RAD" as a preface. Now, the procedures listed in Table 9-4 provide support for both types of drivers. However, to be compatible with previous releases, the "RAD" names are still supported. If you are using a configuration file created in a previous release, you need not modify it to update these names.

IDEVCF.A86 contains DEFINE_DUIB structures for several standard devices. Figure 9-10 shows one of these structures.

```
;
; Shugart 204, unit 1
;
define_duib     <
&               'F1',                   ; Name(14)
&               00BH,                   ; file$drivers
&               0FFH,                   ; functs
&               00H,                    ; flags
&               128,                    ; dev$gran
&               0E900H,03H,             ; dev$size = 256256
&               0,                      ; Device
&               1,                      ; Unit
&               1,                      ; dev$unit
&               initio,                 ; init$io
&               finishio,               ; finish$io
&               queueio,                ; queue$io
&               cancelio,               ; cancel$io
&               dinfo_204,              ; device$info
&               uinfo_shugart,          ; unit$info
&               100,                    ; update$timeout
&               6,                      ; num$buffers
&               129                     ; priority
&>
```

Figure 9-10. Example DUIB Contained in IDEVCF.A86

The DUIB in Figure 9-10 defines an iSBC 204 unit. The name of this DUIB is F1. This name should be used when making an A$PHYSICAL$ATTACH$DEVICE system call. The parameters of the DUIB describe the unit as follows:

- The 0BH value for the file_drivers parameter indicates that file drivers one, two, and four can attach this device-unit (bits 0, 1, and 3 are set).
- The 0FFH value for the functions parameter indicates that all eight functions are valid for this device-unit (bits 0-7 are set).
- The 00H value for the flags parameter indicates that the drive is a single-density, single-sided diskette drive.
- The 0E900H value for the low_dev_size parameter and the 03H value for the high_dev_size parameter indicate a storage capacity of 3E900H bytes, or 256256 decimal bytes.
- The values for the device_number, unit_number, and device_unit_number parameters indicate that this DUIB applies to device-unit 1, which is unit 1 of device 0.
- The init_io, finish_io, queue_io, and cancel_io parameters indicate that this iSBC 204 device has a random access driver. These parameters contain the values listed in Table 9-4 for random access and common driver entry points. IDEVCF.A86 contains EXTRN statements for these entry points.
- The device_info and unit_info parameters indicate that this device has a device information table located at the address of symbol DINFO_204 and this unit has a unit information table located at the address of symbol UNIFO_SHUGART. IDEVCF.A86 contains these tables. They are described in the next section.

DEVICE AND UNIT INFORMATION TABLES

The device_info and unit_info parameters of DEFINE_DUIB refer to tables that you must place in the configuration file which contain specific information that the driver needs. The format of the information in each table depends on the device driver. This section describes the formats of the device information tables and unit information tables for common and random access device drivers. The GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 I/O SYSTEM contains more complete information about these device drivers.

Common Device Driver Tables

To provide device information tables for common device drivers, use the COMMON_DEV_INFO assembly language structure. This structure is defined in file IDEVCF.INC, which is available on the Basic I/O System release

diskette. IDEVCF.A86 contains an $INCLUDE statement for this file, which includes it in the assembly of the configuration file. The format of the COMMON_DEV_INFO structure is as follows:

```
COMMON_DEV_INFO  <
&        level,
&        priority,
&        stack_size,
&        data_size,
&        num_units,
&        device_init,
&        device_finish,
&        device_start,
&        device_stop,
&        device_interrupt
&        >
```

where:

level — WORD specifying an encoded interrupt level at which the device will interrupt. The interrupt task uses this value to associate itself with the correct interrupt level. The values for this field are encoded as follows:

bits	value
15-7	0
6-4	First digit of the interrupt level (0-7)
3	If one, the level is a master level and bits 6-4 specify the entire level number. If zero, the level is a slave level and bits 2-0 specify the second digit
2-0	Second digit of the interrupt level (0-7), if bit 3 is zero.

priority — BYTE specifying the initial priority of the device's interrupt task.

stack_size — WORD specifying the size in bytes of the stack for the user-written device interrupt procedure (and other procedures that it calls). This number should not include stack requirements for the Basic I/O System-supplied procedures. They add their requirements to this number.

data_size — WORD specifying the size in bytes of the user portion of the device-local data. This data is for driver use only. The common driver support procedures supplied by the Basic I/O System allocate their data in addition to this.

num_units — WORD specifying the number of units supported by the driver. Units are assumed to be numbered consecutively, starting with zero.

device_init — WORD specifying the start address of the user-written device initialization procedure.

device_finish — WORD specifying the start address of the user-written device finish procedure.

device_start — WORD specifying the start address of the user-written device start procedure.

device_stop — WORD specifying the start address of the user-written device stop procedure.

device_interrupt — WORD specifying the start address of the user-written device interrupt procedure.

Depending on the actual device driver, you may have to provide additional fields for this structure. The GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 I/O SYSTEM describes all fields of this structure in more detail.

Most common device drivers do not require unit information tables. Therefore, make sure to set the unit_info field of the DEFINE_DUIB structure to zero for any device-units that use common device drivers, unless the particular driver requires unit information.

Random Access Device Driver Tables

To provide the device information tables and unit information tables for random access device drivers, use the RADEV_DEV_INFO and RADEV_UNIT_INFO assembly language structure. These structures are defined in file IDEVCF.INC, which is available on the Basic I/O System release diskette. IDEVCF.A86 contains an $INCLUDE statement for this file, which includes it in the assembly of the configuration file.

The format of the RADEV_DEV_INFO structure is as follows:

```
RADEV_DEV_INFO  <
&       level,
&       priority,
&       stack_size,
&       data_size,
&       num_units,
&       device_init,
&       device_finish,
&       device_start,
&       device_stop,
&       device_interrupt
&       >
```

The fields of this structure are the same as those for the COMMON_DEV_INFO structure and are described in the "Common Device Driver Tables" section of this chapter.

Depending on the actual device driver, you may have to provide additional parameters for this structure. Refer to the "Device Driver Tables for Intel-supplied Device Drivers" section of this chapter for examples of this. The GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 I/O SYSTEM describes all of the fields of this structure in detail.

To provide the unit information table for a random access device driver, use the RADEV_UNIT_INFO assembly language structure. The format of this structure is as follows:

```
RADEV_UNIT_INFO  <
&        track_size,
&        max_retry,
&        0
&        >
```

where:

track_size WORD specifying the size in bytes of one track of a volume of the device. If the controller can handle requests that cross track boundaries, specify 0 for this parameter.

max_retry WORD specifying the maximum number of times an operation should be retried if an error occurs. A value of 9 is recommended.

Depending on the actual device driver, you may have to provide additional parameters for this structure. Refer to the "Device Driver Tables for Intel-supplied Device Drivers" section of this chapter for examples of this. Also refer to the GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 I/O SYSTEM for further information about this structure.

Device-Driver Tables for Intel-supplied Device Drivers

Intel supplies device drivers for the following devices:

iSBC 204 flexible disk controller

iSBC 206 hard disk controller

iSBC 215/iSBX 218/ iSBC 220 winchester/flexible/SMD disk controller

iSBC 254 bubble memory controller

iSBC 86/12A on board USART

byte bucket

All you have to do in order to use these drivers is to fill out the DUIBs, device information tables, and unit information tables as described in this section.

iSBC 204 Driver. The iSBC 204 driver is a random access driver. It supports the following:

- The functions F$READ, F$WRITE, F$SEEK, F$SPECIAL, F$ATTACH$DEVICE, and F$DETACH$DEV. F$OPEN and F$CLOSE are accepted, but the driver performs no operations for these functions. Track formatting and volume change notification are supported via the F$SPECIAL function. Refer to the iRMX 86 I/O SYSTEM REFERENCE MANUAL for further information about these special functions.

- All Intel-supplied file drivers.

- Device granularities of 128 and 512 bytes, software selectable on a per-unit basis.

- Up to four units per controller; two for each 8271 DMA chip. The 8271 chip at chip location A4 of the iSBC 204 board is FDC0, controlling units 0 and 1. The 8271 chip at chip location A6 is FDC1, controlling units 2 and 3.

You must place the following values in the device information table in order to support the iSBC 204 driver:

RADEV_DEV_INFO Field	Value
level	Configuration option
priority	Configuration option
stack_size	20
data_size	127
num_units	1 through 4
device_init	I204INIT
device_finish	DEFAULTFINISH
device_start	I204START
device_stop	DEFAULTSTOP
device_interrupt	I204INTERRUPT

In addition, you must also append the following information to the device information structure:

Type	Value
WORD	Address of the I/O port which matches the board configuration.

Figure 9-11 shows a device information table contained in IDEVCF.A86. This table is the one associated with the DUIB shown in Figure 9-10.

```
;
; General 204 Single-Density floppy device information
;
dinfo_204       radev_dev_info  <
&               058H,                   ; level
&               81,                     ; priority
&               20,                     ; stack$size
&               127,                    ; data$size
&               4,                      ; num$units
&               i204init,               ; device$init
&               defaultfinish,          ; device$finish
&               i204start,              ; device$start
&               defaultstop,            ; device$stop
&               i204interrupt           ; device$interrupt
&>
                dw      0A0H            ; I/O port base (204 specifilc
```

Figure 9-11. Device Information Table for iSBC 204 Device

IDEVCF.A86 also contains EXTRN statements for the symbols referenced in the Figure 9-11.

You must supply the following information in the unit information table in order to support the iSBC 204 driver:

RADEV_UNIT_INFO Field	Value
track_size	26 * 128 or 8 * 512
max_retry	9 (recommended)

In addition, you must append the following information, in sequence, to the unit information structure:

Type	Value
WORD	reserved
WORD	reserved
BYTE	step-rate for drive
BYTE	settle time for drive
BYTE	head load/unload time

Refer to the description of the specify command in the iSBC 204 FLEXIBLE DISKETTE CONTROLLER HARDWARE REFERENCE MANUAL for more information about these values. Figure 9-12 shows a unit information table contained in IDEVCF.A86. This table is associated with the DUIB shown in Figure 9-10.

```
;
; Shugart floppy unit infomation:
;
uinfo_shugart     radev_unit_info <
&                 26 * 128,                     ; track size
&                 9,                            ; max retry
&                 0                             ; reserved
&>
; 204 Specific:
                  dw      4                     ; Reserved
                  db      035H,0DH              ; Fixed initialize values.
                  db      8                     ; step rate
                  db      8                     ; settle
                  db      039H                  ; cnt$load
```

Figure 9-12. Unit Information Table for iSBC 204 Unit

iSBC 206 Driver. The iSBC 206 driver is a random access driver. It supports the following:

- The functions F$READ, F$WRITE, F$SEEK, F$SPECIAL, F$ATTACH$DEV, and F$DETACH$DEV. F$OPEN and F$CLOSE are accepted, but the driver performs no operations for these functions. Track formatting and volume change notification are supported via the F$SPECIAL function. Refer to the iRMX 86 I/O SYSTEM REFERENCE MANUAL for further information about these special functions.

- All Intel-supplied file drivers.

- Device granularities of 128 and 512 bytes, hardware (switch) selectable on a per-spindle basis.

- Up to 16 units per controller.

The iSBC 206 driver can support four spindles, with up to four platters per spindle. Units 0-3 are on the first spindle, 4-7 are on the second, 8-11 are on the third, and 12-15 are on the fourth. Units 0, 4, 8, and 12 are the removable platters.

You must specify the following values in the device information table, in order to support the iSBC 206 driver:

RADEV_DEV_INFO Field	Value
level	Configuration option
priority	Configuration option

RADEV_DEV_INFO Field	Value
stack_size	40
data_size	16
num_units	1 through 16
device_init	I206INIT
device_finish	DEFAULTFINISH
device_start	I206START
device_stop	DEFAULTSTOP
device_interrupt	I206INTERRUPT

In addition, you must also append the following data to the device information structure:

Type	Value
WORD	Address of the I/O port which matches the board configuration.

You must specify the following values in the unit information table, in order to support the iSBC 206 driver:

RADEV_UNIT_INFO Field	Value
track_size	36 * 128 or 12 * 512
max_retry	9 (recommended)

iSBC 215/iSBX 218/iSBC 220 Driver. The iSBC 215/iSBX 218/iSBC 220 driver is a random access driver. It supports the following:

- The functions F$READ, F$WRITE, F$SEEK, F$SPECIAL, F$ATTACH$DEV, and F$DETACH$DEV. F$OPEN and F$CLOSE are accepted, but the driver performs no operations for these functions. Track formatting and volume change notification are supported via the F$SPECIAL function. Refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL for further information about these special functions.

- All Intel-supplied file drivers.

- Device granularities of 128, 256, 512, and 1024 bytes.

- Up to 12 disks per controller.

The iSBC 215/iSBX 218/iSBC 220 controller assumes that units 0-3 are fixed disks on drives 1-4, units 4-7 are removable disks on drives 1-4, and 8-11 are flexible diskette drives 1-4 (when the iSBX 218 board is used in conjunction with the iSBC 215 board). The driver checks only the four least-significant bits of the unit number. The upper bits can be used to identify multiple units on the same disk, as described in this section.

You must specify the following values in the device information table, in order to support the iSBC 215/iSBX 218/ iSBC 220 driver.

RADEV_DEV_INFO Field	Value
level	Configuration option
priority	Configuration option
stack_size	300
data_size	400
num_units	12 or more
device_init	I215INIT
device_finish	DEFAULTFINISH
device_start	I215START
device_stop	DEFAULTSTOP
device_interrupt	I215INTERRUPT

In addition, you must also append the following data the the device information structure:

Type	Value
WORD	Wakeup address offset (normally set to 0)
WORD	Wakeup address base
WORD	Wakeup port address

You should set the wakeup address base and the wakeup port address to the values set with the switches on the iSBC 215 or iSBC 220 controller board.

You must specify the following values in the unit information table, in order to support the iSBC 215/iSBX 218 driver:

RADEV_UNIT_INFO Field	Value
track_size	0
max_retry	9 (recommended)

In addition, you must append the following information, in sequence, to the unit information structure:

Type	Value
WORD	The device code (0 for the iSBC 215 device and 2 for the iSBC 220 device)
WORD	The number of cylinders on the iSBC 215 disk.
BYTE	The number of fixed data heads on the disk drive.
BYTE	The number of removable data heads on the disk drive. For a iSBX 218 flexible disk drive, you should set this to 1 or 2 to indicate single or double sided diskettes. This value should correspond to the flags field of the DUIB for the unit.
BYTE	The number of sectors per track.

Type	Value
BYTE	The number of cylinders set aside for alternate tracks.
DWORD	The starting sector number of the device (normally 0). By using this field and the device size field in the DUIB, you can define several units on different parts of a single disk drive.

iSBC 254 Controller. The iSBC 254 driver is a random access driver. It supports the following:

- The functions F$READ, F$WRITE, F$SEEK, and F$ATTACH$DEVICE. F$DETACH$DEVICE, F$OPEN, and F$CLOSE are accepted, but the driver performs no operations for these functions.
- All Intel-supplied file drivers.
- Device granularities of 64, 128, 256, and 512 bytes, software selectable on a per-unit basis.
- Two types of hardware configuration:

  Each iSBC 254 board as a complete device with one unit.

  Several iSBC 254 boards as separate units of a single imaginary device.

You must place the following values in the device information table in order to support the iSBC 254 driver:

RADEV_DEV_INFO Field	Value
level	Configuration option
priority	Configuration option
stack_size	512
data_size	15
num_units	Configuration option
device_init	DEFAULTINIT
device_finish	DEFAULTFINISH
device_start	I254START
device_stop	DEFAULTSTOP
device_interrupt	I254INTERRUPT

You must specify the following values in the unit information table, in order to support the iSBC 254 driver:

RADEV_UNIT_INFO Field	Value
track_size	0
max_retry	9 (recommended)

In addition, you must append the following information, in sequence, to the unit information structure:

Type	Value
WORD	1
WORD	Base address of the board.
WORD	Number of device-granularity units of memory (64-, 128-, 256-, or 512-byte units) on the iSBC 254 board.

iSBC 86/12A On Board USART. The On Board USART is a Basic I/O System driver interface to the Terminal Handler and the Debugger. It supports the functions F$READ, F$WRITE, F$ATTACH$DEV, and F$DETACH$DEV. This driver interfaces to the Basic I/O System at the DUIB level. It should only be used by the physical file driver. You must specify the following procedure names in the DEFINE_DUIB structure in order to support the On Board USART:

DEFINE_DUIB Field	Procedure Name
init_io	THINITIO
finish_io	THFINISHIO
queue_io	THQUEUEIO
cancel_io	THCANCELIO

The device and unit information pointers are ignored. Set the parameters device_info and unit_info to 0.

