intel®

RUN-TIME SUPPORT MANUAL FOR iAPX 86,88 APPLICATIONS

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

Order Number: 121776-002

intel®

RUN-TIME SUPPORT MANUAL FOR iAPX 86,88 APPLICATIONS

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

Order Number: 121776-002

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.

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

BXP	Intel	iSBC	Multibus
CREDIT	Int$_e$l	iSBX	Multichannel
i	Intelevision	iSXM	Multimodule
ICE	Intellec	Library Manager	Plug-A-Bubble
iCS	Intellink	MCS	PROMPT
i_m	iOSP	Megachassis	RMX/80
iMMX	iPDS	Micromainframe	System 2000
Insite	iRMX	Micromap	UPI

A 1169/684/5.5K JAY

REV.	REVISION HISTORY	DATE
-001	Original issue.	12/81
-002	Modified UDI procedures in Chapters 5, 6 and 7. Appendix A completely revised.	7/82

HOW TO USE THIS MANUAL

The software portion of a microcomputer application consists of more than the applications programs that you write. Intel provides a number of software tools that assist you not only when you are compiling and debugging application programs, but also when these programs are running. This manual focuses on the run-time aids that Intel offers for the iAPX 86,88 family of processors. The documentation of these run-time aids is useful regardless of which Intel languages you are using to implement your application and regardless of which operating systems you are using.

Figure Ø-1 provides a general model of how application software is supported at run-time by layers of software and hardware. This model (at varying levels of refinement) organizes the information contained in the manual.

Layer 1, applications programming, is the subject of Chapter 1 and Chapter 2. Chapter 1 provides an overview of the programming languages that Intel offers. Chapter 2 discusses the development process, with emphasis on the debugging aids available in each development environment. These chapters are primarily introductory, defining the context for later chapters that deal more directly with run-time support.

Layer 2, run-time libraries, is the subject of Chapter 3. This section identifies the run-time support available for each Intel language and explains how to utilize that support.

Chapter 4 explains layer 3 of the model, connecting application to system environment. This section explains how layer 3 contributes to portability of your application from one iAPX 86,88 environment to another. It also deals with some aspects of application programming that might interfere with such portability if not carefully considered in advance. Chapter 4 introduces concepts and terms that are developed in more detail in Chapters 5 through 9.

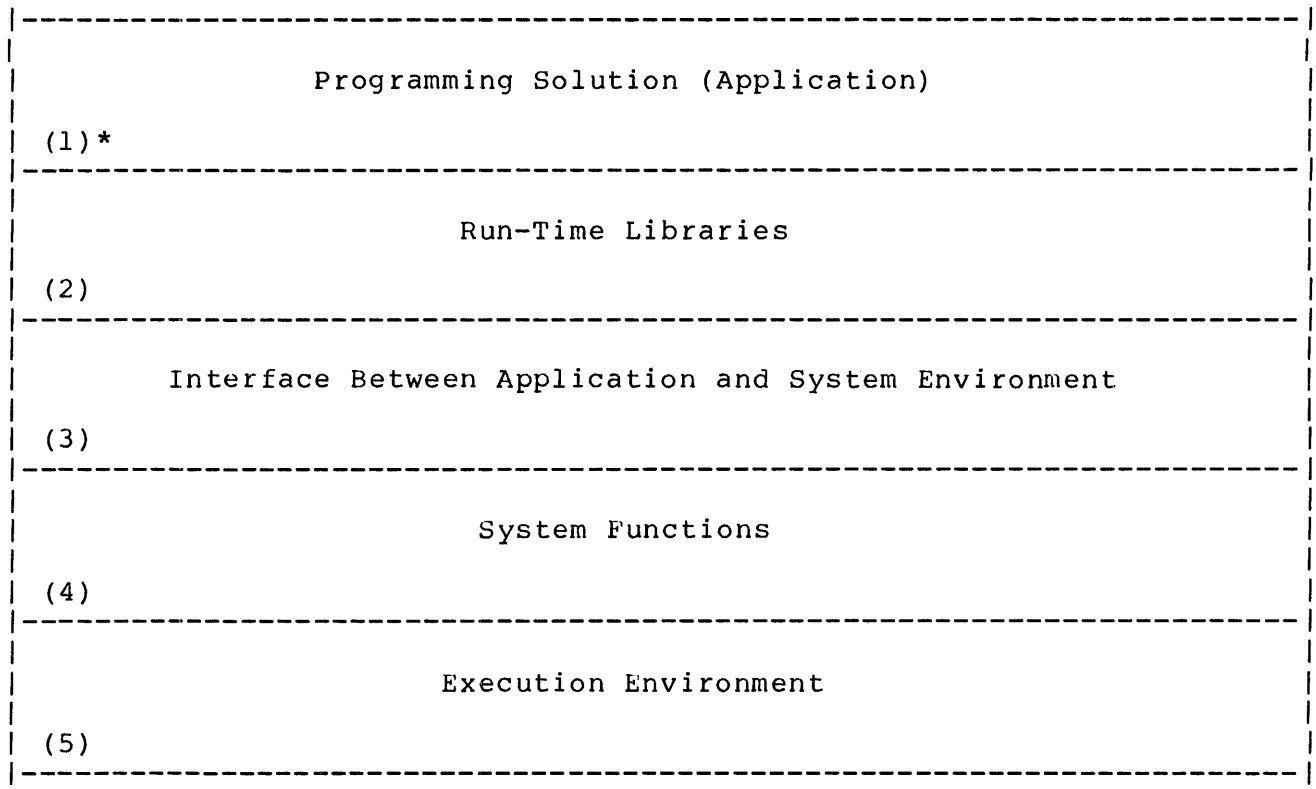
Chapters 5, 6, and 7 concern layer 4. They apply the concepts introduced in Chapter 4 to each of Intel's operating system environments, namely Series III, iRMX 86, and iRMX 88. You may skip chapters that deal with operating systems in which you are not interested.

Chapter 8 tells how to apply the concepts of Chapter 4 when dealing with an operating system other than one provided by Intel.

Chapter 9 gives techniques for configuring Intel's run-time software to better suit your application.

Appendix A presents specifications for the Universal Development Interface (UDI), Intel's standard for layer 3 of the model.

Appendix B explains how to transport an application from a development environment to a target environment that does not include an operating system.



* Level numbers are in parentheses.

Figure Ø-1. General Run-Time Model

AUDIENCE

This manual assumes that you are an applications programmer who:

- Is familiar with programming in an application language (Pascal or FORTRAN) that is being used to develop an application
- Plans to develop an application in a development environment, such as an Intellec Microcomputer Development System or iRMX 86-based OEM system
- Has heard about Intel operating systems, such as iRMX 86 and iRMX 88, but does not necessarily know what run-time support is available or how it is used

If you are planning to use PL/M-86 or ASM-86 to develop your application, this manual will still be of use to you, although much of Chapter 3 does not apply.

The manual may also be used by an operating system programmer to:

- Understand and use Intel-supplied libraries and standard interfaces
- Write an implementation of layer 3 (the Universal Development Interface) for an operating system other than one supplied by Intel
- Implement custom device drivers

RELATED DOCUMENTATION

Throughout this manual are frequent references to related information on specific subjects. These references do not, however, include order numbers and may use abbreviated titles. Use the following lists to look up order numbers and complete titles.

1. Intel languages
   - Pascal-86 User's Guide, 121539
   - FORTRAN-86 User's Guide, 121570
   - PL/M-86 User's Guide, 121636
   - 8086/8087/8088 Macro Assembly Language Reference Manual for 8086-Based Development Systems, 121627
   - An Introduction to ASM86, 121689
2. General debugging aids
   - ICE-86A Microsystem In-Circuit Emulator Operating Instructions for ISIS-II Users, 162554
   - ICE-88 In-Circuit Emulator Operating Instructions for ISIS-II Users, 9800949
3. Intellec Series III Microcomputer Development System
   - Intellec Series III Microcomputer Development System Product Overview, 121575
   - Intellec Series III Microcomputer Development System Console Operating Instructions, 121609
   - Intellec Series III Microcomputer Development System Programmer's Reference Manual, 121618
4. iRMX 86 Operating System
   - Introduction to the iRMX 86 Operating System, 9803124
   - User's Guide for the iSBC 957B iAPX 86,88 Interface and Execution Package, 143979
   - iRMX 86 System Debug Monitor Reference Manual, 143908

- 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 Configuration Guide, 9803126
- Guide to Using iRMX 86 Languages, 143907
- iRMX 86 System Programmer's Reference Manual, 142721
- iRMX 86 Programming Techniques, 142982

5. iRMX 88 Operating System

- Introduction to the iRMX 80/88 Real-Time Multitasking Executives, 143238
- iRMX 88 Installation Instructions, 143241
- iRMX 88 Reference Manual, 143232
- iRMX 80/88 Interactive Configuration Utilities, 142603
- Guide to Writing Device Drivers for the iRMX 86 and iRMX 88 I/O Systems, 142926

6. Utilities

- iAPX 86,88 Family Utilities User's Guide, 121616

7. Numeric Data Processing

- iAPX 86,88 User's Manual, 210201
- 8087 Support Library Reference Manual, 121725
- iSBC 337 Multimodule Numeric Data Processor Hardware Reference Manual, 142887

8. Device Drivers

- iSBC 86/12A Single Board Computer Hardware Reference Manual, 9803074
- Peripheral Design Handbook, 205610

RELATED SOFTWARE VERSION

The information in this manual that relates to specific features of the FORTRAN-86/88 and Pascal-86/88 compilers refers to version 2.Ø or later versions of both compilers.

CONTENTS

CHAPTER 1
INTRODUCTION TO INTEL'S PROGRAMMING LANGUAGES

Application Languages 1-1
FORTRAN-86/88 1-1
Pascal-86/88 1-1
System Implementation Languages 1-1
PL/M-86 1-2
ASM-86 1-2
Connecting Modules Written in Different Languages 1-2
The Linkage Mechanism 1-3
Interface Considerations 1-4
Sharing Data 1-4
Data Types 1-4
Shared Memory Area 1-6
Parameter Passing 1-6
Stack Usage 1-7
Conventions for Register Usage 1-7

CHAPTER 2
THE DEVELOPMENT ENVIRONMENT

Systems 2-1
Intellec Series III Microcomputer Development System 2-1
OEM Systems 2-2
Custom Systems 2-2
Debugging Tools 2-2
The iAPX 86,88 Monitor Program 2-2
The iRMX 86 System Debug Monitor 2-3
The iRMX 86 System Debugger 2-3
Debug-86 2-3
ICE-86 and ICE-88 In-Circuit Emulators 2-3
Translator Support for Symbolic Debugging 2-4

CHAPTER 3
RUN-TIME SUPPORT LIBRARIES

Application Language Run-Time Support 3-1
SIL Run-Time Support 3-2
Non-Mathematical Run-Time Libraries 3-3
8087 Support Library 3-5
Using the Run-Time Libraries 3-8
Version Numbers 3-11
Linking for Reentrancy 3-11
Linking Multi-Language Jobs 3-12
Using the 8087 Support Library with Multi-Language Jobs 3-14
Initialization for Subprograms 3-14
Run-Time Detection of Linkage Errors 3-14

CHAPTER 4
INTERFACE BETWEEN APPLICATION AND OPERATING SYSTEM

Universal Development Interface (UDI) 4-1

Error Reporting 4-2
Interrupt Handling 4-3

CHAPTER 5
SERIES III RUN-TIME SUPPORT

UDI for the Series III Operating System 5-2
Libraries 5-2
Implementation Considerations 5-2
Exception Codes 5-3
Interrupt Handling 5-4
8087 Support 5-4
Reentrancy and Multitasking 5-6
Example Program 5-6
Compiling 5-7
Linking for Series III Execution 5-8
Invoking 5-8
Linking for Execution on an iRMX 86 System 5-9

CHAPTER 6
iRMX 86 RUN-TIME SUPPORT

UDI for the iRMX 86 Operating System 6-1
Libraries 6-1
Implementation Considerations 6-2
Exception Codes 6-4
Interrupt Handling 6-6
Logical Names 6-6
Reentrancy 6-6
Multitasking 6-6
Using Overlays in an iRMX 86 Environment 6-6
Example Program 6-8
Compiling 6-8
Linking 6-9
Invoking 6-9

CHAPTER 7
iRMX 88 RUN-TIME SUPPORT

UDI for the iRMX 88 Operating System 7-1
Implementation Considerations 7-2
Exception Codes 7-3
Libraries, Compiling, Linking 7-4
Interrupt Handling 7-4

CHAPTER 8
RUN-TIME CONSIDERATIONS FOR NON-INTEL OPERATING SYSTEMS

UDI Procedures Used by Run-Time Libraries 8-1
Implementing a Universal Development Interface 8-2

CHAPTER 9
CONFIGURING THE RUN-TIME SYSTEM

Establishing an Alternate Exception Handler 9-1
Eliminating Preconnection Parsing 9-2

Changing Default Preconnections 9-2

APPENDIX A
UDI SPECIFICATIONS

Introduction A-1
Overview. A-1
Utility Procedures. A-1
Memory Management A-2
File Management A-2
Program Control A-4
Exception Handling. A-4
General Assumptions A-8
Multitasking. A-8
Coprocessor Support A-8
Format of Primitives. A-8
Utility and Command Parsing Service Procedures. A-9
DQGETTIME A-9
DQ$DECODE$TIME. A-10
DQGETSYSTEM$ID. A-11
DQGETARGUMENT A-12
DQ$SWITCH$BUFFER. A-15
Memory Management Procedures. A-17
DQ$ALLOCATE A-17
DQ$FREE A-18
DQGETSIZE A-19
DQ$RESERVE$IO$MEMORY A-20
File Connection Procedures. A-22
DQ$ATTACH A-22
DQ$CREATE A-23
DQ$DETACH A-24
DQ$DELETE A-25
DQGETCONNECTION$STATUS A-26
DQ$FILE$INFO. A-28
File Naming Procedures. A-30
DQ$RENAME A-30
DQ$CHANGE$EXTENSION A-31
DQ$CHANGE$ACCESS. A-32
File Usage Procedures A-33
DQ$OPEN A-33
DQ$SEEK A-35
DQ$READ A-37
DQ$SPECIAL. A-38
DQ$WRITE A-40
DQ$TRUNCATE A-41
DQ$CLOSE A-42
Program Control Procedures. A-43
DQ$EXIT A-43
DQ$OVERLAY A-44
Exception Handling Procedures A-45
DQ$TRAP$EXCEPTION A-45
DQGETEXCEPTION$HANDLER A-46
DQ$DECODE$EXCEPTION A-48
DQ$TRAP$CC. A-49
Minimal Primitives Needed For Application Runtime Support A-50
86/88 Family And Operating System Dependencies. A-51

APPENDIX B
WRITING YOUR OWN LOGICAL RECORD SYSTEM

Why Use an Alternate LRS? B-1
What is Involved in Writing an LRS? B-1
Logical Record Interface Specifications B-2
    Reentrancy B-3
    Exception Handling B-3
    Specification Format B-4
    Data Types B-5
    Data Structures B-5
        File Descriptors B-5
        File/Device Driver Tables B-6
    Connection Procedures B-8
        TQ$FILE$DESCRIPTOR B-8
        TQ$DEVICE B-10
    Control Procedures B-12
        TQ$INITIALIZE B-12
        TQGETPRECON B-15
        TQ$EXIT B-17
    Device Driver Procedures B-18
        File Markers B-18
        Buffering B-18
        Open B-19
        Close B-23
        Read B-25
        Write B-27
        Seek B-29
        Skip B-31
        End Record B-32
        Rewind B-34
        Backspace B-36
        End File B-37
    Exception Handler Procedures B-38
        TQSETERH B-38
        TQGETERH B-40
    Memory Management Procedures B-42
        TQ$ALLOCATE B-43
        TQ$FREE B-45
        TQGETSMALL$HEAP B-46
Example Device Drivers B-49

INDEX

ILLUSTRATIONS

FIGURE	TITLE	PAGE
0-1	General Run-Time Model	vi
3-1	Application Language Run-Time Support	3-2
3-2	SIL Run-Time Support	3-3
3-3	Detail of Non-Mathematical Run-Time Support	3-4
3-4	FORTRAN-86/88 Run-Time Libraries	3-9
3-5	Pascal-86/88 Run-Time Libraries	3-10
3-6	8087 Support Libraries	3-10
3-7	Linkage Structure of a Reentrant Pascal-86/88 Job	3-12
3-8	Linkage Structure of a Multi-Language Job	3-13
4-1	UDI Support	4-2
5-1	Intel's Series III Solution	5-1
5-2	Series III 8087 Interrupt Initialization	5-6
5-3	Series III Example Program	5-7
6-1	Intel's iRMX 86 Solution	6-2
6-2	iRMX 86 Example Program	6-8
7-1	Intel's iRMX 88 Solution	7-2
8-1	Interfacing to Non-Intel Operating System	8-1
B-1	Run-Time Support Without an Operating System	B-1
B-2	Table of Address for File/Device Drivers	B-6

TABLES

TABLE	TITLE	PAGE
1-1	Compatible Data Types	1-5
3-1	Contents of Non-Mathematical Run-Time Libraries	3-5
3-2	Contents of Numerics Run-Time Libraries	3-7
4-1	Language Interfaces for Interrupt Processing	4-4
5-1	Series III Exception Codes and Mnemonics	5-4
6-1	iRMX 86 Exception Codes and Mnemonics	6-5
7-1	iRMX 88 Exception Codes and Mnemonics	7-3
B-1	Required Exception Codes	B-4
B-2	Attribute Bit Items for Open	B-20
B-3	Disposition Parameters for Close	B-23
B-4	Mode Parameters for Seek	B-30

CHAPTER 1
INTRODUCTION TO INTEL'S PROGRAMMING LANGUAGES

This chapter introduces you to four Intel languages designed for use with the iAPX 86,88 Family of processors: FORTRAN-86/88, Pascal-86/88, PL/M-86, and ASM-86. Of these, FORTRAN-86/88 and Pascal-86/88 are applications languages, and PL/M-86 and ASM-86 are system implementation languages (SIL's).

APPLICATION LANGUAGES

An application language is designed to let the programmer concentrate on the problem to be solved rather than on the environment in which it is solved. An application language conforms to some industry-wide standard and thereby makes it possible to transport applications written in that language from one environment to another.

FORTRAN-86/88

FORTRAN is the oldest application language designed for expressing formulas. It supports many built-in functions for arithmetic and numeric calculations, including double-precision and extended-precision floating point; it is therefore ideal for scientific calculations. Huge libraries of engineering programs already exist in FORTRAN.

Intel's FORTRAN-86/88 implements a superset of the FORTRAN 77 subset defined by the American National Standards Institute (ANSI). FORTRAN-86/88 also includes additional features helpful in microcomputer applications. Refer to the FORTRAN-86 User's Guide for more detailed information.

Pascal-86/88

Pascal was originally designed by Niklaus Wirth as a language to teach programming. It was designed to be small, easy to understand, and therefore easy to use. It encourages and enforces a rigid structure to instill good programming practices and reduce programming errors. Pascal's structures make it relatively easy to prove that a program does what the programmer intends.

Intel's Pascal-86/88 implements a superset of standard Pascal as defined in the ISO Draft Proposal for a standard Pascal. Pascal-86/88 also includes additional features useful in microcomputer applications. Refer to the Pascal-86 User's Guide for details.

SYSTEM IMPLEMENTATION LANGUAGES

A system implementation language (SIL) is designed to help programmers fully exploit the capabilities of the hardware.

PL/M-86

PL/M-86 provides access to hardware functions while offering the benefits of a high-level language. Block structure, complex expressions, and parameterized procedure and function calls are some of the powerful features of the language. At the same time, machine level features such as port I/O, interrupt handling, and unrestricted pointer manipulation, make PL/M-86 a useful tool for the systems programmer. Refer to the PL/M-86 User's Guide for detailed information.

ASM-86

For systems or applications that require the utmost in efficiency or machine intimacy, Intel offers the 8086/8087/8088 Macro Assembly Language (ASM-86). Refer to An Introduction to ASM86 or the 8086/8087/8088 Macro Assembly Language Reference Manual for more information.

CONNECTING MODULES WRITTEN IN DIFFERENT LANGUAGES

The translators (i.e., compilers and assemblers) for Intel's application languages and SIL's produce object modules. Object modules can be combined to form larger programs by using the utility programs LIB86 and LINK86. (Refer to iAPX 86,88 Family Utilities User's Guide for complete information on using LINK86 and LIB86.) Not only can you combine modules produced by the same translator, but you can also combine modules produced by different translators. For example, you could combine a main program module written in Pascal-86/88 with a module containing a subroutine written in PL/M-86.

The ability to combine object modules gives you several major benefits:

- You can debug smaller, more manageable portions of your application before integrating them into the entire application system.
- You can implement and enforce the principles of stepwise refinement and information hiding to produce more understandable and maintainable programs.
- You can choose the language that is most suitable for each aspect of your application. For example, you might choose Pascal-86/88 as the primary language for your application because its control structures support structured programming principles. However, if your application depends on some involved mathematical calculations, you may wish to use FORTRAN-86/88's powerful mathematical functions to implement these calculations. They would be linked as subroutines to the main Pascal-86/88 program. Suppose that your application has a complex data structure involving many memory cross-references. You may choose to take advantage of PL/M-86's unrestricted pointers and implement a set of PL/M-86 subroutines to maintain this data structure.
- You can connect your application logic to the run-time support logic provided by Intel. Intel's run-time support consists of a number of libraries, each containing procedures that execute a class of run-time functions. You use the LINK86 utility to bind these procedures to your application programs. All Intel run-time libraries documented in this

manual obey PL/M-86 linkage conventions. Linking an Intel run-time procedure to your Pascal-86/88 or FORTRAN-86/88 program is just the same as linking a PL/M-86 procedure that you code yourself.

The following discussion concerning linking of modules written in different languages is primarily intended to help you to link run-time libraries to your FORTRAN-86/88 or Pascal-86/88 programs. However, it is written in a general way that enables you to link from any of Intel's application languages or SIL's to any other.

THE LINKAGE MECHANISM

LINK86 uses four kinds of information to link object modules together:

1. Module identification. A module is a collection of related data and procedures that is treated as a unit by LINK86. If any one data item or procedure is referenced by another module, the entire module is linked. A module is identified by one of the following means:

   - In FORTRAN-86/88, by the initial statement (PROGRAM, FUNCTION, or SUBROUTINE)
   - In Pascal-86/88, by the MODULE heading
   - In PL/M-86, by the name of the outer block
   - In ASM-86, by the NAME directive

2. Public definitions. You must define as "public" any data item or procedure in a module that is to be referenced by another module. Such a definition tells the compiler or assembler to generate address information about the item or procedure in a special format that LINK86 can recognize. Data items or procedures are identified as public by the following means:

   - In FORTRAN-86/88, by the SUBROUTINE and FUNCTION statements (all FORTRAN-86/88 subprograms are automatically public)
   - In Pascal-86/88, by a PUBLIC section for a module that is the same as the module in which the PUBLIC section appears
   - In PL/M-86, by the PUBLIC attribute
   - In ASM-86, by the PUBLIC directive

3. External declarations. When you reference a data item or procedure that is defined as a public object in some other module, you must tell the compiler or assembler that this is an "external" reference. This informs the compiler or assembler that the address of that object is to be supplied by LINK86 after compilation or assembly. Data items or procedures are declared external by the following means:

   - In FORTRAN-86/88, by referencing the procedure in an executable statement (using the EXTERNAL statement where necessary to distinguish an external reference from an internal variable

reference)

- In Pascal-86/88, by a PUBLIC section for a module other than the module in which the PUBLIC section appears
- In PL/M-86, by the EXTERNAL attribute
- In ASM-86, by the EXTRN directive

4. Libraries. A library is a file containing a collection of related modules. The utility LIB86 is used to create and maintain libraries consisting of one or more modules. By giving LINK86 the pathname of a library, you tell LINK86 where to look for the modules it may need to link together. Only those modules actually referenced are included in the output of the linker.

INTERFACE CONSIDERATIONS

If you wish to link modules written in different languages, then you must not only know how to use the above linkage techniques, but you must conside differences in the nature and implementation of the languages to be linked. The following sections identify the most significant interface considerations. For further details, refer to the FORTRAN-86 User's Guide and the Pascal-86 User's Guide.

Sharing Data

When trying to share data among modules written in different languages, you must be aware of the data types supported by each language and the mechanisms by which data can be shared.

DATA TYPES

Each language has its own model of data, which may be only partially compatible with that of other languages. Table 1-1 summarizes the correspondences among data types in iAPX 86,88 languages.

Table 1-1. Compatible Data Types

FORTRAN-86/88	Pascal-86/88	PL/M-86	ASM-86
(none)	CHAR; enumeration, unsigned subrange, or set stored in 8 bits	BYTE	DB
CHARACTER*1	{CHAR, INTEGER}	{BYTE, INTEGER}	{DB, DW}
CHARACTER*n (n>1)	(none)	(none)	(none)
LOGICAL*1	BOOLEAN (*1)	BYTE (*1)	DB (*1)
LOGICAL*2	(none)	WORD (*1)	DW (*1)
LOGICAL*4	(none)	DWORD (*1)	DD (*1)
INTEGER*1	(none)	BYTE (*2)	DB (signed)
INTEGER*2	INTEGER or subrange stored in 16 bits	INTEGER	DW (signed)
INTEGER*4	LONGINT	DWORD (*3)	DD (signed)
(none)	Enumeration or set stored in 16 bits or WORD or subrange in 0..64K-1	WORD	DW
(none)	Pointer to any type	POINTER	DD
(none)	Pointer (SMALL)	POINTER (SMALL)	DW
REAL	REAL	REAL	DD (8087 single precision)
REAL*8 or	LONGREAL	(none)	DQ (8087 double precision)
TEMPREAL	TEMPREAL	(none)	DT (8087 extended precision)
(n)	ARRAY [m..n] of base type	(n)	(none)
(none)	RECORD	STRUCTURE	STRUC

NOTES: (*1) Only rightmost bit significant; remaining bits are undefined except for Pascal-86/88, which requires them to be zero.
(*2) For values 0 through 127 only.
(*3) PL/M-86 DWORD is an unsigned 32-bit number.

SHARED MEMORY AREA

Except for FORTRAN-86/88 modules, data items defined PUBLIC in one module may be declared EXTERNAL by other modules that need to share the items. By this method, the modules actually access the same locations in memory. This method is particularly effective for sharing large data structures.

FORTRAN-86/88 modules share data by use of the COMMON statement. The COMMON statement establishes data segments and controls the layout of data items in those segments. The linker causes segments that are defined in different FORTRAN-86/88 modules but which have the same name to share the same memory area. ASM-86 modules can share these COMMON areas by defining data segments with the same names and formats. The name of a FORTRAN-86/88 COMMON segment is the name supplied in the COMMON list prefixed by the character @ ("at" sign) or just @ for unnamed lists. Other languages can gain access to a COMMON area if one of the FORTRAN-86/88 modules discloses the address of the COMMON area by passing a reference to the first data item in the COMMON area.

PARAMETER PASSING

All of Intel's high-level languages (FORTRAN-86/88, Pascal-86/88, and PL/M-86) support the use of parameters with subprogram definitions and calls. Parameters define passing of arguments to subprograms. Not only can arguments be passed to subprograms within one module but also between modules. All the high-level languages use a common parameter-passing mechanism for the data types they have in common.

The calling module passes arguments via the processor stack in one of two ways:

1. By value. The value of the argument is given to the subprogram.

2. By reference. The address of the argument is given to the subprogram. The subprogram can then use the address to access the value directly.

The called subprogram must know which method is being used for each parameter and must know the format of the data being passed.

The default method of parameter passing in FORTRAN-86/88 is by reference. Pascal-86/88 passes variable (VAR) parameters by reference. PL/M-86 passes by reference when a pointer is used as a parameter.

The non-ANSI function %VAL can be used with FORTRAN-86/88 INTEGER*n and LOGICAL*n parameters to interface with other languages that pass parameters by value. Pascal-86/88 passes value parameters by value. The default method in PL/M-86 is by value.

ASM-86 modules can use either method, but they must conform to the method used by the higher-level language.

Stack Usage

The calling module places arguments on either the 8086 stack or the 8087 register stack from left to right in the order in which they are declared. The first seven floating-point arguments passed by value are placed on the 8087 stack; parameter references and all other arguments (including floating-point arguments after the first seven) are placed on the 8086 stack.

Conventions for Register Usage

Each of the high-level languages has expectations about what registers are changed by subprograms and about the contents of registers when subprograms are called. Intel's high-level languages for the iAPX 86,88 family are compatible with each other in this regard. ASM-86 programmers must be careful to follow the conventions of the high-level languages when interfacing with them.

CHAPTER 2
THE DEVELOPMENT ENVIRONMENT

The hardware and software requirements for running a target application are rarely the same as those for developing the application. The development process requires tools such as editors, compilers, assemblers, linkers, and debuggers, all of which need hardware support in the form of memory, mass storage, and peripherals. If your target hardware is a fully featured OEM system, it may be possible for you to do your development on the same system. However, if your target hardware is a custom, specialized microsystem with minimal memory and peripherals, development must be carried out on a microcomputer development system or an adequately featured OEM system.

This chapter introduces the Intel systems and debugging products that are available to your application development effort.

SYSTEMS

Intellec Series III Microcomputer Development System

To aid you in developing your application, Intel offers the Intellec Series III Microcomputer Development System. The hardware and software features of this system are specially designed to help at every stage of product development. These features include:

- ISIS II operating system, to give you control of the Intellec System's resources.
- CREDIT CRT-based text editor, to help you create and modify source programs.
- Intel's application language and system implementation language translators and utilities.
- Built-in 8086 processor and operating system, to help you test application software modules.
- DEBUG-86 debugging tool, to help you find and eliminate faults in the software being tested on the built-in 8086.
- Hardware support for optional iSBC 957B iAPX 86,88 Interface and Execution Package. If your target system uses one of Intel's iAPX 86,88 single-board computer products such as the iSBC 86/12A, you can install the board in a separate chassis, down-load your programs, and perform board-level debugging with the aid of the Intellec Systems console.
- Hardware support for the optional iSBC 337 Multimodule Numeric Data Processor. This "piggyback" board may be installed either on the built-in 8086 processor board or on the iSBC 957B board. The iSBC 337 Multimodule Numeric Data Processor provides floating-point instruction capability for all Intel languages.

- Hardware and software support for optional In-Circuit Emulator (ICE) products. If your target hardware is a custom design that uses one of Intel's iAPX 86,88 components, the ICE products permit you to connect your Series III to your prototype hardware in place of the iAPX 86,88 component. The ICE software then gives you a "debugging window" into your prototype to help you work out hardware/software integration problems.

OEM Systems

Intel's OEM systems are general purpose microcomputer systems featuring the Multibus system bus, iSBC processor and memory boards, and Winchester and flexible disk storage. Extra slots are available for customer configuration, which might include line printer, CRT interface, or other peripherals.

The features that make Intel OEM systems suitable as a development environment include:

- Intel's application language and SIL translators and utilities
- Text editor
- iRMX 86 Real-Time Multitasking Operating System, designed for interrupt-driven, multitasking applications
- System Debug Monitor, for machine-level debugging
- Debugger module of the operating system, for symbolic debugging of the interactions between application tasks and the operating system

Custom Systems

You can also build your own system using Intel components, boards, and peripherals, as well as Intel's translators and iRMX 86 operating system.

DEBUGGING TOOLS

All of Intel's debuggers give you a "window" into the otherwise invisible process of program execution. You can define points at which you want processing to stop, and you can examine or change the state of data being processed. Intel offers a variety of debugging tools to suit various levels of debugging and to suit the various development environments.

The iAPX 86,88 Monitor Program

The iAPX 86,88 Monitor Program provides machine-level debugging capabilities for iAPX 86,88 single-board computer products (for example, the iSBC 86/12A). The iAPX 86,88 Monitor Program resides in PROM on the board. The monitor enables you to:

- Examine and modify the contents of iAPX 86,88 registers and absolute memory locations (including those of the 8087 Numeric Data Processor)

- Set breakpoints
- Single step program execution
- Do I/O to and from ports
- Move or compare blocks of memory

To learn more about the Monitor Program, refer to the User's Guide for the iSBC 957B iAPX 86,88 Interface and Execution Package.

The iRMX 86 System Debug Monitor

The iRMX 86 System Debug Monitor extends the capabilities of the iAPX 86,88 Monitor Program to include recognizing iRMX 86 system calls and displaying iRMX 86 data structures. For more information, refer to the iRMX 86 System Debug Monitor Reference Manual.

The iRMX 86 System Debugger

The iRMX 86 System Debugger is an optional layer of the iRMX 86 Operating System. Like the System Debug Monitor, the Debugger Module also recognizes iRMX 86 system calls and data structures, but offers several important additional capabilities:

- You can examine or manipulate one task while other tasks continue to run undisturbed.
- You can monitor operating system activity without interfering with execution.
- The Debugger Module can serve as the exception handler for system and application tasks. This feature puts a task into debugging mode only when a system exception occurs.

To learn more about the iRMX 86 Debugger refer to the iRMX 86 Debbuger Reference Manual.

Debug-86

Debug-86 resides in ROM on the Intellec Series III Microcomputer Development System. Debug-86 not only provides access to all of the on-board 8Ø86's registers and ports, but it also permits you to reference the locations of procedures and variables in your application programs by the symbolic names that you assign to them in your source code. Refer to the Intellec Series III Microcomputer Development System Console Operating Instructions for more information on Debug-86.

ICE-86A and ICE-88 In-Circuit Emulators

The ICE-86A and ICE-88 in-circuit emulators are hardware and software

systems that work with Intellec Development Systems to help you develop custom hardware/software systems that use iAPX 86,88 components. ICE hardware consists of three boards that are installed in the chassis of an Intellec System and a cable and buffer box that connect the Intellec System to your hardware. The interface between the in-circuit emulator and your hardware system is implemented at the connector pins of your system's processor chip. These pins carry the information that establishes the characteristics and status of your system. This interface makes it possible for ICE-86A or ICE-88 to emulate the function of the processor chip as it would perform in controlling and reacting to its hardware environment. The in-circuit emulators use the resources of the Intellec System to load programs, to emulate their execution, and to give you the power to monitor and control the emulation process.

ICE-86A and ICE-88 assist in all stages of development:

1. Before prototype hardware is available, the debugging capabilities can be used to facilitate software testing. Intellec System memory can be substituted for missing prototype hardware.

2. To begin integration of software and hardware development efforts, your prototype need consist of no more than a CPU socket. As each section of your hardware is completed, it can be added to the prototype. As prototype memory becomes available, you can replace the Intellec System equivalent. Thus each section of the hardware and software can be "system tested" as it becomes available.

3. When your prototype is complete, it can be tested using the software that will drive the final product. ICE-86A or ICE-88 can be used for real-time emulation of the iAPX 86 or iAPX 88 Processor to debug the system as a complete unit.

The most significant debugging features of the in-circuit emulators include:

- Access to and control of iAPX 86,88 registers, flags, and pins (including those of the 8087 Numeric Data Processor)

- Symbolic references to the data and procedures of your application programs

- A wide range of breakpoint conditions, through comparison of processor chip states with predesignated states

- Selective tracing

To learn more about in-circuit emulators, refer to ICE-86A Microsystem In-Circuit Emulator Operating Instructions for ISIS-II Users and ICE-88 In-Circuit Emulator Operating Instructions for ISIS-II Users.

TRANSLATOR SUPPORT FOR SYMBOLIC DEBUGGING

Several of the above-mentioned debugging aids (namely, Debug-86, ICE-86A, and ICE-88) feature symbolic references to objects in your application programs. This is made possible through use of the DEBUG control in the FORTRAN-86/88, Pascal-86/88, and PL/M-86 compilers and the ASM-86 assembler.

The DEBUG control causes the translator (compiler or assembler) to emit records in the object files that associate symbolic names with memory addresses. These records are then merged as required into the output files of LINK86 and LOC86, so that they are available in any executable module that may be processed by a debugger.

CHAPTER 3
RUN-TIME SUPPORT LIBRARIES

This chapter discusses layer 2 of the model shown in Figure Ø-1: the run-time libraries. Run-time libraries are libraries of procedures that you can link to your application programs. The procedures in the run-time libraries implement common compiler functions, thereby reducing the size of compilers and the code they generate, and they enable you to configure your application to eliminate costly features that are not needed. Configurability is discussed in Chapter 9 and Appendix B. This chapter identifies the libraries offered by Intel and explains their functions and usage.

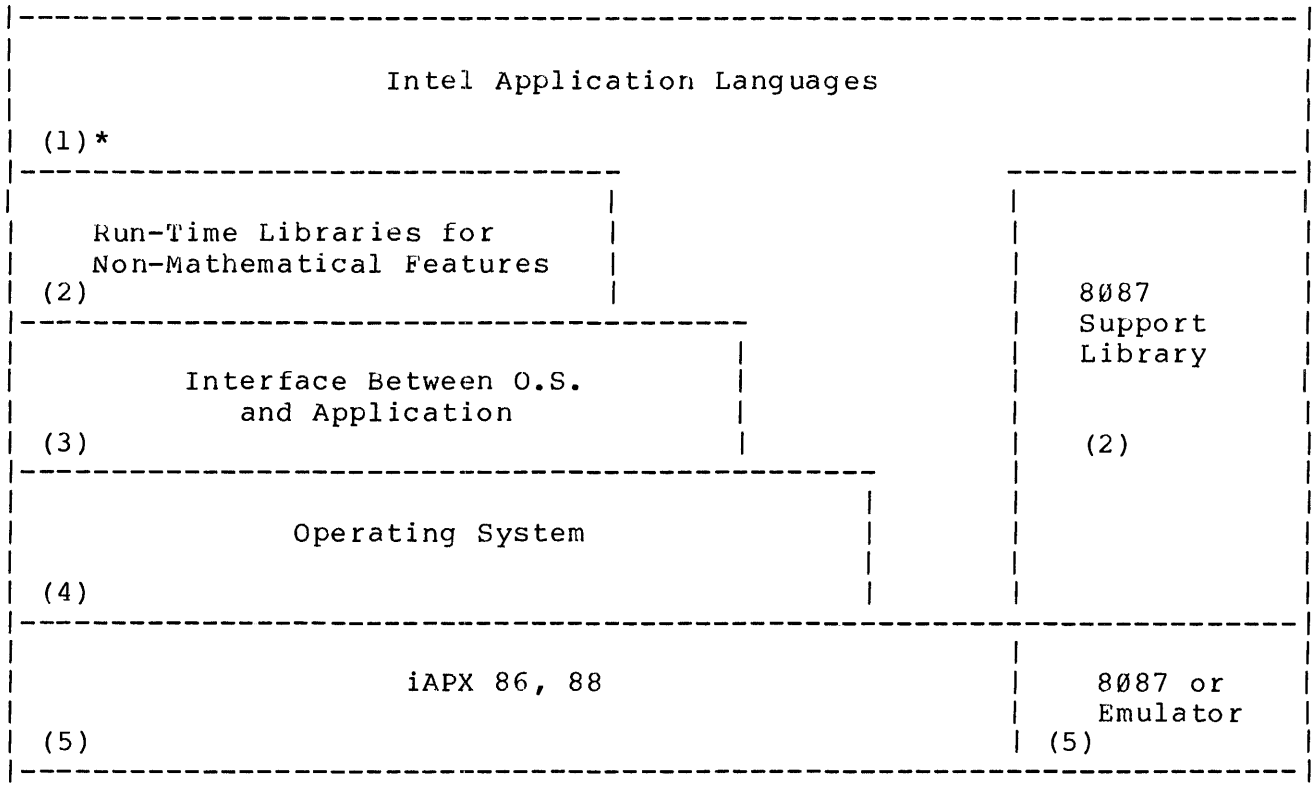
APPLICATION-LANGUAGE RUN-TIME SUPPORT

The application languages FORTRAN-86/88 and Pascal-86/88 have two classes of libraries:

1. Those that support processing of floating-point and mathematical functions using the 8Ø87 Numerics Data Processor or the 8Ø87 NDP Emulator

2. Those that do not deal with mathematical data processing

Figure 3-1 is a refinement of figure Ø-1, showing how these two classes of libraries interface with application languages in an iAPX 86,88 processing environment. Each horizontal line in the diagram represents a direct interface between layers of software or hardware. For example, application languages (through calls generated by the compilers) call procedures in the run-time libraries; the non-mathematical run-time libraries call procedures in layer 3, while the 8Ø87 Support Library either calls procedures in the 8Ø87 Emulator software or executes instructions of a hardware 8Ø87 Numeric Data Processor. The diagram also illustrates that application languages (through calls that you program in the source code) may bypass layer 2 and directly call procedures in layer 3 and layer 4.

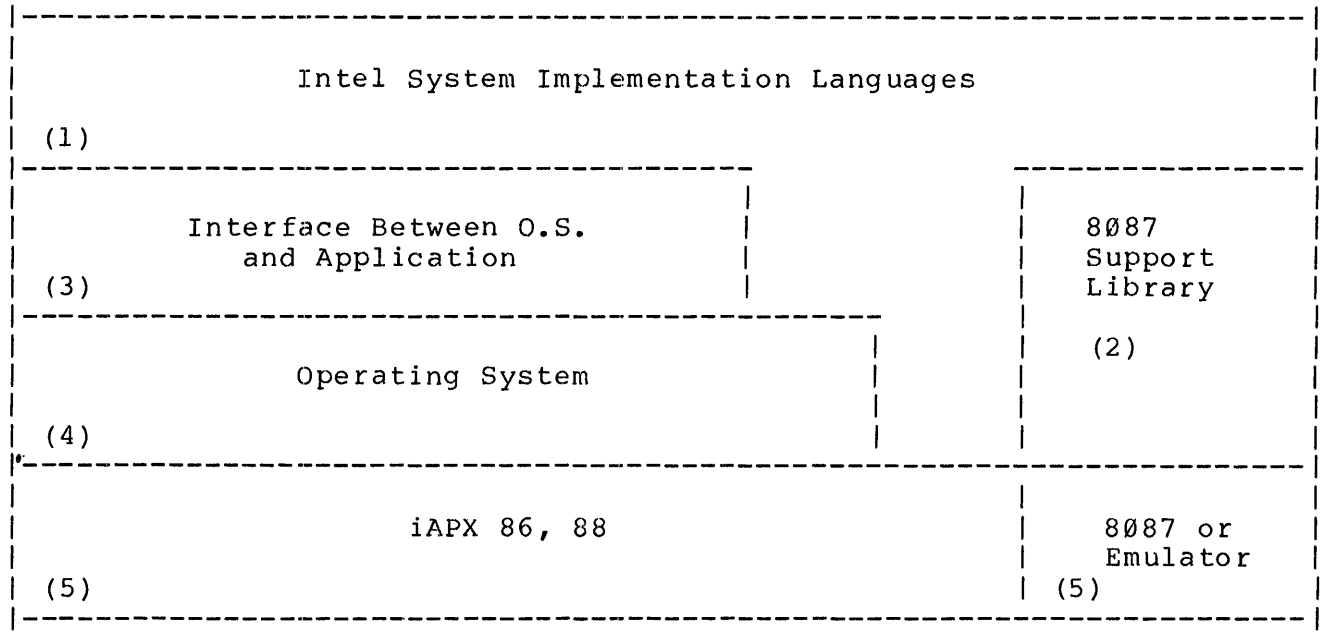


* Level numbers in parentheses correspond to those in figure 0-1

Figure 3-1. Application Language Run-Time Support

SIL RUN-TIME SUPPORT

System implementation languages (SIL's) have no need for the extensive non-mathematical run-time support libraries provided with application languages, since the SIL's do not provide the high-level language features that those libraries implement. However, floating-point arithmetic and mathematical functions using the 8087 Numeric Data Processor or 8087 NDP Emulator are supported by the 8087 Support Library, as illustrated in figure 3-2. Figure 3-2 shows that SIL's interface directly to layer 3. You implement this interface through explicit procedure calls that you write in the source language.



* Level numbers in parentheses correspond to those in figure Ø-1

Figure 3-2. SIL Run-Time Support

NON-MATHEMATICAL RUN-TIME LIBRARIES

The non-mathematical run-time libraries provide run-time execution of such application language features as:

- Sequential and direct access I/O
- Formatted and unformatted I/O
- Console input and output
- File management
- String processing
- Set manipulation
- Integer arithmetic
- Dynamic storage allocation
- Interrupt processing
- Run-time exception handling

You invoke these functions indirectly through your use of application-language syntax.

Figure 3-3 illustrates the non-mathematical run-time interfaces in greater detail.

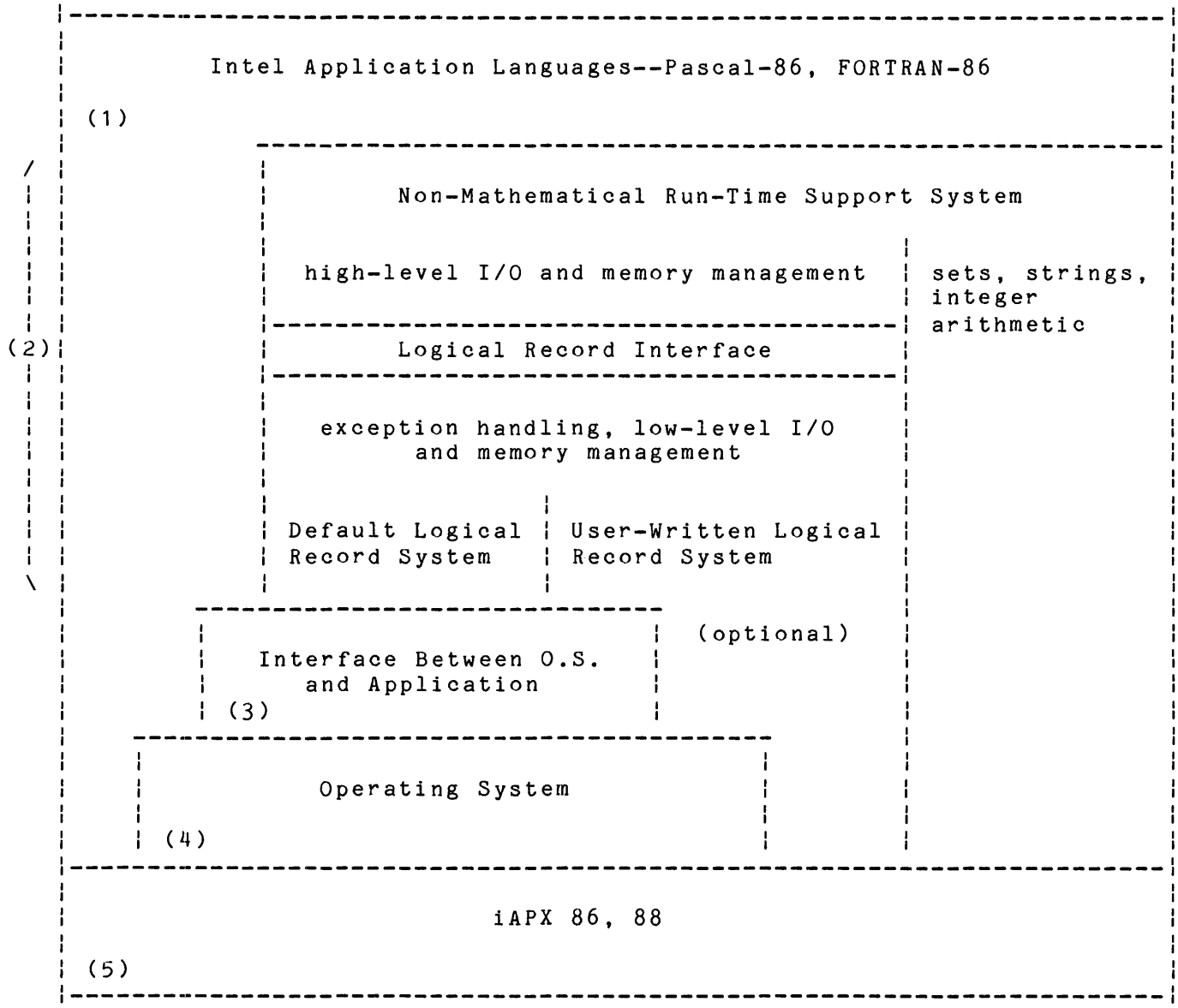


Figure 3-3. Detail of Non-Mathematical Run-Time Support

In figure 3-3, the longer vertical line in the interior of the box labelled "Non-Mathematical Run-Time Support System" divides the functions into two classes:

1. Those that are independent of the operating environment, requiring only the services of an iAPX 86,88 processor for their implementation (for example, procedures that process sets, strings, and integer arithmetic)

2. Those that depend ultimately on further support from the operating environment (for example, procedures that deal with I/O, memory management, and exception handling)

The operating environment assumed by the latter class of functions is the

Logical Record System (LRS). A standardized interface, called the Logical Record Interface (LRI), has been defined for the LRS. The LRS translates from the idealized operating environment defined by LRI onto the real operating environment. This can be done in one of two ways:

1. Intel provides procedures (contained in the run-time libraries) that translate to the Universal Development Interface (UDI) standard for layer 3. Layer 3 and UDI are discussed in following chapters.

2. You can write procedures to replace the Intel-supplied procedures. You need to do this if the target environment for your application does not include an operating system. Appendix B defines LRI and explains how to structure your own LRS.

Table 3-1 lists the FORTRAN-86/88 and Pascal-86/88 non-mathematical run-time libraries, and summarizes the functions performed by each library.

Table 3-1. Contents of Non-Mathematical Run-Time Libraries

Library Name		Description of Contents
FORTRAN-86/88	Pascal-86/88	
F86RN0.LIB	P86RN0.LIB	Non-reentrant interfaces, local data to support reentrancy.
F86RN1.LIB	P86RN1.LIB	OS-independent functions (sets, strings, integer arithmetic, high-level I/O and memory management).
F86RN2.LIB		Edit descriptor tables and null edit descriptor procedures.
F86RN3.LIB	P86RN2.LIB	Default LRS (exception handling, low-level I/O and memory management).
F86RN4.LIB	P86RN3.LIB	Default LRS device driver tables and null device driver procedures.
RTNULL.LIB	RTNULL.LIB	Null procedures linked in place of default LRS libraries when you supply an alternate LRS.

8087 SUPPORT LIBRARY

The 8087 Support Library implements floating-point calculations and mathematical functions in cooperation with the 8087 Numeric Data Processor

or (for those systems which do not have an 8087 processor) the 8087 software emulator. For more information about the 8087, refer to the iAPX 86,88 User's Manual.

The FORTRAN-86/88 and Pascal-86/88 products each include the 8087 Support Library. The compilers generate calls to procedures in these libraries when you program operations on REAL data items or call certain intrinsic mathematical functions. Refer to the FORTRAN-86 User's Guide or the Pascal-86 User's Guide for details of how to use this library.

The PL/M-86 product offers run-time support for operations on data items of type REAL (single precision). The compiler generates calls to these support routines when you use REAL items in arithmetic and relational expressions.

If you wish to use the full power of 8087 numerics processing in your PL/M-86 and ASM-86 programs you must obtain the 8087 Support Library. The 8087 Support Library used with PL/M-86 and ASM-86 is the same as that used with FORTRAN-86/88 and Pascal-86/88. The code generated by the PL/M-86 compiler uses the 8087 Support Library when you program operations on single-precision REAL data items; to use the other mathematical functions in the library, you execute procedure calls with the required parameters. For ASM-86 programs, you must explicitly program calling sequences for the routines in the Library. Refer to the 8087 Support Library Reference Manual for detailed information.

Table 3-2 summarizes the library files contained in the 8087 Support Library along with the floating-point subset available with PL/M-86.

Table 3-2. Contents of Numerics Run-Time Libraries

File Name		Description of Contents
PL/M-86	8087 Support Library	
	DCON87.LIB	Conversion between ASCII decimal formats and internal binary formats.
	CEL87.LIB	Common elementary functions (logarithmic, exponential, trig, hyperbolic, etc.).
	EH87.LIB	Floating-point exception-handling utility procedures.
8087.LIB	8087.LIB	Interface to 8087 processor.
E8087.LIB	E8087.LIB	Interface to 8087 emulator.
	E8087	8087 emulator for use with systems that do not have an 8087 processor.
PE8087		Partial 8087 emulator for simple arithmetic on REAL data types.
	NULL87.LIB	Null procedures linked in place of above libraries when no 8087 is used.

The file DCON87.LIB is used only by PL/M-86 and ASM-86 programs to perform type conversions; type conversion is invoked implicitly in the high-level languages FORTRAN-86/88 and Pascal-86/88.

The file EH87.LIB contains a foating-point exception-handling procedure called FILTER. FILTER implements the proposed IEEE floating-point standard for normalized arithmetic and non-trapping NaN's as defined in "A Proposed Standard for Binary Floating Point Arithmetic," Draft 8.0 of IEEE Task P754, Computer, March 1981, pp. 51-62. Refer to the 8087 Support Library Reference Manual for details about FILTER. The interface libraries 8087.LIB , E8087.LIB, and NULL87.LIB also contain a procedure called FILTER; however, this version of FILTER does nothing except return the value zero. When invoked to report an 8087 exception, the run-time system's error handler

calls FILTER. If you link EH87.LIB ahead of the interface library, then certain exceptions involving denormalized operands and non-trapping NaN's are filtered out. If you do not link EH86.LIB, the run-time system's error handler displays a message and terminates the job when any 8087 exception occurs.

USING THE RUN-TIME LIBRARIES

Figures 3-4, 3-5, and 3-6 summarize the run-time libraries available with each of Intel's languages. Link these libraries as needed to your application programs. Refer to the language manuals and the iAPX 86,88 Family Utilities User's Guide for details.

In figures 3-4, 3-5, and 3-6, the vertical lines in the box titled "8087 Support Library" delineate linkage options that depend on the execution enviroment.

- If your application uses 8087 features and your system includes an 8087 processor, then the LINK86 syntax should include CEL87.LIB, EH87.LIB, 8087.LIB.

- If your application uses 8087 features but your system does not have an 8087 processor, then the LINK86 syntax should include CEL87.LIB, EH87.LIB, E8087.LIB, E8087.

- If your application does not use 8087 features, you may need to link NULL87.LIB to resolve any unneeded references to 8087 Support Library procedures. (For example, the run-time system contains calls to 8087 initialization procedures even though your FORTRAN-86/88 or Pascal-86/88 source programs use no 8087 features.)

Refer to the 8087 Support Library Reference Manual for more information.

Not shown in figures 3-4, 3-5, and 3-6 is RTNULL.LIB. You should link this library to your application in place of the default LRS libraries (F86RN3.LIB and F86RN4.LIB for FORTRAN-86/88, or P86RN2.LIB and P86RN3.LIB for Pascal-86/88) when you supply an alternate LRS. RTNULL.LIB resolves references to procedures that you don't supply in your LRS. Refer to Appendix B for more information on how to write an LRS and on the procedures of RTNULL.LIB.

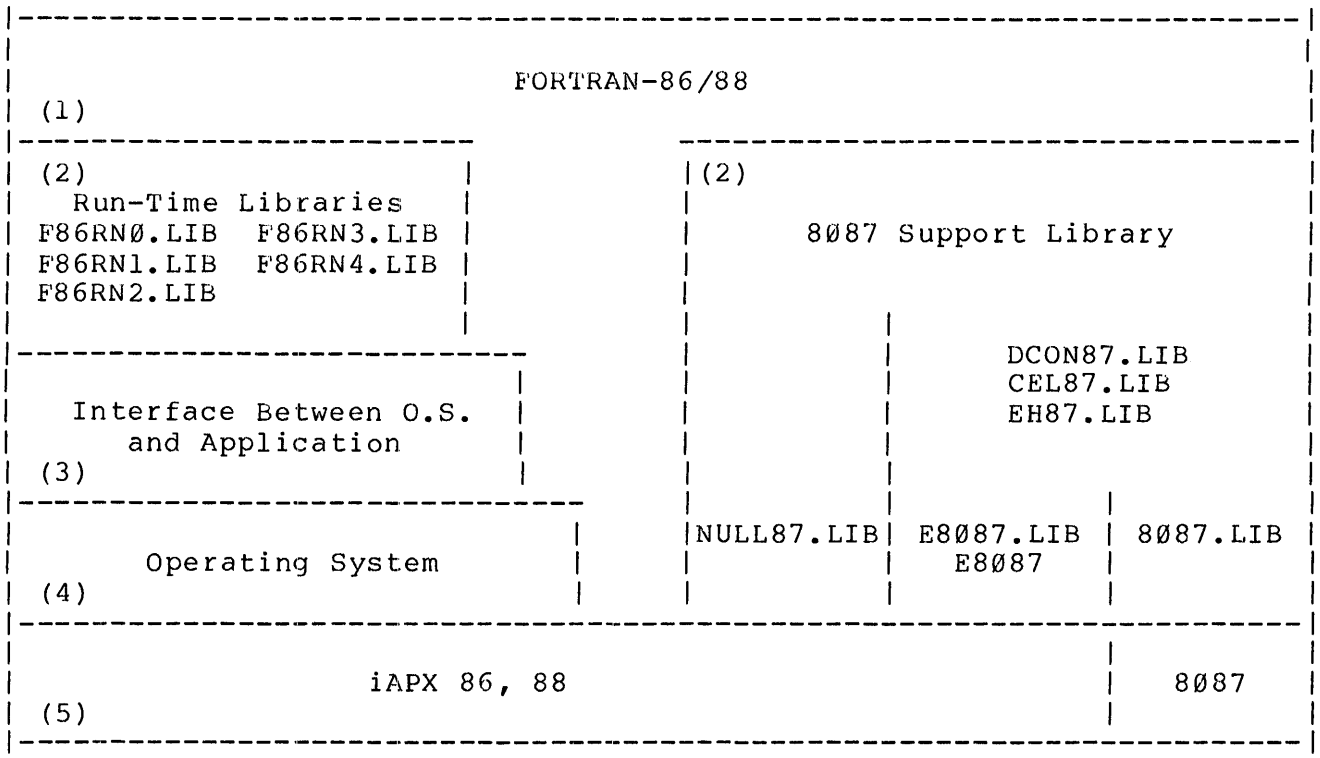


Figure 3-4. FORTRAN-86/88 Run-Time Libraries

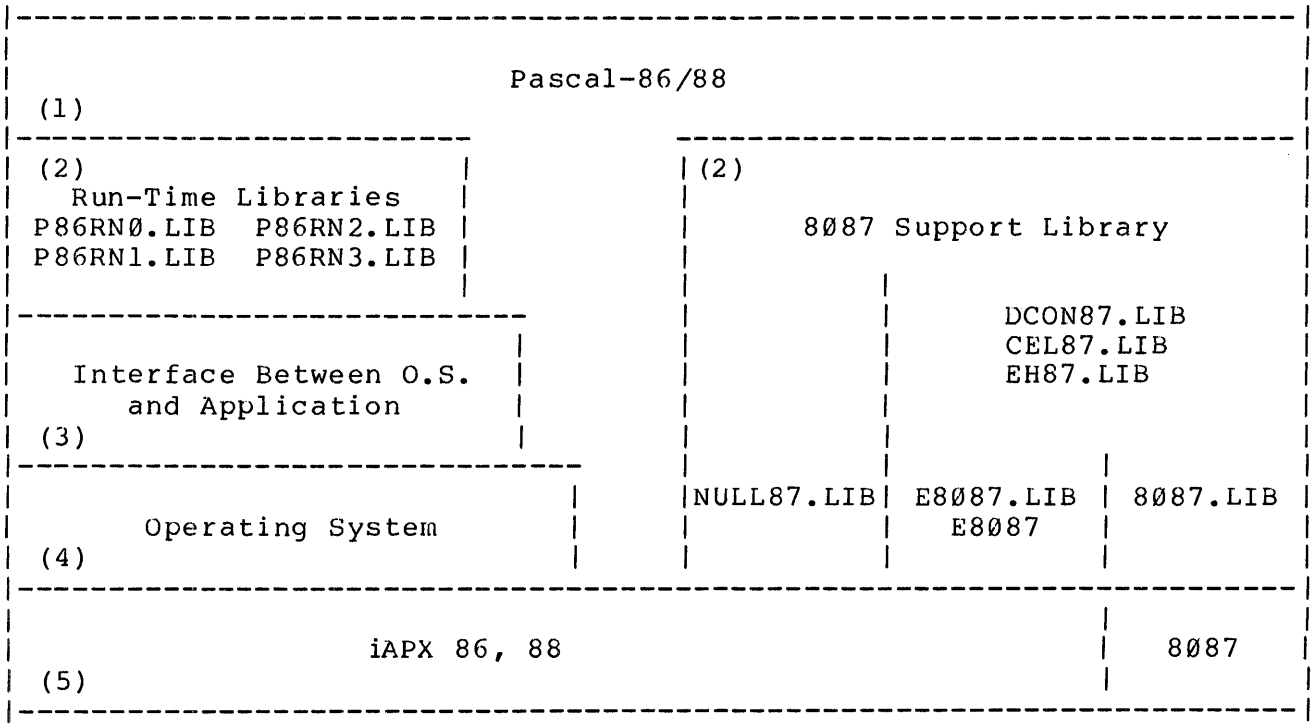


Figure 3-5. Pascal-86/88 Run-Time Libraries

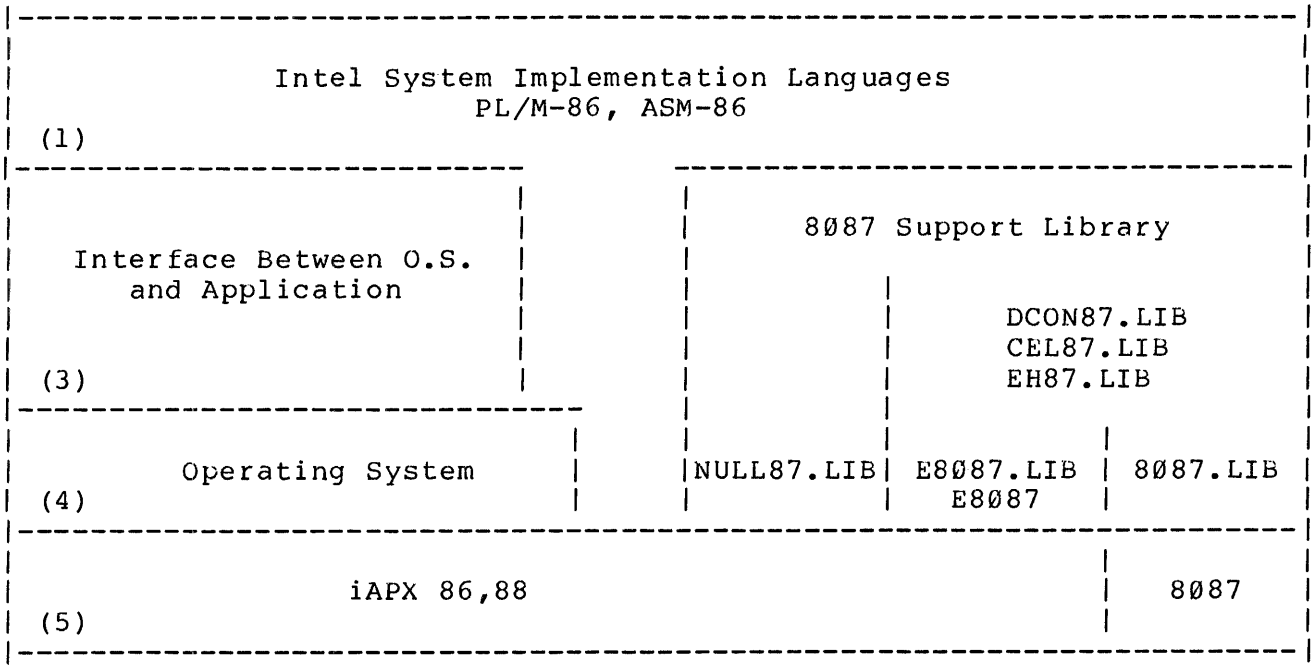


Figure 3-6. 8087 Support Libraries

Version Numbers

Each run-time library file has a version number stored in the file. Always use the latest version of each library.