The USART driver requires a Terminal Handler (or the Debugger's Terminal Handler) to be present in the application system. This Terminal Handler must use input and output mailbox names RQTHNORMIN and RQTHNORMOUT, respectively.

Byte Bucket Driver. The Basic I/O System supports a byte bucket device by providing a byte bucket driver. This driver responds to F$READ, F$WRITE, F$OPEN, F$CLOSE, F$ATTACH$DEVICE, and F$DETACH$DEVICE, but performs no operations for these functions. It returns an exception code on an F$SEEK request. The physical file driver and the stream file driver can make use of the byte bucket device driver.

The byte bucket device driver interfaces to the Basic I/O System at the DUIB level. In order to provide support for the byte bucket driver, you must specify the following values in the DEFINE_DUIB structure:

DEFINE_DUIB Field	Procedure Name
init_io	BYTEBUCKETINITIO
finish_io	BYTEBUCKETFINISHIO
queue_io	BYTEBUCKETQUEUEIO
cancel_io	BYTEBUCKETCANCELIO

GENERAL DEVICE INFORMATION

The Basic I/O System requires that several parameters and RAM-based data structures be generated for device configuration. In order for this to happen, you must include a %DEVICE_TABLES macro call in the configuration file. Place this call outside of all segment definitions in your Basic I/O System configuration module and immediately following the DEFINE_DUIB structures. The released configuration file contains an example macro call in the proper place. Modify this call to reflect your system.

The file IDEVCF.INC contains the definition of %DEVICE_TABLES macro. IDEVCF.A86 contains an $INCLUDE statement for this file, which includes it in the assembly of the configuration file.

The format of the macro call is as follows:

```
%DEVICE_TABLES(num_duib, num_dev_unit, num_devices)
```

where:

num_duib	Number of DUIBs that you have defined with the DEFINE_DUIB structure.
num_dev_unit	Number of device-units that you have defined. This number is 1 + (the largest dev_unit_number parameter). The dev_unit_number parameter was entered with the DEFINE_DUIB structure.
num_devices	Number of devices defined. This number is 1 + (the largest device_number parameter). The device_number parameter was entered with the DEFINE_DUIB structure.

ASSEMBLING THE CONFIGURATION FILES, LINKING AND LOCATING THE BASIC I/O SYSTEM

After you have modified ITABLE.A86 and IDEVCF.A86 to conform to your system requirements, you must assemble them and link and locate the Basic I/O System. IOS.CSD, a SUBMIT file contained on the Basic I/O System release diskette, can be used to perform these functions. In order to use this SUBMIT file, you must first prepare your diskettes and place them in the proper drives of your development system, as explained in the "Linking and Locating the Subsystems" section of Chapter 4. You should also examine ITABLE.A86 and IDEVCF.A86 to make sure that the $INCLUDE statements contain the proper disk identifiers. Then you can enter the following command:

```
SUBMIT :fx:IOS(date, loc_adr)
```

where:

fx	The appropriate disk identifier, indicating the drive containing IOS.CSD.

date — The date on which you submit the file (maximum of nine characters).

loc_adr — The address at which to locate the Basic I/O System. If you want to enter this value as a hexadecimal number, you must include the suffix H. The base portion of this value is the base portion of the Basic I/O System's entry point. The offset portion of the entry point is 0. You must specify the entry point in the %JOB macro call for the Basic I/O System.

This command assembles ITABLE.A86 and IDEVCF.A86, links them together with other modules containing Basic I/O System code, and locates the Basic I/O System at the specified address. It places the located Basic I/O System in file IOS on drive F1. It also places the assembly listing, link map, and locate map on drive F3 in files IOS.LST, IOS.MP1, and IOS.MP2, respectively.

You must specify a %JOB macro in the system configuration file for the Basic I/O System (refer to Chapter 4). In this macro, the entry point depends on the address at which you locate the Basic I/O System (CS:0). The data segment should be specified as 0 (the Basic I/O System assigns its own data segment).

BASIC I/O SYSTEM INITIALIZATION

The Basic I/O System defines a public symbol in which it returns its initialization status. This symbol, RQ$AIOS$INIT$ERROR, is defined by the Basic I/O System as follows:

```
DECLARE
    RQ$AIOS$INIT$ERROR      STRUCTURE(
        EXCEP$INDEX         WORD,
        EXCEP$CODE          WORD) PUBLIC;
```

If the Basic I/O System initializes properly, it sets itself up as an Operating System extension and returns a value of 0 in the EXCEP$CODE field. If the Basic I/O System does not initialize properly, it sets these fields as follows:

EXCEP$INDEX — An index in the initialization task indicating where the task failed.

EXCEP$CODE — The first exception code that the initialization task incurred.

CHAPTER 10. CONFIGURING THE APPLICATION LOADER

The iRMX 86 Application Loader provides the capability to load iAPX 86 object files from disk into memory under the control of the iRMX 86 Operating System. Application Loader configuration involves selecting parameters of the Application Loader that you wish to include in your application system and selecting the type of loading that the Application Loader can do. You perform these operations by modifying an Intel-supplied Application Loader configuration file and calling an Intel-supplied SUBMIT file. The configuration file, LCONFG.P86, is a PL/M-86 source file which is contained on the Application Loader release diskette. As released, this file selects default parameters. To change these parameters, you must modify this file, compile it, link it with the rest of the Application Loader modules, and locate the Application Loader at an absolute address. The Intel-supplied SUBMIT file, LOADER.CSD performs the compile, link, and locate operations, as well as selecting the type of Application Loader to include. This chapter describes these files.

MODIFYING LCONFG.P86

You can select parameters that are associated with the Application Loader by making modifications to LCONFG.P86. Figure 10-1 lists the portion of the released LCONFG.P86 that defines these parameters.

```
loader$config: DO;

DECLARE buf$size            LITERALLY '1024';   /*  BYTES  */
DECLARE rdbuf$size          LITERALLY '1024';   /*  BYTES  */


DECLARE lbuf$size             WORD PUBLIC DATA(buf$size + 11);
DECLARE l$rdbuf$size          WORD PUBLIC DATA(rdbuf$size);

DECLARE L$DEFAULT$MEMPOOL   WORD PUBLIC DATA(50H); /* PAGES */

END loader$config;
```

Figure 10-1. Application Loader Configuration File (LCONFG.P86)

In order to select parameter values, you must change the values specified with the LITERALLY statements in LCONFG.P86. The following paragraphs discuss the parameters.

buf$size	Size of an Application Loader internal buffer. This is the maximum allowable length of the iterative data block portion of a data record. Unless you are familiar with the formats of absolute data records, you should not change this parameter from its default of 1024.
rdbuf$size	Size of an I/O System buffer used for reading the object file from disk.
l$default$mempool	Default memory pool size, in 16-byte paragraphs, that jobs will have when created with the S$LOAD$IO$JOB and A$LOADIOJOB system calls. This value is used for both the maximum and minimum pool sizes.

COMPILING LCONFG.P86, LINKING AND LOCATING THE LOADER

After you have made any necessary modifications to the Application Loader configuration file, LCONFG.P86, you must compile it and link and locate the Application Loader. LOADER.CSD, a SUBMIT file contained on the Application Loader release diskette, can be used to perform these functions. In order to use this SUBMIT file, you must first prepare your diskettes and place them in the proper drives of your development system, as explained in the "Linking and Locating the Subsystems" section of Chapter 4. Then you can enter the following command:

```
SUBMIT   :fx:LOADER(date, loc_adr, code_type, load_job)
```

where:

fx	The appropriate disk identifier, indicating the drive containing LOADER.CSD.
date	The date on which you submit the file (maximum of nine characters).
loc_adr	The address at which to locate the Application Loader. If you want to enter this value as a hexadecimal number, you must include the suffix H. The base portion of this value is the base portion of the Application Loader's entry point. The offset portion of the entry point is 0. You must specify the entry point in the %JOB macro call for the Application Loader.

code_type — The type of code that the Application Loader can load. Enter one of the following values for this parameter:

value	type of code
A	Absolute code only.
P	Absolute and position independent code (PIC).
L	Absolute, PIC, and load-time locatable (LTL) code.
O	Absolute, PIC, LTL, and overlay code.

load_job — The types of job loading that the Application Loader can perform. Enter one of the following values for this parameter:

value	type of job loading
N	No job loading (neither A$LOAD$IO$JOB nor S$LOADIOJOB are supported). The Application Loader can handle absolute code and overlays only.
A	Asynchronous job loading. (S$LOAD$IO$JOB is not supported).
S	Both synchronous and asynchronous job loading.

This command compiles LCONFG.P86, links it together with the rest of the Application Loader, and locates the Application Loader at the specified address. It places the located Application Loader on drive F1 in file LOADER. It also places the compilation listing, link map, and locate map on drive F3 in files LOADER.LST, LOADER.MP1, and LOADER.MP2, respectively.

You must specify a %JOB macro in the system configuration file for the Application Loader (refer to Chapter 4). In this macro, the entry point depends on the address at which you locate the Application Loader (CS:0). The data segment base should be specified as 0 (the Application Loader assigns its own data segment).

CHAPTER 11. CONFIGURING THE BOOTSTRAP LOADER

The Bootstrap Loader is used to load the iRMX 86 Operating System and/or application programs into memory from mass storage and begin execution of the system. The Bootstrap Loader consists of two stages, the first of which must reside in ROM, and the second of which resides on mass storage. The first stage contains a device driver (or drivers) which loads the second stage into memory. This chapter provides configuration parameters for the first stage and the drivers. The second stage is not configurable; it is put on mass storage automatically when the mass storage volume is formatted.

FIRST STAGE CONFIGURATION

First stage configuration consists of:

- Selecting the features that you wish to include in the Bootstrap Loader.
- Listing the characteristics of the possible devices from which the Bootstrap loader can read.

You perform both of these operations by making modifications to a Bootstrap Loader first stage configuration file. This file, BS1.A86, is an assembly language source file which is contained on the Bootstrap Loader release diskette. As released, BS1.A86 defines an example configuration. Figure 11-1 lists the portion of BS1.A86 that defines the features and devices. To change the configuration of the Bootstrap Loader, you must modify this file, assemble it, and link it with the rest of the Bootstrap Loader object files and libraries.

```
        name    bs1

$include(:f1:bs1.inc)
%console
%manual
%auto
%device(f0, 0, deviceinit204, deviceread204)
%device(f1, 1, deviceinit204, deviceread204)
%device(f2, 2, deviceinit204, deviceread204)
%device(f3, 3, deviceinit204, deviceread204)
%device(d0, 0, deviceinit206, deviceread206)
%device(w0, 0, deviceinit215, deviceread215)
%device(wf0, 8, deviceinit215, deviceread215)
%device(wf1, 9, deviceinit215, deviceread215)
%device(wf2, 10, deviceinit215, deviceread215)
%device(wf3, 11, deviceinit215, deviceread215)
%device(b0, 0, deviceinit254, deviceread254)
%end
```

Figure 11-1. First Stage Configuration File (BS1.A86)

The first stage configuration file consists of a series of macro calls which define the features and devices. These macros include:

%CONSOLE
%MANUAL
%AUTO
%DEVICE
%END

The file BS1.INC, which is available on the Bootstrap Loader release diskette, contains the definitions of all of the macros which you can call in the first stage configuration file. BS1.A86 contains an $INCLUDE statement for BS1.INC which includes it in the assembly of BS1.A86.

%CONSOLE MACRO

This optional macro call indicates whether or not the user can supply, from the console, the name of the file to be loaded. If you include the %CONSOLE call in your configuration file, the user has the option of entering the file name. If you omit the %CONSOLE call, the Bootstrap Loader uses a default file name. The format of the %CONSOLE call is as follows:

%CONSOLE

%MANUAL MACRO

This optional macro indicates whether or not the user can supply, from the console, the name of the device to load from. If you include the %MANUAL call in your system, the user has the option of entering the name of the device to load from, as well as the name of the file to be loaded. If you omit the %MANUAL call from your system, the user cannot specify the device name.

If you include the %MANUAL call in the first stage configuration file, the %CONSOLE and %AUTO calls are automatically included, without having to specify them. The format of the %MANUAL call is as follows:

```
%MANUAL
```

%AUTO MACRO

This optional macro indicates whether or not the Bootstrap Loader can select devices automatically. If you include the %AUTO call in your first stage configuration file and the user does not specify a device name from the console, the Bootstrap Loader tries to initialize, in order, each of the devices specified with %DEVICE calls (described in the next section). It scans through the devices repeatedly until it successfully initializes one. When it does succeed in initializing a device, it uses that device from which to load. If you omit the %AUTO call from your first stage configuration file, the Bootstrap Loader can use just one device. The format of the %AUTO call is as follows:

```
%AUTO
```

%DEVICE MACRO

This macro defines the possible devices from which the Bootstrap Loader can load. You must include at least one %DEVICE call in your first stage configuration file. You can include more than one if you also include the %MANUAL or %AUTO calls. The format of the %DEVICE call is as follows:

```
%DEVICE(name, unit, device$init, device$read)
```

where:

name	Name of the device from which to load.
unit	Unit number of the device.

device$init — Address of a procedure that the Bootstrap Loader calls to initialize the device. Intel supplies this procedure for iSBC 204, iSBC 206, iSBC 215, iSBC 220, and iSBC 254 devices. To use one of these procedures, specify one of the following values:

device	value
iSBC 204	DEVICEINIT204
iSBC 206	DEVICEINIT206
iSBC 215	DEVICEINIT215
iSBC 220	DEVICEINIT215
iSBC 254	DEVICEINIT254

device$read — Address of a procedure that the Bootstrap Loader calls to read the device. Intel supplies this procedure for iSBC 204, iSBC 206, iSBC 215, iSBC 220, and iSBC 254 devices. To use one of these procedure, specify one of the following values:

device	value
iSBC 204	DEVICEREAD204
iSBC 206	DEVICEREAD206
iSBC 215	DEVICEREAD215
iSBC 220	DEVICEREAD215
iSBC 254	DEVICEREAD254

%END MACRO

This macro denotes the end of the first stage configuration file. You must include the %END call as the last statement of the first stage configuration file. The format of the %END call is as follows:

```
%END
```

DRIVER CONFIGURATION

Driver configuration consists of providing the elementary device driver procedures that the Bootstrap Loader calls when it initializes and reads from the device. You can either include the routines provided with the Bootstrap Loader for the iSBC 204, 206, 215, 220, and 254 devices, or you can write your own driver procedures for other devices.

INTEL-SUPPLIED PROCEDURES

Intel supplies elementary device driver procedures which can be used with iSBC 204, 206, 215, 220, and 254 devices. In order to include these procedures in your Bootstrap Loader, you can make modifications (if necessary) to driver configuration files contained on the Bootstrap Loader release diskette, assemble the files, and link them with the rest of the Bootstrap Loader object files and libraries. The following sections describe these files.

iSBC 204 Device Driver

The file B204.A86 is a device configuration file which places iSBC 204 device driver procedures in the Bootstrap Loader. This file is an assembly language source file which is contained on the Bootstrap Loader release diskette. Figure 11-2 lists the portion of B204.A86 that includes the driver procedures.

```
$include(:f2:b204.inc)

%b204(0A0H, 128, 26)
```

Figure 11-2. Driver Configuration File (B204.A86)

B204.A86 contains two statements, an $INCLUDE statement and a %B204 macro call. The $INCLUDE statement includes file B204.INC in the assembly of B204.A86. B204.INC contains the definition of the %B204 macro and is available on the Bootstrap Loader release diskette. The %B204 call causes the configuration of the iSBC 204 driver routines. You can modify the %B204 call to reflect your system. The format of the %B204 call is as follows:

```
%B204(io_base, sector_size, track_size)
```

where:

io_base	Base I/O port number which is selected on the iSBC 204 board.
sector_size	Sector size of the device in bytes.
track_size	Track size of the device in sectors.