To find the version number of a run-time library file use the LIB86 utility. For example, to find the version number of the FORTRAN-86/88 library file F86RN4.LIB, execute LIB86 and enter the command:

```
LIST F86RN4.LIB
```

LIB86 produces a listing such as the following:

```
F86RN4.LIB
  COPYRIGHT_INTEL_CORP_yyyy
  VERSIONNUMBERVØnm
        .
        .
        .
```

The symbols "nm" indicate that this is version n.m of the library file. The version number may also appear as:

```
VERSIONNUMBERVnPm
```

This indicates a library file of version n.m. Note that E8Ø87 is not a library file. Its version number is incorporated in the module name for E8Ø87, which appears on a link map.

Linking for Reentrancy

If an application job consists of more than one task, you may wish to link a single copy of the run-time libraries to several tasks. If, however, one task may be arbitrarily interrupted to execute another task that shares any of the same libraries, the procedures in the shared libraries must be reentrant. All of the procedures in the runtime libraries except for those in F86RNØ.LIB and P86RNØ.LIB are reentrant and may be shared. With F86RNØ.LIB and P86RNØ.LIB, however, each task must be linked to its own copy.

Figure 3-7 illustrates the permissible library sharing for a two-task Pascal-86/88 job. A FORTRAN-86/88 job would be similar, except that the four libraries F86RN1.LIB, F86RN2.LIB, F86RN3.LIB, and F86RN4.LIB can be shared; F86RNØ.LIB must be linked to each task. Each task must be a main module to allow separate initialization of the run-time system.

Note that tasks that are compiled according to the SMALL model of segmentation and are linked as shown in figure 3-7 cannot use the default version of the LRS procedure TQGETSMALL$HEAP. The tasks would interfere with each other in their use of the heap. To find out how to supply an alternate implementation of TQGETSMALL$HEAP, refer to Appendix B.

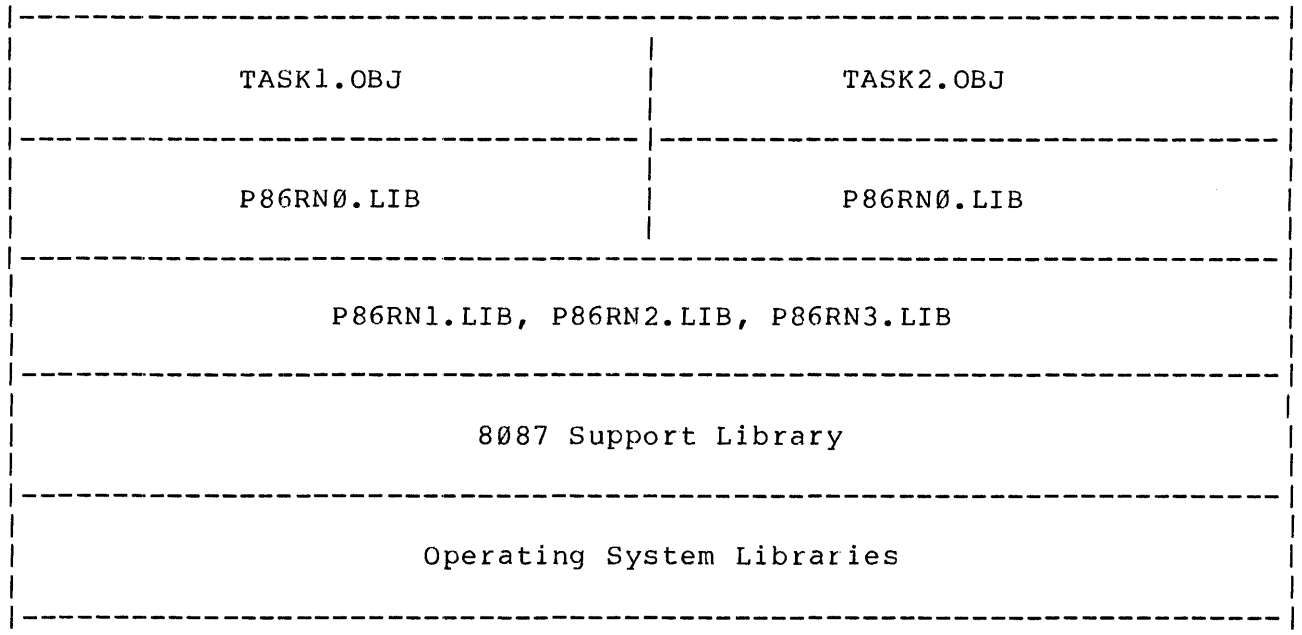


Figure 3-7. Linkage Structure of a Reentrant Pascal-86/88 Job

To create a job with the structure illustrated in figure 3-7, you must first link each task separately to P86RNØ.LIB. For example:

```
LINK86 TASK1.OBJ, P86RNØ.LIB TO TASK1.RNØ

LINK86 TASK2.OBJ, P86RNØ.LIB TO TASK2.RNØ
```

Then the two resulting files may be linked to the remaining libraries; for example:

```
LINK86 TASK1.RNØ, TASK2.RNØ, &
    P86RN1.LIB,      &
    P86RN2.LIB,      &
    P86RN3.LIB,      &
    NULL87.LIB,      &
    <O.S. library>   &
TO MYJOB1 BIND
```

This produces warning messages about duplicate symbols, since two copies of P86RNØ.LIB publics are being linked. These warnings may be ignored, or, if there is no requirement that TASK1 and TASK2 publics be available, the NOPUBLICS or the PURGE control may be used in the first two linkage steps.

Such a job is reentrant only if the operating system supports reentrancy. Refer to Chapter 4 and following chapters for more information on Intel's operating systems.

Linking Multi-Language Jobs

When a program contains modules written in both FORTRAN-86/88 and

Pascal-86/88, you may link both to the same pair of LRS libraries (either F86RN3.LIB and F86RN4.LIB, or P86RN2.LIB and P86RN3.LIB). Be sure to use the latest pair of libraries.

Figure 3-8 illustrates the possible sharing of libraries in a job that contains four reentrant tasks, two written in FORTRAN-86/88 and two written in Pascal-86/88. In this example the LRS libraries from FORTRAN-86/88 are used by all tasks.

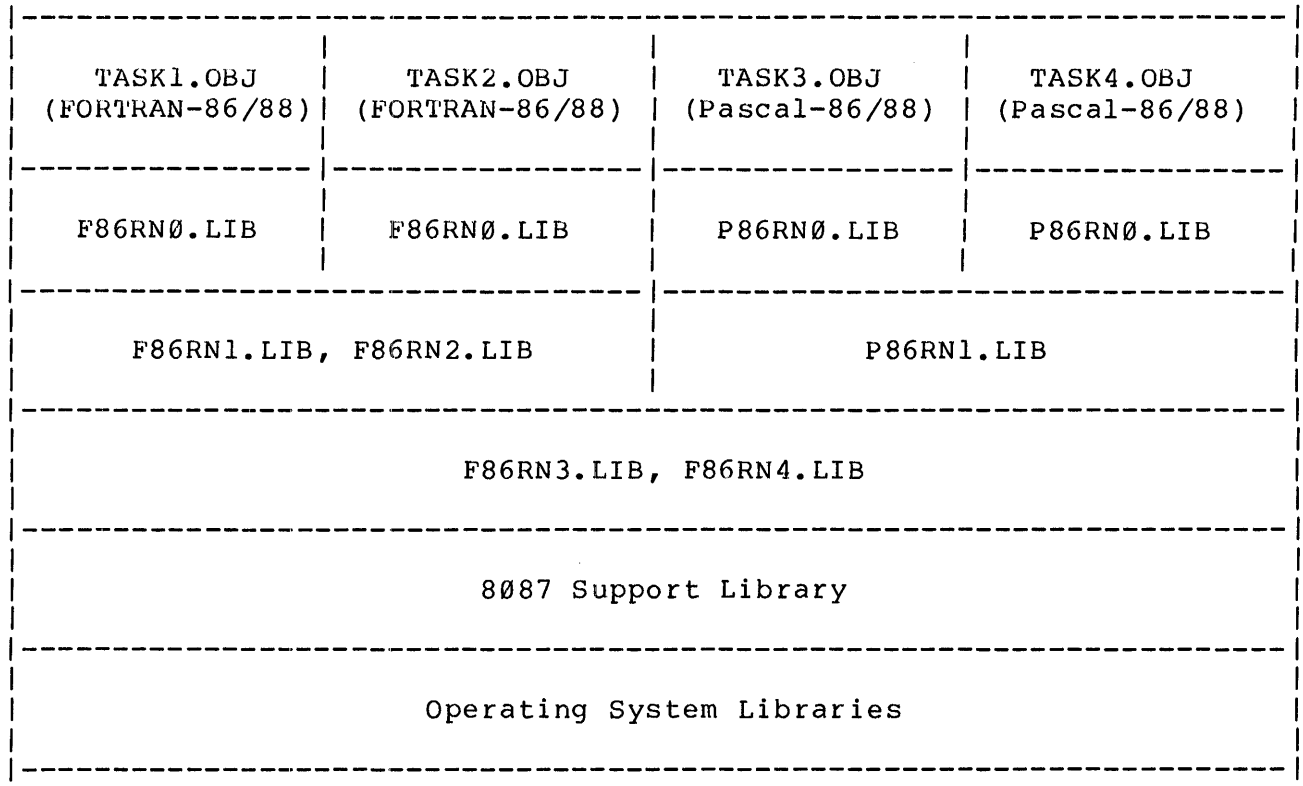


Figure 3-8. Linkage Structure of a Multi-Language Job

To create a job with the structure illustrated by figure 3-8, use a sequence of LINK86 commands such as these:

```
LINK86 TASK1.OBJ, F86RN0.LIB TO TASK1.FN0

LINK86 TASK2.OBJ, F86RN0.LIB TO TASK2.FN0

LINK86 TASK3.OBJ, P86RN0.LIB TO TASK3.PN0

LINK86 TASK4.OBJ, P86RN0.LIB TO TASK4.PN0

LINK86 TASK1.FN0, TASK2.FN0, F86RN1.LIB, F86RN2.LIB TO T1N2.LNK

LINK86 TASK3.PN0, TASK4.PN0, P86RN1.LIB TO T3N4.LNK

LINK86 T1N2.LNK, T3N4.LNK, &
    F86RN3.LIB,     &
    F86RN4.LIB,     &
    NULL87.LIB,     &
    <O.S. library>  &
TO MYJOB2 BIND
```

Using the 8087 Support Library with Multi-Language Jobs

When PL/M-86 or ASM-86 modules are included in a job with FORTRAN-86/88 and Pascal-86/88 modules, the PL/M-86 and ASM-86 modules can use the 8087 Support Library from either FORTRAN-86/88 or Pascal-86/88. The full 8087 emulator (E8087) from the 8087 Support Library can be used in place of the PL/M-86 partial emulator (PE8087).

Initialization for Subprograms

When FORTRAN-86/88 or Pascal-86/88 subprograms are linked to a main module written in ASM-86 or PL/M-86, you must code explicit calls in the main module to initialize both the non-mathematical run-time system and the 8087 Support Library. To initialize the non-mathematical run-time system, call TQ_001; to terminate the non-mathematical run-time system, call TQ_999. Note that only one call to TQ_001 and one call to TQ_999 are required, even though the program contains both FORTRAN-86/88 and Pascal-86/88 subprograms. To initialize the 8087 Support Library procedures and the 8087, call INITFP. Following are example external declarations that you must include in a PL/M-86 module for these procedures. All are FAR procedures.

```
TQ_001:  PROCEDURE EXTERNAL;
    END;

TQ_999:  PROCEDURE EXTERNAL;
    END;

INITFP:  PROCEDURE EXTERNAL;
    END;
```

Run-Time Detection of Linkage Errors

Some cases of incorrect linkage of run-time libraries are detected at run

time. This fact is reported to the current exception handler via the exception codes 1300H and 8017H. If either of these exception conditions occurs, check the commands that you gave to LINK86 to be sure that all the needed libraries are linked in the proper order. Exception code 8017H may also result from other conditions; refer to the FORTRAN-86/88 or Pascal-86/88 User's Guide for more information.

CHAPTER 4
INTERFACE BETWEEN APPLICATION AND OPERATING SYSTEM

This chapter deals with operating system interfaces in a general way. It introduces terms and concepts that should be considered early in the development process, if you want an application that can be transported from one operating environment to another. The terms and concepts discussed in this chapter are applied to each of Intel's operating systems in Chapters 5, 6, and 7.

UNIVERSAL DEVELOPMENT INTERFACE (UDI)

Intel has defined the Universal Development Interface (UDI) so that you can transport your applications from one operating environment to another (for example, from the development environment to the target production environment). UDI is a specification of a set of procedure calls that are used to request operating system functions. The kinds of functions that are available through UDI procedure calls include:

- Creating and breaking connections to data files
- Opening, reading, seeking, writing, and closing data files
- Changing names of data files
- Controlling program execution
- Controlling memory allocation
- Handling system exception conditions
- Controlling the processing of console input
- Parsing the text of a command line
- Fetching the current date and time
- Fetching the name of the operating system

For complete specification of UDI, refer to appendix A.

Each Intel operating system for the iAPX 86,88 Family provides a Universal Development Interface or a subset thereof. The UDI specifications are implemented in one of two ways, depending on the operating system:

1. By the operating system itself, whose system calls follow the UDI specifications

2. By modules that translate from the UDI standard to the actual operating system calls

In either case, the UDI interfaces are implemented as libraries that must be linked to your application modules.

Figure 4-1 illustrates the second method of UDI implementation, showing how UDI libraries (layer 3) fit into the run-time support model. The figure illustrates three points worth considering in more detail:

1. The non-mathematical run-time libraries make UDI calls when they need operating system services to carry out their functions. The run-time libraries, therefore, are operating-system independent and can be transported to any operating environment that supports UDI.

2. You can make UDI calls directly from your application if you need

operating system services beyond those supplied by the run-time libraries. Adhering to UDI specifications ensures that your application remains operating-system independent and transportable.

3. You can make operating system calls directly from your application (subject to system and language restrictions). If you do so, however, you may not be able to transport your application to another operating environment.

If your application must run under an operating system other than one supplied by Intel, you can make your application transportable to that environment, too. To do so, you need to implement a library of routines that translate from the UDI specifications to that operating system's native interfaces. For more information on how to do this, refer to Chapter 8.

As Figure 4-1 illustrates, the 8087 Support Library does not depend on an operating system; therefore it can be tranported to any iAPX 86,88 operating environment.

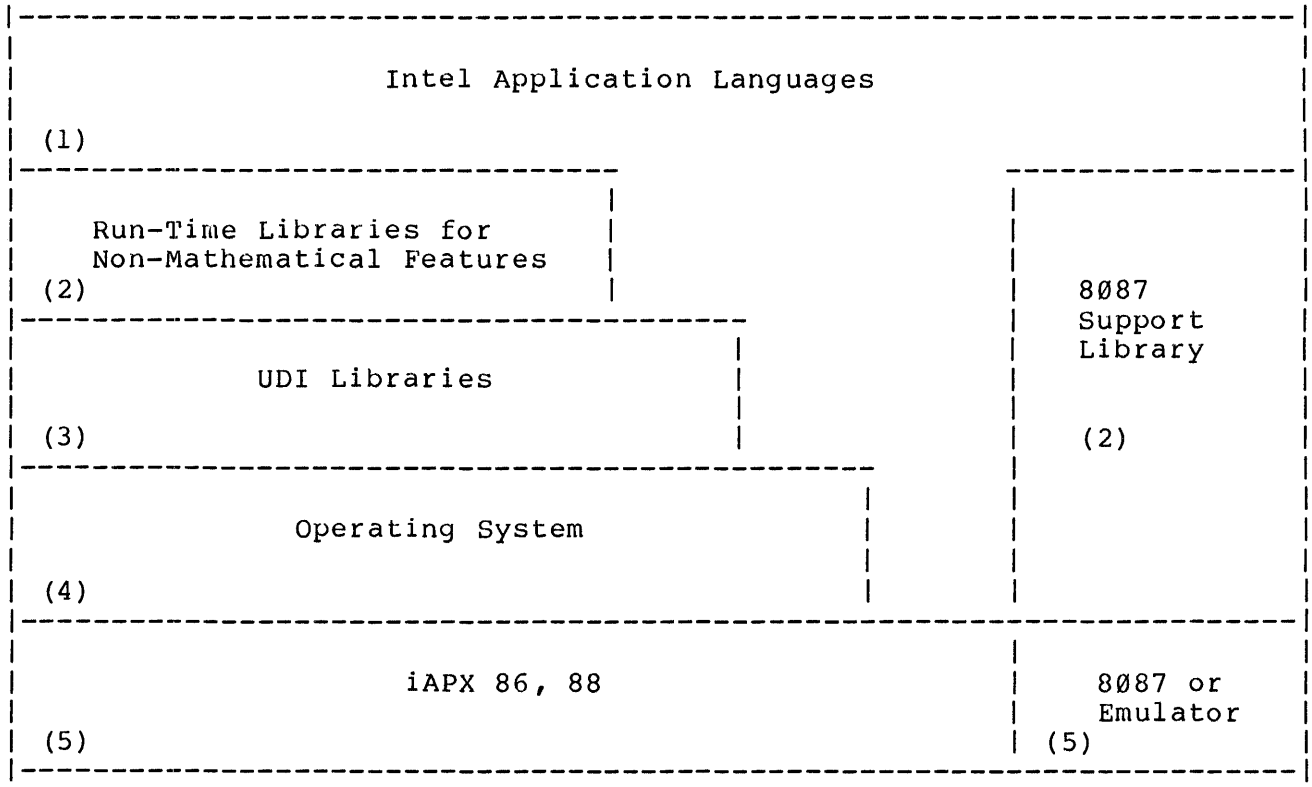


Figure 4-1. UDI Support

ERROR REPORTING

UDI procedures return a condition code that indicates the results of executing a UDI procedure. You must check the condition code after each UDI call to determine whether the procedure produces the results that you

expect. Certain condition codes are identified in the UDI specifications (Appendix A); however, each operating system is free to return additional condition codes. The condition codes returned by one operating system may not be the same as those returned by another.

You should take care to design your error-handling logic to accomodate the condition codes returned in all of the environments in which your application may run.

INTERRUPT HANDLING

There are two different styles of interface available for interrupt processing:

1. Special interrupt-related constructs in the programming languages

2. Calls to operating-system procedures

Using the language constructs for interrupt processing is adequate for environments where there is no operating system or where the operating system does not provide any other interface. Where the operating system does provide an interrupt-processing interface, that interface should be used. Using the operating-system interface ensures that the operating system has enough information to effectively control the operating environment.

To simplify transportation of your application between an environment where language constructs are used and an environment where operating-system calls are used, you should isolate all interrupt processing in a separate module. By having a separate interrupt-processing module:

1. You can hide from other application modules any knowledge of which interface is actually in use.

2. You can test each interface separately, before linking to the rest of the application.

All of Intel's operating systems for the iAPX 86,88 Family provide interrupt handlers for interrupts caused by the run-time system. These interrupts include:

- Interrupt Ø--Divide exception
- Interrupt 4--Overflow
- Interrupt 5--Range check
- Interrupt 16--Floating-point exception
- Interrupt 17--Case range, procedure stack overflow

If you are not using one of Intel's operating systems but are using the run-time system, you must supply your own interrupt handlers for these interrupt levels.

Table 4-1 identifies for each language the syntax or logic that is used for

interrupt processing.

Table 4-1. Language Interfaces for Interrupt Processing

INTERRUPT PROCESSING FUNCTION	PROGRAMMING LANGUAGE INTERFACE			
	Pascal-86/88	FORTRAN-86/88	PL/M-86	ASM-86
Associate procedure with interrupt level.	SETINTERRUPT built-in procedure or INTERRUPT compiler control	SETINT built-in procedure	INTVECTOR/ NOINTVECTOR compiler control, SET$INTERRUPT built-in, INTERRUPT procedure attribute	You declare interrupt vector.
Save and restore state.	INTERRUPT control in interface specs.	INTERRUPT compiler control	INTERRUPT procedure attribute	You code.
Finish processing interrupt.	END statement in interrupt procedure	RETURN or END statement in interrupt procedure	RETURN or END statement in interrupt procedure	IRET
Reentrancy	All procedures are reentrant.	REENTRANT compiler control	REENTRANT procedure attribute	Use stack-relative variables
Enable and disable interrupts.	ENABLEINTERRUPT, DISABLEINTERRUPT built-in procedures	Automatically disabled on entry and enabled on exit	ENABLE, DISABLE statements	STI CLI
Cause interrupt from software.	CAUSEINTERRUPT built-in procedure		INTERRUPT$PTR built-in, CALL interrupt procedure	INT INTO

CHAPTER 5
SERIES III RUN-TIME SUPPORT

The Intellec Series III contains two processors: an 8085 and an 8086. ISIS-II is the operating system that runs on the 8085 processor, and the Series III Operating System runs on the 8086 processor. The Series III Operating System takes control when you enter the ISIS-II command RUN to execute programs on the 8086. To fulfill many of its functions, the Series III Operating System communicates with ISIS-II which continues to run on the 8085.

The Series III Operating System provides support for application programs being developed for the 8086 and for application development tools that run on the 8086. Figure 5-1 illustrates how the Series III Operating System fits into the run-time support model. Details on the Series III Operating System are to be found in the Intellec Series III Microcomputer Development System Programmer's Reference Manual.

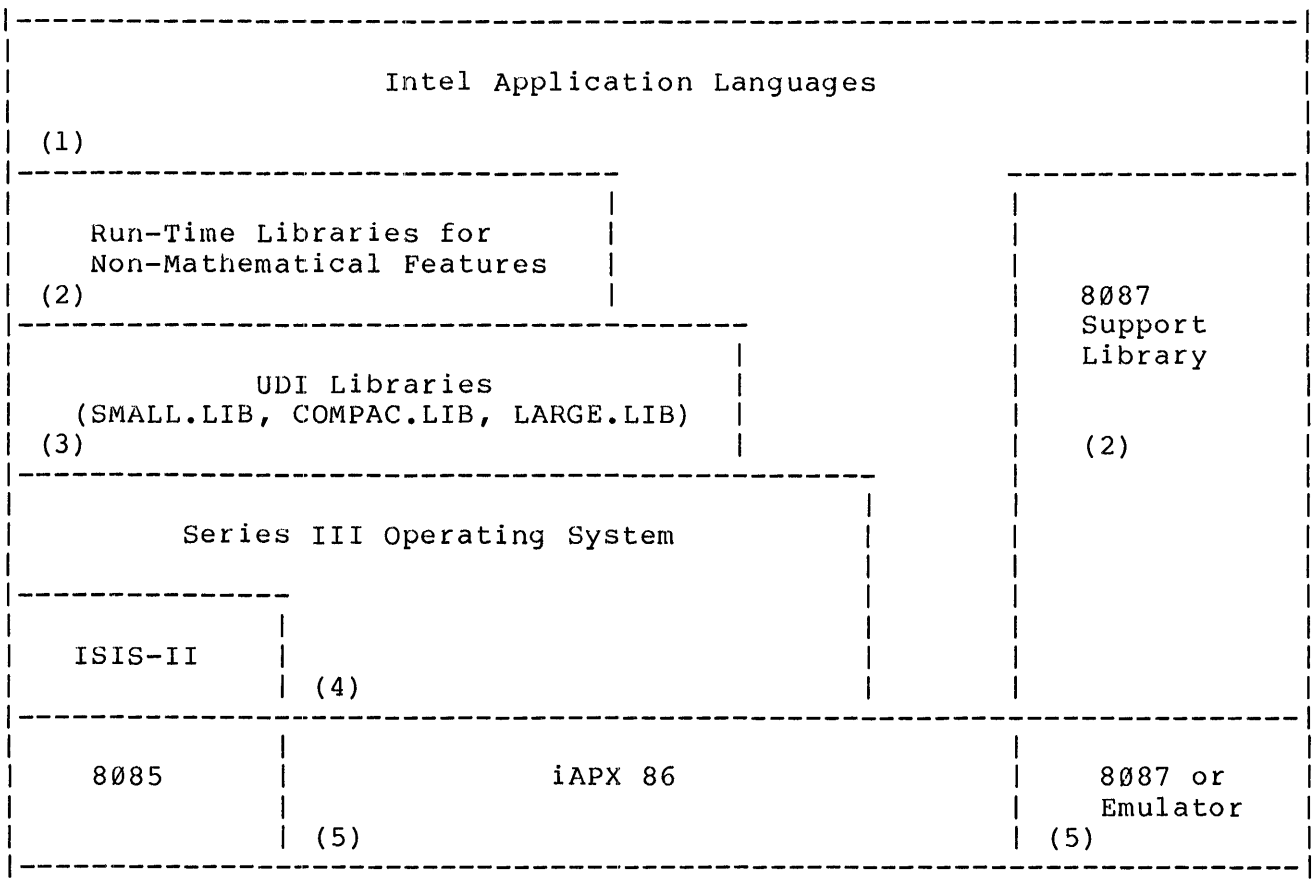


Figure 5-1. Intel's Series III Solution

UDI FOR THE SERIES III OPERATING SYSTEM

Programs interface with the Series III Operating System according to the UDI standard defined in Appendix A. All the procedure calls defined by the standard are supported.

Libraries

The Series III Operating System provides the UDI interface in the form of libraries. The libraries are named SMALL.LIB, COMPAC.LIB, and LARGE.LIB. To execute an application program which uses UDI, you must link the program to the library or libraries that match the program's segmentation scheme. (LARGE.LIB may also be used for the MEDIUM model of segmentation.)

Implementation Considerations

The UDI specifications (as documented in Appendix A) permit variations in implementation that depend on the style and capablities of individual operating systems. The Series III variations are identified in this section, along with any deviations from the UDI standard. References to particular procedures are understood to refer to the Series III versions of these procedures.

DQ$ATTACH

This procedure supports a maximum of 12 files attached at any one time. Only one connection can be established to a particular disk file. However, multiple connections to physical files (for example, :LP:) and logical devices (for example, :BB: and :CI:) are allowed. The logical file :WORK: may not be attached (use DQ$CREATE instead).