The iSBC 204 device driver uses the following values for drive parameters:

parameter	value
step rate	25 milliseconds
head settling time	20 milliseconds
head load time	60 milliseconds

These values refer to 8-inch drives. The values are sufficient for most flexible diskette drives.

iSBC 206 Device Driver

The file B206.A86 is a device configuration file which places iSBC 206 device driver procedures in the Bootstrap Loader. This file is an assembly language source file which is contained on the Bootstrap Loader release diskette. Figure 11-3 lists the portion of B206.A86 that includes the driver procedures.

```
$include(:f2:b206.inc)

%b206(068H)
```

Figure 11-3. Driver Configuration File (B206.A86)

B206.A86 contains two statements, an $INCLUDE statement and a %B206 macro call. The $INCLUDE statement includes file B206.INC in the assembly of B206.A86. B206.INC contains the definition of the %B206 macro and is available on the Bootstrap Loader release diskette. The %B206 call causes the configuration of the iSBC 206 driver routines. You can modify the %B206 call to reflect your system. The format of the %B206 call is as follows:

```
%B206(io_base)
```

where:

io_base — Base I/O port number which is selected on the iSBC 206 board.

iSBC 215/220 Device Driver

The file B215.A86 is a device configuration file which places iSBC 215 or iSBC 220 device driver procedures in the Bootstrap Loader. This file is an assembly language source file which is contained on the Bootstrap Loader release diskette. Figure 11-4 lists the portion of B215.A86 that includes the driver procedures.

```
$include(:f2:b215.inc)

%b215(70H, 256, 2, 0, 9, 1024, 5)
```

Figure 11-4. Driver Configuration File (B215.A86)

B215.A86 contains two statements, an $INCLUDE statement and either a %B215 macro call or a %B220 macro call. The $INCLUDE statement includes file B215.INC in the assembly of B215.A86. B215.INC contains the definitions of the %B215 and %B220 macros and is available on the Bootstrap Loader release diskette. The %B215 call causes the configuration of the iSBC 215 driver routines. The %B220 call (with the same parameter values) causes the configuration of the iSBC 220 driver routines. You can modify either of these calls to reflect your system. The format of the two calls are as follows:

```
%B215(wakeup, cylinders, fixed_heads, removable_heads, sectors,
      dev_gran, alternates)
```

or

```
%B220(wakeup, cylinders, fixed_heads, removable_heads, sectors,
      dev_gran, alternates)
```

where:

wakeup	Base address of the wakeup port
cylinders	Number of cylinders on the disk drive or drives. All drives used by the Bootstrap Loader must have the same characteristics.
fixed_heads	Number of heads on fixed platters.
removable_-heads	Number of heads on removable platters.

sectors	Number of sectors per track.
dev_gran	Number of bytes per sector.
alternates	Number of alternate cylinders.

iSBC 254 Device Driver

The file B254.A86 is a device configuration file which places iSBC 254 device driver procedures in the Bootstrap Loader. This file is an assembly language source file which is contained on the Bootstrap Loader release diskette. Figure 11-5 lists the portion of B254.A86 that includes the driver procedures.

```
$include(:f2:b254.inc)

%b254(040H, 64, 1, 8192)
```

Figure 11-5. Driver Configuration File (B254.A86)

B254.A86 contains two statements, an $INCLUDE statement and a %B254 macro call. The $INCLUDE statement includes file B254.INC in the assembly of B254.A86. B254.INC contains the definition of the %B254 macro and is available on the Bootstrap Loader release diskette. The %B254 call causes the configuration of the iSBC 254 driver routines. You can modify the %B254 call to reflect your system. The format of the %B254 call is as follows:

```
%B254(io_base, dev_gran, num_boards, board_size)
```

where:

io_base	Base I/O port number which is selected on the iSBC 254 board.
dev_gran	Page size, in bytes.
num_boards	Reserved field which must be set to 1.
board_size	Size, in pages, of the iSBC 254 board.

USER-SUPPLIED PROCEDURES

If you have devices other than iSBC 204, 206, 215, 220, or 254 devices that you want to use with the Bootstrap Loader, you must write device driver routines for these devices, specify the addresses of these routines in the first stage configuration file, assemble them, and link them to the rest of Bootstrap Loader object files and libraries. You must supply the following two procedures for each type of device that you wish to support:

device initialization	This procedure must determine whether the device is ready and then perform any necessary initialization.
device read	This procedure must read data from the device.

The Bootstrap Loader expects each of these procedures to follow the PL/M-86 large model of computation, be of type FAR, and use 32-bit pointers. If you are coding your routines in PL/M-86, you should specify the ROM control in order to permit the Bootstrap Loader to function in ROM. The iRMX 86 LOADER REFERENCE MANUAL describes how to create these procedures.

You can use any names you want for your device initialization and device read procedures. However, you must specify the names of the procedures in the %DEVICE macro call for the device, when you create the first stage configuration file.

ASSEMBLING THE CONFIGURATION FILES, LINKING AND LOCATING THE BOOTSTRAP LOADER

After you have made any necessary modifications to the Bootstrap Loader configuration files, BS1.A86, B204.A86, B206.A86, B215.A86, and B254.A86 and have created any necessary device driver procedures, you should assemble these files and link and locate the Bootstrap Loader. BS1.CSD, a SUBMIT file contained on the Bootstrap Loader release diskette, can be used to perform these functions. In order to use this SUBMIT file, you must first prepare your diskettes and place them in the proper drives of your development system, as explained in "Linking and Locating the Subsystems" section of Chapter 4. You should also examine the configuration files to make sure that the $INCLUDE statements contain the proper disk identifiers. Then you can enter the following command:

```
SUBMIT :fx:BS1(date, rom_loc_addr, ram_loc_addr)
```

where:

fx	The appropriate disk identifier, indicating the drive containing BS1.CSD
date	The date on which you submit the file (maximum of nine characters).

rom_loc_addr — The address at which to locate the first stage of the Bootstrap Loader (the CODE segment). This address specifies the location of the ROM-resident portion of the Bootstrap Loader. If you want to specify a hexidecimal value for this parameter, you must use the suffix H (and the prefix 0, if the value begins with a letter).

ram_loc_addr — The address at which to locate the STACK, DATA, and BOOT segments of the Bootstrap Loader. This address specifies location of the RAM-resident portion of the Bootstrap Loader. If you want to specify a hexidecimal value for this parameter, you must use the suffix H (and the prefix 0, if the value begins with a letter).

If you have written and compiled your own device driver procedures, you should modify BS1.CSD in order to link these procedures in with the remainder of the Bootstrap Loader. To do this, place the names of your device driver object files in the LINK86 input list immediately before the line containing:

```
:f1:bs1.lib     &
```

If you plan to run the Bootstrap Loader on an iAPX 86-based microcomputer system that does not include an iSBC 957A monitor and you wish to use the console for input/output (%MANUAL or %CONSOLE calls present in the configuration file), you must supply procedures that read from and write to the console. The Bootstrap Loader release diskette includes a PL/M-86 source file which contains procedures to do this. You can examine this file, BCICO.P86, modify the procedures to suit your needs, compile it, and link it to the rest of the Bootstrap Loader object files and libraries. You can also use these routines as examples if you need to supply console input and console output functions for any of your other routines.

To include the read and write procedures as part of your Bootstrap Loader, you should add three lines to the BS1.CSD SUBMIT file. Add the first two lines immediately before the LINK86 statement, in order to compile the routines. These line are:

```
PLM86 :fx:BCICO.P86 LARGE ROM OPTIMIZE(3)     &
       PRINT(:fx:BCICO.LST) DATE(%0) CODE
```

Add the third line immediately following the LINK86 invocation, in order to link the routines in with the remainder of the Bootstrap Loader. This line is:

```
:F1:BCICO.OBJ,     &
```

If you include this line, LINK86 will generate a warning message similar to:

```
WARNING 25: EXTRA START ADDRESS IGNORED
```

This is a normal message; it does not indicate an error condition.

When locating the Bootstrap Loader, the location of the CODE segment determines which locations of ROM the Bootstrap Loader needs. The location of the STACK, DATA, and BOOT segments determines which locations of RAM the Bootstrap Loader needs. (The Bootstrap Loader reads its second stage into the BOOT segment during initialization.) You do not have to reserve memory for these RAM segments with %SAB macro calls.

After you have located the Bootstrap Loader, you should burn the code segment into PROM. The second stage portion of the loader will be placed on disk automatically, when you use the Files Utility or the Human Interface to format the disk. Refer to the iRMX 86 INSTALLATION GUIDE for further information concerning the Files Utility and the iRMX 86 HUMAN INTERFACE REFERENCE MANUAL for information concerning the Human Interface.

CHAPTER 12. CONFIGURING THE EXTENDED I/O SYSTEM

Extended I/O System configuration involves the following three operations:

- Selecting the system calls of the Extended I/O System that you want to include in your application system and discarding the rest.

- Selecting the logical devices that you want the Extended I/O System to initialize.

- Selecting I/O jobs that you want the Extended I/O System to create during system initialization.

You perform all of these operations by making modifications to the following files, all of which are contained on the Extended I/O System release diskette:

File	Purpose
ETABLE.A86	System call configuration
EDEVCF.A86	Logical device configuration
EJOBCF.A86	I/O job configuration

Figure 12-1 illustrates the structure of these files. As released, these files define the full complement of system calls, as well as a standard group of logical devices and jobs. To eliminate system calls from your Extended I/O System, or make changes to the logical device or job configuration, you must make changes to these files, assemble them, link them to the rest of the Extended I/O System object files and libraries, and locate the Extended I/O System at an absolute address. The remainder of this chapter describes these processes.

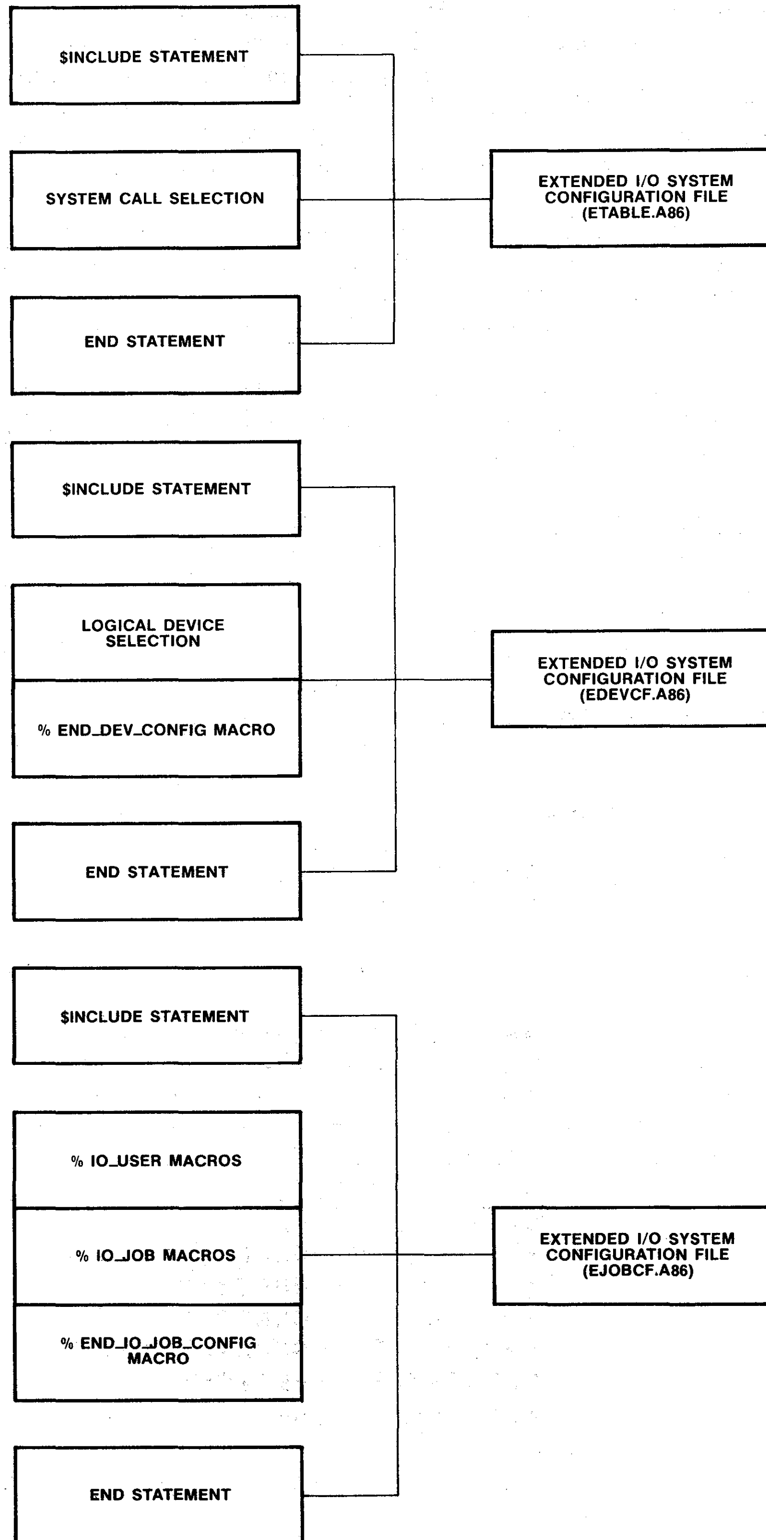


Figure 12-1. Structure of Extended I/O System Configuration Files

SELECTING SYSTEM CALLS (ETABLE.A86)

ETABLE.A86 consists of a series of macro calls which correspond in name to the system calls of the Extended I/O System. Each macro gives directions to the assembler to include code for that system call in the Extended I/O System. To exclude a system call from your Extended I/O System, delete the metacharacter (%) of the associated macro call, and replace it with the comment character (;). By doing this, you change the macro call into a comment and prevent the assembler from evaluating it.

The file ETABLE.MAC, which is available on the Extended I/O System release diskette, contains the definitions of all macros called in ETABLE.A86. ETABLE.A86 contains an $INCLUDE statement for ETABLE.MAC, which includes it in the assembly of ETABLE.A86.

Figure 12-2 lists the released ETABLE.A86 file. If you do not modify this file, it will include the full complement of Extended I/O System system calls in your application system.

```
NAME     ETABLE

$INCLUDE(:F2:ETABLE.MAC)

;
;     JOB INTERFACE
;
                     %RQCREATEIOJOB
                     %RQEXITIOJOB
;
;     CONFIGURATION INTERFACE
;
                     %RQLOGICALATTACHDEVICE
                     %RQLOGICALDETACHDEVICE
;
;    SYNCHRONOUS INTERFACE
;
                     %RQSCREATEFILE
                     %RQSATTACHFILE
                     %RQSDELETECONNECTION
                     %RQSLOOKUPCONNECTION
                     %RQSCATALOGCONNECTION
                     %RQSUNCATALOGCONNECTION
                     %RQSCREATEDIRECTORY
                     %RQSDELETEFILE
```

Figure 12-2. System Configuration File (ETABLE.A86)

```
%RQSRENAMEFILE
%RQSCHANGEACCESS
%RQSOPEN
%RQSCLOSE
%RQSREADMOVE
%RQSWRITEMOVE
%RQSSEEK
%RQSTRUNCATEFILE
%RQSGETFILESTATUS
%RQSGETCONNECTIONSTATUS
%RQSSPECIAL

END
```

Figure 12-2. System Call Configuration File (ETABLE.A86) (continued)

SELECTING LOGICAL DEVICES (EDEVCF.A86)

EDEVCF.A86 consists of a series of macro calls that associate logical names with physical device-units. When the Extended I/O System is initialized, it creates Logical Device Objects for the devices and catalogs these objects in the root job's object directory under the specified logical names. The first time these logical names are used as the prefix portions of path names, the Extended I/O System creates device connections.

One of the macros called in EDEVCF.A86 is the %DEV_INFO_BLOCK macro. The format of a call to this macro is very similar to the format of the LOGICAL$ATTACH$DEVICE system call (described in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL). The format is as follows:

```
%DEV_INFO_BLOCK('log_name', 'dev_name', file_driver)
```

where:

log_name	Logical name under which the device-unit is to be cataloged in the root object directory. This name can consist of one to twelve characters. You must enclose this name in single quotes.
dev_name	Name of the device-unit to be assigned a logical name. Use the name associated with the DUIB of this device-unit for this parameter. (DUIBs are described in the "Device-Unit Information Block" section of Chapter 9.) You must enclose this parameter in single quotes.

file_driver — Type of files which can reside on this device. Possible values include:

PHYSICAL
STREAM
NAMED

The other macro called in EDEVCF.A86 is the %END_DEV_CONFIG macro. The format of this macro is:

```
%END_DEV_CONFIG(buffer_size)
```

where:

buffer_size — Suggested size of the buffers that the Extended I/O System uses when it transfers information to and from files. The actual buffer size is the largest multiple of the device granularity that does not exceed the buffer_size value. The device granularity is a Basic I/O System configuration parameter (refer to Chapter 9).

A call to this macro specifies the buffer size and designates the end of the logical device configuration .

The file EDEVCF.MAC, which is available on the Extended I/O System release diskette, contains the definitions of the macros called in EDEVCF.A86. EDEVCF.A86 contains an $INCLUDE statement for EDEVCF.MAC, which includes it in the assembly of EDEVCF.A86.

Figure 12-3 lists EDEVCF.A86 as it is released with the Extended I/O System. This file contains %DEV_INFO_BLOCK macro calls for several commonly used device-units. You should modify this file to reflect your hardware environment.

```
NAME            EDEVCF

CGROUP          GROUP   CODE
                        ASSUME  CS : CGROUP

$INCLUDE(:F2:EDEVCF.MAC)

;
;     BYTE-BUCKET
;
          %DEV_INFO_BLOCK('BB','BB',PHYSICAL)
```

Figure 12-3. Logical Device Configuration File (EDEVCF.A86)

```
;
;     TERMINAL
;
          %DEV_INFO_BLOCK('TO','TO',PHYSICAL)
;
;     SHUGART 204, UNIT 0, DRIVE 0
;
          %DEV_INFO_BLOCK('FO','FO',NAMED)
;
;     SHUGART 204, UNIT 1, DRIVE 1
;
          %DEV_INFO_BLOCK('F1','F1',NAMED)
;
;     218 WINCHESTER FLOPPY SS/SD, UNIT 0, DRIVE 0
;
          %DEV_INFO_BLOCK('WFO','WFO',NAMED)
;
;     218 WINCHESTER FLOPPY SS/SD, UNIT 1, DRIVE 1
;
          %DEV_INFO_BLOCK('WF1','WF1',NAMED)
;
;     STREAM
;
          %DEV_INFO_BLOCK('STREAM','STREAM',STREAM)

          %END_DEV_CONFIG(1024)

                    END
```

Figure 12-3. Logical Device Configuration File (EDEVCF.A86) (continued)

SELECTING I/O JOBS (EJOBCF.A86)

EJOBCF.A86 consists of a series of macro calls that direct the Extended I/O System to create I/O jobs at system initialization time. The I/O jobs created become children of the Extended I/O System initialization job. The macros called in EJOBCF.A86 are:

%IO_USER
%IO_JOB
%END_IO_JOB_CONFIG

The file EJOBCF.MAC, which is available on the Extended I/O System release diskette, contains the definitions of all macros called in EJOBCF.A86. EJOBCF.A86 contains an $INCLUDE statement for EJOBCF.MAC, which includes it in the assembly of ETABLE.A86. Figure 12-4 lists EJOBCF.A86 as released with the Extended I/O System. This file contains macro calls for one typical job. You must modify this file to reflect the needs of your application system.

If you want to include the Human Interface in your configured system, you must modify EJOB.A86 to specify the Human Interface as an I/O job. Refer to Chapter 13 for further information.

```
AME             EJOBCF

GROUP           GROUP   CODE
                        ASSUME   CS : CGROUP

INCLUDE(:F2:EJOBCF.MAC)

  USER 'WORLD' DEFINITION

    %IO_USER('WORLD', OFFFFH)

  ETOS TEST JOB

    %IO_JOB('TO','WORLD',260H,OFFFFH,0:0,0,0,155,1800:0,1A00,0:0,1200,0)

      %END_IO_JOB_CONFIG(40)

                END
```

Figure 12-4. I/O Job Configuration File (EJOBCF.A86)

%IO_USER MACRO

The %IO_USER macro defines users that are later specified in the %IO_JOB calls. You must define each user with %IO_USER before you refer to it, and you must include one %IO_USER macro for each object that you define. The format of the call to %IO_USER is as follows:

    %IO_USER('user_name', user_id)

where:

user_name	Name of the user. This name must be enclosed in single quotes.

user_id	A 16-bit value that specifies the id of of the user .

Your EJOBCF.A86 file must contain at least one %IO_USER call which defines a user whose name is WORLD and whose id is OFFFFH. The released version of EJOBCF.A86 contains such a call.

%IO_JOB MACRO

The %IO_JOB macro defines the I/O jobs to be created. You must include one macro call for each I/O job that you want the Extended I/O System to create. The format of the call to the %IO_JOB macro is very similar to the format of the CREATEIOJOB system call. A short description of the %IO_JOB parameters is included in this section, but for a complete description refer to the description of the CREATEIOJOB system call in the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL. The format of the call to %IO_JOB is as follows:

```
%IO_JOB('default_prefix', 'default_user', pool_min, pool_max,
        excep_handler_addr, excep_mode, job_flags, task_prior,
        task_start_addr, data_segment, stack_addr, stack_size,
        task_flags)
```

where:

default_prefix	Logical name specifying the default prefix for the job. If you omit this parameter, the default prefix for the Extended I/O System's initialization job is used. This parameter must be enclosed in single quotes.
default_user	Name of the default user for this job. You must have previously defined this user name with a call to %IO_USER. This parameter must be enclosed in single quotes.
pool_min	Minimum allowable size of the new job's memory pool, in 16-byte paragraphs. The Extended I/O System uses this parameter as the initial size of the memory pool for the new job.
pool_max	Maximum allowable size of the new job's memory pool, in 16-byte paragraphs.
excep_handler_-addr	Hexadecimal pointer to the new job's default exception handler, in the form base:offset. A value of 0:0 indicates that the job uses the Extended I/O System exception handler. (The Extended I/O System exception handler is declared at system configuration time in the %JOB macro. Refer to the "%JOB Macro" section of Chapter 4 for further information.)

excep_mode — Encoded value which tells the Extended I/O System when to pass control to the exception handler. Encode this value as follows:

Value	Control Passes to Exception Handler
0	Never
1	On programmer errors only
2	On environmental conditions only
3	On all exceptional conditions

job_flags — Information that tells the Nucleus whether to validate objects used as parameters in system calls. Bits in this word are interpreted as follows:

bit	Meaning
15-2	Reserved.
1	If set to 0, the Nucleus validates objects used as parameters. If set to 1, the Nucleus performs no validation.
0	Reserved.

task_prior — Priority of the initial task in the newly created job. Specify a value in the range 0 to 255 decimal. A value of zero for this parameter indicates that the initial task has a priority equal to the maximum priority of initial job of the Extended I/O System.

task_start_addr — Hexadecimal pointer to the first instruction of the new job's initial task, in the form base:offset.

data_segment — Value to which the initial task's DS and ES registers are initialized. A value of zero indicates that the initial task assigns the data segment.

stack_addr — Hexadecimal pointer (in the form base:offset) of the stack for the initial task. A value of 0:0 causes the Nucleus to allocate a stack to the task and initialize the SS register to the base address of this segment and the SP register to the value of the stack_size parameter. It is recommended that you specify 0:0 for this parameter. This permits dynamic stack allocation.

stack_size — Size in bytes of the stack for the initial task.

task_flags — A Word that specifies whether the new job's initial task contains floating-point instructions. The bits (where bit 15 is the high-order bit) have the following meanings:

bit	meaning
15-1	Reserved.
0	If set to 1, the initial task uses floating-point instructions. These instructions require the 8087 NDP for execution. If set to 0, the initial task does not contain floating-point instructions.

%END_IO_JOB_CONFIG MACRO

The %END_IO_JOB_CONFIG macro indicates the end of I/O job configuration. You should place this macro call at the end of EJOBCF.A86. The format of the call to this macro is as follows:

```
%END_IO_JOB_CONFIG(dir_size)
```

where:

dir_size — Maximum allowable number of entries in the directories of I/O jobs. A value of zero indicates that no directories are to be created for I/O jobs.

ASSEMBLING THE CONFIGURATION FILES, LINKING AND LOCATING THE EXTENDED I/O SYSTEM

After you have made any necessary modifications to the Extended I/O System configuration files, ETABLE.A86, EDEVCF.A86, and EJOBCF.A86, you must assemble them and link and locate the Extended I/O System. EIOS.CSD, a SUBMIT file contained on the Extended I/O System release diskette, can be used to perform these functions. In order to use this SUBMIT file, you must prepare your diskettes and place them in the proper drives as explained in the "Linking and Locating the Subsystems" section of Chapter 4. You should also examine ETABLE.A86, EDEVCF.A86, and EJOBCF.A86 to make sure that the $INCLUDE statements contain the proper disk identifiers. You can then enter the following command:

```
SUBMIT    :fx:EIOS( date, loc_adr)
```

where:

fx — The appropriate disk identifier, indicating the drive containing EIOS.CSD.

date — The date on which you submit the file (maximum of nine characters).

loc_adr — The address at which to locate the Extended I/O System. If you want to enter this value as a hexadecimal number, you must include the suffix H. The base portion of this value is the base portion of the Extended I/O System's entry point. The offset portion of the entry point is 0. You must specify the entry point in the %JOB macro call for the Extended I/O System.

This command assembles ETABLE.A86, EDEVCF.A86, and EJOBCF.A86, links them together with the other modules of the Extended I/O System, and locates the Extended I/O System at the specified address. It places the located Extended I/O System in file EIOS on drive F1. It also places the assembly listing, link map, and locate map on drive F3 in files EIOS.LST, EIOS.MP1, and EIOS.MP2, respectively.

You must specify a %JOB macro in the system configuration file for the Extended I/O System (refer to Chapter 4). In this macro, the entry point depends on the address at which you locate the Extended I/O System (CS:0). The data segment should be specified as 0 (the Extended I/O System assigns its own data segment).

EXTENDED I/O SYSTEM INITIALIZATION

The Extended I/O System defines a public symbol, RQ$EIOS$INIT$ERROR, in which it returns its initialization status. If the Extended I/O System initializes properly, it attaches all logical devices specified in the configuration file (EDEVCF.A86), sets itself up as an operating system extension, and sets RQ$EIOS$INIT$ERROR to zero. If the Extended I/O System does not initialize correctly, it sets RQ$EIOS$INIT$ERROR to a nonzero value.

Once the initialization is complete, users can create and attach files on the devices specified in EDEVCF.A86 (unless they are off-line, in which case an exceptional condition code is returned). If one of these devices is switched from on-line to off-line, the Extended I/O System automatically marks all synchronous connections to that device as invalid (returns the EBADSYNC$CONN condition code) and detaches the device. When the unit is switched back on, the Extended I/O System automatically attaches it the first time a user tries to create or attach a file on it. The Extended I/O System performs this service only for all devices that it attaches.

CHAPTER 13. CONFIGURING THE HUMAN INTERFACE

Human Interface configuration involves the following operations:

- Designating prefixes and subpaths for the logical names required by the Human Interface
- Specifying the Human Interface sign-on
- Specifying the maximum number of characters in a command name
- Specifying the list of directories that the Human Interface searches when it tries to load a command.

You perform these operations by making modifications to an Intel-supplied Human Interface configuration file. This file, HCONFG.A86, is a PL/M-86 source file which is contained on the Human Interface release diskette. As released, HCONFG.A86 defines a default Human Interface. To make changes, you must modify HCONFG.A86, compile it, link it with the rest of the Human Interface object files and libraries, and locate the Human Interface at an absolute address. The following sections describe this configuration process.

MODIFYING HCONFG.A86

HCONFG.A86 consists of a series of PL/M-86 DECLARE statements which identify the characteristics of the Human Interface. Figure 13-1 lists the portion of the released HCONFG.A86 that you can modify. To change the configuration information, you must modify the values associated with the variables in the DECLARE statements. The following paragraphs list the variables you can change.

```
HCONFG: DO;

              /* path names that will represent default directories */

DECLARE
    H$system$directory(*)  BYTE  PUBLIC DATA(10,':F0:SYSTEM'), /* :SYSTEM
    H$prog$directory(*)    BYTE  PUBLIC DATA(8,':F0:PROG'),    /* :PROG:
    H$default$dir(*)       BYTE  PUBLIC DATA(4,':F0:'),        /* :$: */
    H$work$directory(*)    BYTE  PUBLIC DATA(8,':F0:WORK');    /* :WORK:

DECLARE
    H$signon(*)  BYTE  PUBLIC DATA(15,'iRMX 86 HI V1.0'),

    H$command$name$max  WORD  PUBLIC DATA(50),  /* command name max size

    H$prefixes(*)  BYTE  PUBLIC DATA(2,  /* number of prefixes */
         6,':PROG:',      /* first prefix */
         8,':SYSTEM:');   /* second prefix */
```

Figure 13-1. Human Interface Configuration File (HCONFG.P86)

The first four variables define paths for which the Human Interface assigns logical names. These paths are specified in the form of STRINGs. The first portion of each string (the length) is a number which equals the number of characters in the second portion of the string. The second portion of the string consists of the prefix and subpath. Maintain this format when modifying the variables. If you specify a 0 for the length of any string, the Human Interface will not create the corresponding logical name.

H$SYSTEM$DIRECTORY(*) — A BYTE array containing a STRING which defines the prefix and subpath of a directory that the Human Interface associates with logical name :SYSTEM:.

H$PROG$DIRECTORY(*) — A BYTE array containing a STRING which defines the prefix and subpath of a directory that the Human Interface associates with logical name :PROG:.

H$DEFAULT$DIR(*) — A BYTE array containing a STRING which defines the prefix that the Human Interface associates with logical name :$:.

H$WORK$DIRECTORY(*)	A BYTE array containing a STRING which defines the prefix that the Human Interface associates with logical name :WORK:.

The following variable is also specified in the form of a STRING. It defines the Human Interface sign-on.

H$SIGN$ON(*)	A BYTE array containing a STRING which defines the Human Interface sign-on characters. These characters are displayed on the user terminal when the Human Interface begins running.

The next variable defines the number of characters in a command name.

H$COMMAND$NAME$MAX	A WORD specifying the maximum number of characters in a command name. This includes the prefix and subpath portions of the command name.

The following variable defines directories that the Human Interface searches when looking for a user-specified file. These directories are specified in the form of a STRING table. A STRING table is a BYTE array whose first byte specifies the number of strings in the table. The remaining bytes of the STRING table specify the actual STRINGs.

H$PREFIXES(*)	A BYTE array, in the form of a STRING table, indicating the directories that the Human Interface searches, in order, when looking for user-specified files. As many as 255 directories can be specified in this STRING table. The STRINGs can contain either logical names (for existing files) or pathnames. When the Human Interface user specifies a pathname that does not begin with ($) or (:), the Human Interface appends the pathname from the command line to the end of the first directory name in the STRING table and searches the resulting path for the file. If the file is not found, the Human Interface repeats the process for the remaining directories. If a directory name in the STRING table includes a pathname (that is, it is more than just a logical name), it must include the (/) as the last character. The (/) is needed because the Human Interface appends the user-specified file name directly to the end of the directory name.

HCONFG.A86 also contains definitions of variables that are not described in this manual. These variables are defined in the configuration file to allow for future expansion of the configuration options. They are not currently user selectable. Do not modify the value of any variable that is not described in this section.

COMPILING HCONFG.A86, LINKING AND LOCATING THE HUMAN INTERFACE

HI.CSD, a SUBMIT file contained on the Human Interface release diskette, can be used to link and locate the Human Interface. In order to use this SUBMIT file, you must first prepare your diskettes and place them in the proper drives of your development system, as explained in the "Linking and Locating the Subsystems" section of Chapter 4. Then you can enter the following command:

```
SUBMIT    :fx:HI( date, loc_adr)
```

where:

fx	The appropriate disk identifier, indicating the drive containing HI.CSD.
date	The date on which you submit the file (maximum of nine characters).
loc_adr	The address at which to locate the Human Interface. If you want to enter this value as a hexadecimal number, you must include the suffix H. The base portion of this value is the base portion of the Extended I/O System's entry point. The offset portion of the entry point is 0. You must specify the entry point in the %IO_JOB macro call for the Human Interface. You specify this macro call during Extended I/O System configuration.

This command compiles HCONFG.P86, links together the modules that make up the Human Interface, and locates the Human Interface at the specified address. It places the located Human Interface in file HI on drive F1. It also places the link and locate maps on drive F3 in files HI.MP1 and HI.MP2 respectively.

Unlike the other iRMX 86 subsystems, the Human Interface does not require a %JOB macro in the system configuration file. Instead, the Extended I/O System must create the Human Interface as an I/O job. To ensure that this process takes place, you must include an %IO_JOB macro for the Human Interface in the Extended I/O System configuration file (EJOBCF.A86). Refer to Chapter 12 for more information about the %IO_JOB macro. In this %IO_JOB macro, the entry point depends on the address at which you locate the Human Interface (CS:0). The data segment base should be specified as 0 (the Human Interface initializes its own data segment).

HUMAN INTERFACE REQUIREMENTS

In order to run the Human Interface, you must include the following iRMX 86 subsystems in your application system.