DQ$CHANGE$ACCESS

For ISIS files CLASS is ignored. The ISIS WRITE PROTECT attribute will be OFF (0) if the ACCESS bits for UPDATE and DELETE or WRITE and DELETE are ON (1). Otherwise, the WRITE PROTECT attribute is ON (1).

For NDS II files the values are as specified by the DQ$CHANGE$ACCESS procedure in Appendix A.

DQ$CREATE

This procedure internally opens the file to check whether it exists and then closes it. This action can impact the limit on the number of open connections.

DQ$DELETE

You may not delete a file that is connected.

DQ$EXIT

The completion$code is not used by the Series III Operating System.

DQ$FILE$INFO

For ISIS files, OWNER is always returned as "4,ISIS", LENGTH is the current file length, TYPE is 1 for ISIS.DIR and 0 for all other files. OWNER$ACCESS and WORLD$ACCESS is 02H if the WRITE PROTECT and/or FORMAT attribute is set, otherwise they are 0FH. CREATE$TIME is always 0 while LASTMODTIME is 0 if ISIS DIRTY attribute is off (0) and 1 if it is on (1).

For NDS II files the values are as specified by the DQ$FILE$INFO procedure in Appendix A.

DQGETTIME, DQ$DECODE$TIME

System time is not maintained by the Series III Operating System. E$SUPPORT is returned.

DQ$OPEN

This procedure is restricted to six files maximum open at one time including :CI: and excluding :CO:.

DQ$RENAME

Renaming a file to which a connection has been established is valid. The connection to the renamed file remains established.

DQ$RESERVE$IO$MEMORY

Returns E$MEM if NUMBER$FILES exceeds twelve files.

DQ$TRUNCATE

During the truncate operation, a workfile is opened and remains open until truncation completes. This may impact the limit of six open files.

Exception Codes

Every UDI call returns an exception code that specifies the status of the call. If the operating system returns an exception code of zero, it did not find any errors when it processed the call. However, if the operating system returns a non-zero exception code, it did find errors.

Table 5-1 lists all the exception codes, along with their mnemonics, that the Series III Operating System can return from its UDI calls. For

definitions of the conditions and details as to which routines can return which conditions, refer to the Intellec Series III Microcomputer Development System Programmer's Reference Manual.

Table 5-1. Series III Exception Codes and Mnemonics

HEX CODE	MNEMONIC	HEX CODE	MNEMONIC
0000	E$OK	0104	E$NOPEN
0002	E$MEM	0105	E$OPEN
0020	E$FEXIST	0106	E$OREAD
0021	E$FNEXIST	0107	E$OWRITE
0023	E$SUPPORT	0108	E$PARAM
0026	E$FACCESS	0109	E$PTR
0028	E$SHARE	010A	E$SIX
0029	E$SPACE	010C	E$SYNTAX
0081	E$STRING$BUF	010E	E$UNSAT
0101	E$CONTEXT	010F	E$ADDRESS
0102	E$CROSSFS	0110	EBADFILE
0103	E$EXIST		

INTERRUPT HANDLING

The Intellec Series III System maps the seven Multibus interrupt lines (INT0 through INT7) onto interrupt vector entries numbered 56 through 63; therefore your application may not use interrupts 56 through 63 for software interrupts. Interrupt vector entries available for user software include 64 through 183. Refer to the Intellec Series III Microcomputer Development System Programmer's Reference Manual for details.

8087 SUPPORT

You may incorporate an 8087 Numeric Data Processor in your Intellec Series III System by installing the iSBC 337 Multimodule Numeric Data Processor. Refer to the iSBC 337 Multimodule Numeric Data Processor Hardware Reference Manual for more information. You must also incorporate in your application two software procedures that help to handle 8087 interrupts.

When the iSBC 337 Multimodule NDP is installed in a Series III system, the interrupt output of the 8087 (MINT) is connected to the IR7 pin of the 8259A Programmable Interrupt Controller, which associates the 8087 interrupt with interrupt type number 63. The run-time system, however, expects the 8087 interrupt to arrive at interrupt number 16. To translate from interrupt 63 to interrupt 16, you must link to your application programs an interrupt procedure such as the one shown in figure 5-2.

If necessary the run-time system writes at entry 16 of the system interrupt vector the address of the interrupt procedure that is to process 8087 interrupts. To find the location of that interrupt procedure, the run-time

system calls a procedure of the form shown below:

```
TQ$WHERES$TRAP87:
    PROCEDURE (handler$ptr$ptr) WORD REENTRANT PUBLIC;
    DECLARE handler$ptr$ptr POINTER;
    ;
    END;
```

The parameter handlerptrptr points to a four-byte area where TQ$WHERES$TRAP87 stores a long pointer containing the address of the interrupt handler procedure that is to handle 8087 interrupts.

The WORD returned by TQ$WHERES$TRAP87 contains either the value 16, the number of the interrupt vector entry (ultimately) associated with the 8087, or the value zero, indicating that the operating system has already set up an interrupt procedure for handling 8087 interrupts.

The default version of TQ$WHERES$TRAP87 in the run-time libraries returns a value of zero. However, the Series III Operating System does not initialize the interrupt vector for 8087 interrupt handling. Therefore, you must supply a version of TQ$WHERES$TRAP87 that returns the value 16 and writes out the address of an 8087 interrupt procedure. Intel provides an interrupt procedure (with the PUBLIC identifier TQ_TRAP87) that fields 8087 interrupts and calls the current exception handler. You may use the address of TQ_TRAP87 in your version of TQ$WHERES$TRAP87. You must link your version of TQ$WHERES$TRAP87 before the run-time libraries, so that the linker fetches your version in place of the default version. Figure 5-2 shows an example of a TQ$WHERES$TRAP87 procedure that uses the address of TQ_TRAP87.

```
$LARGE

MYSIII8087CONFIG:  DO;

TQ_TRAP87:
    PROCEDURE EXTERNAL;
    END;

TQ$WHERES$TRAP87:
    PROCEDURE (ADDR$PTR) WORD REENTRANT PUBLIC;
    DECLARE ADDR$PTR              POINTER,
            DESIRED$TRAP$HANDLER
                BASED ADDR$PTR    POINTER;

    DISABLE;
    OUTPUT(0C2H) = INPUT(0C2H) AND 7FH;
        /* Change 8259A mask to enable IR7. */
    ENABLE;

    /* Set address of default 8087 trap handler. */
    DESIRED$TRAP$HANDLER = @TQ_TRAP87;

    RETURN (16); /* Tell run-time system which interrupt
                    table entry to initialize with
                    address in desired$trap$handler.     */

    END TQ$WHERES$TRAP87;

MY$8087$TRAP:
    PROCEDURE INTERRUPT 63; /* Maps to IR7 in Series III. */

    CAUSE$INTERRUPT (16);   /* Translate to interrupt 16. */

    OUTPUT(0C0H) = 67H;     /* Send specific EOI for IR7. */

    END MY$8087$TRAP;

END MYSIII8087CONFIG;
```

Figure 5-2. Series III 8087 Interrupt Initialization

REENTRANCY AND MULTITASKING

The Series III Operating System is designed for use by a single operator and supports neither reentrancy nor multitasking.

EXAMPLE PROGRAM

The following example illustrates the process of using an Intellec Series

III Microcomputer Development System to compile, link (both to the UDI and to the run-time libraries), load, and execute a simple Pascal-86/88 program (shown in figure 5-3).

Although this program makes no direct UDI calls, the code generated by the Pascal-86/88 translator does call UDI procedures for I/O to the console. The example shows how to link this program both for execution on the Series III, and for execution on an iRMX 86 system.

```
(* Convert a number of inches into yards, feet, and inches *)

PROGRAM inch(input,output);
VAR yards, feet, f_inch, number : integer;
    quitchar : char;
PROCEDURE convert (ins : integer ; VAR y, f, i : integer);

    BEGIN
        y := ins DIV 36;
        ins := ins MOD 36;
        f := ins DIV 12;
        i := ins MOD 12;
    END;

BEGIN
    REPEAT
        writeln; writeln;
        write('Number of inches is: ');
        readln(number);
        writeln;
        convert(number, yards, feet, f_inch);
        writeln(yards:4, ' yards, ',
                feet:1, ' feet, and ',
                f_inch:2, ' inches');
        writeln; writeln;
        write('Another number--y or n? :');
        read(quitchar);
    UNTIL NOT (quitchar in ['Y','y'])
END.
```

Figure 5-3. Series III Example Program

Compiling

The following line invokes the Series III Pascal-86/88 compiler. This example assumes the compiler is in the system and that the program is to be compiled on the default device.

```
-RUN PASC86 INCHES.SRC
```

The compiler places the object module in file INCHES.OBJ and produces a listing file called INCHES.LST. If the compiler finds no errors, it responds as follows:

```
SERIES-III Pascal-86, Vx.y
PARSE(Ø), ANALYSE(Ø), NOXREF, OBJECT

    COMPILATION OF INCHES COMPLETED, Ø ERRORS DETECTED,
    END OF Pascal-86 COMPILATION.
```

Linking for Series III Execution

If there are no errors, you are ready to link the compiled program to the necessary run-time support libraries and the large model UDI library as follows.

```
-RUN :F1:LINK86 INCHES.OBJ, &
>>P86RNØ.LIB, &
>>P86RN1.LIB, &
>>P86RN2.LIB, &
>>P86RN3.LIB, &
>>NULL87.LIB, &
>>LARGE.LIB   &
>>TO :F1:INCHES.86 BIND
```

The BIND control directs LINK86 to produce :F1:INCHES.86 in a load time locatable format. Refer to the iAPX 86,88 Family Utilities User's Guide for more information concerning the BIND control.

The system responds as follows:

```
SERIES-III 8Ø86 LINKER, Vx.y
```

LINK86 produces a map file (INCHES.MP1) and a loadable module (INCHES.86).

Invoking

You can now invoke the program. Type

```
-RUN :F1:INCHES
```

The program responds with

```
Number of inches is :
```

Assume you enter "38". The program displays the number in yards, feet, and inches.

```
1 yard, Ø feet, and 2 inches
```

The program then asks if you want to enter another number.

```
Another number--y or n?:
```

Answer with N if you wish to exit the program.

Linking for Execution on an iRMX 86 System

After testing the program on the Series III system, you are ready to relink the compiled program for running on the target system, which in this example is an iRMX 86 system. You use the linker again, this time substituting the iRMX 86 large model UDI library and omitting the .86 suffix from the name of the output file:

```
-RUN :F1:LINK86 INCHES.OBJ, &
>>P86RN0.LIB, &
>>P86RN1.LIB, &
>>P86RN2.LIB, &
>>P86RN3.LIB, &
>>NULL87.LIB, &
>>URXLRG.LIB &
>>TO :F1:INCHES BIND MEMPOOL(+2000H)
```

The BIND control directs LINK86 to produce :F1:INCHES in a load time locatable format. The MEMPOOL (+2000H) control dynamically allocates 2000H more bytes of memory for the program and connections. Refer to the iAPX 86,88 Family Utilities User's Guide for more information concerning the BIND and MEMPOOL controls.

The system responds as follows:

```
SERIES-III 8086 LINKER, Vx.y
```

LINK86 produces a map file (INCHES.MP1) and a loadable module (INCHES).

Refer to the Guide to Using iRMX 86 Languages for further information concerning compiling, linking, and executing application programs you want to run on the iRMX 86 Operating System.

CHAPTER 6
iRMX™ 86 RUN-TIME SUPPORT

The Intel iRMX 86 Operating System is a software package designed for use with the Intel iAPX 86- and iAPX 88-based microcomputers. It is a powerful and flexible system around which you can build your application.

UDI FOR THE iRMX 86 OPERATING SYSTEM

The iRMX 86 Operating System consists of a number of layers. In order to use the complete UDI, you must have the Human Interface, Application Loader, Extended I/O System, Basic I/O System and Nucleus. The following list contains short descriptions of the layers you need to support the UDI.

- **Nucleus** — The Nucleus 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.
- **Basic I/O System** — The Basic I/O System provides asynchronous file access capabilities for software running under the supervision of the Nucleus.
- **Extended I/O System** — The Extended I/O System provides high level, synchronous file access capabilities for software running under the supervision of the Nucleus.
- **Application Loader** — The Application Loader provides the capability to load object files into memory from disk under the control of the operating system.
- **Human Interface** — The Human Interface provides an interactive interface between a user and the software running under the supervision of the Nucleus.

You must include these layers when you configure the iRMX 86 Operating System. You must also include the UDI layer when you configure. Refer to the iRMX 86 Configuration Guide for more information.

Programs interface with the iRMX 86 Operating System according to the standard defined in Appendix A. All the procedure calls defined by the standard are supported.

Libraries

The iRMX 86 Operating System provides the UDI interface in the form of libraries. In order to execute an application program (on the iRMX 86 Operating System) that uses UDI, you must link the program to one of three iRMX 86 UDI libraries. These libraries are called LARGE.LIB, COMPAC.LIB, and SMALLL.LIB. If your program corresponds to the LARGE or MEDIUM model of

segmentation, link it to LARGE.LIB. If your program corresponds to the SMALL or COMPACT models of segmentation, link it to SMALLL.LIB or COMPAC.LIB, respectively. After you link your programs to the UDI libraries, you can put the resultant code in RAM or ROM.

The iRMX 86 Programming Techniques manual and the iRMX 86 Configuration Guide discuss selecting a model of segmentation. While these models deal with the PL/M 86 language, they apply to ASM86 and Pascal-86/88 as well. In contrast, Fortran-86/88 always requires the LARGE library.

Refer to the iAPX 86,88 Family Utilities User's Guide for more details concerning linking to libraries.

Figure 6-1 shows how the iRMX 86 Operating System fits the model of Intel solutions. Note that both the 8087 and the emulator are included in this model. However, if you are running a multitasking 8087 program on the iRMX 86 Operating System you cannot use the emulator.

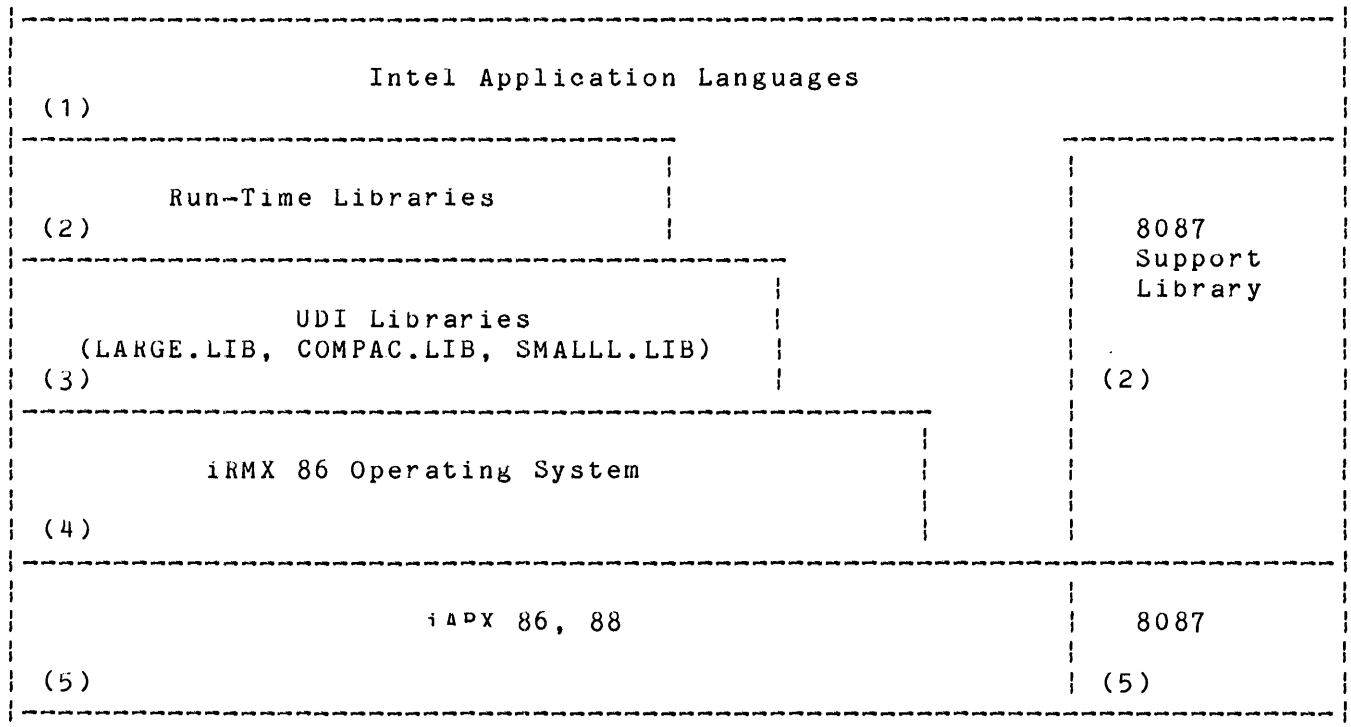


Figure 6-1. Intel's iRMX 86 Solution

Implementation Considerations

The UDI specifications (as documented in appendix A) permit variations in implementation that depend on the style and capabilities of individual operating systems. The iRMX 86 variations are identified in the following sections, along with any deviations from the UDI standard. References to particular procedures are understood to refer to the iRMX 86 versions of these procedures.

DQ$ALLOCATE

The iRMX 86 Operating System allocates bytes in multiples of 16 whereas the UDI standard specifies allocation of the exact number of bytes you request. If you request a number of bytes that is not a multiple of 16, the iRMX 86 Operating System rounds this number to the next higher multiple of 16. DQGETSIZE returns the number of bytes actually allocated.

DQ$ATTACH

Attaching a file that is already connected is valid. A connection to the existing file is made, and all prior connections remain established.

Attaching the logical file :WORK: is valid for reading only. The first read operation returns the end-of-file exception code.

DQ$DECODE$EXCEPTION

The DQ$DECODE$EXCEPTION call returns iRMX 86 messages for the exception codes in a format specified by the Human Interface rather than returning UDI exception codes. Refer to the iRMX 86 Human Interface Reference Manual for information about iRMX 86 messages.

DQ$DELETE

When this procedure is called, the file associated with the specified path name is marked for deletion. If connections to the specified file exist, they remain valid until detached by calling either DQ$DETACH or DQ$EXIT. Refer to the iRMX 86 Extended I/O System Reference Manual for more information concerning connections.

DQGETCONNECTION$STATUS

DQGETCONNECTION$STATUS returns either a value of 0 (zero) or a value of 3 for the seek mode. This is unlike the UDI standard, which allows an operating system to support (independently) either a forward or a backward seek.

DQGETTIME

This UDI call returns the date and time. On the iRMX 86 Operating System, the date and time are set when you enter the DATE and TIME commands (iRMX 86 Human Interface Reference Manual) or invoke the RQSETTIME system call (iRMX 86 Basic I/O System Reference Manual).

DQ$RENAME

Renaming a file to which a connection has been established is valid. The connection to the renamed file remains established.

Exception Codes

Every UDI call returns an exception code that specifies the status of the call. If the operating system returns an exception code of zero, it did not find any errors when it processed the call. However, if the operating system returns a non-zero exception code, it did find errors. Table 6-1 lists all the exception codes and the mnemonics that the iRMX 86 Operating System can for its UDI calls.

Table 6-1. iRMX 86 Exception Codes and Mnemonics

HEX CODE	MNEMONIC	HEX CODE	MNEMONIC
0000	E$OK	0063	EBADSEGMENT
0001	E$TIME	0064	E$CHECKSUM
0002	E$MEM	0065	E$EOF
0003	E$BUSY	0066	E$FIXUP
0004	E$LIMIT	0067	ENOLOADER$MEM
0005	E$CONTEXT	0068	ENOMEM
0006	E$EXIST	0069	ERECFORMAT
0007	E$STATE	006A	ERECLENGTH
0008	ENOTCONFIGURED	006B	ERECTYPE
0009	E$INTERRUPT$SATURATION	006C	ENOSTART
000A	E$INTERRUPT$OVERFLOW	006D	EJOBSIZE
0020	E$FEXIST	006E	E$OVLY
0021	E$FNEXIST	006F	E$LOADER$SUPPORT
0022	E$DEVFD	0080	E$LITERAL
0023	E$SUPPORT	0081	E$STRING$BUFFER
0024	E$EMPTY$ENTRY	0082	E$SEPARATOR
0025	EDIREND	0083	E$CONTINUED
0026	E$FACCESS	0085	E$LIST
0027	EFTYPE	0087	E$PREPOSITION
0028	E$SHARE	0089	E$CONTROL$C
0029	E$SPACE	008B	E$EXTRA$SO
002A	E$IDDR	8000	E$ZERO$DIVIDE
002B	E$IO	8001	E$OVERFLOW
002C	E$FLUSHING	8002	E$TYPE
002D	E$ILLVOL	8003	EBBOUNDS
002E	EDEVOFFLINE	8004	E$PARAM
0040	ELOGNAME$SYNTAX	8005	EBADCALL
0041	E$CANNOT$CLOSE	8006	EEARRAY$BOUNDS
0042	EIOMEMORY	8007	ENDPERROR
0044	E$MEDIA	8017	E$CHECK$EXCEPTION
0045	ELOGNAME$NEXIST	8020	E$IFDR
0046	ENOTOWNER	8021	ENOUSER
0047	EIOJOB	8022	ENOPREFIX
0050	EIOCLASS	8040	ENOTPREFIX
0051	EIOSOFT	8041	ENOTDEVICE
0052	EIOHARD	8042	ENOTCONNECTION
0053	EIOPRINT	8060	EJOBPARAM
0054	EIOWPROT	8080	E$PARSE$TABLES
0060	EABSADDRESS	8083	E$DEFAULT$SO
0061	EBADGROUP	8084	E$STRING
0062	EBADHEADER		

For more information about these exception calls, refer to the iRMX 88 Human Interface Reference Manual.

INTERRUPT HANDLING

Programs that run under the iRMX 86 Operating System should use iRMX 86 interrupt management techniques to handle interrupts. The UDI libraries do not include interrupt management. If you wish to use interrupts for application specific functions rather than iRMX 86-provided device drivers and timers, you must write interrupt handlers and possibly interrupt tasks. To handle interrupts, you should use iRMX 86 system calls to process interrupts and set up interrupt levels; you should not use direct programming language statements to enable and disable interrupt levels.

The iRMX 86 Operating System reserves certain interrupts for special purposes. Interrupts 56-63 are reserved for external interrupts using the 8259A master levels; interrupts 64-127 are also reserved for external interrupts using the 8259A slave levels. Refer to the iRMX 86 Nucleus Reference Manual for more information concerning interrupt management techniques.

LOGICAL NAMES

The UDI uses certain logical names to mean special things. For example, :LP: means "line printer", :CO: means "console output", and :CI: means "console input." When you configure your operating system, be sure to assign these reserved logical names to the correct devices. Failure to do this will cause your UDI to differ from the UDI standard. Refer to the iRMX Operator's Manual for a more complete explanation of logical names. Refer to the iRMX 86 Configuration Guide for more information concerning configuring the iRMX 86 Operating System.

REENTRANCY

UDI libraries are fully reentrant.

MULTITASKING

The UDI libraries are fully compatible with a multitasking environment. However, there are no UDI calls to create and delete tasks. While the iRMX 86 Operating System allows you to divide your application programs into tasks, doing so takes you outside the scope of the UDI. However, if you want to take advantage of iRMX 86 multitasking under UDI you must use RQ$CREATE$TASK and related iRMX 86 system calls to manipulate tasks. These system calls are fully described in the iRMX 86 Nucleus Reference Manual).

USING OVERLAYS IN AN iRMX 86 ENVIRONMENT

If your assembly language or PL/M-86 programs use overlays and use UDI calls to load the overlays (the DQ$OVERLAY procedure), you should take care to ensure that you link the UDI library to your program correctly. The iAPX 86, 88 Family Utilities User's Guide contains an example of linking an overlay program. This example lists a two-step link process, as follows:

1. Link the root and each of the overlays separately, specifying the

OVERLAY control, but not the BIND control, in each LINK86 command.

2. Link all the output modules together in one module, specifying the BIND control, but not the OVERLAY control.

This is the same process that you should use when linking your iRMX 86 overlay programs. However, you must ensure that you link the entire UDI library to the root portion of the program and not to any of the overlays. To do this, use the INCLUDE control to include the UDI externals file (UDI.EXT) with the PL/M-86 compilation of the root portion of the program. By including this file with the root, you make external references to all UDI routines from that root. Then when you link the root to the UDI library, LINK86 pulls in all of the UDI routines, not just the ones called in the root. Since you are linking the UDI library to the root only, this prevents you from having unsatisfied externals when you link the root to the overlays.

For example, suppose your program consists of three files, ROOT.OBJ, OV1A.OBJ, and OV2A.OBJ, the root and overlay files, respectively. You have compiled these program modules with the PL/M-86 compiler and included the UDI externals file UDI.EXT with the compilation of the root. Assuming that LINK86 resides on the default logical device in directory SYSTEM and that the object files reside in :F1:PROG, the following LINK86 commands will link the overlay program and produce an executable module. This happens in two steps.

1. The first three LINK86 commands separately link the root and overlay portions of the program. The root portion of the program is linked to the UDI library.

```
-LINK86 :F1:PROG/ROOT.OBJ,      &
**:F1:PROG/LARGE.LIB OVERLAY


iRMX 86 8086 LINKER Vx.y


-LINK86 :F1:PROG/OV1A.OBJ OVERLAY(OVERLAY1)


iRMX 86 8086 LINKER Vx.y


-LINK86 :F1:PROG/OV2A.OBJ OVERLAY(OVERLAY2)


iRMX 86 8086 LINKER Vx.y
```

2. The next LINK86 command links together in one module all the output modules produced in the first step.

```
-LINK86 :F1:PROG/ROOT.LNK,       &
**:F1:PROG/OV1A.LNK,       &
**:F1:PROG/OV2A.LNK         &
**TO :F1:PROGRAM1  BIND  MEMPOOL(+2000H)
```

EXAMPLE PROGRAM

The following example illustrates the process of using an iRMX 86-based system to compile, link (both to the UDI and to the run-time libraries), loading, and execute a simple Pascal-86/88 program (shown in figure 6-2).

Although this program makes no direct UDI calls, the code generated by the Pascal-86/88 translator does call UDI procedures for I/O to the console. If you compile and link this program as shown in the example, you can run it on the iRMX 86 Operating System.

```
(* Convert a number of inches into yards, feet, and inches *)

PROGRAM inch(input,output);
VAR yards, feet, f_inch, number : integer;
    quitchar : char;
PROCEDURE convert (ins : integer ; VAR y, f, i : integer);

    BEGIN
        y := ins DIV 36;
        ins := ins MOD 36;
        f := ins DIV 12;
        i := ins MOD 12;
    END;

BEGIN
    REPEAT
        writeln; writeln;
        write('Number of inches is: ');
        readln(number);
        writeln;
        convert(number, yards, feet, f_inch);
        writeln(yards:4, ' yards, ',
                feet:1, ' feet, and ',
                f_inch:2, ' inches');
        writeln; writeln;
        write('Another number--y or n? :');
        read(quitchar);
    UNTIL NOT (quitchar in ['Y','y'])
END.
```

Figure 6-2. iRMX 86 Example Program

Compiling

The following line invokes the iRMX 86 Pascal-86/88 compiler on an iRMX 86 system. This example assumes the compiler is in the system and that the program is to be compiled on the default device.

```
-PASC86 INCHES.SRC
```

The compiler places the object module in file INCHES.OBJ and produces a listing file called INCHES.LST. If the compiler finds no errors, it responds as follows:

```
iRMX 86 Pascal-86/88, V1.0
PARSE(0), ANALYSE(0), NOXREF, OBJECT

    COMPILATION OF INCHES COMPLETED, 0 ERRORS DETECTED,
    END OF Pascal-86/88 COMPILATION.
```

Linking

If there are no errors, you are ready to link the compiled program to the necessary run-time support libraries and the large model UDI library as follows.

```
-:F1:LINK86 INCHES.OBJ, &
**P86RN0.LIB, &
**P86RN1.LIB, &
**P86RN2.LIB, &
**P86RN3.LIB, &
**NULL87.LIB, &
**LARGE.LIB &
**TO :F1:INCHES BIND MEMPOOL(+2000H)
```

The BIND control directs LINK86 to produce :F1:INCHES in a load time locatable format. The MEMPOOL (+2000H) control dynamically allocates 2000H of memory for the program and connections. Refer to the iAPX 86,88 Family Utilities User's Guide for more information concerning the BIND and MEMPOOL controls.

The system responds as follows:

```
iRMX 86 8086 LINKER, Vx.y
```

LINK 86 produces a map file (INCHES.MP1) and a loadable module (INCHES).

Invoking.

You can now invoke the program. Type

```
-:f1:inches
```

The program responds with

```
Number of inches is :
```

Assume you enter "38". The program displays the number in yards, feet, and inches.

```
1 yard, 0 feet, and 2 inches
```

The program then asks if you want to enter another number.

```
Another number--y or n?:
```

Answer with N if you wish to exit the program.

Refer to the iAPX 86, 88 Family Utilities User's Guide for further information concerning compiling, linking and executing application programs you want to run on the iRMX 86 Operating System.

CHAPTER 7
iRMX™ 88 RUN-TIME SUPPORT

The iRMX 88 Real-Time Multitasking Executive is a small, high-performance operating system that provides functions needed in software that monitors and/or controls external events occurring asynchronously in real time.