Nucleus
Debugger or Terminal Handler
Basic I/O System
Extended I/O System
Application Loader
Human Interface

The Nucleus, Basic and Extended I/O Systems, and the Application Loader must be configured with the system calls required by the Human Interface. Appendix C lists these requirements. The following sections outline any additional requirements of the subsystems.

TERMINAL HANDLER OR DEBUGGER REQUIREMENTS

The Human Interface communicates to the terminal via the Basic I/O System, which in turn uses the Terminal Handler (or the Debugger's Terminal Handler). In order for this to happen, the Basic I/O System requires that a Terminal Handler with input and output mailbox names RQTHNORMIN and RQTHNORMOUT, respectively, be present in the application system. These mailbox names are configuration options of the Terminal Handler. Refer to Chapter 7 for further information. The Basic I/O System is unable to communicate through a Terminal Handler with different input and output mailbox names.

The Human Interface library, HI.LIB, contains a module that implements control-C semantics. To use this module, you must link it to the Debugger or the Terminal Handler, depending on which you use with the Human Interface. To link this module, modify the Terminal Handler or Debugger SUBMIT file (MTH.CSD or DB.CSD) to include the following line:

```
:fx:HI.LIB(HCONTC),          &
```

Place this line in the LINK86 input list in the place designated for the control-C semantics file, as indicated in Chapters 7 and 8.

BASIC I/O SYSTEM REQUIREMENTS

The Human Interface uses the Basic I/O System to perform I/O to secondary storage devices. In order to run the Human Interface, the Basic I/O System must contain the following file drivers:

physical
stream
named

Ensure that these file drivers are selected in the Basic I/O System configuration file (ITABLE.A86). Refer to Chapter 9 for information.

The Basic I/O System must also contain DUIBs for the devices used by the Human Interface. In particular, the IDEVCF.A86 configuration file must contain DUIBs for devices with the following device names:

TO	Terminal device
BB	Byte bucket device
STREAM	Stream file device

The released version of IDEVCF.A86 contains DUIBs for these devices. You must ensure that IDEVCF.A86 contains DUIBs for any other devices used by the Human Interface, such as disk drives.

The Basic I/O System must also contain device drivers for each of the devices used by the Human Interface. In particular, you must ensure that IDEVCF.A86 includes the On Board USART driver to enable the I/O System to communicate with the terminal via the Terminal Handler (or the Debugger's Terminal Handler).

EXTENDED I/O SYSTEM REQUIREMENTS

The TO, BB, and STREAM devices must be logically attached when the Human Interface starts processing. Therefore, the Extended I/O System configuration file (EDEVCF.A86) must contain %DEV_INFO_BLOCK macro calls for each of these devices. The macro calls should assign the logical names to be the same as the device names. As released, EDEVCF.A86 contains macro calls for these devices. Refer to Chapter 12 for further information.

The Extended I/O System must create the Human Interface as an I/O job. Thus EJOBCF.A86 must contain an %IO_JOB macro call for the Human Interface. The default prefix for the Human Interface job is the logical name of the Human Interface's terminal (TO). As released, EJOBCF.A86 contains a macro call for the Human Interface job.

When specifying the %JOB macro for the Extended I/O System, you must ensure that its memory pool is large enough to include the Human Interface. To do this, you can specify a value of OFFFFH for the pool_max parameter (refer to Chapter 4 for more information about the %JOB macro). You should also specify a value of OFFFFH for the pool_max parameter in the Human Interface's %IO_JOB macro call. However, if you specify OFFFFH for the Extended I/O System and the Human Interface, it is recommended that you specify equal pool_min and pool_max values in all other %JOB and %IO_JOB calls to prevent these other jobs from borrowing memory.

CREATING HUMAN INTERFACE VOLUMES

Before you can initially use the Human Interface, you must create iRMX 86 volumes that are properly formatted and contain the Human Interface commands. Intel supplies one such volume in the iRMX 86 release package. This volume is a preconfigured flexible diskette in iRMX 86 format (with a granularity of 128 bytes) that contains the Human Interface commands. You can use this diskette if your iRMX 86 system contains a flexible diskette drive connected to either an iSBC 204 controller or an iSBC 215/218 controller.

If your iRMX 86 system does not contain a flexible diskette drive, you must use the Files Utility to create your first Human Interface volume. Use the Files Utility FORMAT command to format the iRMX 86 volume and the Files Utility UPCOPY command to copy the Human Interface commands from the Human Interface release diskette to the formatted iRMX 86 volume. Refer to the iRMX 86 INSTALLATION GUIDE for a description of the Files Utility.

CREATING HUMAN INTERFACE COMMANDS

One of the primary functions of the Human Interface is to execute files of object code contained on secondary storage devices. It does this by loading the code into iRMX 86 memory and creating jobs for this code. The Human Interface is released with several such files which contain the Intel-supplied Human Interface commands. You can also create your own files of object code for the Human Interface to load as jobs. The procedure for creating your own commands depends on the kind of development system you use.

USING A SERIES III DEVELOPMENT SYSTEM

If you use a Series III development system to develop your commands, you can produce load-time locatable code (LTL), position independent code (PIC), or absolute code. To create LTL or PIC commands, which the Human Interface can load anywhere in dynamic memory, perform the following steps:

1. Compile your code using the PL/M-86 compiler or assemble it using the 8086/8087/8088 Macro Assembler.

2. Use LINK86 to link your code together with the necessary libraries and create an LTL or PIC module. Enter the LINK86 command as follows:

```
RUN     :fx:LINK86                                                 &
        :fx:command.obj,                                           &
        :fx:HPIFC.LIB,                                             &
        :fx:LPIFC.LIB,                                             &
        :fx:EPIFC.LIB,                                             &
        :fx:IPIFC.LIB,                                             &
        :fx:RPIFC.LIB                                              &
    TO  :fx:command                                                &
        PRINT(:fx:command.mpl) SYMBOLCOLUMNS(2)                    &
        OBJECTCONTROLS(PURGE)                                      &
        PRINTCONTROLS(LINES, PUBLICS, NOCOMMENTS, SYMBOLS)  &
        BIND SEGSIZE(STACK(stacksize)) MEMPOOL(minsize,maxsize)
```

where:

:fx:	The appropriate disk identifiers, indicating the drives containing the modules.
command.obj	File containing the assembled or compiled object code for your command. You can link in several files or libraries at this point, if necessary.
HPIFC.LIB LPIFC.LIB EPIFC.LIB IPIFC.LIB RPIFC.LIB	Interface libraries for the Human Interface, Application Loader, Extended I/O System, Basic I/O System, and Nucleus. These libraries satisfy external references caused by making system calls.
command	File on which LINK86 places the linked command. After transferring this file to an iRMX 86 secondary storage device, you can invoke the command by entering its pathname.
command.mpl	File on which LINK86 places the link map.
stacksize	Size, in bytes, of the stack needed by the command and any system calls that the command makes. The Human Interface uses this value when it creates a job for the command.
minsize maxsize	Minimum and maximum sizes of the command's dynamic memory pool, in bytes. The Human Interface uses these values when it creates a job for the command.

The code is now ready to be transferred to an iRMX 86 secondary storage device. You do not need to process the code further with LOC86.

3. Use the Files Utility or the Human Interface UPCOPY command to copy the linked command from the development system disk to an iRMX 86 secondary storage device. At this point you can invoke the command by entering its pathname at the Human Interface terminal.

You can also create your commands as absolute object modules, if you wish. To do this, use the output file produced by LINK86 as input to LOC86, and use the ADDRESSES control to specify absolute addresses for the code.

There are limitations to commands containing absolute code. The next section discusses these limitations further.

USING A SERIES II DEVELOPMENT SYSTEM

If you use a Series II development system to develop your commands, you can produce only absolute object code which the Human Interface must load into one particular area of memory. To create absolute commands, perform the following steps:

1. Compile your code using the PL/M-86 compiler or assemble it using the 8086/8087/8088 Macro Assembler.

2. Use LINK86 to link your code together with the necessary iRMX 86 interface libraries. Enter the LINK86 command as follows:

```
:fx:LINK86                                              &
    :fx:command.obj,                                    &
    :fx:HPIFC.LIB,                                      &
    :fx:LPIFC.LIB,                                      &
    :fx:EPIFC.LIB,                                      &
    :fx:IPIFC.LIB,                                      &
    :fx:RPIFC.LIB                                       &
TO  :fx:command.lnk  PRINT(:fx:command.mpl)
```

   where the parameters of this command mean the same as they do when entered with the Series III version of the LINK86 command. Notice that when using the Series II development system, you do not preface the LINK86 command with RUN and you do not use the SYMBOLCOLUMNS, OBJECTCONTROLS, PRINTCONTROLS, BIND, MEMPOOL, and SEGSIZE controls. These are not supported with the Series II version of LINK86. With the exception of BIND and MEMPOOL, you enter the omitted controls in the LOC86 command.

3. Use LOC86 to assign absolute addresses to the linked module created by LINK86. Enter the LOC86 command as follows:

```
:fx:LOC86  :fx:command.lnk  TO  :fx:command                &
           ORDER (CLASSES (CODE, DATA, STACK, MEMORY))     &
           ADDRESSES (CLASSES (CODE (absolute_address)))   &
           SEGSIZE (STACK (stacksize))                     &
           MAP PRINT (:fx:command.mp2) SYMBOLCOLUMNS(2)    &
           OBJECTCONTROLS(PURGE)                           &
           PRINTCONTROLS(LINES,PUBLICS,NOCOMMENTS,SYMBOLS)
```

   where:

command.lnk	Name of the link file produced previously by LINK86.

command	Name of the file in which LOC86 writes the absolute module. After transferring this file to an iRMX 86 secondary storage device, you can invoke the command by entering its pathname.
absolute_address	Absolute starting location of the code segment of the command. LOC86 locates the remaining segments after the code segment. You must reserve these areas of iRMX 86 memory during configuration with the %SAB macro (refer to Chapter 4).
stacksize	Size, in bytes, of the stack needed by the command and any system calls that the command makes.
command.mp2	Name of the file in which LOC86 writes the locate map.

4. Use the Files Utility or the Human Interface UPCOPY command to copy the located command from the development system disk to an iRMX 86 secondary storage device. If you have reserved the areas of iRMX 86 memory that the command needs with the %SAB macro during system configuration, you can invoke the command by entering its pathname at the Human Interface terminal.

ADDITIONAL REQUIREMENTS FOR ABSOLUTE CODE

When the commands you create contain absolute object code, you must take steps during the configuration process to ensure that this code loads and executes correctly, without affecting the remainder of the application system.

Since absolute code contains the addresses at which it is to reside as part of the code, the Application Loader, when called by the Human Interface, cannot load this code at any convenient place in iRMX 86 memory. Instead, it must load this code at the exact place specified in the code. If that place in iRMX 86 memory contains other objects (as it might if the memory is part of a dymanic memory pool), the act of loading the file can harm or destroy other tasks. If the place in memory contains part of the Operating System, the results can be worse. In order to ensure that no damage occurs when the Application Loader loads absolute files, you must use the %SAB macro call to reserve areas into which the Application Loader (when called by the Human Ingerface) will later load code (refer to Chapter 4 for a description of the %SAB macro). If you do this, no other objects will use the specified areas of memory, and the Application Loader can safely load your commands into that area for execution.

If you create your commands as LTL or PIC modules on a Series III development system, you do not have to reserve memory with %SAB macros. The Application Loader loads LTL and PIC modules into convenient areas of dynamic memory.

APPENDIX A. EXAMPLE SYSTEM CONFIGURATION

One of the iRMX 86 release diskettes contains a demonstration system. This system consists of the following:

- Nucleus
- Debugger
- root job
- application job consisting of a BASIC interpreter

The system contained on the release diskette has already been configured. In order to run it, make sure that your hardware is assembled correctly and load the system into your iSBC 86/12A single board computer. Refer to the iRMX 86 INSTALLATION GUIDE for further information on using this system. This appendix, however, shows how to use the procedures described previously in this manual and build the demonstration system from its individual parts.

In order to build the demonstration system, this appendix assumes that you have the following:

- An INTELLEC Series II Microcomputer Development System with at least four disk drives.
- A system disk containing ASM86, LINK86, and LOC86.
- The Nucleus release diskette, the Debugger release diskette, the demonstration system release diskette, and the iSBC 957A release diskette.

This appendix uses the SUBMIT files provided on the subsystem release diskettes and described in the previous chapters of this manual to link and locate the Nucleus and Debugger. The demonstration system release diskette contains different SUBMIT files that link and locate the Nucleus, Debugger, and TBASIC interpreter. These SUBMIT files do not, however, follow the disk drive conventions outlined in Chapter 4. They assume that you have only two disk drives in your development system. You can use the files on the demonstration system release diskette, but this appendix does not describe the commands used to submit them.

This Appendix also makes assumptions about terminal characteristics, naming files, and placing files on diskettes. These are made for convenience, not out of necessity. It also assumes that you are going to use the iSBC 957A package to load the system into the iSBC 86/12A single board computer.

PREPARE A MEMORY MAP

The first step in the configuration process is preparing the memory map to describe the general layout of the system. Figure A-1 contains such a memory map.

There are several things to notice about this memory map. They are:

- The highest RAM address recorded indicates that this system has 128K of RAM.
- The iSBC 957A package will be used to load this system into memory. Thus space is allocated in the memory map for the iSBC 957A monitor.
- Space is left for the Bootstrap Loader, should you want to add it to your application system at a later date. The space left in this example is sufficient for most configurations of the Bootstrap Loader. However, if you intend to bootstrap from a device with a 1024-byte sector size or larger, you should leave more room, starting your system at 200:0.
- The modules will be located in memory as described in Chapter 4.

iRMX 86™ SYSTEM MEMORY MAP WORKSHEET

Configuration file name: ______________________

Start address/ Data segment base	Module	Length	Absolute Address
			FFFF:F
	(reserved)		
	reset vector		FFFF:0
	iSBC 957A monitor		FE00:0
	Highest RAM address		1FFF:F
			______
	Application Job		______
			______
			______
	Root job		______
			______
			______
	Debugger		______
			______
			______
	Nucleus		1C0:0
			1BF:F
	(space for Bootstrap Loader)		6F:F
	iSBC 957A monitor data		40:0
	Interrupt vector		0:0

Figure A-1. Preparing the Memory Map

CONFIGURE THE SUBSYSTEMS AND LINK AND LOCATE THE SYSTEM

The next steps you must perform involve preparing configuration files for the Nucleus and the Debugger, assembling these files, and linking and locating all of the pieces of your system. You should fill out the memory map each time you locate a module, in order to keep track of the modules and the memory that they require. Figure A-2 shows a filled out memory map. The following sections refer to this figure.

iRMX 86™ SYSTEM MEMORY MAP WORKSHEET

Configuration file name: ____________________

Group	Start address/ Data segment base	Module	Length	Absolute Address
				FFFF:F
		(reserved)		
		reset vector		
				FFFF:0
		iSBC 957A monitor		
				FE00:0
		Highest RAM address		
				1FFF:F
D	SS=11C5:0			
				127A:0
D	length=0A04	Application Job		
				0E70:0
D	CSTART=0E70:90			
				______
				______
C		Root job		
				0E10:0
				______
				0E0E:F
B		Debugger		
				6B2:0
				______
				6B1:B
A		Nucleus		
				1C0:0
A				
				1BF:F
A		(space for Bootstrap Loader)		
				6F:F
A		iSBC 957A monitor data		
				40:0
A		Interrupt vector		
				0:0

Figure A-2. Completed Memory Map

PREPARE, LINK, AND LOCATE THE NUCLEUS

You should start the configuration process with the Nucleus. To do this, place the diskettes in the proper drives of the development system (system diskette in drive F0, configuration diskette in drive F1, Nucleus release diskette in drive F2, and scratch and listing diskette in drive F3). Then copy the Nucleus configuration file, NTABLE.A86, from the release diskette to your configuration diskette. Modify NTABLE.A86 so that it includes only those features and system calls that your application system requires. (You do not need to modify NDEVCF.A86 if you are running on the iSBC 86/12A board.) Figure A-3 shows a modified version of this file that specifies the system calls and features needed by the TBASIC interpreter.

```
$INCLUDE(:F2:NTABLE.MAC)
$EJECT
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;       NUCLEUS FEATURE CONFIGURATION TABLE
;
;       TO LEAVE OUT A FEATURE, CHANGE THE '%' TO THE COMMENT
;       CHARACTER ';' .
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

                %PARAMETER_VALIDATION
                %SYSTEM_EXCEPTION_HANDLER

$EJECT
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;       NUCLEUS PRIMITIVE CONFIGURATION TABLE
;
;       TO CONFIGURE A NUCLEUS PRIMITIVE OUT OF THE iRMX 86 SYSTEM,
;       REPLACE THE '%' BY THE COMMENT CHARACTER ';'.  CALLS TO
;       PRIMITIVES NOT CONFIGURED IN THE SYSTEM WILL RESULT IN AN
;       E$NOT$CONFIGURED EXCEPTIONAL CONDITION.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


                %RQGETTYPE
                %RQDISABLEDELETION
                %RQENABLEDELETION
                %RQCATALOGOBJECT
                %RQUNCATALOGOBJECT
                %RQLOOKUPOBJECT
                ;RQCREATEEXTENSION
                ;RQDELETEEXTENSION
                ;RQCREATECOMPOSITE
```