UDI FOR THE iRMX 88 OPERATING SYSTEM

The iRMX 88 Operating System offers a subset of UDI adequate to support the application language run-time libraries. This implementation of UDI is comprised of the following UDI calls:

DQ$ALLOCATE	DQGETEXCEPTION$HANDLER
DQ$ATTACH	DQGETSIZE
DQ$CLOSE	DQ$OPEN
DQ$CREATE	DQ$READ
DQ$DECODE$EXCEPTION	DQ$RENAME
DQ$DELETE	DQ$SEEK
DQ$DETACH	DQ$SPECIAL
DQ$EXIT	DQ$TRAP$EXCEPTION
DQ$FREE	DQ$TRUNCATE
DQGETARGUMENT	DQ$WRITE
DQGETCONNECTION$STATUS	

The remaining UDI calls are not available in the iRMX 88 Executive. They are:

DQ$CHANGE$ACCESS	DQ$DECODE$TIME	DQ$FILE$INFO
DQ$CHANGE$EXTENSION	DQ$OVERLAY	DQ$RESERVE$IO$MEMORY
DQGETSYSTEM$ID	DQ$SWITCH$BUFFER	DQ$TRAP$CC
DQGETTIME		

The iRMX 88 subsystems required to support UDI are:

- The Nucleus, which coordinates all the many concurrent activities in an executing iRMX 88-based application system
- The Free Space Manager, which maintains a pool of free RAM, allocating blocks to tasks on request and reclaiming blocks returned by tasks
- The I/O System, which provides file management capabilities, allowing tasks to perform I/O to and from secondary storage devices and other devices such as operator terminals, without disrupting normal processing

Figure 7-1 shows how the iRMX 88 Executive fits the model of Intel solutions. Note that levels (3) and (4) are shown as being a single, integrated level. This is because the iRMX 88 Executive contains UDI calls, rather than being a separate entity that is called by UDI.

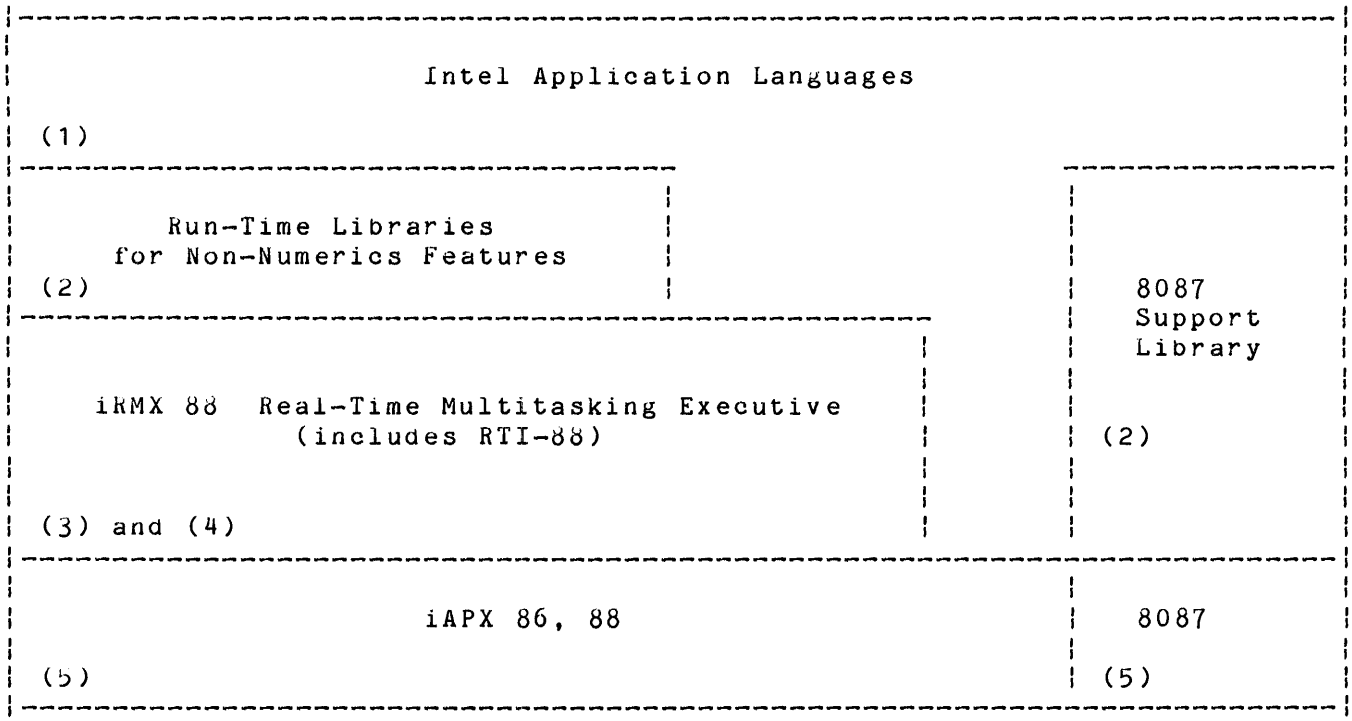


Figure 7-1. Intel's iRMX 88 Solution

Implementation Considerations

The UDI specifications (as documented in Appendix A) permit variations in implementation that depend on the style and capablities of individual operating systems. The variations from UDI as defined in Appendix A are identified in the following sections. References to particular procedures ar understood to refer to the iRMX 88 implementation of these procedures.

DQ$ATTACH

Attaching a file that is already connected is valid. A connection to the existing file is made, and all prior connections remain established.

An attempt to attach the file :WORK: returns the exception code E$SUPPORT.

DQ$DECODE$EXCEPTION

The DQ$DECODE$EXCEPTION call returns a string of zero length (one byte containing 0), rather than returning a string with a condition code and other information.

DQ$DELETE

When this procedure is called, the file associated with the specified path

name is marked for deletion. If connections to the specified file exist, they remain valid until detached by calling either DQ$DETACH or DQ$EXIT.

DQGETARGUMENT

The DQGETARGUMENT call always places a string of zero length (one byte containing 0) at the location pointed to by the ARGUMENT$P parameter, and it always returns the value 0DH (ASCII carriage return).

DQGETTIME

System time is not maintained by the iRMX 88 Real-Time Multitasking Executive.

DQ$RENAME

Renaming a file to which a connection has been established is valid. Any connection to the renamed file remains established.

DQ$TRAP$EXCEPTION

In iRMX 88-based systems, there is a single, system-wide exception handler; therefore, when a task calls DQ$TRAP$EXCEPTION, the exception handler changes for that task and for all other tasks in the system as well.

Exception Codes

Each of the RTI-88 procedures returns an exception code when called. Table 7-1 lists the exception codes that these procedures can return.

Table 7-1. iRMX 88 Exception Codes and Mnemonics

HEX CODE	MNEMONIC
0000	E$OK
0002	E$MEM
0005	E$CONTEXT
0006	E$EXIST
0008	ENOTCONFIGURED
0020	E$FEXIST
0021	E$FNEXIST
0023	E$SUPPORT
0029	E$SPACE
002B	E$IO
002C	E$FLUSHING
0044	E$MEDIA
8004	E$PARAM
8020	E$IFDR

For information about the handling of exceptional conditions (those other than E$OK), and for information about which exceptional conditions are returned by each of the RTI-88 calls, see the iRMX 88 Reference Manual.

Libraries, Compiling, and Linking

The process of configuring an iRMX 88 application (during which the requirements of the application are defined) occurs at the console of an Intellec Development System where you carry on a dialogue with a special software module called the Interactive Configuration Utility (ICU). The ICU produces a SUBMIT file containing the linking commands needed to produce a working application system. Before submitting the file, you must make available all of the task modules, which must have been compiled under either the COMPACT or LARGE model of segmentation.

INTERRUPT HANDLING

In iRMX 88-based systems, interrupts are handled in either of two ways, depending upon how the programmer wanted that particular level of interrupt to be serviced.

One way is to have an interrupt task wait at a special interrupt exchange. (An exchange is a place where tasks wait for messages that are sent either by other tasks or by the Nucleus.) Each time an interrupt arrives at that level, the Nucleus sends a message to the exchange. The task receives the message, services the interrupt, and then waits again at the exchange.

The other way is for a special interrupt service routine to receive immediate control whenever an interrupt arrives. This routine can service the interrupt itself, or it can invoke an interrupt task that services the task and then returns control to the interrupt service routine. The interrupt service routine finally signals to the Nucleus that it is finished and relinquishes control.

For more information about interrupt handling in the iRMX 88 environment, see the iRMX 88 Reference Manual.

CHAPTER 8
RUN-TIME CONSIDERATIONS FOR NON-INTEL OPERATING SYSTEMS

Even though the target operating system for your application is not one supplied by Intel, you can still take advantage of Intel's application languages and application-language run-time support. By implementing your own Universal Development Interface (UDI), you can translate from the environment assumed by Intel's application-language run-time support libraries to the actual environment provided by your operating system. Figure 8-1 illustrates the role of a user-written UDI in interfacing to a foreign operating system.

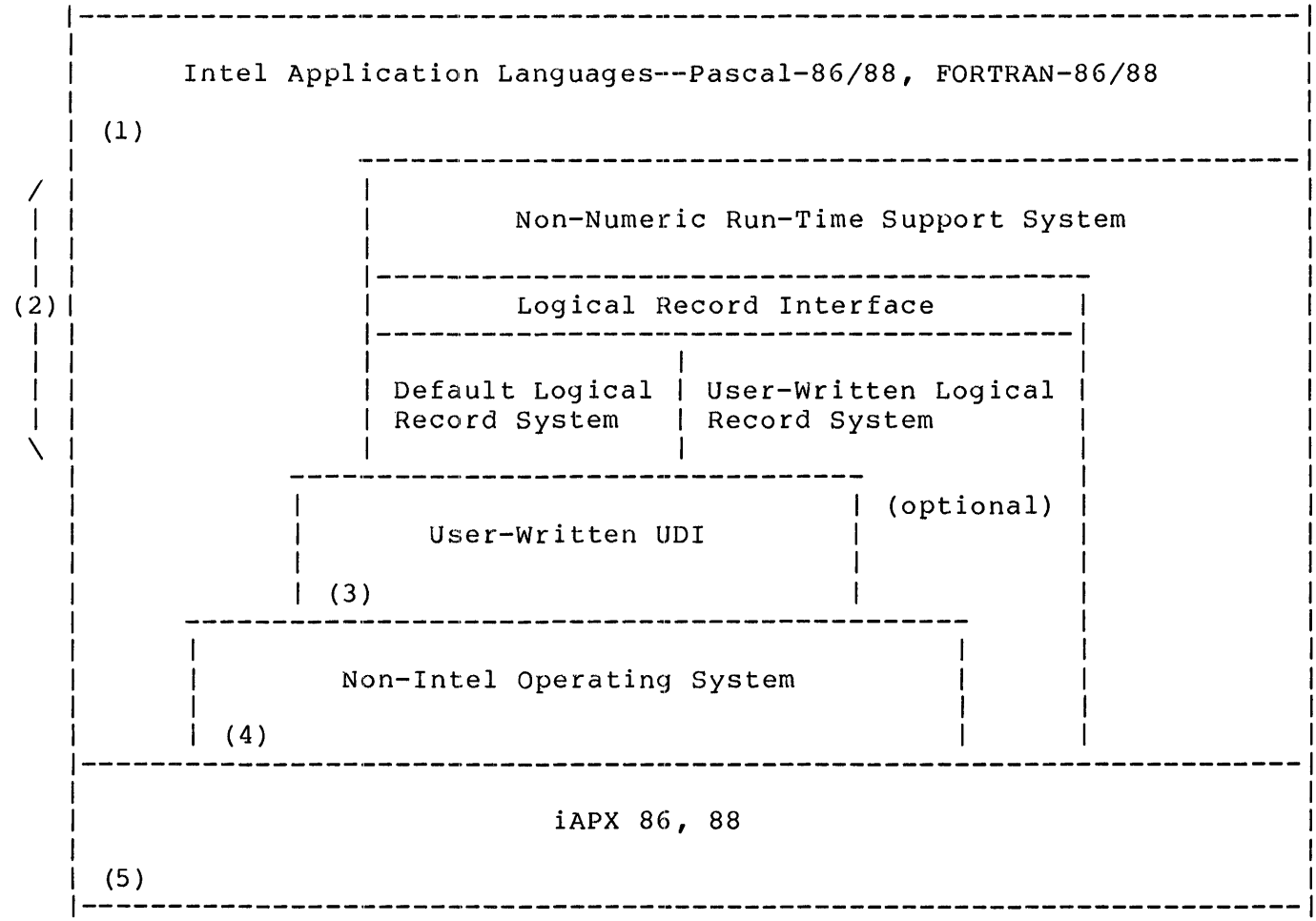


Figure 8-1. Interfacing to Non-Intel Operating System

UDI PROCEDURES USED BY RUN-TIME LIBRARIES

To implement your UDI, follow the specifications in Appendix A. The run-time system does not use all of the procedure interfaces defined for UDI in Appendix A. The UDI procedures actually used by the run-time system include:

DQ$ALLOCATE
DQ$ATTACH
DQ$CLOSE
DQ$CREATE
DQ$DELETE
DQ$DETACH
DQ$EXIT
DQ$FREE
DQGETARGUMENT
DQGETCONNECTION$STATUS
DQGETEXCEPTION$HANDLER
DQGETSIZE
DQ$OPEN
DQ$READ
DQ$SEEK
DQ$SPECIAL (type 1 only)
DQ$TRAP$EXCEPTION
DQ$TRUNCATE
DQ$WRITE

This list is subject to change; however, LINK86 will flag as UNRESOLVED EXTERNALS any UDI routines that are used by the run-time system but are not implemented.

Note that the 8087 Support Library does not depend on operating system support and therefore can be transported to any operating environment.

IMPLEMENTING A UNIVERSAL DEVELOPMENT INTERFACE

An implementation of UDI can take one of two forms:

1. A library of modules that embody all the interface logic and are linked to applications programs. This approach may be feasible when the functions performed by your operating system correspond closely to those of UDI.

2. A stand-alone program that acts as a sub-operating-system, running under control of your operating system. This approach gives more control to your UDI. It may be necessary to use this approach when the dissimilarity between the interfaces of your operating system and those of UDI force you to perform operating-system-like functions in your UDI implementation. You will still need to have a library of interface routines that transform a UDI procedure call in your application modules into an interrupt that invokes your sub-operating-system.

Following is a checklist to help you evaluate how much logic you will need to write in order to implement a UDI for your operating system. Ask yourself these questions, substituting the name of your operating system in place of MYOS:

- Command Language

    1. Does MYOS make the command tail available for parsing?

2. Is the syntax of UDI's predefined path names (for example, :CI:, :CO:, :WORK:) compatible with the command language of MYOS?

3. Does MYOS preprocess the command tail in ways that are incompatible with UDI? (For example, does it translate lower-case characters into upper-case even when enclosed in quotation marks?) If so, are these differences significant in my application?

- File management

  1. Is disk addressing in MYOS character-oriented as in UDI, or must I translate character addresses into some other form (such as sector addresses)?

  2. Does MYOS implement file connection (CREATE, ATTACH, DETACH) separately from file OPEN and CLOSE operations, or is translation required?

  3. Can attached files be described with a 2-byte value (a UDI CONNECTION), or is a longer value needed?

  4. Does MYOS handle work files as defined for UDI, or do I have to keep track of work files and delete them on termination?

  5. Does MYOS limit the size of disk files to less than the size permitted by UDI?

  6. Does MYOS support a file truncation operation as in DQ$TRUNCATE, or do I have to simulate truncation?

  7. Does MYOS handle console I/O as defined for UDI, with both line-edited and transparent modes; or do I have to write the logic to do this?

- Memory management

  1. Is memory management in MYOS adequate to support UDI, or do I need to include a memory manager in my implementation of UDI?

CHAPTER 9
CONFIGURING THE RUN-TIME SYSTEM

One of the purposes of the run-time system is to make it possible for you to eliminate costly features that are unnecessary in your application and to provide alternative implementations of certain features. Appendix B discusses alternatives to the Intel-supplied Logical Record System (LRS). This chapter discusses tailoring of various other features of the run-time system.

ESTABLISHING AN ALTERNATE EXCEPTION HANDLER

The Intel-supplied run-time system provides a UDI standard exception handler. This exception handler displays a message and terminates the job (except for 8087 exceptions, which are discussed later). You can write your own exception handler to take whatever action is appropriate for your application. Appendix A presents the UDI standard interface for exception handlers. The run-time system establishes its exception handler as the current exception handler in the procedure TQ$ESTART. TQ$ESTART is called during initialization.

You can establish your exception handler as the current exception handler by writing your own version of TQ$ESTART and linking it in place of the default TQ$ESTART. TQ$ESTART is a FAR procedure and has no parameters. Your version should call the run-time procedure TQSETERH as follows:

```
TQ$SET$ERH:  PROCEDURE (handler$ptr) EXTERNAL;
    DECLARE handler$ptr  POINTER;
    END;

CALL TQ$SET$ERH (@MYHANDLER);
```

Link your version of TQ$ESTART before the run-time libraries, so that the linker uses your version instead of the version in the run-time libraries.

The default exception handler calls the 8087 Support Library procedure FILTER (using the alias TQ_312) to handle 8087 exceptions. You may wish to do the same in your exception handler. Refer to the 8087 Support Library Reference Manual for details about FILTER. The interface to FILTER is defined as follows:

```
FILTER:  PROCEDURE (errors87) BYTE EXTERNAL;
    DECLARE errors87 WORD;
    END;
```

The parameter errors87 is the 8087 status word that is passed to your exception handler.

The BYTE returned by FILTER indicates whether FILTER was able to handle the exception. FILTER retries the instruction according to various options of the proposed IEEE floating-point standard. A value of zero means that FILTER could not handle the exception; your exception handler should then take appropriate recovery action. A value of 0FFH means that FILTER has recovered from the exception; your exception handler may return to your application

without further action, provided it can restore the state of the iAPX 86,88 Processor.

FILTER pops the parameter errors87 from the 8Ø86 stack before returning.

ELIMINATING PRECONNECTION PARSING

Preconnection parameters are used in the command line that invokes a FORTRAN-86/88 or Pascal-86/88 program to associate external path names with internal program objects (the program-parameter-list of the PROGRAM statement in Pascal-86/88; unit numbers in FORTRAN-86/88). The run-time system calls its procedure TQ$PARSECL during initialization to parse any preconnection parameters that may have been entered in the command line.

If your application does not use preconnection, you may wish to eliminate the preconnection code. You can do so by supplying a version of TQ$PARSECL such as this:

```
DECLARE TQ$DEFAULTPL POINTER EXTERNAL;

TQ$PARSECL:  PROCEDURE (LIST$ADDR$PTR) PUBLIC;
    DECLARE LIST$ADDR$PTR POINTER,
    DECLARE LIST$ADDR BASED LIST$ADDR$PTR POINTER;

    LIST$ADDR = @TQ$DEFAULT$PL;

    END;
```

TQ$DEFAULTPL is a list, supplied by the run-time system, of the standard preconnections.

Link your version of TQ$PARSECL before the run-time libraries, so that the linker uses your version instead of the default version.

CHANGING DEFAULT PRECONNECTIONS

The run-time system provides a default preconnection list with the PUBLIC name TQ$DEFAULTPL. TQ$DEFAULTPL is a linked list each entry of which has the following format:

For the last entry in the linked list, the LOCATION OF NEXT ENTRY is zero, or else the last entry is a null entry indicated by a STRING LENGTH of zero. Each (non-null) STRING contains one preconnection assignment in the same form as it would appear in a program invocation line. The strings contained in TQ$DEFAULTPL are the standard preconnections defined by FORTRAN-86/88 and Pascal-86/88:

```
UNIT5=:CI:
UNIT6=:CO:
INPUT=:CI:
OUTPUT=:CO:
```

To supply different preconnections, code a data structure that has the public name TQ$DEFAULTPL and contains the preconnections needed by your application. If you need no preconnections, code your list with a single null entry. Link your version of TQ$DEFAULTPL before the run-time libraries.

APPENDIX A
UDI SPECIFICATIONS

INTRODUCTION

The Universal Development Interface (UDI) is an interface to the operating system that provides the user a standard set of support procedures. These procedures are supported on all of Intel's 86/88 family operating systems.

The purpose of UDI is two-fold. First, it presents a consistent model of operating system support by providing a set of procedures for the programmer to write applications to run on the different operating systems supporting UDI. Secondly, application programs become portable in that operating system dependencies have not been programmed into the application code.

It is the goal of UDI to achieve object level portability as software is moved across the various operating systems using the same OMFs for a particular CPU. The object level programs normally can be ported to another operating system by simply linking in the proper UDI support for that operating system. When software is ported to a new CPU, source level portability with minimal code changes can be due to a change in op-codes, object module formats, and the technology level of the CPU.

OVERVIEW

The operating system procedures provided by UDI can be divided into five general categories. These are utility, memory management, file management, program control, and exception handling. In this section the general background and definitions needed in order to discuss the basic model of each of these areas is given. The dependencies for each Intel iAPX-86, 88 operating system are discussed in chapters 5, 6, & 7.

Utility Procedures

Utility procedures provide for date and time stamping, system identification, and the ability to pass parameters to a program. Parameters are passed to a program by parsing the command line, that is, the text line that invokes the program, for example,

```
PASC86 PROG.SRC NOLIST<cr>
```

The first argument in the text (in this case PASC86) is the name of the program file to be loaded for execution. Some operating systems accept additional syntax before the program-file argument; for example, the Debugger in a Series III environment might be invoked as follows:

```
DEBUG PASC86 PROG.SRC NOLIST<cr>
```

The portion of the command line after any such preliminary syntax (i.e., beginning with the name of the program file) is called the command tail. In the preceding DEBUG example, the command tail is simply

PASC86 PROG.SRC NOLIST< cr>

Since the arguments are passed as they are parsed, the program must save any of them that will be reused later. This general line parsing capability is available to the programmer for parsing other input.

Memory Management

When a program is loaded, it is allocated a specific amount of memory for its code and data. The portion of memory not occupied by loaded code and data is available through procedures in this section for dynamic use either by the program or by the operating system. There is no guarantee that the unallocated memory is contiguous.

File Management

To the user UDI makes no major distinction between logical secondary storage files and physical devices other than the naming conventions. Thus we will use the term file to indicate a logical secondary storage file or a UDI recognized device.

Those files and their corresponding predefined names a user can assume to be supported are

Console Input	:CI:
Console Output	:CO:
Line Printer	:LP:
Byte Bucket	:BB:
Work Files	:WORK:

and optionally

Serial Input	:TI:
Serial Output	:TO:

UDI distinguishes between

1. Establishing an association between a program and a file.
2. Operating on the file.

The association between a program and a file is known as a connection.

A connection is established by DQ$ATTACH or DQ$CREATE and severed by DQ$DETACH. When your program establishes a connection via DQ$ATTACH or DQ$CREATE, it receives a CONNECTION token of the type WORD from the operating system. This connection is then used in all further communications with the operating system that have to do with the associated file. This distinction allows for the time-consuming operations involving directory searches to be separated from the resource operation of memory allocation.

The procedure DQ$OPEN prepares an already established connection for subsequent input/output operations. Input and output are performed by DQ$READ and DQ$WRITE. When reading from the console input device you may select the mode of input using DQ$SPECIAL. Line edited and a non-edited transparent mode are supported. DQ$SEEK is used to change position in the file without transferring data. DQ$CLOSE is used to close a file when input/output is finished. Closing a file frees buffer space.

Compared to the processing involved in DQ$ATTACH, DQ$CREATE, and DQ$DETACH the amount of processing done by DQ$OPEN and DQ$CLOSE is relatively minimal. Once a connection is established, it may be opened and closed as often as necessary.

Memory may be reserved for system I/O space by calling DQ$RESERVE$IO$MEMORY and declaring the maximum number of buffers and files to be attached at any one time. Use of this procedure assures the user that when memory is dynamically allocated sufficient memory will remain to allocate I/O buffers and thus in turn to open the desired number of files.

The user may assume that by calling DQ$RESERVE$IO$MEMORY sufficient I/O space will be reserved to attach at least twelve files and open at least six files.

No assumption is made as to the type of directory structure. Path names are character strings (normally with special delimiters) that are used to identify files. The rules for forming path names are operating system dependent. To ensure that your programs remain operating system independent,

- Never examine file names.
- Modify file name strings only by calling the UDI procedure DQ$CHANGE$EXTENSION.
- Use only

  1. Path names supplied by the user
  2. Path names created by calling DQ$CHANGE$EXTENSION
  3. Predefined file names

Anything written to the byte bucket :BB: disappears immediately. A read from it returns an end of file condition.

The work file :WORK: may be created any number of times (within the limit on number of connections and opens), each time creating a new work file. Each work file is temporary; it is automatically deleted when the file is detached.

UDI-conforming systems need not include a serial port; but, if they do, they must recognize the device reserved names representing serial input :TI: and serial output :TO:.

In order that input and output from devices may be redirected to secondary storage, input devices are always attached while output devices are created. Thus, Console Input and Serial Input, if supported, may only be connected with DQ$ATTACH. Console Output and Serial Output, if supported, may only be connected with DQ$CREATE.

A DQ$OPEN requesting zero buffers is honored, making interactive communication feasible.

Program Control

DQ$EXIT terminates a program. All files are closed and those resources allocated by the program are freed.

DQ$OVERLAY provides the ability to invoke overlays that have been pre-structured by the system linker. Only one layer of overlays is allowed and the overlay procedure may be called only from the root of the program.

Exception Handling

UDI standardizes the handling of exceptional conditions by system software. Applications may follow the same standard for error conditions that they detect. The standard includes

- A format for encoding exceptional conditions
- A classification of condition codes
- An interface that permits you to establish a procedure for handling certain exceptional conditions
- An interface that permits any executing software to locate the current exception handler
- An interface for reporting exception conditions to the current exception handler task

Detected exceptions are classified as either environmental conditions or programmer errors.

Environment conditions are generally caused by exceptional conditions outside the control of a program; for example, "file not found" or "insufficient memory". Every UDI procedure (except DQ$EXIT) has an argument, called excep$p, that points to a WORD in which the operating system returns an exception code. If the returned exception code is nonzero, it identifies the environmental exception. This type of exception does not pass control to the current exception handler, thus after every UDI call the program should check the exception word to determine whether an environmental exceptional condition prevented successful completion of the procedure.

The list of possible exceptional conditions that follows this section is not exhaustive. It is intentionally restricted to those specific errors that a programmer may need to test against. The operating system is free to return additional exceptional conditions. A programmer can detect those as only being non-zero.

Programmer errors are typically caused by coding errors (for example, "bad parameter"), but "divide-by-zero", "overflow", "range check", "special bounds check", and errors detected by the Numerics Data Coprocessor are also referred to as programmer errors.

When a programmable error occurs, the operating system passes control to the current exception handler. Each operating system supplies a default error handler. The action of this handler is to display an exception message and terminate the program. However, you may establish your own procedure to handle programmer errors by using DQ$TRAP$EXCEPTION.

Your application programs may also report error conditions to the current error handler. The UDI procedure DQGETEXCEPTION$HANDLER is used to fetch the address of the current handler so that your programs can formulate an indirect call.

When a console operator types control-C, a special type of exception occurs. The default action is to cancel the current interactive program.

Exceptional conditions may originate in any part of the system, including the operating system, the UDI procedures, the run-time support system, or the application. In addition, certain processor-generated interrupts are fielded by the operating system and reported to the exception handler. Codes for some of the possible conditions are defined by UDI. These should be used wherever applicable. Other operating system codes may exist, but they are not part of UDI.

PROGRAMMER ERRORS

Symbolic Name	Hex Value	Meaning
E$ZERO$DIVIDE	8000	Integer divide by zero.
E$OVERFLOW	8001	Integer overflow.
E$ARRAY$BOUNDS	8006	Bounds check violation.
E$NDP	8007	Exception detected by the Numerics Data Co-Processor.
E$SPECIAL$BOUNDS$CHECK	8017	Case range, procedure. Stack overflow.
(Run-Time Support)	9100 to 95FF	(Reserved)
(Other)		Other exception codes are system dependent and not defined by UDI.

ENVIRONMENTAL CONDITIONS

Symbolic Name	HEX VALUE	MEANING
E$OK	0	The operation completed normally.
E$MEM	2	Insufficient memory for requested operation.
E$FEXIST	20	The named file exists.
E$FNEXIST	21	The file does not exist.
E$SUPPORT	23	Device or system does not support requested operation.
E$FACCESS	26	Access to a file is denied.
E$SHARE	28	File may not be shared.
E$SPACE	29	Insufficient space on direct-access device.
E$STRING$BUF	81	The string is too long.
(Run-Time Support)	1100 to 15FF	(Reserved)
(Other)		Other exception codes are system dependent and not defined by UDI.

An exception handler must be a FAR procedure that conforms to the following interface specification:

```
exception$handler:
     PROCEDURE (excep$code, param$num, reserved, NDP$status)
               PUBLIC;
     DECLARE excep$code  WORD,
             param$num   WORD,
             reserved    WORD,
             NDP$status  WORD;
     END;
```

excep$code is a code that indicates what exception has occurred.

param$num is the number of the erroneous parameter in the called procedure (1 for the first parameter, 2 for the second, etc.). If the condition was not caused by an erroneous parameter, param$num is zero.

reserved is reserved for future use.

If excep$code is E$NDP, NDP$status contains the value of the numerics data processor status word. The numerics data co-processor exception will have been cleared when the handler gets control.

An exception handler may exit either by jumping to a label in the main module or by executing a long RETURN. If the exception handler RETURNs, it must preserve the registers and flags of the interrupted procedure and thus should be coded in assembly language.

DATA TYPES

The following data types are used in the specification:

BYTE	--	An eight-bit item.
BOOLEAN	--	A BYTE taking on the values TRUE (OFFH) and FALSE (OH).
STRING	--	A sequence of bytes, the first of which contains the length (in bytes) of the remaining portion of the string. A length of zero indicates a null string.
WORD	--	A two-BYTE item (16 bits).
DWORD	--	A four-BYTE item (32 bits).
CONNECTION	--	A two-byte identifier of an attachment between a program and a file.
POINTER	--	The address of a storage location. Two bytes under the SMALL model of segmentation; four bytes in other models. A four-byte pointer consists of a selector and a two-byte offset from the selector.
SELECTOR	--	The base portion of a four-byte POINTER.

GENERAL ASSUMPTIONS

Multitasking

The UDI libraries are fully compatible with a multitasking environment. However, UDI does not define procedures to create, delete, or synchronize tasks.

Coprocessor Support

The use of coprocessors such as numerics or I/O coprocessors are not excluded from being used in a UDI application. If the operating system allows for concurrency of multiple programs then the operating system must preserve and restore the state of the coprocessors as appropriate.

Format of Primitives

In the following specifications, the formal definition of the interface to each procedure is presented in PL/M. For the parameters printed in lower case, you may substitute your own identifiers.

The parameter excep$p is not described with each procedure since it is common to all procedures (except DQ$EXIT). It points to a WORD item in your program in which the operating system places the exception code.

If you are programming in assembly language, refer to the example calling sequences at the end of each specification for the order in which to push parameters onto the stack. The examples shown are directly applicable to programs (or subsystems) compiled according to the COMPACT and LARGE models of segmentation. For the SMALL model, do not push the segment register before pushing the register containing the parameter.

DQGETTIME

Description

DQGETTIME returns the current date and time in character format. This procedure should be replaced by the more general procedure DQ$DECODE$TIME that returns both the binary DWORD system time and date and the decoded ASCII strings.

Declaration Syntax

```
DQ$GET$TIME:
     PROCEDURE (dt$p, excep$p) EXTERNAL;
     DECLARE dt$p     POINTER,
             excep$p  POINTER;
     END;
```

Output Parameters

dt$p is a pointer to a structure that you declare in your program. This structure has the form

```
DECLARE DT STRUCTURE
           (DATE (8) BYTE,
            TIME (8) BYTE);
```

DATE has the form MM/DD/YY for month, day, and year. TIME has the form HH:MM:SS for hours, minutes, and seconds. The value for hours ranges from 0 through 23.

Standard Exception Codes

E$OK .

Comments

There is no requirement that any given operating system update either the time or the date value.

Example Calling Sequence

```
CALL DQ$GET$TIME (@DT, @ERR);     PLM
```

DQ$DECODE$TIME

Description

DQ$DECODE$TIME decodes the operating system dependent time and date DWORD into ASCII date and time strings. It may also be used to return the current date and time in either binary DWORD format or as a decoded ASCII string.

Declaration Syntax

```
DQ$DECODE$TIME:
     PROCEDURE (dt$P, excep$p) EXTERNAL;
     DECLARE dt$p     POINTER,
             excep$p  POINTER;
     END;
```

Input Parameters

dt$p is a pointer to a user declared structure of the following form:

```
DELCARE DT STRUCTURE
  (SYSTEM$TIME  DWORD,
   DATE(8)      BYTE,
   TIME(8)      BYTE);
```

system$time is an operating system dependent formatted DWORD containing the time and date. If system$time is zero then the system clock is first read to obtain the current date and time. If system$time is non-zero, it is simply decoded into ASCII date and time strings.

Output Parameters

system$time will contain the binary format of the current date and time if selected by the input value zero. The specified format is in seconds beginning with January 1, 1978.

DATE has the form MM/DD/YY for month, day, and year. TIME has the form HH:MM:SS for hours, minutes, and seconds. The value for hours is in the range from 0 to 23.

Standard Exception Codes

EOK, ESUPPORT

Comments

There is no requirement that a system maintain date and time.

Example Calling Sequence

```
CALL DQ$DECODE$TIME (dt$p, excep$p);      PLM
```

DQGETSYSTEM$ID

Description

DQGETSYSTEM$ID returns a string that identifies the operating system.

Declaration Syntax

```
DQ$GET$SYSTEM$ID:
     PROCEDURE (id$p, excep$p) EXTERNAL;
     DECLARE id$p     POINTER,
             excep$p  POINTER;
     END;
```

Output Parameters

id$p must point to a buffer of at least 21 bytes in length that you define in your program. The length of the remainder of the string (in bytes) is contained in the first byte of the output area. The text returned is the name of the operating system, for example

```
                SERIES-III
...or...
                iRMX 86
```

Standard Exception Codes

E$OK

Comments

Any program that examines the returned string is not operating-system independent.

Example Calling Sequence

```
CALL DQ$GET$SYSTEM$ID (@ID, @ERR);
```

DQGETARGUMENT

Description

DQGETARGUMENT parses either text in the tail of a command line or (with the help of DQ$SWITCH$BUFFER) text you have read into your program. This function is primarily used for parsing arguments to a user's program.

Declaration Syntax

```
DQ$GET$ARGUMENT:
     PROCEDURE (argument$p, excep$p) BYTE EXTERNAL;
     DECLARE argument$p  POINTER,
             excep$p     POINTER;
     END;
```

Output Parameters

argument$p points to an area that you declare in your program to receive an argument string from the command tail. This area must be at least 81 bytes long. The actual length of the output string (in bytes) is stored in the first byte of this area.

This is a typed procedure (a function). The value of the procedure is a BYTE containing the delimiter that terminates the argument. A delimiter is returned only if the exception code is zero. Delimiters include

, ( ) = # ! % ' ~ + - ; & | [ ] < > and DEL

as well as any characters with hexadecimal values between 0 and 20H (space) or between 7FFFH and 0FFFFH. The operating system may screen certain control characters such as

; & | [ ] < > and DEL

that are likely to have special meaning to the operating system and therefore may never appear as outputs of DQGETARGUMENT when parsing a command tail.

The following rules apply to the arguments and delimiters returned by DQGETARGUMENT:

- Multiple adjacent blanks separating two arguments are treated as one blank. One or more blanks adjacent to any other delimiter are ignored. A tab is treated as a blank and returned as a blank.

- Lowercase characters are converted to uppercase unless part of a quoted string.
- If two delimiters are adjacent, the argument returned has length zero.
- Strings enclosed within a matching pair of single or double quotes are considered literals. The enclosing quotes are not returned as part of the argument. Quotes can be included inside a quoted string by using quotes of the other type or by doubling the quote character.
- If an argument contains more than 80 characters the first 80 are returned with exception E$STRING$BUF. The user can obtain the rest of the argument or the next 80 characters by calling DQGETARGUMENT again.
- The command tail is exhausted when the delimiter is CR .

Standard Exception Codes

EOK, ESTRING$BUF

Comments

The operating system's command line interpreter (CLI) may pre-edit the command line (removing comments and continuation characters) before your program is executed. The command line interpreters of some operating systems may make additional modifications to the command line.

Example Calling Sequences

```
DELIM = DQ$GET$ARGUMENT (@ARG, @ERR);      PLM

        LEA     AX,ARG      ;              ASM
        PUSH    DS          ; 1
        PUSH    AX          ; 2
        LEA     AX,ERR
        PUSH    DS          ; 3
        PUSH    AX          ; 4
        CALL    DQGETARGUMENT
        MOV     DELIM,AL
```

Example Usage

The following examples illustrate the arguments and delimiters returned by successive calls to DQGETARGUMENT:

- PLM.86 LINKER.PLM PRINT(:LP:) NOLIST

ARGUMENT LENGTH	VALUE	DELIMITER
8	PLM86.86	(space)
10	LINKER.PLM	(space)
5	PRINT	(
4	:LP:	)
6	NOLIST	CR

- PLM86.86 MODULE.SRC PRINT(:F1:THISIS.IT) OPTIMIZE(0)& TITLE('MY MODULE')

ARGUMENT LENGTH	VALUE	DELIMITER
8	PLM86.86	(space)
10	MODULE.SRC	(space)
5	PRINT	(
13	:F1:THISIS.IT	)
8	OPTIMIZE	(
1	0	)
5	TITLE	(
9	MY MODULE	)
0		CR

- LINK86.86 :F4:X.OBJ,LLIB(MODL),SYSTEM.LIB & (:F3:FUNNY.LIB(MOD1)) PUBLICS MAP

ARGUMENT LENGTH	VALUE	DELIMITER
9	LINK86.86	(space)
9	:F4:X.OBJ	,
4	LLIB	(
4	MODL	)
0		,
10	SYSTEM.LIB	(
13	:F3:FUNNY.LIB	(
4	MOD1	)
0		)
7	PUBLICS	(space)
3	MAP	CR

DQ$SWITCH$BUFFER

Description

DQ$SWITCH$BUFFER is used with DQGETARGUMENT to parse syntax contained within your program; for example, Intel's translators call this procedure along with DQGETARGUMENT to process control lines imbedded in source files as if they were invocation commands.

Declaration Syntax

```
DQ$SWITCH$BUFFER:
     PROCEDURE (buffer$p, excep$p) WORD EXTERNAL;
     DECLARE buffer$p  POINTER,
             excep$p   POINTER;
     END;
```

Input Parameters

buffer$p points to the beginning of the text to be parsed. It would normally be set to the first character after the '$' of a control line.

Output Parameters

This is a typed procedure (function). The value returned by the function is a WORD, containing the offset (from the start of the buffer) of the first character past the latest delimiter returned by DQGETARGUMENT.

Standard Exception Codes

E$OK

Comments

This procedure should not be called until the entire command tail of the invocation line has been parsed; there is no way to switch back to the original command tail, since only the operating system knows where its buffer is.

DQ$SWITCH$BUFFER actually has two uses:

1. To start parsing at a new location
2. To return the current position in the text

The first time you call this procedure, it switches to the indicated location and returns a value of zero. If, after parsing the text with DQGETARGUMENT, you wish to find the current location in the buffer being parsed, call DQ$SWITCH$BUFFER again. For this second invocation, the value returned is the offset from the start of the buffer to the first character past the last delimiter returned by DQGETARGUMENT.

If you then wish to continue scanning with DQGETARGUMENT, you must reset the buffer to be scanned to the location at which you desire to resume parsing.

Text parsed after using DQ$SWITCH$BUFFER is not pre-edited by the command line interpreter as is the case when reading from the console in the line edit mode.

Example Calling Sequences

```
ARG_COUNT = DQ$SWITCH$BUFFER (@COMMAND_BUF, @ERR);      PLM

     LEA        AX,COMMAND_BUF;                         ASM
     PUSH       DS          ; 1
     PUSH       AX          ; 2
     LEA        AX,ERR
     PUSH       DS          ; 3
     PUSH       AX          ; 4
     CALL       DQSWITCHBUFFER
     MOV        ARG_COUNT,AX
```

DQ$ALLOCATE

Description

DQ$ALLOCATE requests that a specific amount of contiguous free memory be added to that used by the calling program.

Declaration Syntax

```
DQ$ALLOCATE:
     PROCEDURE (size, excep$p) SELECTOR EXTERNAL;
     DECLARE size     WORD,
             excep$p  POINTER;
     END;
```

Input Parameters

The number of bytes of memory being requested is specified by size. A size of zero means a request for 64K bytes.

Output Parameters

This is a typed procedure (function). It returns a SELECTOR the contents of which depend on whether the requested memory is available:

- If enough memory is available, the selector value returned represents the start of the acquired memory segment.
- If the request fails, the procedure returns a SELECTOR of OFFFFH, and the exception code is E$MEM.

Standard Exception Codes

EOK, EMEM

Example Calling Sequences

```
ARRAY_BASE = DQ$ALLOCATE (128, @ERR);      PLM

     MOV       AX,80H     ;                 ASM
     PUSH      AX         ; 1
     LEA       AX,ERR
     PUSH      DS         ; 2
     PUSH      AX         ; 3
     CALL      DQALLOCATE
     MOV       ARRAY_BASE,AX
```

DQ$FREE

Description

DQ$FREE returns to the memory manager a segment of memory acquired earlier by DQ$ALLOCATE.

Declaration Syntax

```
DQ$FREE:
     PROCEDURE (segment, excep$p) EXTERNAL;
     DECLARE segment  SELECTOR,
             excep$p  POINTER;
     END;
```

Input Parameters

segment is a SELECTOR representing a memory segment to be freed.

Output Parameters

None

Standard Exception Codes

E$OK

Comments

You cannot return a part of a segment allocated by DQ$ALLOCATE; you can only return the entire segment.

Example Calling Sequences

```
CALL DQ$FREE (ARRAY_BASE, @ERR);      PLM

     PUSH      ARRAY_BASE; 1          ASM
     LEA       AX,ERR
     PUSH      DS          ; 2
     PUSH      AX          ; 3
     CALL      DQFREE
```

DQGETSIZE

Description

DQGETSIZE returns the size of an already allocated memory segment.

Declaration Syntax

```
DQ$GET$SIZE:
     PROCEDURE (segment, excep$p) WORD EXTERNAL;
     DECLARE segbase  SELECTOR,
             excep$p  POINTER;
     END;
```

Input Parameters

segbase is a SELECTOR for a memory segment.

Output Parameters

This is a typed procedure (function) that returns a WORD containing the size (in bytes) of the indicated segment. A size of zero means 64K bytes.

Standard Exception Codes

E$OK

Comments

Relocatable PL/M-86 programs that are compiled under the SMALL model of segmentation and have expanding data segments can determine the size of their data segments with a statement of the form

```
DQ$SIZE = DQ$GET$SIZE (STACKBASE, @EXCEP);
```

Example Calling Sequence

```
ARRAY_SIZE = DQ$GET$SIZE (ARRAY_BASE, @ERR);
```

DQ$RESERVE$IO$MEMORY

Description

DQ$RESERVE$IO$MEMORY informs the operating system of the maximum number of files that will be attached and the maximum number of buffers that will be requested during the execution of a particular program. It requests that the system reserve enough memory to assure that the creates, attaches, and opens will be successful. If this function is not called, this specification does not require that implementations reserve any specific amount of memory for attaches and opens (although individual implementations may choose to do so). In other words, the default value for the two maximums specified in this function is zero. Further, the user may assume that at least twelve attaches and six opens will be supported by calling DQ$RESERVE$IO$MEMORY, although some operating systems may allow for more.

Declaration Syntax

```
DQ$RESERVE$IO$MEMORY:
     PROCEDURE (number$files, number$buffers, excep$p) EXTERNAL;
     DECLARE number$files    WORD,
             number$buffers  WORD,
             excep$p         POINTER;
     END;
```

Input Parameters

number$files is the maximum number of files that will be attached at any one time. If this value is exceeded, the application takes the responsibility for ensuring that the additional memory is preconfigured or available for allocation when the calls to DQ$ATTACH and DQ$CREATE are made.

number$buffers is the maximum number of buffers that will be required for any concurrent set of open files. More precisely, it is a bound on the sum of the values specified in the num$buf parameters in any set of calls to DQ$OPEN for which corresponding calls to DQ$CLOSE have not been made. If this limit is exceeded, the application takes the responsibility for ensuring that the additional memory is available for allocation when the calls to DQ$OPEN are made.

Standard Exception Codes

EOK, EMEM

Comments

The purpose of this function is to allow an application to foreshadow calls to DQ$ATTACH and DQ$OPEN that it will make. By warning the operating system about these calls, the system can reserve memory ahead of time so that intervening calls to DQ$ALLOCATE do not prevent the attaches and opens from being successful. Successive calls to this function are valid. They simply change the current number of buffers requested for the corresponding program. A request to increase the number of buffers can fail due to lack of memory, especially if calls to DQ$ALLOCATE have been made since the previous call to DQ$RESERVE$IO$MEMORY. By the same token, the original call to DQ$RESERVE$IO$MEMORY should occur before the first DQ$ALLOCATE to maximize the chance that it will be successful.

The size and number of buffers reserved by DQ$RESERVE$IO$MEMORY is operating system dependent and in some cases may be configurable. Each implementation of UDI should use a buffer size that maximizes the chance that a set of opens to commonly-used devices will succeed. The implementation may also include recovery mechanisms such as drawing upon other available memory or reducing the number of buffers used for a particular open (at the expense of subsequent performance).

Example Calling Sequence

```
DQ$RESERVE$IO$MEMORY (NUMBER-FILES, NUMBER-BUFFERS, @ ERR); PLM
```

DQ$ATTACH

Description

DQ$ATTACH creates a connection to an existing file.

Declaration Syntax

```
DQ$ATTACH:
     PROCEDURE (path$p, excep$p) CONNECTION EXTERNAL;
     DECLARE path$p   POINTER,
             excep$p  POINTER;
     END;
```

Input Parameters

path$p points to a STRING that contains the pathname of the file. If the named file does not exist, the operation fails.

Output Parameters

This is a typed procedure (function). If the procedure is successful, it returns a connection of type CONNECTION to the named file.

Standard Exception Codes

EOK, EFNEXIST, EMEM, ESUPPORT

Comments

Attaching an output device such as Console Output will return E$SUPPORT. The result of attaching :WORK: is Operating System dependent. The result of attaching a file that is already connected is Operating System dependent. Attempting to attach console output or line printer returns E$SUPPORT. E$MEM is returned if insufficient memory exists to attach the file.

Example Calling Sequence

```
INPUT_CONN = DQ$ATTACH (@FILE_NAME, @ERR);     PLM
```

Description

DQ$CREATE creates a connection to a new file.

Declaration Syntax

```
DQ$CREATE:
     PROCEDURE (path$p, excep$p) CONNECTION EXTERNAL;
     DECLARE path$p   POINTER,
             excep$p  POINTER;
     END;
```

Input Parameters

path$p points to a STRING containing a pathname. If a file of the same name already exists and is not connected, either it is truncated (deleting any data therein) or it is deleted and a new file is created with the same name. This action is performed before completion of the procedure. If the file exists but is connected and open, an error condition results.

Output Parameters

This is a typed procedure (function). If the procedure is successful, it returns a connection of type CONNECTION to the new file. An attempt to create console input returns an E$SUPPORT exception code.

Standard Exception Codes

EOK, EMEM, E$SPACE, E$SUPPORT

Comments

Creating an input device such as Console Input will return an E$SUPPORT error.

Example Calling Sequence

```
OUTPUT_CONN = DQ$CREATE (@NEW_FILE, @ERR);
```

DQ$DETACH

Description

DQ$DETACH breaks the connections established by DQ$ATTACH or DQ$CREATE.

Declaration Syntax

```
DQ$DETACH:
     PROCEDURE (conn, excep$p) EXTERNAL;
     DECLARE conn     CONNECTION,
             excep$p  POINTER;
     END;
```

Input Parameters

conn identifies the connection. If the connection is open, it is closed before being detached.

Output Parameters

None

Standard Exception Codes

E$OK

Comments

Example Calling Sequences

```
CALL DQ$DETACH (INPUT_CONN, @ERR);      PLM

     PUSH       INPUT_CONN; 1           ASM
     LEA        AX,ERR
     PUSH       DS          ; 2
     PUSH       AX          ; 3
     CALL       DQDETACH
```

DQ$DELETE

Description

DQ$DELETE eliminates an existing file.

Declaration Syntax

```
DQ$DELETE:
     PROCEDURE (path$p, excep$p) EXTERNAL;
     DECLARE path$p   POINTER,
             excep$p  POINTER;
     END;
```

Input Parameters

path$p points to a STRING containing the name of the file to be deleted.

Output Parameters

None

Standard Exception Codes

EOK, EFNEXIST, E$FACCESS

Comments

The results of trying to delete an attached file are operating system dependent.

Example Calling Sequences

```
CALL DQ$DELETE (@FILE_NAME, @ERR);      PLM

     LEA        AX,FILE_NAME;           ASM
     PUSH       DS        ; 1
     PUSH       AX        ; 2
     LEA        AX,ERR
     PUSH       DS        ; 3
     PUSH       AX        ; 4
     CALL       DQ$DELETE
```

DQGETCONNECTION$STATUS

Description

DQGETCONNECTION$STATUS returns information associated with a file connection.

Declaration Syntax

```
DQ$GET$CONNECTION$STATUS:
     PROCEDURE (conn, info$p, excep$p) EXTERNAL;
     DECLARE conn      CONNECTION,
             info$p    POINTER,
             excep$p   POINTER;
     END;
```

Input Parameters

conn identifies a connection established earlier by DQ$ATTACH or DQ$CREATE.

Output Parameters

info$p points to a structure of the following form that you declare to receive the connection data:

```
DECLARE INFO STRUCTURE
(OPEN              BOOLEAN,
 ACCESS            BYTE,
 SEEK              BYTE,
 FILE$PTR          DWORD);
```

These fields are interpreted as follows:

OPEN:
TRUE if connection is open, otherwise FALSE.

ACCESS:
Access privileges of the connection. The right is granted if the corresponding bit is set.

BIT	ACCESS	BYTE VALUE
0	delete	1
1	read	2
2	write	4
3	update	8 (read and write)
4-7	reserved (must be 0)	

SEEK:
Types of seek supported.

BIT	ACCESS	BYTE VALUE
0	seek forward	1
1	seek backward	2
2-7	reserved (must be 0)	

FILE$PTR:
Forms a 4-byte unsigned integer (DWORD) that indicates the current position in the file. The position is expressed as the number of the current byte from the beginning of the file with byte 0 being the first byte. This field is undefined if the file is not open or if backward seek is not supported by the device (for example, the printer cannot be rewound).

Standard Exception Codes

E$OK

Comments

Example Calling Sequences

```
CALL DQ$GET$CONNECTION$STATUS (INPUT_CONN, @FILE_STATUS, @ERR); PLM

        PUSH    INPUT_CONN; 1                                   ASM
        LEA     AX,FILE_STATUS
        PUSH    DS          ; 2
        PUSH    AX          ; 3
        LEA     AX,ERR
        PUSH    DS          ; 4
        PUSH    AX          ; 5
        CALL    DQGETCONNECTIONSTATUS
```

DQ$FILE$INFO

Description

DQ$FILE$INFO returns the file information normally associated with user security and accounting.

Declaration Syntax

```
DQ$FILE$INFO:
     PROCEDURE (conn, mode, file$info$p excep$p) EXTERNAL;
     DECLARE conn          CONNECTION,
             mode          BYTE,
             file$info$p   POINTER,
             excep$p       POINTER;
     END;
```

Input Parameters

conn identifies the connection of a currently attached file.

mode indicates whether the file owner is to be identified.

BYTE VALUE	USAGE
0	owner name or identification is not to be returned.
1	owner name or identification is to be returned.

Output Parameters

file$info$p points to the structure that you declare to receive the file information;

```
DECLARE FILE$INFO STRUCTURE
(OWNER(15)        STRING,
LENGTH$OF$FILE    DWORD,
TYPE              BYTE,
OWNER$ACCESS      BYTE,
WORLD$ACCESS      BYTE,
CREATE$TIME       DWORD,
LAST$MOD$TIME     DWORD,
RESERVED(20)      BYTE);
```

These fields are interpreted as follows:

OWNER:
A string that identifies the system name of the owner of the file.

TYPE:
Indicates the usage of the file.

VALUE	FILE TYPE
0	data file
1	directory file
2	reserved
.	
.	
.	
255	

OWNER$ACCESS, WORLD$ACCESS:
Describes the access rights of the file owner and the rest of the world.

BIT	ACCESS
0	delete
1	read (data), display (directory)
2	write (data), addentry (directory)
3	update (read and write)
4-7	reserved

CREATE$TIME, LAST$MOD$TIME:
Indicates the date and time of creation and last modification for a file. If the file has been created but not modified, the LASTMODTIME should be the same as the CREATE$TIME. A modification consists of a write or truncate on the file. The contents of the time DWORD is the number of seconds since January 1, 1978. This may be decoded into an ASCII string with DQ$DECODE$TIME.

Standard Exception Codes

EOK, ESUPPORT

Example Calling Sequence

```
CALL DQ$FILE$INFO (conn, mode, file$info$p, excep$p);
```

DQ$RENAME

Description

DQ$RENAME changes the name of a file.

Declaration Syntax

```
DQ$RENAME:
     PROCEDURE (old$p, new$p, excep$p) EXTERNAL;
     DECLARE old$p    POINTER,
             new$p    POINTER,
             excep$p  POINTER;
     END;
```

Input Parameters

old$p points to a STRING containing the pathname of the file to be renamed.

new$p points to a STRING giving a new pathname for the file. The exception code E$FEXIST is returned if the new name already exists.

Output Parameters

None

Standard Exception Codes

EOK, EFEXIST, E$FNEXIST

Comments

Both the old name and the new must refer to files on the same volume.

The results of trying to rename an attached file are operating system dependent.

Example Calling Sequence

```
CALL DQ$RENAME (@FILE_NAME, @NEW_FILE, @ERR);     PLM
```

DQ$CHANGE$EXTENSION

Description

DQ$CHANGE$EXTENSION changes or adds the extension at the end of a file name; for example, :F4:FILE.SRC can be changed to :F4:FILE.OBJ or to :F4:FILE.LST.

Declaration Syntax

```
DQ$CHANGE$EXTENSION:
     PROCEDURE (path$p, extension$p, excep$p) EXTERNAL;
     DECLARE path$p       POINTER,
             extension$p  POINTER,
             excep$p      POINTER;
     END;
```

Input Parameters

path$p points to a STRING containing the pathname to be changed.

extension$p points to a three character extension that is to be added to the pathname. The three character extension may not contain delimiters recognized by DQGETARGUMENT but may contain trailing blanks.

Output Parameters

None

Standard Exception Codes

EOK, ESTRING$BUF

Comments

If the first character addressed by extension$p is a space, any prior extension is deleted (including the preceding period).

Example Calling Sequence

```
CALL DQ$CHANGE$EXTENSION (@FILE_NAME, @EXTENSN, @ERR);
```

DQ$CHANGE$ACCESS

Description

DQ$CHANGE$ACCESS changes the access rights of the owner of the file or the world.

Declaration Syntax

```
DQ$CHANGE$ACCESS:
     PROCEDURE (path$p, class, access, excep$p) EXTERNAL;
     DECLARE path$p   POINTER,
             class    BYTE,
             access   BYTE,
             excep$p  POINTER;
     END;
```

Input Parameters

path$p points to a STRING containing the pathname of the file whose access rights is to be changed.

class specifies the class of users whose access rights are to be changed.

Value	Access rights of
0	owner of file
1	world
2-255	reserved

access specifies the type of access to be granted to the class of file users specified. If all bits are set to 0 the specified users access to the file will be denied. If any bits are set to 1 the access is granted as indicated:

Bit	Access
0	delete (data), delete (directory)
1	read (data), display (directory)
2	write (data), add entry (directory)
3	update (read and write)

Output Parameters

None

Standard Exception Codes

EOK, EFNEXIST, E$FACCESS, E$SUPPORT

Comments

The privilege to use this procedure is assured to the owner of the file. The granting of this privilege to other users is operating system dependent. If the privilege is not granted, the error E$FACCESS is returned. E$FNEXIST indicates the file does not exist; E$SUPPORT indicates an attempt to change the access rights of a non-disk file. The access rights of the file will be changed immediately but will not affect other connections to the file until they are detached.

DQ$OPEN

Description

DQ$OPEN opens a previously established connection. The open process involves allocating buffers, checking access privileges, setting the file pointer to the start of the file and, in general, preparing the connection for read or write commands.

Declaration Syntax

```
DQ$OPEN:
     PROCEDURE (conn, access, num$buf, excep$p) EXTERNAL;
     DECLARE conn     CONNECTION,
             access   BYTE,
             num$buf  BYTE,
             excep$p  POINTER;
     END;
```

Input Parameters

conn represents a connection established earlier via DQ$ATTACH or DQ$CREATE.

access specifies the type of access desired:

VALUE	INTERPRETATION
1	read access only
2	write access only
3	update (both read and write)

num$buf specifies an optimal number of buffers, and should have one of the values 0, 1, or 2. Zero means that no buffering should occur; each DQ$READ or DQ$WRITE should result in a physical I/O operation (as is necessary for interactive devices). If you normally execute a seek before doing a read or write (as for random disk file processing), num$buf should be 1. In all other cases it should be 2 for double buffering (as for sequential file processing).

Interactive programs should open console input, console output, serial input, and serial output with num$buf set to zero to eliminate buffering. Such a request is honored by the operating system so that interactive communication is feasible. For all other files, num$buf is used for optimization, but the operating system may provide other buffering according to its buffering algorithms.

Output Parameters

None

Standard Execution Codes

EOK, EMEM, E$FACCESS, E$SHARE

Comments

The use of DQ$OPEN must not violate physical limitations; for example, the line printer must not be opened for read or update. The byte bucket may be opened with any type of access. The act of opening a file does not change file contents.

An open with access = 1 may allow file sharing for readers only. Successful opens with access = 2 or 3 guarantee exclusive access to the file until it is closed. Console input, console output, and byte bucket are exceptions, since any of these may be attached and opened as many times as needed up to the limit as set by DQ$RESERVE$IO$MEMORY. Support for multiple connections to line printer is operating-system dependent.

A successful open guarantees that subsequent DQ$READs, DQ$WRITEs or both (as implied by access) will be valid.

Example Calling Sequences

```
CALL DQ$OPEN (INPUT_CONN, 1, 2, @ERR);      PLM

     PUSH      INPUT_CONN; 1                 ASM
     MOV       AL,1H     ; read only
     PUSH      AX        ; 2
     MOV       AL,2H     ; double buffering
     PUSH      AX        ; 3
     LEA       AX,ERR
     PUSH      DS        ; 4
     PUSH      AX        ; 5
     CALL      DQOPEN
```

DQ$SEEK

Description

DQ$SEEK changes the file position pointer.

Declaration Syntax

```
DQ$SEEK:  PROCEDURE
     (conn, mode, offset, excep$p) EXTERNAL;
     DECLARE conn          CONNECTION,
             mode          BYTE,
             offset        DWORD,
             excep$p       POINTER;
     END;
```

Input Parameters

conn represents an open connection.

mode indicates the type of seek required.

VALUE	INTERPRETATION
1	Move file pointer back by offset.
2	Move file pointer to offset.
3	Move file pointer forward by offset.
4	Move file pointer to end of file minus offset.

offset forms a four-byte (DWORD) unsigned integer that represents either a position in the file or the number of bytes to move the file position pointer, depending on the setting of mode.

Output Parameters

None

Standard Exception Codes

EOK, ESUPPORT

Comments

If you seek past end of file, a subsequent DQ$WRITE causes the file to be extended. A subsequent DQ$READ returns an end-of-file condition.

A DQ$SEEK to a position before the beginning of a file is the same as a DQ$SEEK to the beginning of the file. A seek to 0 goes to the beginning of the file.

Example Calling Sequences

```
CALL DQ$SEEK (INPUT_CONN, 3, 1442040, @ERR);      PLM

     PUSH      INPUT_CONN; 1                      ASM
     MOV       AL,3H     ; seek forward
     PUSH      AX        ; 2
     MOV       AX,16H    ;     22
     PUSH      AX        ; 3
     MOV       AX,0F8H   ;     248
     PUSH      AX        ; 4
     LEA       AX,ERR
     PUSH      DS        ; 5
     PUSH      AX        ; 6
     CALL      DQSEEK
```

In this example the pointer is moved forward by 22 times 2**16 plus 248 bytes, which evaluates to 1,442,040 bytes.