Figure A-3. Example Nucleus Configuration File

```
                  ;RQDELETECOMPOSITE
                  ;RQINSPECTCOMPOSITE
                  ;RQALTERCOMPOSITE
                  ;RQFORCEDELETE
                  %RQCREATEJOB
                  ;RQDELETEJOB
                  ;RQOFFSPRING
                  %RQCREATETASK
                  %RQDELETETASK
                  %RQSUSPENDTASK
                  %RQRESUMETASK
                  %RQSLEEP
                  %RQGETTASKTOKENS
                  %RQGETPRIORITY
                  ;RQSETPRIORITY
                  %RQCREATEMAILBOX
                  %RQDELETEMAILBOX
                  %RQSENDMESSAGE
                  %RQRECEIVEMESSAGE
                  %RQCREATESEMAPHORE
                  %RQDELETESEMAPHORE
                  %RQSENDUNITS
                  %RQRECEIVEUNITS
                  ;RQCREATEREGION
                  ;RQDELETEREGION
                  ;RQSENDCONTROL
                  ;RQRECEIVECONTROL
                  ;RQACCEPTCONTROL
                  %RQCREATESEGMENT
                  %RQDELETESEGMENT
                  %RQGETSIZE
                  ;RQGETPOOLATTRIB
                  ;RQSETPOOLMIN
                  ;RQSETOSEXTENSION
                  %RQSETINTERRUPT
                  ;RQENTERINTERRUPT
                  %RQENABLE
                  %RQDISABLE
                  ;RQRESETINTERRUPT
                  ;RQGETLEVEL
                  %RQEXITINTERRUPT
                  %RQSIGNALINTERRUPT
                  %RQWAITINTERRUPT
                  ;RQGETEXCEPTIONHANDLER
                  ;RQSETEXCEPTIONHANDLER
                  %RQSIGNALEXCEPTION

END
```

Figure A-3. Example Nucleus Configuration File (continued)

Next, copy the Nucleus SUBMIT file, NUCLUS.CSD from the release diskette on drive F2 to the configuration diskette on drve Fl. Modify the ASM86 command in NUCLUS.CSD so that it reads the configuration file (NTABLE.A86) from drive Fl instead of from drive F2. Then call the version of NUCLUS.CSD that resides on Fl to assemble the configuration files and link and locate the Nucleus. From the memory map in Figure A-1, you can see that the Nucleus should be located at address 1C0:0. This allows room for the interrupt vector and the iSBC 957A monitor at lower addresses, and also allows you to include the Bootstrap Loader in your application system at a later date. Therefore, enter the following command:

```
SUBMIT :Fl:NUCLUS(date, 1C00H)
```

where date is the date on which you submit the file. NUCLUS.CSD places the located Nucleus in file NUCLUS on drive Fl. It places the locate map in file NUCLUS.MP2 on drive F3. Figure A-4 shows the important parts of the Nucleus locate map.

```
ISIS-II MCS-86 LOCATER, V1.3 INVOKED BY:
LOC86                                        &
      :f3:nuclus.lnk  TO :f1:nuclus          &
      MAP PRINT(:f3:nuclus.mp2)              &
      NOLINES NOCOMMENTS NOSYMBOLS           &
      SEGSIZE(stack(0))                      &
      ORDER(classes(code, data))             &
      ADDRESSES(classes(code(01C00H)))
WARNING 26:  DECREASING SIZE OF SEGMENT
  SEGMENT:  STACK

SYMBOL TABLE OF MODULE NBEGIN
READ FROM FILE :F3:NUCLUS.LNK
WRITTEN TO FILE :F1:NUCLUS

BASE    OFFSET TYPE SYMBOL                    BASE    OFFSET TYPE SYMBOL

01C0H   0000H   PUB  NBEGIN

MEMORY MAP OF MODULE NBEGIN
READ FROM FILE :F3:NUCLUS.LNK
WRITTEN TO FILE :F1:NUCLUS

SEGMENT MAP

START      STOP      LENGTH ALIGN NAME                  CLASS

00000H     003FFH     0400H   A    (ABSOLUTE)
01C00H     0645BH     485CH   W    CODE                  CODE
0645CH     06475H     001AH   W    OBJ_SEG               CODE
                        .
                        .
                        .
```

Figure A-4. Nucleus Locate Map

```
069E0H    069F3H     0004H   G   LIB_87_INIT
069E4H    069F4H     0000H   W   STACK              STACK
069F0H    06B1BH     012CH   G   ISTACK             STACK
06B1CH    06B1CH     0000H   W   MEMORY             MEMORY <--(A1)


GROUP MAP

ADDRESS   GROUP OR SEGMENT NAME
06550H    DGROUP
          DATA
          READYLISTROOT_DATA
          KSTACK
          DELAYLISTROOT_DATA
          IDLETASK_DATA
          ISTACK
          INTVEC_REG_SEG
          EXT_REG_SEG
          JOB_REG_SEG
          SEM_REG_SEG
          MAIL_REG_SEG
          OD_REG_SEG
          POOL_REG_SEG
          DELETION_REG_SEG
01C00H    CGROUP
          CODE
          OBJ_SEG
          JOB_SEG
          TASK_SEG
          MB_SEG
          SEM_SEG
          REG_SEG
          FS_SEG
          INT_SEG
             •
             •
             •
```

Figure A-4. Nucleus Locate Map (continued)

As you can see from arrow A1 in Figure A-4, the next available memory location is 6B1:C. The last location used by the Nucleus was 6B1:B. Record these values on the memory map as shown in portions A and part of B in Figure A-2.

PREPARE, LINK, AND LOCATE THE DEBUGGER

You should continue the configuration process with the Debugger. This example assumes that you can use the released versions of the Debugger SUBMIT file and configuration file. Thus you do not need to copy any files to your configuration diskette and make modifications. Just make sure that you place the diskettes in the proper drives of your development system (system diskette in drive F0, configuration diskette in drive F1, Debugger release diskette in drive F2, and scratch and listing diskette in drive F3). Then call the Debugger SUBMIT file, DB.CSD, to assemble the DTHCNF.A86 and link and locate the Debugger. Since you have already located the Nucleus, you know that the last memory location used is 6B1BH. Therefore, use a figure of 6B20H in the call to DB.CSD. This call appears as follows:

```
SUBMIT :F2:DB(date, 6B20H)
```

where date is the date on which you submit the file. DB.CSD places the located Debugger in file DB on drive F1. It places the locate map in file DB.MP2 on drive F3. Figure A-5 shows the important parts of this locate map.

```
ISIS-II MCS-86 LOCATER, V1.3 INVOKED BY:
LOC86                                            &
       :f3:db.lnk  TO :f1:db                     &
       MAP PRINT(:f3:db.mp2)                     &
       NOLINES NOCOMMENTS NOSYMBOLS              &
       SEGSIZE(stack(0))                         &
       ORDER(classes(code, data))                &
       ADDRESSES(classes(code(06B20H)))
WARNING 26:  DECREASING SIZE OF SEGMENT
  SEGMENT:  STACK

SYMBOL TABLE OF MODULE DBUGA
READ FROM FILE :F3:DB.LNK
WRITTEN TO FILE :F1:DB

BASE   OFFSET TYPE SYMBOL                   BASE   OFFSET TYPE SYMBOL

06B2H  0000H  PUB  RQDBUGINIT

MEMORY MAP OF MODULE DBUGA
READ FROM FILE :F3:DB.LNK
WRITTEN TO FILE :F1:DB

SEGMENT MAP

START    STOP     LENGTH ALIGN NAME                CLASS

06B20H   0DBFDH   70DEH   W    CODE                CODE
0DBFEH   0DC03H   0006H   B    TH_CNF_SEG1         CODE
```

Figure A-5. Debugger Locate Map

```
0DC04H    0DC09H     0006H    B    TH_CNF_SEG2        CODE
0DC0AH    0DC11H     0008H    B    TH_CNF_SEG3        CODE
0DC12H    0DC15H     0004H    B    TH_CNF_SEG4        CODE
0DC16H    0DC17H     0002H    B    TH_CNF_SEG5        CODE
0DC18H    0DC22H     000BH    B    NORM_IN_MBX_SE     CODE
                                  -G
0DC23H    0DC2EH     000CH    B    NORM_OUT_MBX_S     CODE
                                  -EG
0DC30H    0E0EFH     04C0H    W    DATA               DATA
0E0F0H    0E0F0H     0000H    G    ??SEG
0E0F0H    0E0F0H     0000H    W    STACK              STACK
0E0F0H    0E0F0H     0000H    W    MEMORY             MEMORY <--(B1)


GROUP MAP

ADDRESS   GROUP OR SEGMENT NAME
0DC30H    DGROUP
          DATA
06B20H    CGROUP
          CODE
          TH_CNF_SEG1
          TH_CNF_SEG2
          TH_CNF_SEG3
          TH_CNF_SEG4
          TH_CNF_SEG5
          NORM_IN_MBX_SEG
          NORM_OUT_MBX_SEG
```

Figure A-5. Debugger Locate Map (continued)

Arrow B1 shows the only important piece of information in Figure A-5 that you should record on the memory map. It identifies the next available memory location, address 0E0F:0. The last location used by the Debugger is 0E0E:F. Record these addresses on the memory map. This is shown in portions B and C of Figure A-2. You need to know the next available address in order to leave enough space for the root job and correctly locate the application job.

The entry point of the Debugger is determined from the address at which you locate the Debugger. The base portion of the entry point is the base portion of the location address (6B2). The offset portion of the entry point is 0. Thus the Debugger entry point is 6B2:0. You must supply this address later in the %JOB macro call for the Debugger.

ALLOW SPACE FOR THE ROOT JOB

Use the address 0E10:0 as the starting address of the root job. Record this value and the estimated size on the memory map. You do not know the exact size since you have not created the root job yet. However, for this system, use a size estimate of 600H bytes. Add this value to the starting address and record the result, 0E70:0, on the memory map. Use this value as the starting address of the application job. Portion C of Figure A-2 shows this.

LINK THE APPLICATION JOB

Next, use LINK86 to link the modules of the TBASIC interpreter job together. Before you do this, however, copy the file SBCIOL.LIB from the iSBC 957A release diskette to your configuration diskette. Also copy the interface library, RPIFL.LIB, from the Nucleus release diskette to your configuration diskette. Place the diskettes in the proper drives of the development system (system diskette in drive F0, configuration diskette in drive F1, demonstration system release diskette in drive F2, and scratch and listing diskette in drive F3), and enter the following command:

```
LINK86  :F2:TBASIC.OBJ,                                     &
        :F2:PTHIO.OBJ,                                      &
        :F2:PTOKEN.OBJ,                                     &
        :F2:PERR.OBJ,                                       &
        :F2:PINT.LIB,                                       &
        :F1:SBCIOL.LIB,                                     &
        :F1:RPIFL.LIB                                       &
 TO  :F3:TBASIC.LNK MAP PRINT(:F3:TBASIC.MP1)
```

The files linked in this process contain the following information:

TBASIC.OBJ	This file contains the initialization task and the interpreter.
PERR.OBJ	This file contains error printing routines.
PTOKEN.OBJ	This file contains a token scanner for PL/M-86 routines.
PTHIO.OBJ	This file contains the interface between the Terminal Handler and the interpreter.
PINT.LIB	This file contains a library of PL/M-86 routines that interface with the iRMX 86 Operating System.
SBCIOL.LIB	This file contains the iSBC 957A library.
RPIFL.LIB	This file contains the interface library for application jobs.

LINK86 places the linked application job in file TBASIC.LNK on drive F3.

LOCATE THE APPLICATION JOB

After linking the application job, use LOC86 to assign absolute locations. By examining the memory map, you can see that the next available location is 0E70:0. Use LOC86 to locate the application job there. To do this, enter the following command:

```
LOC86    :F3:TBASIC.LNK TO :F1:TBASIC                  &
         ORDER (CLASSES (CODE, DATA, STACK, MEMORY))   &
         ADDRESSES (CLASSES (CODE (0E700H)))           &
         MAP PRINT (:F3:TBASIC.MP2)                    &
         OBJECTCONTROLS(NOLINES,NOCOMMENTS,            &
                        NOPUBLICS,NOSYMBOLS)
```

LOC86 places the located application job in file TBASIC on drive F1. It places the locate map in file TBASIC.MP2 on drive F3. Figure A-6 shows the important portions of the application job locate map.

```
ISIS-II MCS-86 LOCATER, V1.3 INVOKED BY:
LOC86                                                    &
     :f3:tbasic.lnk  TO :f1:tbasic                       &
     order (classes (code, data, stack, memory ))        &
     addresses (classes (code (00E700H)))                    &
     map print (:f3:tbasic.mp2)                          &
     OBJECTCONTROLS(NOLINES,NOCOMMENTS,                  &
                      NOSYMBOLS,NOPUBLICS)               &
;

SYMBOL TABLE OF MODULE TBASIC
READ FROM FILE :F3:TBASIC.LNK
WRITTEN TO FILE :F1:TBASIC

BASE   OFFSET TYPE SYMBOL                BASE   OFFSET TYPE SYMBOL

0E70H  00CEH  PUB  WSTART                0E70H  0090H  PUB  CSTART  <-(D2)
1089H  008BH  PUB  TXTBGN                1089H  108BH  PUB  TXTEND
1089H  0004H  PUB  VARBGN                1084H  0051H  PUB  TXTUNF
0F20H  01F6H  PUB  GETCHAR               0F20H  0199H  PUB  FLUSHINPUT
0F20H  0148H  PUB  PUTCHAR               0F20H  00F4H  PUB  FLUSHOUTPUT
0F20H  0024H  PUB  INITIHIO              0F44H  003DH  PUB  TOKENIZE
0F44H  001CH  PUB  HEX                   0F44H  0004H  PUB  HEXCHARS
0F86H  005EH  PUB  PERROR                0F8FH  0014H  PUB  PCRTSEMA

                        •
                        •
                        •
```

Figure A-6. Application Job Locate Map

```
MEMORY MAP OF MODULE TBASIC
READ FROM FILE :F3:TBASIC.LNK
WRITTEN TO FILE :F1:TBASIC

MODULE START ADDRESS  PARAGRAPH = 0E70H  OFFSET = 0090H
SEGMENT MAP

START     STOP      LENGTH ALIGN NAME              CLASS

0E700H    0F208H    0B09H   G    CODE              CODE
0F20AH    0F443H    023AH   W    TERMINALHANDLE    CODE
                               -RIO_CODE
0F444H    0F86BH    0428H   W    TOKENIZE_CODE     CODE
0F86CH    0F8FCH    0091H   W    PERROR_CODE       CODE
                      •
                      •
                      •
11C44H    11C44H    0000H   W    INITTASKSIGNAL    DATA
                               -_DATA
11C50H    12653H    0A04H   G    STACK             STACK  <--(D3)
12660H    12660H    0000H   G    ??SEG
12660H    1279EH    013FH   W    CONST             CONST
127A0H    127A0H    0000H   W    MEMORY            MEMORY <--(D1)


GROUP MAP

ADDRESS   GROUP OR SEGMENT NAME
10840H    DGROUP
          CONST
          DATA
          DATA2
          STACK
          MEMORY
0E700H    CGROUP
          CODE
```

Figure A-6. Application Job Locate Map (continued)

As with the Debugger locate map, there are three pieces of important information in Figure A-6 which you must record on the memory map. Arrows D1, D2, and D3 mark them.

Arrow D1 shows the next available memory location, 127A:0. Record this value on the memory map. It will be used when when calling the %SAB macro to reserve memory for the application system. This is shown in portion D of Figure A-2.

Arrow D2 shows the entry point of the first-level job's initialization task, CSTART. Record the address of CSTART, 0E70:090, on the left portion of the memory map, near the other information for the application job. This is shown in portion D of Figure A-2. You must later provide this information in the %JOB macro call for the application job.