DQ$READ

Description

DQ$READ fetches data from an open file.

Declaration Syntax

```
DQ$READ:
     PROCEDURE (conn, buf$p, count, excep$p) WORD EXTERNAL;
     DECLARE conn     CONNECTION,
             buf$p    POINTER,
             count    WORD,
             excep$p  POINTER;
     END;
```

Input Parameters

conn represents an open connection.

buff$p points to an area to be used for input.

count specifies the desired number of bytes to be read.

Output Parameters

buff$p points to a buffer area, at least count bytes long, that you allocate to receive the data read.

This is a typed procedure (function). It returns as type WORD the number of bytes actually transferred. This number equals count unless an error occurs or end of file is encountered. If the procedure returns a value less than requested and an exception code of E$OK, end of file was encountered.

Standard Exception Codes

EOK, ESUPPORT

Comments

The requested number of bytes is read from the file at the current file pointer location. The file pointer is updated by the value returned. When reading from the console input device, DQ$READ supports a line-edited and transparent read. The default mode is line-edited and can be changed by calling DQ$SPECIAL. In the line-edited mode the carriage return character CR is always converted to CRLF, a carriage return/line feed sequence of characters. The user should supply a large enough buffer to hold the input line and the CRLF. In the case of single character reads an additional read will be needed to clear the LF.

Example Calling Sequence

```
BYTES_READ = DQ$READ (INPUT_CONN, @IN_BUFFER, BUFFER_COUNT, @ERR);
```

DQ$SPECIAL

Description

DQ$SPECIAL specifies whether subsequent reads from the console input are to be in the line-edited or transparent mode.

Declaration Syntax

```
DQ$SPECIAL:
     PROCEDURE (type, conn$p, excep$p) EXTERNAL;
     DECLARE type     BYTE,
             conn$p   POINTER,
             excep$p  POINTER;
     END;
```

Input Parameters

conn$p points to a connection previously established by a DQ$ATTACH of console input.

type is a BYTE whose value indicates the mode of input.

Value	Input Mode
1	All characters except control-C and control-D are placed uninterpreted in the user's buffer and are not echoed to the screen. This is referred to as transparent mode.
2	Characters are placed in a system buffer and are interpreted for editing commands. Upon a carriage return, the system buffer is passed to the user's buffer. This is referred to as line-edited mode.
3	Similar to (1) except only those characters already in the system input buffer are returned. This is referred to as polling mode.

The default type from the console device when a program starts executing is 2 (line editing).

Output Parameters

None

Standard Exception Codes

EOK, ESUPPORT

Comments

Line editing of input means that the console operator has the opportunity to correct typing errors. Data from the console is not actually returned by a DQ$READ until the operator types CR. Editing characters (such as the backspace character) are removed from the input. The characters used for editing are operating-system dependent. The final CR is always converted to the CRLF.

Interactive programs often need to obtain characters from the console as they are typed. This is made possible by using DQ$SPECIAL to switch to transparent mode. In transparent mode, all characters typed except control-C and control-D are placed in the buffer. The DQ$READ function returns when count characters have been typed. The system does not echo the characters read in transparent mode.

When console input is assigned to a console, the four characters, control-C, control-D, ESCAPE, and control-Z, are likely to have special meaning to the operating system and may never appear in the input stream.

Regardless of whether transparent or line-edited mode is in effect, the abort character (control-C) and the debugger character (control-D) have the usual effect.

Another type (4) may be supported by some operating systems. This is not part of the UDI standard, and programs that use this type are operating-system dependent. It is defined here to ensure consistency of use. If type is 4 and the E$OK condition code is returned, then conn$p is assumed to point to a used-declared structure of the form

```
DECLARE CRT$ID BASED CONN$P STRUCTURE
          (CRT$ID$NUM     WORD,
           PATHNAME$LEN   BYTE,
           PATHNAME (45)  BYTE);
```

An exception code of E$OK with CRT$ID$NUM=0 indicates an undefined CRT. PATHNAME$LEN=0 indicates there is no system CRT configuration file PATHNAME defined. CRTIDNUM=1 is used to indicate the Series-IV CRT.

This mechanism is used by screen-oriented programs to identify the console CRT and/or the configuration file containing CRT characteristics.

When DQ$SPECIAL is used to switch from line-oriented mode (type 2) to one of the transparent modes (type 1 or 3), the contents of the line currently being line-edited are not carried over to the new mode of input. When DQ$SPECIAL is used to switch from one of the transparent modes to line-edited mode, any type-ahead input will be available in the new mode. The contents of the line currently being line-edited will remain unchanged when switching from mode 2 to mode 2.

Example Calling Sequences

```
CALL DQ$SPECIAL (1, @CI_CONN, @ERR);          PLM

     MOV       AL,1H    ; transparent input    ASM
     PUSH      AX       ; 1
     LEA       AX,CI_CONN
     PUSH      DS       ; 2
     PUSH      AX       ; 3
     LEA       AX,ERR
     PUSH      DS       ; 4
     PUSH      AX       ; 5
     CALL      DQSPECIAL
```

DQ$WRITE

Description

DQ$WRITE transfers data from main memory to a file.

Declaration Syntax

```
DQ$WRITE:
     PROCEDURE (conn, buf$p, count, excep$p) EXTERNAL;
     DECLARE conn     CONNECTION,
             buf$p    POINTER,
             count    WORD,
             excep$p  POINTER;
     END;
```

Input Parameters

conn represents an open connection to a file.

buf$p points to the start of the data to be written.

count is the number of bytes to be written.

Output Parameters

None

Standard Exception Codes

EOK, ESUPPORT, E$SPACE

Comments

The error E$SPACE is returned when there is not enough space on the secondary storage device to extend file resulting from the write. Writing begins at the current position indicated by the file pointer, which is zero if no prior reads, seeks, or writes to this file have occurred.

Writing beyond end of file causes the file to be extended.

Example Calling Sequence

```
CALL DQ$WRITE (OUTPUT_CONN, @OUT_BUFFER, 256, @ERR);
```

Description

DQ$TRUNCATE frees previously occupied space from the current file pointer position to end of file.

Declaration Syntax

```
DQ$TRUNCATE:
     PROCEDURE (conn, excep$p) EXTERNAL;
     DECLARE conn     CONNECTION,
             excep$p  POINTER;
     END;
```

Input Parameters

conn represents a connection to a file that is open for write or update.

Output Parameters

None

Standard Exception Codes

EOK, ESUPPORT

Comments

This procedure frees all previously allocated secondary storage space from the location indicated by the file pointer to the end of the file. (If the pointer is at or past end of file, truncation has no effect.) The effects of truncating console input, console output, line printer, and work files are operating system dependent.

Example Calling Sequences

```
CALL DQ$TRUNCATE (EDIT_CONN, @ERR);      PLM

     PUSH      EDIT_CONN; 1               ASM
     LEA       AX,ERR
     PUSH      DS        ; 2
     PUSH      AX        ; 3
     CALL      DQTRUNCATE
```

DQ$CLOSE

Description

DQ$CLOSE waits for completion of I/O operations taking place on the file (if any), ensures that output buffers are empty, and frees buffers.

Declaration Syntax

```
DQ$CLOSE:
     PROCEDURE (conn, excep$p) EXTERNAL;
     DECLARE conn     CONNECTION,
             excep$p  POINTER;
     END;
```

Input Parameters

conn represents an open connection.

Output Parameters

None

Standard Exception Codes

E$OK

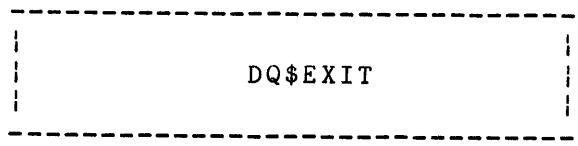
Comments

Once closed, a connection may be either re-opened or detached.

Example Calling Sequences

```
CALL DQ$CLOSE (OUTPUT_CONN, @ERR);

     PUSH       OUTPUT_CONN; 1
     LEA        AX,ERR
     PUSH       DS           ; 2
     PUSH       AX           ; 3
     CALL       DQCLOSE
```



Description

DQ$EXIT terminates a program. All files are closed and all resources are freed.

Declaration Syntax

```
DQ$EXIT:
     PROCEDURE (completion$code) EXTERNAL;
     DECLARE completion$code  WORD;
     END;
```

Input Parameters

completion$code indicates the nature of the termination. It must contain one of the following values:

VALUE	INTERPRETATION
0	Termination was normal.
1	Warning messages were issued.
2	Errors were detected.
3	Fatal errors were detected.
4	The program was aborted.

Output Parameters

This procedure has no exception pointer as an argument; a DQ$EXIT call can never generate an exception.

Standard Exception Codes

None

Comments

The support of the completion$code is operating system dependent.

Example Calling Sequence

```
CALL DQ$EXIT (3);
```

DQ$OVERLAY

Description

DQ$OVERLAY causes loading of an overlay.

Declaration Syntax

```
DQ$OVERLAY:
     PROCEDURE (name$p, excep$p) EXTERNAL;
     DECLARE name$p   POINTER,
             excep$p  POINTER;
     END;
```

Input Parameters

name$p points to a STRING containing the name of the overlay to be loaded. This name must be the same as used in the LINK86 OVERLAY control to name the overlay.

Output Parameters

None

Standard Exception Codes

EOK, ESUPPORT

Comments

Only one level of overlays is allowed; therefore, this procedure may be called only from the root (non-overlaid) phase.

Example Calling Sequences

```
CALL DQ$OVERLAY (@OVLY_NAME, @ERR);      PLM

     LEA        AX,OVLY_NAME;            ASM
     PUSH       DS       ; 1
     PUSH       AX       ; 2
     LEA        AX,ERR
     PUSH       DS       ; 3
     PUSH       AX       ; 4
     CALL       DQOVERLAY
```

DQ$TRAP$EXCEPTION

Description

DQ$TRAP$EXCEPTION substitutes an alternate programmer exception handler for the default programmer exception handler provided by the operating system.

Declaration Syntax

```
DQ$TRAP$EXCEPTION:
     PROCEDURE (address$p, excep$p) EXTERNAL;
     DECLARE address$p  POINTER,
             excep$p    POINTER;
     END;
```

Input Parameters

address$p is the address of a four-byte area containing a long pointer to the entry point of the alternate exception handler. A long pointer has the form:

```
DECLARE LONG$P STRUCTURE
               (LONG$OFFSET  WORD,
                LONG$BASE    SELECTOR);
```

Output Parameters

None

Standard Exception Codes

E$OK

Comments

A PL/M procedure compiled under the SMALL model of segmentation can neither serve as an exception handler nor form a long pointer to an exception handler in another module.

Example Calling Sequence

```
CALL DQ$TRAP$EXCEPTION (@HANDLER_ADDRESS, @ERR);
```

DQGETEXCEPTION$HANDLER

Description

DQGETEXCEPTION$HANDLER fetches the address of the current programmer exception handler.

Declaration Syntax

```
DQ$GET$EXCEPTION$HANDLER:
     PROCEDURE (handler$p, excep$p) EXTERNAL;
     DECLARE handler$p  POINTER,
             excep$p    POINTER;
     END;
```

Output Parameters

handler$p points to a four-byte area declared in your program that the system fills with a long pointer to the current avoidable-exception handler. This is the address specified in the last call to DQ$TRAP$EXCEPTION, if it has been called; otherwise the value returned is the address of the system default handler.

Standard Exception Codes

E$OK

Comments

The output of this procedure is always a four-byte pointer, even if called from a program compiled under the SMALL model of segmentation.

This pointer has two primary uses:

- It is used to formulate an indirect call to the current error handler. You should always call DQGETEXCEPTION$HANDLER before calling the exception handler to ensure that the address used in the call refers to the most recently established handler.
- It can be saved to later restore the current exception handler after another handler has been temporarily substituted.

Example Calling Sequences

```
Call DQ$GET$EXCEPTION$HANDLER (@HANDLER_ADDRESS, @ERR);      PLM

     LEA        AX,HANDLER_ADDRESS;                          ASM
     PUSH       DS        ; 1
     PUSH       AX        ; 2
     LEA        AX,ERR
     PUSH       DS        ; 3
     PUSH       AX        ; 4
     CALL       DQGETEXCEPTIONHANDLER
```

Example Usage

```
$LARGE
                .
                .
                .
DECLARE HANDLER$ADDRESS  POINTER,
        ERR$CODE         WORD;

DQ$GET$EXCEPTION$HANDLER:
     PROCEDURE (ADDRESS$P, EXCEP$P) EXTERNAL;
     DECLARE ADDRESS$P  POINTER,
             EXCEP$P    POINTER;
     END;
                .
                .
                .
CALL DQ$GET$EXCEPTION$HANDLER (@HANDLER$ADDRESS, @ERR$CODE);

CALL HANDLER$ADDRESS (1525H, 0, 0, 0);  /* Indirect call */
                .
                .
                .
```

DQ$DECODE$EXCEPTION

Description

DQ$DECODE$EXCEPTION supplies a message that describes the meaning of an exception code.

Declaration Syntax

```
DQ$DECODE$EXCEPTION:
     PROCEDURE (exception$code, message$p, excep$p) EXTERNAL;
     DECLARE exception$code  WORD,
             message$p       POINTER,
             excep$p         POINTER;
     END;
```

Input Parameters

exception$code contains the exception code to be translated.

message$p points to an area used for output.

Output Parameters

message$p is a pointer to an 81-byte area you declare to receive the exception message. The first byte of the message holds the actual length of the string. The string contains the error code and whatever additional information the operating system can provide. The message will not terminate with a CRLF.

Standard Exception Codes

E$OK

DQ$TRAP$CC

Description

DQ$TRAP$CC establishes an alternate procedure to receive control when the operator types control-C at the physical console.

Declaration Syntax

```
DQ$TRAP$CC:
     PROCEDURE (cc$procedure$p, excep$p) EXTERNAL;
     DECLARE cc$procedure$p  POINTER,
             excep$p         POINTER;
     END;
```

Input Parameters

cc$routine$p is the entry point address of your control-C handler.

Output Parameters

None

Standard Exception Codes

E$OK

Comments

The default control-C handler terminates the program. By supplying your own control-C handler you can take other action.

When the control-C handler receives control, the registers and flags are those of the program that set the control-C handler. The control-C handler must preserve all flags and registers, and execute a long return at completion.

> **Note**
>
> This condition is not supportable in all multi-tasking environments and is likely to be changed in the future. In order to safely avoid the multi-tasking dilemma of receiving an asynchronous control-C when the registers are not those of the event that is meant to be interrupted the user's handler should set a flag which the main procedure can poll at a convenient place.

The control-C character that activates the handler must come from the user's physical console even in the case where the :CI: file has been redirected. If a DQ$READ of console input is outstanding when control-C is typed and you have supplied a control-C procedure, after the control-C procedure has been executed the DQ$READ procedure returns a value of zero and a result code of E$OK.

Example Calling Sequence

```
CALL DQ$TRAP$CC (@CC_HANDLER, @ERR);     PLM
```

MINIMAL PRIMITIVES NEEDED FOR APPLICATION RUNTIME SUPPORT

The iAPX 86, 88 high level language run-time support interface is a subset of UDI. The following procedures may be supported in a minimal fashion by returning E$SUPPORT as an exception code. This exception code is used to indicate to any call to the procedure that its service was not performed.

DQ$CHANGE$EXTENSION
DQGETSYSTEM$ID
DQ$OVERLAY
DQ$TRAP$CC
DQ$RENAME
DQ$RESERVE$IO$MEMORY
DQ$SWITCH$BUFFER
DQ$FILE$INFO
DQ$DECODE$TIME

Although the above procedures are not presently used by the high level language run-time support package, there is no guarantee that they will not be used at some point in the future.

The following procedures that are called by the high-level language run-time support may be supported with the indicated minimal functionality:

DQ$DECODE$EXCEPTION may return a string containing the ASCII error code without the corresponding message.

DQGETARGUMENT may return a string of zero length at the location pointed to be ARGUMENT$P, and a function value of an ASCII carriage return (0DH).

DQ$SPECIAL need only support the type parameter with value 2. Any other value may return the exception code E$SUPPORT. (The terminal driver need only support the line-editing mode.)

Device Names

Console Input	:CI:
Console Output	:CO:
Line Printer	:LP:
Word Directory Prefix	:WORK:
Serial Input	:TI:
Serial Output	:TO:
Byte Bucket	:BB:

Interrupt and Exception Mapping

0	E$ZERO$DIVIDE	integer divide by zero
4	E$OVERFLOW	integer overflow
5	E$ARRAY$BOUNDS	bounds check violation
16	E$NDP	8087 numerics data processor
17	E$SPECIAL$BOUNDS$CHECK	software interrupt used for runtime special bounds checking

Calling Conventions

UDI conforms to PL/M usage. Only registers CS, DS, SS, IP, SP, and BP are preserved by the UDI system calls. Other registers and flags will be undefined on return from a UDI call.

Numerics Data Coprocessor (8087) Support Initial State (optional)

If the operating system allows more than one 8087 program or task to execute concurrently, then the operating system is responsible for saving and restoring the 8087 state as required so that programs cannot interfere with each other. The operating system must also associate 8087 execution interrupts with the correct program.

Attributes of the 8087 processor must also be maintained for every program that uses it. These attributes and their initial values are

PRECISION	=	64 bits
ROUNDING	=	round to nearest or even
CLOSURE	=	projective affinity
ERROR MASKS	=	all 8087 errors unmasked
INTERRUPT MASK	=	8087 error interrupt unmasked

A program may change these values; the system will maintain a separate value of the 8087 Control Word for each program.

APPENDIX B
WRITING YOUR OWN LOGICAL RECORD SYSTEM

WHY USE AN ALTERNATE LRS?

The Logical Record System (LRS) supplied by Intel with the application languages FORTRAN-86/88 and Pascal-86/88 provides I/O, memory-management, and exception-handling support by calling on operating system functions through the Universal Development Interface (UDI). If, however, the target environment for your FORTRAN-86/88 or Pascal-86/88 application does not include an operating system, you need to write an LRS that provides I/O, memory-management, and exception-handling support for your application.

If your target environment includes an operating system that does not support UDI, refer to Chapter 8 and Appendix A, which provide information on how to implement UDI.

WHAT IS INVOLVED IN WRITING AN LRS?

Figure B-1 illustrates how a user-written LRS fits into the run-time support model when there is no operating system.

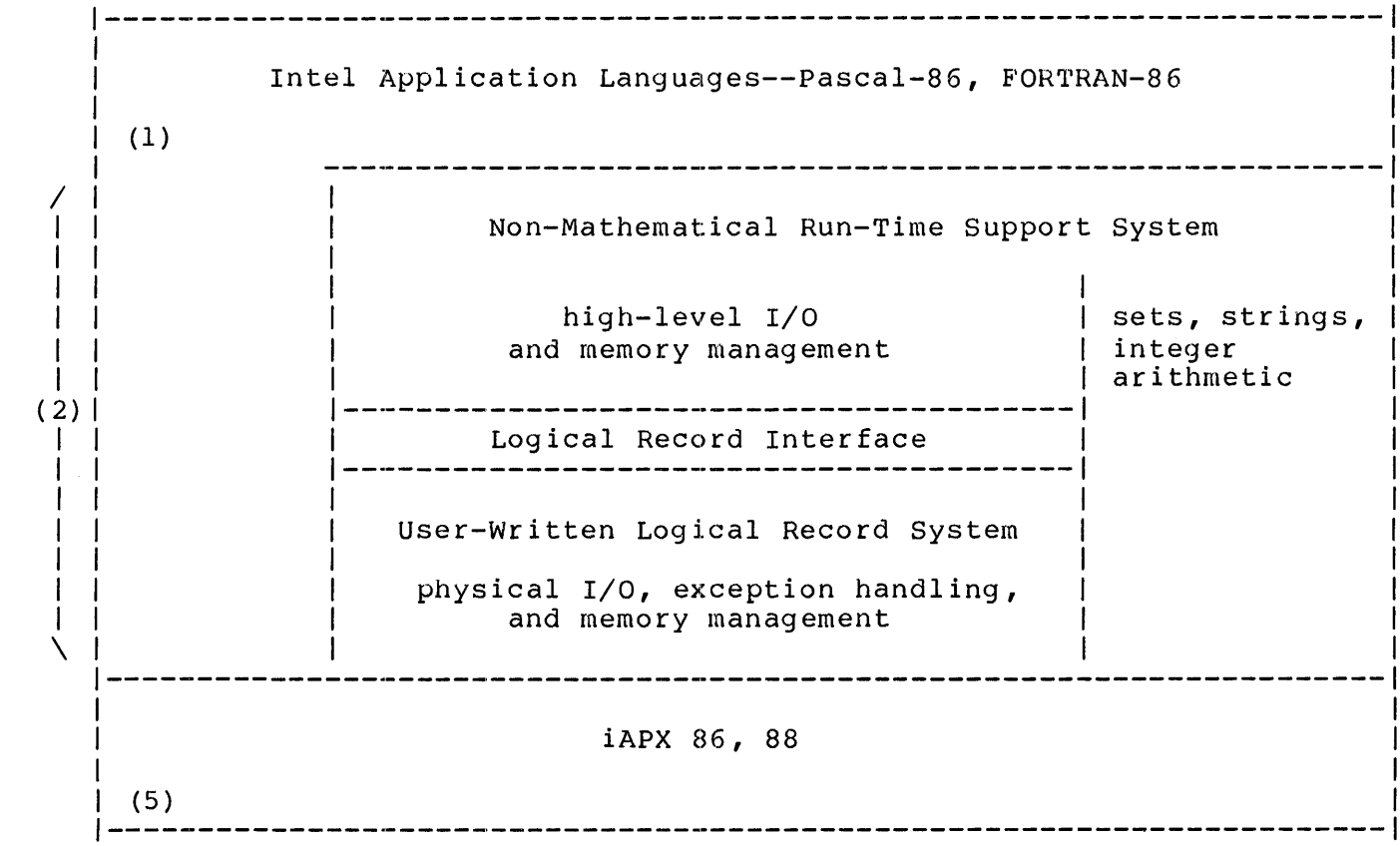


Figure B-1. Run-Time Support Without an Operating System

A standard interface, called the Logical Record Interface (LRI), has been defined for the LRS. The LRI consists of the following components:

- Control procedures

    1. TQ$INITIALIZE
    2. TQGETPRECON
    3. TQ$EXIT

- Input-output support

    1. Data structures

        File-Device Descriptors
        Device Driver Tables

    2. Connection procedures

        TQ$FILE$DESCRIPTOR
        TQ$DEVICE

    3. Device driver procedures

        Open
        Close
        Read
        Write
        Seek
        Skip
        End record
        Rewind
        Backspace
        End file

- Exception handler procedures

    1. TQSETERH
    2. TQGETERH

- Memory management procedures

    1. TQ$ALLOCATE
    2. TQ$FREE
    3. TQGETSMALL$HEAP

You construct your LRS by supplying substitutes for some or all of the above data structures and procedures in accordance with the LRI specifications. Most applications do not require all of these interfaces, and neither FORTRAN-86/88 nor Pascal-86/88 require all of them. Read the following specifications for more details on how the LRS works and when each of these interfaces is required.

LOGICAL RECORD INTERFACE SPECIFICATIONS

The following sections define a standard interface to the Logical Record

System.

Reentrancy

If your application consists of several concurrent tasks, you may wish to link all the tasks to a single copy of the run-time libraries. If you do so and if any of your application tasks that use the LRS may be arbitrarily interrupted to execute other LRS tasks, the LRS procedures must be reentrant. Except where noted in the following specifications, the procedures of the default LRS are reentrant. To make your LRS procedures reentrant, too, you must adhere to these guidelines:

- Local variables must be kept on the stack. (In PL/M-86, this is accomplished by the REENTRANT attribute of a procedure declaration.)
- No LRS procedure may assign a value to a fixed memory location.
- The code must not modify itself. (This requirement prohibits overlays.)
- Sub-procedures called by LRS procedures must also be reentrant.

The LRS procedure interfaces are designed to help make these guidelines attainable.

Exception Handling

Many of the LRS procedures return codes that identify exceptional conditions that arise from executing those procedures. The run-time system handles exceptions according to the UDI standard. (Refer to the UDI Specifications in Appendix A.) The default LRS contains an exception handler that displays a message at the system console and terminates the job. You may, however, substitute your own UDI standard exception handler by supplying a TQ$ESTART procedure. Refer to the TQ$INITIALIZE procedure for details of how to do this.

The exception codes listed in table B-1 are recognized by the run-time system and are used to control its execution. They do not necessarily result in calling the current exception handler. You must return these codes from your LRS to signal the indicated conditions.

Table B-1. Required Exception Codes

SYMBOLIC NAME	HEX VALUE	MEANING
E$OK	0	The operation completed normally.
E$PRECON	1501	Invalid syntax has been detected in command line preconnection parameters.
E$NWRITE	1503	The file is not open for writing.
E$NREAD	1504	The file is not open for reading.
E$EOF	15FF	A read was attempted but the file was positioned at end of file.
E$EOR	15FE	A read was attempted past the end of a record.
E$PATHEQ	15FD	The name of the file already open was not equal to the name specified.
E$MEM	0002	Insufficient memory for requested operation.
E$FEXIST	0020	The file, which should not have existed, does in fact exist.
E$FNEXIST	0021	The file, which should have existed, does not in fact exist.

Your LRS procedures may return other codes to signal other exceptional conditions. The range of codes from 1520H through 15EFH is reserved for your use. When the run-time system encounters any of these codes (except E$OK) or any other codes it does not recognize, it calls the current exception handler.

Specification Format

The specifications of each of the LRS procedures are divided into identical sections. The DESCRIPTION section gives a brief summary of the function performed by the routine and indicates under what conditions you might wish to provide your own version of the routine.

The PROCEDURE INTERFACE section shows the general form of the PL/M-86 syntax you should use to declare the procedure if you write your own version in PL/M-86. Identifiers printed in lower case are merely descriptive; you may use other identifiers in your own procedure declarations. PL/M-86 is used as the standard for definition since all LRS procedures conform to PL/M-86 usage in calling sequence and register usage.

If a procedure returns an exception code, the REQUIRED EXCEPTION CODES section defines which codes have special meaning to the run-time system. Your LRS procedures may return other codes to denote other exceptional conditions. Codes that the run-time system does not recognize are passed to the current exception handler.

The run-time library RTNULL.LIB contains minimal versions of the LRS procedures. The sections entitled RTNULL VERSION define the actions taken by the procedures in RTNULL.LIB. You may find that some of the RTNULL versions suit the needs of your application. You can use the procedures in RTNULL.LIB if you are supplying an entire LRS and therefore not linking to the libraries that contain the default LRS. If you link RTNULL.LIB after the libraries that contain your LRS, any LRI procedures that you do not supply are linked from RTNULL.LIB.

Following the specifications you will find an example implementation of a set of specialized device drivers.

Data Types

The following data types are used in the specifications:

BYTE	--	An eight-bit item.
WORD	--	A two-BYTE item.
POINTER	--	Equivalent to PL/M-86 type POINTER (four bytes).
SELECTOR	--	Equivalent to PL/M-86 type SELECTOR. A 16-bit iAPX 86,88 paragraph number (the base portion of a four-byte POINTER).

Data Structures

Two types of data structures are central to the operation of an LRS:

1. File descriptors
2. File/device driver tables

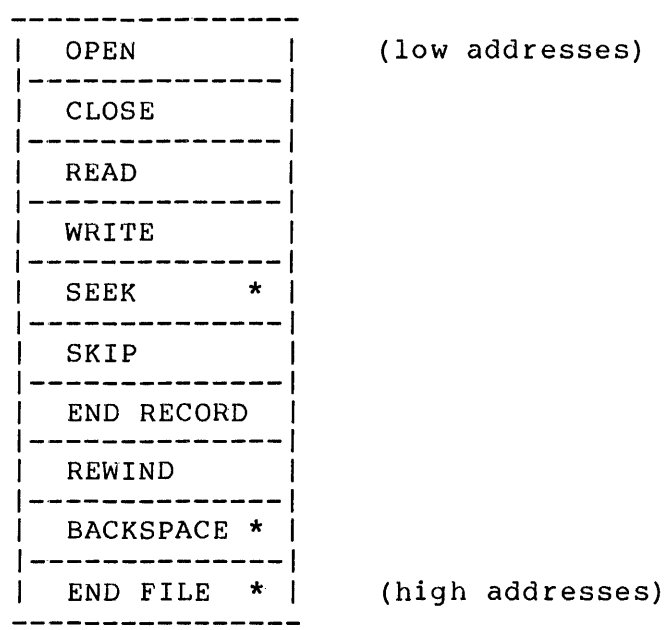
FILE DESCRIPTORS

A block of memory, called a file descriptor, is needed to store attributes of an active file. Memory space for file descriptors is supplied by the LRI procedure TQ$FILE$DESCRIPTOR. Each file descriptor is 48 bytes long and begins at a paragraph boundary (i.e., at an address that is evenly divisible by 16). The first (lowest addressed) 16 bytes of a file descriptor are used by device drivers. If you are supplying all of the device driver procedures for a file (i.e. you are not using any of the default driver procedures), your drivers may use this area for information specific to the file (for example, for storing the open attributes of the file). The remaining 32 bytes are reserved for higher levels of the run-time system.

FILE/DEVICE DRIVER TABLES

Routines that actually transfer data and communicate with external files or devices are called file/device drivers or just device drivers. Because of the unique requirements of different devices and file types, there is a need for different device drivers. The default logical record system supplies two sets of drivers: one for formatted files and one for unformatted files. Each set contains ten procedures to perform ten file actions: open, close, read, write, seek, skip to the end of a record, mark the end of a record, rewind, backspace, and mark the end of a file.

You can replace or supplement the default device drivers by writing your own driver routines in accord with the LRI procedure specifications that follow. The mechanism for connecting your device drivers to the run-time system consists of device driver tables used together with the LRI procedure TQ$DEVICE. A device driver table is an array of long pointers to the entry points of the ten device driver procedures. When you supply a set of device driver routines, you must also supply a device driver table containing the addresses of those routines. The run-time system uses these addresses to form indirect calls to your driver routines. In addition, you must supply your own TQ$DEVICE routine to inform the run-time system which device driver table to use for any given file. The device driver table is illustrated in figure B-2.