Arrow D3 shows the stack segment starting address and length. Record the starting address, 11C5:0, and the length, 0A04, on the left portion of the memory map, near the other information for the application job. This is shown in portion D of Figure A-2. This job has a statically allocated stack and so you must provide this information in the %JOB macro call.

This application job assumes that the code segment and the data segment are the same. Therefore, it is not necessary to record any information about the data segment in the memory map.

BUILD THE CONFIGURATION FILE

After you have located the Nucleus, the Debugger, and the application job and filled out the memory map, you have enough information to build the configuration file needed by the root job. This involves creating a file containing an $INCLUDE statement, a %SYSTEM macro call, %SAB macro calls, %JOB macro calls and an %END macro call. The following sections show filled out worksheets for these macros and discuss the parameters. Then the actual configuration file itself is shown.

%JOB MACRO CALLS

For this system, you must make two %JOB calls; one for the Debugger and one for the application job. The order in which you include these calls in the configuration file is important because that is the order in which the jobs are initialized when the system starts running. Make the %JOB call for the Debugger first.

Debugger %JOB Call

Figure A-7 shows the completed worksheet for the Debugger's %JOB call.

Macro call: JOB (defines first-level jobs) - for Debugger

Number of calls required one for each first-level job

CONFIGURATION FILE NAME: CONFIG.A86

FORMAT:

	parameter	type	suggested default	value
%JOB	(directory_size,	word	(0)	0
	pool_min,	word		1FFH
	pool_max,	word	(0FFFFH)	1FFH
	max_objects,	word		0FFFFH
	max_tasks,	word		0FFFFH
	max_job_priority	byte		0
	exception_handler_entry,	addr	(0:0)	0:0
	exception_handler_mode,	byte	(1)	1
	job_flags,	word	(0)	1
	init_task_priority,	byte	(0)	0
	init_task_entry,	addr		6B2:0000
	data_segment_base,	base	(0)	0
	stack_pointer,	addr	(0:0)	0:0
	stack_size,	word	(512)	512
	task_flags)	word	(0)	0

NOTES:

1. Type addr is specified as base:offset

2. Types addr and base must be entered as hexadecimal numbers without the suffix H. Types word and byte default to decimal, but will accept all radix suffixes.

Figure A-7. Completed Debugger %JOB Macro Worksheet

The parameters are described in the following:

directory_size through max_tasks	The first five parameters affect the Debugger while it is running. Enter these parameters as shown in Figure A-7.
max_job_priority	The 0 indicates that the priority of the root job is the maximum priority for tasks in this job.
exception_handler_-entry	The 0:0 value means that the default system exception handler identified in the %SYSTEM call is used.
exception_handler_mode	The 1 indicates that the exception handler is called in the event of programming errors.
job_flags	The 0 indicates that the Nucleus does not validate parameters.
init_task_priority	This value was listed in Table 4-1.
init_task_entry	This value depends on the address specified when locating the Debugger. It is always (base of the location address):0.
data_segment_base	The 0 indicates that the Debugger assigns its own data segment.
stack_pointer	The 0:0 indicates that the Nucleus allocates the stack segment.
stack_size	This value was listed in Table 4-1.
task_flags	The 0 indicates that the Debugger does not use the 8087 NDP component.

Application Job %JOB Call

Figure A-8 shows the completed %JOB macro worksheet for the application job.

Macro call: JOB (defines first-level jobs) - for TBASIC

Number of calls required: one for each first-level job

CONFIGURATION FILE NAME: CONFIG.A86

FORMAT:

	parameter	type	suggested default	value
%JOB	(directory_size,	word	(0)	20
	pool_min,	word		1FFH
	pool_max,	word	(0FFFFH)	0FFFFH
	max_objects,	word		0FFFFH
	max_tasks,	word		0FFFFH
	max_job_priority,	byte		0
	exception_handler_entry,	addr	(0:0)	0:0
	exception_handler_mode,	byte	(1)	1
	job_flags,	word	(0)	1
	init_task_priority,	byte	(0)	131
	init_task_entry,	addr		0E70:090
	data_segment_base,	base	(0)	0
	stack_pointer,	addr	(0:0)	11C5:0
	stack_size,	word	(512)	0A04H
	task_flags)	word	(0)	0

NOTES:

1. Type addr is specified as base:offset

2. Types addr and base must be specified as hexadecimal numbers without the suffix H. Types word and byte default to decimal but will accept all radix suffixes.

Figure A-8. Completed Application Job %JOB Macro Worksheet

The values shown in Figure A-8 are very similar to those shown in Figure A-7. The major differences are outlined as follows:

init_task_priority	A value of 131 is the recommended value for this job.
init_task_entry	This value was taken from the memory map.
data_segment_base	This 0 indicates that the task assigns the data segment.
stack_pointer	This job has a statically allocated stack. This value was taken from the memory map.
stack_size	This value was taken from the memory map.

%SAB MACRO CALLS

This system uses two %SAB calls, one for the memory needed by the Nucleus and the first-level jobs, and the other for the remainder of the address space over 128K. Figure A-9 contains the completed worksheet for the two %SAB calls.

Macro call: SAB (for system address blocks)

Number of calls required: one or more

CONFIGURATION FILE NAME: ______

FORMAT:

	parameter	type	suggested default	value
%SAB	(start_base,	base		0
	end_base,	base		127A
	type)	see note 1	U	U
%SAB	(start_base,	base		2000
	end_base,	base		FFFF
	type)	see note 1	U	U
%SAB	(start_base,	base		
	end_base,	base		
	type)	see note 1	U	

NOTES:

1. The type parameter is reserved for future use. Enter the character U for this parameter.
2. A SAB is declared between start_base:0 and end_base:F, inclusive.
3. Types addr and base must be entered as hexadecimal numbers without the suffix H. Types word and byte default to decimal but will accept all radix suffixes.

Figure A-9. Completed %SAB Macro Worksheet

The first %SAB call shown in Figure A-9 reserves the memory needed for the entire application system. The end_base parameter is taken from the memory map. It includes the estimate for the root job.

The second %SAB call shown in Figure A-9 reserves memory that is not actually in the system. This system has only 128K bytes of memory. Thus addresses 2000:0 to 0FFFF:F are not used. Reserving these locations speeds the system initialization process.

%SYSTEM MACRO CALL

Figure A-10 shows the completed worksheet for the %SYSTEM call. You must place this call last in the configuration file.

Macro call: SYSTEM (system parameters)

Number of calls required: exactly one

CONFIGURATION FILE NAME: CONFIG.A86

FORMAT:

	parameter	type	suggested default	value
%SYSTEM	(nucleus_entry,	base		1C0
	rod_size,	word	(0)	20
	min_trans_size,	word	(64)	64
	debugger,	see note 1	(A)	A
	default_e_h_provided,	see note 2	(N)	D
	mode)	word		1

NOTES:

1. Valid entries for the debugger parameter include:

   A Debugger available
   N No Debugger available

2. Valid entries for the default_e_h_provided parameter include:

   Y Yes
   D Debugger
   N No

3. Types addr and base must be entered as hexadecimal numbers without the suffix H. Types word and byte default to decimal, but will accept all radix suffixes.

Figure A-10. Completed %SYSTEM Macro Worksheet

The parameters are described in the following:

nucleus_entry	This value is taken from the memory map.
rod_size and min_trans_size	These values affect the system at run time. Use the values listed in Figure A-10.
debugger	The A indicates that the Debugger is available.
default_e_h_provided	The D indicates that the Debugger is used for the default exception handler.
mode	The 1 indicates that the Operating System transfers control to the exception handler (the Debugger) in the event of a programming error condition.

CREATE THE ACTUAL CONFIGURATION FILE

After you have filled out the macro worksheets, you can create the configuration file. To do this, create a file called CONFIG.A86 on your configuration diskette, and copy the information from the worksheets into it as well as an $INCLUDE statement for file CTABLE.MAC and an %END call. The statements in this file appear as follows:

```
$INCLUDE (:F2:CTABLE.MAC)
%SAB (0, 127A, U)
%SAB (2000, FFFF, U)
%JOB(0,1FFH,1FFH,0FFFH,0FFFFH,0,0:0,1,1,0,6B2:000D,0,0:0,512,0)
%JOB(20,1FFH,0FFFFH,0FFFFH,0FFFFH,0,0:0,1,1,131,0E70:090,0,11C5:0,
     0A04H,0)
%SYSTEM (1C0, 20, 64, A, D, 1)
%END
```

GENERATE THE ROOT JOB

You can now use the CROOT.CSD SUBMIT file to assemble the configuration file, link the root job, and locate the root job. Place the diskettes in the proper drives of your development system (system diskette in drive F0, coniguration diskette in drive F1, Nucleus release diskette in drive F2, scratch and listing diskette in drive F3), and enter the following command:

```
SUBMIT :F2:CROOT(CONFIG, date, 0E100H)
```

Where date can be in any form, as long as it does not exceed nine characters.

This command assembles the configuration file, links it to the root job, and locates the root job at the correct address. LOC86 places the located root job in file CONFIG on drive F1. It also places the locate map for the root job in file CONFIG.MP2 on drive F3. You can use the locate map to determine the actual size of the root job. When you configure your ROM/RAM system, you can update the memory map to reflect this value.

LOAD THE SYSTEM

At this point you can use either the ICE-86 in-circuit emulator or the iSBC 957A package to load the system into memory. When you do, load the following files, in order, from your configuration diskette:

NUCLUS
DEBUGR
TBASIC
CONFIG

APPENDIX B. BURNING THE NUCLEUS INTO 2732 PROM

If you use the Universal PROM Mapper (UPM) version 3.0 to burn code into PROM, you cannot load the entire Nucleus with a single UPM READ command. In order to burn the Nucleus (and possibly some of your other programs), you must burn 16K byte pieces of the code into PROM. This appendix describes the procedures required to do this. It also lists the required hardware and software. Although this appendix refers specifically to the Nucleus, you can use the procedures described here to burn any large module into PROM.

REQUIREMENTS

In order to use the procedure outlined in this appendix, you must have the following hardware and software.

- A linked Nucleus (NUCLUS.LNK)
- LOC86 software
- A UPP universal PROM Programmer with a 2732 personality module
- 8 erased 2732A PROM modules

With this hardware and software you can use the procedures in the following sections to place the Nucleus code into PROM.

LOCATE THE NUCLEUS

Use LOC86 to locate the Nucleus for a ROM/RAM configuration. Use a command similar to the following:

```
LOC86   NUCLUS.LNK TO ROMNUC                              &
        ORDER (CLASSES (DATA, STACK, MEMORY))             &
        SEGSIZE (STACK (0))                               &
        ADDRESSES (CLASSES (CODE (rom_address),           &
                            (DATA (ram_address)))         &
        MAP PRINT (map_file)                              &
        OBJECTCONTROLS(NOLINES,NOCOMMENTS,
                       NOPUBLICS,NOSYMBOLS)
```

Chapter 5 describes the parameters of this command in detail. However, for the discussion in this appendix, the important parameter is rom_address. This parameter specifies the address in ROM where the Nucleus will reside.

Also examine the locate map to determine the exact size of the code class. You need this information to determine the number of pieces to burn.

BURN THE CODE INTO PROM

To burn the code into PROM, insert the 2732 personality module into the appropriate program socket (this appendix assumes socket 2). Then enter the commands shown in Figure B-1. These commands are structured so that you can place them in a SUBMIT file. The CNTL/E (control/E) characters in the figure return control to you so that you can insert a PROM into the UPP. After doing this, enter another CNTL/E to return control to the SUBMIT file. Make sure to place a new PROM into the UPP before each PROGRAM statement.

```
UPM
2732
SOCKET=2

    READ OBJECT FILE :F2:ROMNUC FROM 0 TO 3FFFH START 0E8000H
    STRIP LOW FROM 0 TO 3FFFH INTO 4000H
    STRIP HI FROM 0 TO 3FFFH INTO 6000H
CNTL/E    PROGRAM FROM 4000H TO 4FFFH START 0
CNTL/E    PROGRAM FROM 6000H TO 6FFFH START 0
CNTL/E    PROGRAM FROM 5000H TO 5FFFH START 0
CNTL/E    PROGRAM FROM 7000H TO 7FFFH START 0

    READ OBJECT FILE :F2:ROMNUC FROM 0 TO 2AF9H START 0EC000H
    STRIP LOW FROM 0 TO 2AF9H INTO 4000H
    STRIP HI FROM 0 TO 2AF9H INTO 6000H
CNTL/E    PROGRAM FROM 4000H TO 4FFFH START 0
CNTL/E    PROGRAM FROM 6000H TO 6FFFH START 0
CNTL/E    PROGRAM FROM 5000H TO 557DH START 0
CNTL/E    PROGRAM FROM 7000H TO 757DH START 0

EXIT
```

Figure B-1. UPM SUBMIT File to Burn the Nucleus into PROM

BURNING THE NUCLEUS INTO 2732 PROM

The commands in Figure B-1 assume that the Nucleus code class ranges from 0E8000H to 0EEAF:9. You must modify the SUBMIT file to specify the correct addresses for your system. To give you a better understanding of how to do this, the individual UPM commands used to burn the first portion of the Nucleus are listed and discussed in detail.

READ OBJECT FILE :F2:ROMNUC FROM 0 TO 3FFFH START 0E8000H

This command reads the first piece of the object file from disk into a 16K INTELLEC memory buffer. The logical addresses of the memory buffer are 0 through 3FFFH. The absolute address of the module is specified as 0E8000H.

STRIP LOW FROM 0 TO 3FFFH INTO 4000H

This command separates the even address (low order) bytes from the file and copies them into another memory buffer.

STRIP HI FROM 0 TO 3FFFH INTO 6000H

This command separates the odd address (high order) bytes from the file and copies them into another memory buffer.

CNTL/E PROGRAM FROM 4000H TO 4FFFH START 0

This command burns the first half of the low order bytes into PROM. Make sure that you insert a 2732 PROM into the UPP before entering this command. Include the CNTL/E character only if you use a SUBMIT file. This character returns control to you so that you can insert the PROM. After you do, enter another CNTL/E and processing resumes.

CNTL/E PROGRAM FROM 6000H TO 6FFFH START 0

This command burns the first half of the high order bytes into PROM.

CNTL/E PROGRAM FROM 5000H TO 5FFFH START 0

This command burns the second half of the low order bytes into PROM.

CNTL/E PROGRAM FROM 7000H TO 7FFFH START 0

This command burns the second half of the high order bytes into PROM.

The remainder of the commands in Figure B-1 function similarly. For further information about UPM, refer to the UNIVERSAL PROM PROGRAMMER USER'S MANUAL.

After you have burned all the PROMs, plug them into the memory board and test the system.

APPENDIX C. SYSTEM CALL USAGE

This appendix lists the system calls used by fully-configured versions of the optional subsystems. This information is important when you decide which system calls to include in your final application system. Table C-1 lists the system calls used by the Terminal Handler, Table C-2 lists those used by the Debugger, Table C-3 lists those used by the I/O System, Table C-4 lists those used by the Extended I/O System, Table C-5 lists those used by the Application Loader, and Table C-6 lists those used by the Human Interface.

Table C-1. System Calls Used by the Terminal Handler

NUCLEUS SYSTEM CALLS	
CATALOG$OBJECT	GET$SIZE
CREATE$MAILBOX	GET$TASK$TOKENS
CREATE$SEGMENT	GET$TYPE
CREATE$TASK	RECEIVE$MESSAGE
DELETE$SEGMENT	SEND$MESSAGE
DISABLE	SET$INTERRUPT
ENABLE	SIGNAL$INTERRUPT
END$INIT$TASK	WAIT$INTERRUPT
EXIT$INTERRUPT	

Table C-2. System Calls Used by the Debugger

NUCLEUS SYSTEM CALLS	
CATALOG$OBJECT	GET$SIZE
CREATE$JOB	GET$TASK$TOKENS
CREATE$MAILBOX	LOOKUP$OBJECT
CREATE$SEGMENT	RECEIVE$MESSAGE
CREATE$TASK	RESUME$TASK
DELETE$SEGMENT	SEND$MESSAGE
DISABLE	SET$INTERRUPT
DISABLE$DELETION	SET$PRIORITY
ENABLE	SIGNAL$INTERRUPT
ENABLE$DELETION	SLEEP
END$INIT$TASK	SUSPEND$TASK
EXIT$INTERRUPT	WAIT$INTERRUPT
GET$PRIORITY	

Table C-3. System Calls Used by the I/O System

NUCLEUS SYSTEM CALLS		
CATALOG$OBJECT	DISABLE$DELETION	SEND$CONTROL
CREATE$COMPOSITE	ENABLE$DELETION	SEND$MESSAGE
CREATE$EXTENSION	END$INIT$TASK	SET$INTERRUPT
CREATE$MAILBOX	FORCE$DELETE	SETOSEXTENSION
CREATE$REGION	GET$LEVEL	SIGNAL$EXCEPTION
CREATE$SEGMENT	GET$TASK$TOKENS	SIGNAL$INTERRUPT
CREATE$TASK	GET$TYPE	SLEEP
DELETE$COMPOSITE	LOOKUP$OBJECT	UNCATALOG$OBJECT
DELETE$MAILBOX	RECEIVE$CONTROL	WAIT$INTERRUPT
DELETE$REGION	RECEIVE$MESSAGE	
DELETE$SEGMENT	RESET$INTERRUPT	
DELETE$TASK	RESUME$TASK	

Table C-4. System Calls Used By the Extended I/O System

NUCLEUS SYSTEM CALLS		BASIC I/O SYSTEM CALLS
CATALOG$OBJECT	DELETE$MAILBOX	A$ATTACH$FILE
CREATE$COMPOSITE	DELETE$SEGMENT	A$CHANGE$ACCESS
CRETE$EXTENSION	GET$TASK$TOKENS	A$CLOSE
CREATE$JOB	GET$TYPE	A$CREATE$DIRECTORY
CREATE$MAILBOX	LOOKUP$OBJECT	A$CREATE$FILE
CREATE$REGION	RECEIVE$CONTROL	A$DELETE$CONNECTION
CREATE$SEGMENT	RECEIVE$MESSAGE	A$DELETE$FILE
CREATE$TASK	SEND$CONTROL	AGETCONNECTION$STATUS
DELETE$COMPOSITE	SEND$MESSAGE	AGETFILE$STATUS
DELETE$JOB	SETOSEXTENSION	A$OPEN
	UNCATALOG$OBJECT	A$PHYSICAL$ATTACH$DEVICE
		A$PHYSICAL$DETACH$DEVICE
		A$READ
		A$RENAME$FILE
		A$SEEK
		A$SPECIAL
		A$TRUNCATE
		A$WRITE
		CREATE$USER

Table C-5. System Calls Used by the Application Loader

NUCLEUS SYSTEM CALLS	I/O SYSTEM SYSTEM CALLS	EXTENDED I/O SYSTEM CALLS (if load job features are included)
CATALOG$OBJECT	A$ATTACH$FILE	CREATEIOJOB
CREATE$MAILBOX	A$DELETE$CONNECTION	S$ATTACH$FILE
CREATE$SEGMENT	A$CLOSE	S$DELETE$CONNECTION
CREATE$TASK	A$OPEN	
DELETE$JOB	A$READ	
DELETE$MAILBOX	A$SEEK	
DELETE$SEGMENT		
DELETE$TASK		
END$INIT$TASK		
GET$POOL$ATTRIB		
LOOKUP$OBJECT		
RECEIVE$MESSAGE		
SEND$MESSAGE		
SET$EXCEPTION$HANDLER		
SETOSEXTENSION		

Table C-6. System Calls Used by the Human Interface

NUCLEUS SYSTEM CALLS	
CATALOG$OBJECT	GET$TYPE
CREATE$MAILBOX	LOOKUP$OBJECT
CREATE$REGION	RECEIVE$CONTROL
CREATE$SEGMENT	RECEIVE$MESSAGE
CREATE$SEMAPHORE	RECEIVE$UNITS
CREATE$TASK	SEND$CONTROL
DELETE$JOB	SEND$MESSAGE
DELETE$MAILBOX	SEND$UNITS
DELETE$SEGMENT	SET$EXCEPTION$HANDLER
END$INIT$TASK	SETOSEXTENSION
GET$SIZE	
GET$TASK$TOKENS	

BASIC I/O SYSTEM CALLS	
A$ATTACH$FILE	A$WRITE
A$DELETE$CONNECTION	GET$TIME
AGETCONNECTION$STATUS	SET$TIME
A$OPEN	
A$PHYSICAL$ATTACH$DEVICE	
A$READ	

EXTENDED I/O SYSTEM CALLS	
EXITIOJOB	SGETFILE$STATUS
S$ATTACH$FILE	S$OPEN
S$CHANGE$ACCESS	S$READ$MOVE
S$CREATE$FILE	S$RENAME$FILE
S$DELETE$CONNECTION	S$SEEK
S$DELETE$FILE	S$SPECIAL
SGETCONNECTION$STATUS	S$TRUNCATE
SGETEXTENSION$DATA	S$WRITE$MOVE

APPLICATION LOADER SYSTEM CALLS
A$LOAD
A$LOAD$IO$JOB

INDEX

Underscored entries are primary references.

204 driver 9-31, 11-5
206 driver 9-33, 11-6
215 driver 9-34, 11-7
218 driver 9-34
220 driver 9-34, 11-7
254 driver 9-36, 11-8
957A package 4-40, B-2
2732 PROM B-1
8087 4-27, 6-9
8253 6-8
8259A 6-7

absolute code 13-10
application job link procedures 4-8, 4-11
Application Loader 10-1
    entry point 10-3
    system call usage C-3
assembling the root job 4-38
%ATTACH_DEVICE_TASK_PRIO macro 9-6
%AUTO macro 11-3

Basic I/O System 4-34, 9-1
    entry point 4-37, 9-39
    features 9-18
    initialization 9-39
    interfaces 9-3, 9-5
    system call usage C-2
baud rate 7-3
BCICO.P86 11-10
Bootstrap Loader 11-1
    driver configuration 11-4
BS1.A86 11-1
building the configuration file 4-22
burning the Nucleus into PROM B-1
B204.A86 11-5
B206.A86 11-6
B215.A86 11-7
B254.A86 11-7

common driver 9-25
    tables 9-27
COMMON_DEV_INFO structure 9-28
compact model 4-14, 4-27
component configuration 6-6, 7-1
condition code files 3-2

configuration
    Application Loader 10-1
    Basic I/O System 9-1
    Bootstrap Loader 11-1
    Debugger 8-1
    environment 1-3
    Extended I/O System 12-1
    file 4-22, 4-37, A-15, A-23
    Human Interface 13-1
    interface 9-3
    Nucleus 6-1
    overview 2-1
    ROM/RAM-based system 5-1
    Terminal Handler 7-1
    types 1-5
%CONSOLE macro 11-2
control-C semantics 7-8, 8-2
creating the configuration file 4-37, A-23
creation of tasks 1-3
CROOT.CSD 4-38, A-23

data segment allocation 4-27
date/time interface 9-4
DB.CSD 8-2
Debugger 4-32, 4-34, 8-1, 13-5
    entry point 8-3
    system call usage C-2
default exception handler 4-32
DEFINE_DUIB structure 9-21
demonstration system A-1
describing I/O devices 9-20
%DEV_INFO_BLOCK macro 12-4
device granularity 9-23
%DEVICE macro 11-3
device number 9-23
device numbering 9-20
%DEVICE_TABLES macro 9-38
device-information table 9-24, 9-27
device-unit information blocks 9-21
device-unit number 9-20, 9-23
devices 9-20
diskette preparation 4-9
DTHCNF.A86 8-1
DUIB 9-20, 9-21
%DUMMY_TIMER macro 9-19

edge triggering 6-8
EEXCEP.LIT 3-3
EIOS.EXT 3-3
%END macro 11-4
%END_DEV_CONFIG macro 12-5
%END_IO_JOB_CONFIG macro 12-10

entry point
    Application Loader 4-37, 10-3
    Basic I/O System 4-37, 9-39
    Debugger 4-37, 8-3
    Extended I/O System 4-37, 12-11
    Human Interface 13-4
    Terminal Handler 4-37, 7-9
EPIFC.LIB 4-14
EPIFL.LIB 4-14
EDEVCF.A86 12-1
EDEVCF.MAC 12-5
EJOBCF.A86 12-1
ETABLE.A86 12-1
ETABLE.MAC 12-3
example system configuration A-1
exception handler 4-25, 4-32, 4-39, 6-4
Extended I/O System 12-1, 13-6
    entry point 4-37, 12-11
    initialization 12-11
    I/O jobs 12-6
    logical devices 12-4
    system calls 12-3
external declaration 3-2

file driver 9-5
    global data 9-5
    tables 9-7
file/connection interface 9-5
FILE_DRIVER_INFO structure 9-7
first-level jobs 1-3
floating-point instructions 4-27, 6-9

general device information 9-38
general system layout 4-1, 5-2
generating the root job 4-38, A-23
global data 9-5
granularity 9-23

HCONFG.P86 13-1
HI.CSD 13-4
HI.EXT 3-3
HI.LIB 7-8, 8-2, 13-5
high location of modules 4-2
HPIFC.LIB 4-14
HPIFL.LIB 4-14
Human Interface 13-1
    commands 13-7
    entry point 4-37, 13-4
    requirements 13-5
    volumes 13-7

ICE-86 in-circuit emulator 4-40
IDEVCF.A86 9-1
IDEVCF.INC 9-3
in-circuit emulator 4-40

include files 3-2, 9-3
initial system 1-3, 3-3
initialization 3-3, 9-39
Intel-supplied device drivers 9-30
interface libraries 4-14
interfaces 9-3, 9-5
I/O devices 9-20
%IO_JOB macro 12-8, 13-6
%IO_USER macro 12-7
IOS.CSD 9-38
IOS.EXT 3-3
IOS_FILE_DRIVER structure 9-7
IPIFC.LIB 4-14
IPIFL.LIB 4-14
iSBC 204 driver 9-31, 11-5
iSBC 206 driver 9-33, 11-6
iSBC 215 driver 9-34, 11-7
iSBC 218 driver 9-34,
iSBC 220 driver 9-34, 11-7
iSBC 254 driver 9-36, 11-8
iSBC 957A package 4-40
ITABLE.A86 9-1
ITABLE.INC 9-3

%JOB macro 4-23, 4-34, 5-1, A-15
job preparation 3-1
jobs 1-3

language requirements 3-2
large model 3-2, 4-27
layout of the system 4-1, 5-2
LCONFG.P86 10-1
level triggering 6-8
LEXCEP.LIT 3-3
linking
    application jobs 4-13, A-12
    Application Loader 10-2
    Basic I/O System 9-38
    Bootstrap Loader 11-9
    Debugger 8-2, A-10
    Extended I/O System 12-10
    Human Interface 13-4
    Nucleus 6-11, A-6
    root job 4-38, A-23
    subsystems 4-8
    Terminal Handler 7-7
LINK86 4-13
LOADER.CSD 10-2
LOADER.EXT 3-3
loading the system 4-39, A-24

locating
    application jobs 4-14, 5-4, A-13
    Application Loader 10-2
    Basic I/O System 9-38
    Bootstrap Loader 11-9
    Debugger 8-2, A-10
    Extended I/O System 12-10
    Human Interface 13-4
    Nucleus 6-11, A-6
    RAM-based systems 4-1
    ROM/RAM-based system 5-2, 5-4
    root job 4-38, A-23
    subsystems 4-8, 5-2, 5-4
    Terminal Handler 7-8
LOC86 4-14, 5-4, B-2
logical devices 12-4
low location of modules 4-2
LPIFC.LIB 4-14
LPIFL.LIB 4-14

mailboxes 7-6
%MASTER_PIC macro 6-7
MCONFG.A86 7-1
medium model 3-2, 4-14, 4-27
memory address space 4-28, 5-1,
memory map 4-3, 5-3, A-2
minimizing memory address space 5-1
multiple Terminal Handlers 7-9
%MTH macro 7-3
MTH.CSD 7-7

named file driver 9-17
    tables 9-13
named files 9-6
NDEVCF.A86 6-1, 6-6
NDP 4-27, 6-9
%NDP_SUPPORT macro 6-9
NEXCEP.LIT 3-3
%NO_ALLOCATE macro 9-19
%NO_CREATE_FALSE macro 9-19
%NO_TRUNCATE macro 9-19
non-file connection interfaces 9-3
NTABLE.A86 6-1
Nucleus 6-1
    component configuration 6-6
    default configuration 6-10
    INCLUDE files 3-2
    initialization errors 6-11
    internal features 6-3
    link and locate procedures 6-11, A-6
    maximal and minimal configuration 6-10
    root task errors 6-12
    system calls 6-4
NUCLUS.CSD 6-11
NUCLUS.EXT 3-3

%NUM_FILE_DRIVERS macro 9-6
numbering devices 9-20

on board USART 9-37
optional subsystems 1-1, 3-5, 4-8, 4-33
order of modules 4-2, 4-38
overview of configuration 2-1

parameter
    interface 9-3
    validation 6-3
physical file driver 9-16
    tables 9-8
physical files 9-6
PIC 6-7
PIT 6-8, 7-5
PL/M-86 3-1, 4-27
power-fail interface 9-3
preparing
    application jobs 3-1
    diskettes 4-9
    jobs for system configuration 3-1
    memory maps 4-3, 5-3, A-2
    subsystems 3-5
procedural overview 2-1
programmable interrupt controller 6-7
programmable interval timer 6-8
PROM B-1

RADEV_DEV_INFO structure 9-29
RADEV_UNIT_INFO structure 9-30
random access driver 9-25
    tables 9-29
REQ_FILE_DRIVER structure 9-7
ROM control 3-2
ROM/RAM location process 5-4
ROM/RAM-based system 5-1
root job 1-3, 3-3, 4-16, 4-38, A-12, A-23
RPIFC.LIB 4-14
RPIFL.LIB 4-14
RQENDINIT$TASK 3-4
RQTHNORMIN 7-6, 13-5
RQTHNORMOUT 7-6, 13-5
RQSYSEX 4-32

%SAB macro 4-28, 5-1, A-19
sample system configuration A-1
Series II development system 13-9
Series III development system 4-10, 13-7
    configuration files 4-11
    SUBMIT files 4-11
size control considerations 3-3, 4-27
%SLAVE_PIC macro 6-8
stack allocation 4-15, 4-27

SUBMIT files 4-8
    Application Loader 10-2
    Basic I/O System 9-38
    Bootstrap Loader 11-9
    Debugger 8-1
    Extended I/O System 12-10
    Human Interface 13-4
    Nucleus 6-11
    root job 4-38
    Terminal Handler 7-7
synchronous initialization 3-4
system
    address block 4-28
    calls 9-15, C-1
    configuration file 4-22, 4-37, A-16, A-23
    layout 4-1, 5-2
    loading 4-40, A-24
    testing 4-40, 5-5
    type 4-2
%SYSTEM macro 4-30, 4-36, A-21

tables for file drivers 9-8
tasks 1-3
Terminal Handler 4-34, 7-1, 13-5
    component configuration 7-1
    entry point 7-9
    system call usage C-1
testing the system 4-40, 5-5
%TH_19200_BAUD_COUNT macro 7-2
%TH_CHAR_LENGH macro 7-5
%TH_INT_LEVELS macro 7-6
%TH_MAILBOX_NAMES macro 7-6
%TH_TIMER macro 7-5
%TH_USART macro 7-4
timer 6-8, 7-5
%TIMER macro 6-8
%TIMER_TASK_PRIO macro 9-7
type
    configuration 1-5
    system 4-2

unit number 9-20, 9-23
unit-information table 9-24, 9-30
Universal PROM Mapper B-1
UPM B-1
UPP B-1
USART 7-4, 9-37

worksheets
    %JOB macro 4-24
    memory map 4-5
    %SAB macro 4-29
    %SYSTEM macro 4-31

REQUEST FOR READER'S COMMENTS

Intel Corporation attempts to provide documents that meet the needs of all Intel product users. This form lets you participate directly in the documentation process.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this document.

1. Please specify by page any errors you found in this manual.

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 document on a scale of 1 to 10 with 10 being the best rating. ____________

NAME ______________________ DATE __________

TITLE ______________________

COMPANY NAME/DEPARTMENT ______________________

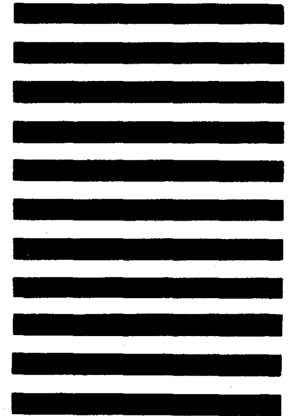
ADDRESS ______________________

CITY ______________ STATE ______ 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.



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 79 BEAVERTON, OR

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
5200 N.E. Elam Young Pkwy.
Hillsboro, Oregon 97123

O.M.S. Technical Publications

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

Printed in U.S.A.