* Not used for Pascal-86/88 support.

Figure B-2. Table of Addresses for File/Device Drivers

Note that the order of entries in the table is important. The function of a driver routine determines its position in the table. For example, when the run-time system needs to perform a file open, it always calls the first routine in the table. Since the run-time system forms indirect calls using the address in the device driver tables, you may assign any names to your

driver routines.

The following example shows one way of setting up a device driver table in PL/M-86.

```
DECLARE VT_DRIVER_TABLE (10) POINTER;

VT_DRIVER_TABLE (0) = @VT_OPEN;
VT_DRIVER_TABLE (1) = @VT_CLOSE;
VT_DRIVER_TABLE (2) = @VT_READ;
VT_DRIVER_TABLE (3) = @VT_WRITE;
VT_DRIVER_TABLE (4) = @VT_SEEK;
VT_DRIVER_TABLE (5) = @VT_SKIP;
VT_DRIVER_TABLE (6) = @VT_END_RECORD;
VT_DRIVER_TABLE (7) = @VT_REWIND;
VT_DRIVER_TABLE (8) = @VT_BACKSPACE;
VT_DRIVER_TABLE (9) = @VT_END_FILE;
```

Connection Procedures

The following procedures are used to establish connections at run time between the logical record system and your device driver procedures.

DESCRIPTION

This routine supplies space for file descriptors. It is called by the run-time system before each file open if the run-time system does not already have a free file descriptor. The default version of TQ$FILE$DESCRIPTOR calls the UDI procedure DQ$ALLOCATE. You should provide your own version of TQ$FILE$DESCRIPTOR if one of the following conditions prevail:

1. Your system does not include DQ$ALLOCATE.

2. Your application requires only a fixed number of file descriptors, thereby enabling you to use a simpler allocation algorithm.

PROCEDURE INTERFACE

```
TQ$FILE$DESCRIPTOR:
    PROCEDURE (fd$sel$p) WORD PUBLIC REENTRANT;
    DECLARE fd$sel$p  POINTER;
    DECLARE fd BASED fd$sel$p SELECTOR;
    DECLARE err$code WORD;
    ;
    RETURN err$code;
    END;
```

INPUT PARAMETERS

fdselp contains the address of the output item fd.

OUTPUT

fd identifies a 48-byte area that the run-time system may use for a file descriptor.

This is a typed procedure (function). The value of the procedure is a WORD (here called err$code) that indicates the result of calling this procedure.

REQUIRED EXCEPTION CODES

E$OK

COMMENTS

The run-time system does not return space when a file descriptor is freed by a file close. However, when a file is to be opened and a free file descriptor exists, the run-time system re-uses that space and does not call TQ$FILE$DESCRIPTOR.

RTNULL VERSION

The version of TQ$FILE$DESCRIPTOR in RTNULL.LIB provides six file descriptors in a statically allocated memory space.

DESCRIPTION

This procedure determines which set of device drivers is to be used for a given file. The procedure is called prior to every file open. If you wish to provide your own device drivers you must supply a TQ$DEVICE procedure, since it is the sole means of enabling the run-time system to call the drivers in your LRS.

PROCEDURE INTERFACE

```
TQ$DEVICE:
    PROCEDURE (name$p, name$length, driver$table$addr$p)
        WORD PUBLIC REENTRANT;
    DECLARE name$p                POINTER,
            name$length           BYTE,
            driver$table$addr$p   POINTER;
    DECLARE driver$table$addr
      BASED driver$table$addr$p   POINTER;
    DECLARE err$code WORD;
    ;
    RETURN err$code;
    END;
```

INPUT PARAMETERS

name$p points to the path name of the file that is to be opened, and name$length contains its length. If you want your device drivers to operate on some, but not all, files, your TQ$DEVICE must examine the path name to determine which device drivers to use. For example, if your application requires a special set of device drivers to handle a videotape machine, you may decide to select them whenever the logical device :VT: is used as the pathname. Take care not to create any unnecessary system dependencies by the way your TQ$DEVICE examines path names. Do not change the path name; the run-time system assumes that the contents of the area addressed by name$p are not changed.

driver$table$addr$p points to an area that contains the address (driver$table$addr) of one of the default device-driver tables. Do not change this area if you wish to use the default table.

OUTPUT

If you do wish to substitute one of your device-driver tables for the table determined by the run-time system, overwrite driver$table$addr with the address of your table.

This is a typed procedure (function). The value of the procedure is a WORD (here called err$code) that indicates the result of calling this procedure.

REQUIRED EXCEPTION CODES

E$OK

COMMENTS

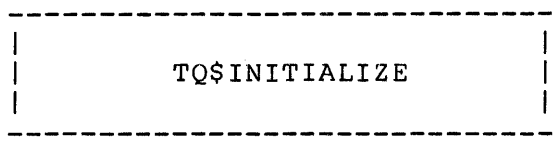
RTNULL VERSION

There is no version of TQ$DEVICE in RTNULL.LIB. If your application does no I/O, the compilers do not generate calls to TQ$DEVICE. If you are supplying your own device drivers, you must code your own version of TQ$DEVICE.

RTNULL.LIB does, however, contain null versions of the default driver tables and driver routines. These cause the processor to halt, if they are ever called. Therefore, if you do not want to use any of the default drivers, your TQ$DEVICE routine must always overwrite the default driver table address with the address of one of your driver tables.

Control Procedures

The procedures in this section control execution of the LRS.



DESCRIPTION

This procedure sets up an exception handler, processes file preconnection parameters that may be entered in the program invocation line, and performs any other initialization of the LRS. TQ$INITIALIZE is called at the beginning of program execution.

PROCEDURE INTERFACE

```
TQ$INITIALIZE:
    PROCEDURE (list$addr$p) WORD PUBLIC REENTRANT;
    DECLARE list$addr$p  POINTER;
    DECLARE list$addr BASED list$addr$p WORD;
    DECLARE err$code WORD;
    ;
    RETURN err$code;
    END;
```

INPUT PARAMETERS

list$addr$p contains the address of the output item list$addr.

OUTPUT

TQ$INITIALIZE must fill list$addr with the address of the list that it creates of preconnection parameters. This value is passed later to TQGETPRECON. (If you code your TQ$INITIALIZE to be reentrant, the preconnection list can not be stored in memory local to TQ$INITIALIZE.) If your application does not use preconnection, list$addr may be filled with a zero pointer or a pointer to a dummy list entry (five bytes of zeros).

This is a typed procedure (function). The value of the procedure is a WORD (here called err$code) that indicates the result of calling this procedure.

REQUIRED EXCEPTION CODES

E$OK

COMMENTS

The default version of TQ$INITIALIZE does nothing but make the following calls to subsidiary LRS procedures and check their returned condition codes. All of these functions should return E$OK if they complete successfully.

```
ECODE = TQ$ESTART;          /* Set up exception handler.  */
ECODE = TQ$PARSECL(list$addr$p); /* Parse command line. */
ECODE = TQ$INITIO;          /* Initialize I/O system.     */
ECODE = TQ$INITMM;          /* Initialize memory manager. */
```

Associated with the procedure TQ$PARSECL is the data structure TQ$DEFAULTPL. You may either selectively override any of these subsidiary features in the LRS, or override the entire TQ$INITIALIZE procedure.

The default TQ$ESTART establishes the LRS's exception handler as the current exception handler by calling the UDI procedure DQ$TRAP$EXCEPTION. The LRS's exception handler displays a message on the console and causes the job to be aborted. You need to supply your own version of TQ$ESTART if you wish to use another exception handler.

The default TQ$PARSECL parses any preconnection parameters that may have been entered in the program invocation command. Preconnection parameters associate external path names with internal program objects (the program-parameter-list of the PROGRAM statement in Pascal-86/88; unit numbers in FORTRAN-86/88). TQ$PARSECL fills list$addr with a POINTER to the list that it creates of preconnection parameters. This pointer is passed later to TQGETPRECON. If your application does not use dynamic preconnection, you may wish to provide a version of TQ$PARSECL that merely fills list$addr with a POINTER to TQ$DEFAULTPL.

The data structure TQ$DEFAULTPL contains the default proconnections defined by FORTRAN-86/88 and Pascal-86/88. The defaults are:

- For FORTRAN-86/88...

  1. UNIT5 = :CI:
  2. UNIT6 = :CO:

- For Pascal-86/88...

  1. INPUT = :CI:
  2. OUTPUT = :CO:

You may supply your own version of TQ$DEFAULTPL if you wish to change the default preconnections. Refer to the discussion of TQGETPRECON for the format of a preconnection list.

The default TQ$INITIO does nothing. You may supply your own version of TQ$INITIO to perform any initialization required by your device drivers.

The default TQ$INITMM does nothing. You may supply your own version of TQ$INITMM to perform any initialization required by your memory management routines TQ$ALLOCATE, TQ$FREE, and TQGETSMALL$HEAP.

If you supply your own version of TQ$INITIALIZE, you must perform your own

preconnection parsing and supply your own preconnection lookup routine (TQGETPRECON) and your own exception handling routines (TQSETERH, TQGETERH, and an exception handler). Supplying your own version of TQ$INITIALIZE overrides the default initialization required by these default LRS procedures.

RTNULL VERSION

The null version of TQ$INITIALIZE sets up the default preconnections, but does no preconnection parsing and does not establish a default exception handler. It does initialize the null version of TQ$FILE$DESCRIPTOR. (Refer to the discussion of TQ$FILE$DESCRIPTOR.)

The null version of TQ$ESTART does nothing (i.e., if an exception handler is already established, it remains so).

The null version of TQ$PARSECL returns a pointer to TQ$DEFAULTPL.

The null version of TQ$DEFAULTPL is an empty list (no preconnections).

TQGETPRECON

DESCRIPTION

This procedure is called before every file open to look up any preconnection parameter that may have been specified for the file.

PROCEDURE INTERFACE

```
TQ$GET$PRECON:
    PROCEDURE (unit, l$filename$p, l$filename$length,
               p$filename$addr$p, precon$root) BYTE
        PUBLIC REENTRANT;
    DECLARE unit                BYTE,
            l$filename$p        POINTER,
            l$filename$length   BYTE,
            p$filename$addr$p   POINTER,
            precon$root         SELECTOR;
    DECLARE p$filename$addr
      BASED p$filename$addr$p   POINTER;
    DECLARE p$filename$length   BYTE;
    ;
    RETURN p$filename$length;
    END;
```

INPUT PARAMETERS

The input parameters depend on whether the calling program is written in FORTRAN-86/88 or Pascal-86/88:

- If l$filename$length is zero, then the calling program is written in FORTRAN-86/88 and unit contains the unit number of the file.
- Otherwise, the calling program is written in Pascal-86/88; l$filename$p points to a string containing the logical file name (Pascal-86/88 file variable) and l$filename$length contains the (non-zero) length of the string.

p$filename$addr$p points to the output item p$filename$addr.

precon$root indicates the beginning of the preconnection list to be used for converting the unit number or logical file name to a physical file name (actual path name). (This will help you if you design your LRS procedures to be reentrant, since the preconnection list will not be in memory that is local to this procedure.) Refer to the TQ$INITIALIZE procedure for information on how the preconnection list is built.

OUTPUT

This is a typed procedure (function). The value of the procedure is a BYTE that indicates the length of the physical file name (path name) that corresponds to the input unit number or logical file name. If no corresponding entry is found in the preconnection list or if your application does not use preconnection, the value returned by the procedure must be zero.

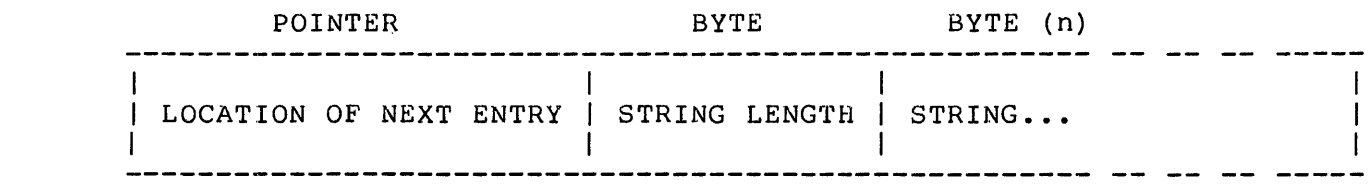
If a corresponding entry is found, place a pointer to the physical file name in p$filename$addr, and return the length of the physical file name.

REQUIRED EXCEPTION CODES

This procedure returns no explicit exception code. If no match is found in the preconnection list, the procedure must return a value of zero.

COMMENTS

If you do not supply your own version of TQ$INITIALIZE or TQ$PARSECL, the parameter precon$root of TQ$GET$PRECON points to a linked list, each entry of which has the following format:



For the last entry in the linked list, the LOCATION OF NEXT ENTRY is zero, or else the last entry is a null entry indicated by a STRING LENGTH of zero. Each (non-null) STRING contains one preconnection assignment in the same form as appears in the program invocation line; for example:

```
DATAFILE=:F1:CUST12.DAT          (Pascal-86/88)
UNIT4=:LP:                       (FORTRAN-86/88)
```

RTNULL VERSION

The version of TQGETPRECON in RTNULL.LIB always returns a length of zero.

TQ$EXIT

DESCRIPTION

This procedure terminates execution of the job. You must supply your own version of this procedure if your system does not have the UDI procedure DQ$EXIT. (The default version of TQ$EXIT calls DQ$EXIT.)

PROCEDURE INTERFACE

```
TQ$EXIT:
    PROCEDURE (termination$type) PUBLIC REENTRANT;
    DECLARE termination$type  WORD;
    ;
    END;
```

INPUT PARAMETERS

termination$type has a value of zero or one. Zero means normal termination; one indicates that termination is due to an exception.

OUTPUT

(None.)

REQUIRED EXCEPTION CODES

(None.)

COMMENTS

This procedure must not attempt to return control to the calling procedure.

RTNULL VERSION

The RTNULL.LIB version of TQ$EXIT causes the processor to halt.

Device Driver Procedures

The device driver procedures perform I/O operations for a file. When you supply your own device drivers, the code you write is concerned only with generalized aspects of I/O such as buffering, transferring bytes of data to and from physical devices, and recognizing record boundaries. You do not have to be concerned with assembling bytes into integers, reals, strings, or other item types. This work is done by higher levels of the run-time system.

FILE MARKERS

The model of file processing assumed by the LRI uses three kinds of file markers:

1. End-of-record mark
2. End-of-file mark
3. Current file position pointer

Your device drivers may implement these markers in any way that is appropriate for the files and devices you deal with. The following suggestions illustrate some of the possible implementations:

- The end-of-record mark may be a CR-LF (carriage return, line feed) pair in a text file, the inter-record gap on a magnetic tape file, or whatever form of record seperator is suitable for the file or device.

- For disk storage devices, the end-of-file mark may be a position pointer. Magnetic tape files might use a tape-mark as the end-of-file mark.

- The current file position pointer marks a specific character, which is not necessarily at the beginning or end of a record.

BUFFERING

No file buffering is performed in the run-time system above the LRI level. The default device drivers (i.e., those in the LRS supplied by Intel) provide buffering (except for text files and formatted output to non-seekable devices). If you are supplying your own device drivers, you must also provide whatever buffering is necessary for your application.

OPEN

DESCRIPTION

The run-time system calls open before performing any input or output on a file. The open routine makes the connection to a physical file, allocates buffers, sets the file pointer to the beginning of the file, and otherwise prepares the file for subsequent I/O operations.

The default open procedure uses the UDI procedures DQ$OPEN, DQ$ATTACH, DQ$DETACH, DQ$CREATE, DQ$ALLOCATE, DQ$FREE, DQ$SPECIAL, and DQ$GET$CONNECTION$STATUS. You must supply your own open procedure if your system is missing any of these interfaces.

PROCEDURE INTERFACE

```
open:
    PROCEDURE (fd, name$p, name$length, attribute,
               rec$length) WORD PUBLIC REENTRANT;
    DECLARE fd             SELECTOR,
            name$p         POINTER,
            name$length    BYTE,
            attribute      WORD,
            rec$length     WORD;
    DECLARE err$code       WORD;
    ;
    RETURN err$code;
    END;
```

INPUT PARAMETERS

fd identifies the file descriptor for the file to be affected.

name$p points to an area containing the path name of the file, and name$length contains its length in bytes. The path name comes either from the FORTRAN-86/88 or Pascal-86/88 module or from the TQGETPRECON routine. (Do not modify the contents of the area addressed by name$p. The run-time system assumes that the contents of this area remain unmodified.) If name$length is zero, the driver should assume that it is a scratch file (work file) to be deleted after the close operation.

The attribute WORD contains several bit parameters that provide more information about the file. Your open driver may either use or ignore these parameters, depending on their relevance to the files and devices the driver is designed to support. Some of the parameters may be relevant to other driver procedures that will be called later to operate on the file. In this case you should store the parameters in the user portion of the file descriptor for the file. The bit parameters are defined in table B-2. The bits are numbered from least significant (bit

zero) to most significant (bit 15).

rec$length contains the record length to be associated with the file. A value of zero indicates that the record length is variable.

Table B-2. Attribute Bit Items for Open

BITS	VALUE	INTERPRETATION
1,0		Status of file.
	00	Unknown status; file may or may not already exist. (This is the default setting and the only setting for Pascal-86/88.)
	01	New file; file must not already exist.
	10	Old file; file must already exist.
	11	Scratch file; file must be deleted on execution of close.
2		Access to file.
	0	Sequential access; seek driver routine will not be called. (This is the only setting for Pascal-86/88.)
	1	Direct access; seek driver routine may be called.
3		Format of file.
	0	Unformatted; file contains binary data.
	1	Formatted; file contains character data (typically ASCII-encoded and printable on line printer).
4		(Reserved.)
5		Record delimiter.
	0	None. The first character of a record is treated as any other character. (This is the only setting for Pascal-86/88.)
	1	The first character of every record is a vertical spacing control for a printer.

Table B-2. Attribute Bit Items for Open (cont'd.)

BITS	VALUE	INTERPRETATION
6		Interactivity.
	Ø	Not an interactive file.
	1	Possibly interactive file. The device should be treated as an interactive console (no buffering). For FORTRAN-86/88 CARRIAGE = CONSOLE files, this is used for the $ (dollar-sign) edit descriptor, which can suppress the record terminator in sequential formatted output statements. In Pascal-86/88 this is used to indicate TEXTFILE output.
7		Path check.
	Ø	No action required. (This is the only setting for Pascal-86/88 support.)
	1	Compare the path name supplied with this call to open against that associated with the open connection. If they are unequal, return the E$PATHEQ exception code without taking any other action. If they are equal, accept the new open attributes without changing the file pointer. This feature is used to support the FORTRAN-86/88 OPEN statement.
9,8		I/O mode.
	ØØ	Destructive write only.
	Ø1	Read only.
	1Ø	(Reserved.)
	11	Update. (FORTRAN-86/88 only.)
1Ø-15		(Reserved.)

OUTPUT

This is a typed procedure (function). The value of the procedure is a WORD (here called err$code) that indicates the result of calling this procedure.

REQUIRED EXCEPTION CODES

E$OK. For FORTRAN-86/88, also E$FEXIST (for a new file that already exists), E$FNEXIST (for an old file that does not exist), and E$PATHEQ (for failure of a path check).

COMMENTS

RTNULL VERSION

Causes the processor to halt.

CLOSE

DESCRIPTION

This procedure closes a file, flushing out any data that may be left in buffers and returning buffer space.

PROCEDURE INTERFACE

```
close:
    PROCEDURE (fd, dispose) WORD PUBLIC REENTRANT;
    DECLARE fd          SELECTOR,
            dispose     BYTE;
    DECLARE err$code    WORD;
    ;
    RETURN err$code;
    END;
```

INPUT PARAMETERS

fd identifies the file descriptor for the file to be affected.

dispose contains several bit items that specify what to do with the file. These bits are defined in table B-3. The bits are numbered from least significant (bit zero) to most significant (bit seven).

Table B-3. Disposition Parameters for Close

BITS	VALUE	INTERPRETATION
1,0		Disposition of file:
	00	Dispose as appropriate for status of file as specified in open. (For example, if work file, delete.) This is the only setting for Pascal-86/88.
	01	Do not delete the file.
	10	Delete the file.
	11	(Reserved.)
2-7		(Reserved.)

OUTPUT

This is a typed procedure (function). The value of the procedure is a WORD (here called err$code) that indicates the result of calling this procedure.

REQUIRED EXCEPTION CODES

E$OK

COMMENTS

Note that, when a file is closed, the space used by its file descriptor is not returned. The file descriptor is re-used by the run-time system if another file is opened, and TQ$FILE$DESCRIPTOR is not called before that open.

RTNULL VERSION

Causes the processor to halt.

READ

DESCRIPTION

This procedure reads a specified number of bytes from a file.

PROCEDURE INTERFACE

```
read:
    PROCEDURE (fd, buffer$p, count,
               actual$count$p) WORD PUBLIC REENTRANT;
    DECLARE fd              SELECTOR,
            buffer$p        POINTER,
            count           WORD,
            actual$count$p  POINTER;
    DECLARE actual$count BASED actual$count$p WORD;
    DECLARE err$code        WORD;
    ;
    RETURN err$code;
    END;
```

INPUT PARAMETERS

fd identifies the file descriptor for the file to be affected.

count specifies the number of bytes to be read.

buffer$p and actual$count$p contain the addresses of output items.

OUTPUT

Store the data read by this procedure at the location addressed by buffer$p.

Fill actual$count with the number of bytes actually read.

This is a typed procedure (function). The value of the procedure is a WORD (here called err$code) that indicates the result of calling this procedure.

REQUIRED EXCEPTION CODES

E$OK; E$EOF when end of file is encountered before count bytes are read; E$EOR when end of record is encountered before count bytes are read.

COMMENTS

The run-time system calls this routine to read all or part of a record. When only part of a record is read, several calls may be necessary to read the entire record. The E$EOR exception signals when the end of a record is reached. For files with fixed length records, the run-time system never tries to read more bytes than the fixed number defined for the record. However, this driver must be able to recognize the end of a record, and be able to position the file pointer to the beginning of the next record before the next read operation.

RTNULL VERSION

Causes the processor to halt.

WRITE

DESCRIPTION

This procedure writes a specified number of characters to a file.

PROCEDURE INTERFACE

```
write:
    PROCEDURE (fd, buffer$p, count) WORD PUBLIC REENTRANT;
    DECLARE fd         SELECTOR,
            buffer$p   POINTER,
            count      WORD;
    DECLARE err$code   WORD;
    ;
    RETURN err$code;
    END;
```

INPUT PARAMETERS

fd identifies the file descriptor for the file to be affected.

buffer$p points to the beginning of the data to be written.

count specifies the number of bytes to be written.

OUTPUT

This is a typed procedure (function). The value of the procedure is a WORD (here called err$code) that indicates the result of calling this procedure.

REQUIRED EXCEPTION CODES

E$OK

COMMENTS

The run-time system may call this procedure several times in order to complete an entire record, even if the records are fixed in length. When the record is complete, the run-time system calls the end$record driver to mark the end of the record. It is the responsibility of write to perform any output buffering that may be desired.

RTNULL VERSION

Causes the processor to halt.

SEEK

DESCRIPTION

This routine is called for files opened for direct access to position the file pointer before a read or write operation. (This routine is not used by the Pascal-86/88 run-time system.)

PROCEDURE INTERFACE

```
seek:
    PROCEDURE (fd, mode, high$offset, low$offset) WORD
        PUBLIC REENTRANT;
    DECLARE fd            SELECTOR,
            mode          BYTE,
            low$offset    WORD,
            high$offset   WORD;
    DECLARE err$code      WORD;
    ;
    RETURN err$code;
    END;
```

INPUT PARAMETERS

fd identifies the file descriptor for the file to be affected.

low$offset and high$offset together form a four-byte (DWORD) unsigned integer (here called offset) that represents either a position in the file or the number of bytes to move the file position pointer, depending on the setting of mode.

mode indicates the type of seek required. The values of mode are defined in table B-4.

Table B-4. Mode Parameters for Seek

VALUE	INTERPRETATION
Ø	Seek to the record number specified in offset. Note that the first record of a file is record number 1.
1	Move file pointer back by offset bytes within current record.
2	Set file pointer to offset within current record.
3	Move file pointer forward by offset bytes within current record.
4	Move file pointer to end of file.

OUTPUT

This is a typed procedure (function). The value of the procedure is a WORD (here called err$code) that indicates the result of calling this procedure.

REQUIRED EXCEPTION CODES

E$OK

COMMENTS

Modes 1 through 4 are not currently supported or required.

RTNULL VERSION

Causes the processor to halt.

SKIP

DESCRIPTION

This procedure moves the file pointer forward to the beginning of the next sequential record. The procedure is called when processing of a record is finished, even though only part of the record was read (as caused by a slash (/) in a FORTRAN-86/88 format or by READLN in Pascal-86/88).

PROCEDURE INTERFACE

```
skip:
    PROCEDURE (fd) WORD PUBLIC REENTRANT;
    DECLARE fd        SELECTOR;
    DECLARE err$code  WORD;
    ;
    RETURN err$code;
    END;
```

INPUT PARAMETERS

fd identifies the file descriptor for the file to be affected.

OUTPUT

This is a typed procedure (function). The value of the procedure is a WORD (here called err$code) that indicates the result of calling this procedure.

REQUIRED EXCEPTION CODES

E$OK; E$EOF if end of file is encountered.

COMMENTS

RTNULL VERSION

Causes the processor to halt.

END RECORD

DESCRIPTION

This procedure marks the end of a record. The run-time system calls this routine every time output to a record (even a fixed-length record) is completed (as caused by a slash format in FORTRAN-86/88 or by a Pascal-86/88 WRITELN).

PROCEDURE INTERFACE

```
end$record:
    PROCEDURE (fd) WORD PUBLIC REENTRANT;
    DECLARE fd        SELECTOR;
    DECLARE err$code  WORD;
    ;
    RETURN err$code;
    END;
```

INPUT PARAMETERS

fd identifies the file descriptor for the file to be affected.

OUTPUT

This is a typed procedure (function). The value of the procedure is a WORD (here called err$code) that indicates the result of calling this procedure.

REQUIRED EXCEPTION CODES

E$OK

COMMENTS

Your driver should mark the file in a manner appropriate for the particular device and access mode being used. (For example, for text files you may want to write carriage return and line feed characters.) For files with fixed-length records, the driver may either increment the record pointer or pad the balance of the record with a distinguishable character. A call to this driver implies that the program has terminated output to the record, and that the rest of the record is either undefined or defined by this driver.

RTNULL VERSION

Causes the processor to halt.

REWIND

DESCRIPTION

This routine sets the file pointer to the beginning of file and performs any device control functions necessary to rewind the physical device.

PROCEDURE INTERFACE

```
rewind:
    PROCEDURE (fd, mode) WORD PUBLIC REENTRANT;
    DECLARE fd         SELECTOR,
            mode       BYTE;
    DECLARE err$code   WORD;
    ;
    RETURN err$code;
    END;
```

INPUT PARAMETERS

fd identifies the file descriptor for the file to be affected.

mode specifies access rights to the file after it is rewound. The values correspond to those of bits 8 and 9 of the attribute parameter of the open driver:

VALUE	INTERPRETATION
Ø	Destructive write only.
1	Read only.
2	(Reserved.)
3	Update.

OUTPUT

This is a typed procedure (function). The value of the procedure is a WORD (here called err$code) that indicates the result of calling this procedure.

REQUIRED EXCEPTION CODES

E$OK

COMMENTS

RTNULL VERSION

Causes the processor to halt.

BACKSPACE

DESCRIPTION

This procedure positions the file pointer to the beginning of the previous record. (It is not used for Pascal-86/88 support.)

PROCEDURE INTERFACE

```
backspace:
    PROCEDURE (fd) WORD PUBLIC REENTRANT;
    DECLARE fd        SELECTOR;
    DECLARE err$code  WORD;
    ;
    RETURN err$code;
    END;
```

INPUT PARAMETERS

fd identifies the file descriptor for the file to be affected.

OUTPUT

This is a typed procedure (function). The value of the procedure is a WORD (here called err$code) that indicates the result of calling this procedure.

REQUIRED EXCEPTION CODES

E$OK

COMMENTS

RTNULL VERSION

Causes the processor to halt.

END FILE

DESCRIPTION

This procedure marks the current position of the file pointer as the end of the file. (This procedure is not used for Pascal-86/88 support.)

PROCEDURE INTERFACE

```
end$file:
    PROCEDURE (fd) WORD PUBLIC REENTRANT;
    DECLARE fd        SELECTOR;
    DECLARE err$code  WORD;
    ;
    RETURN err$code;
    END;
```

INPUT PARAMETERS

fd identifies the file descriptor for the file to be affected.

OUTPUT

This is a typed procedure (function). The value of the procedure is a WORD (here called err$code) that indicates the result of calling this procedure.

REQUIRED EXCEPTION CODES

COMMENTS

If there is data in the file beyond the location indicated by the current file pointer, that data is truncated.

RTNULL VERSION

Causes the processor to halt.

Refer also to the TQ$INITIALIZE procedure.

DESCRIPTION

This procedure establishes the address of the exception handler to be used in processing all subsequent exceptional conditions. The run-time system calls this procedure from TQ$INITIALIZE once at the start of program execution to establish the default exception handler. You must supply your own version of TQSETERH if your system does not have the UDI procedure DQ$TRAP$EXCEPTION.

PROCEDURE INTERFACE

```
TQ$SET$ERH:
    PROCEDURE (handler$addr) PUBLIC REENTRANT;
    DECLARE handler$addr  POINTER;
    ;
    END;
```

INPUT PARAMETERS

handler$addr is the address of the new exception handler.

OUTPUT

(None.)

REQUIRED EXCEPTION CODES

(None.)

COMMENTS

The handler address should be stored so that it is accessible by TQGETERH. At any one time, only one exception handler can be in effect for the set of modules linked to one LRS; therefore, handler$addr may be stored in local memory without limiting reentrancy of the LRS.

In multitasking systems a contention condition can occur wherein one task has updated only one word of the handler address when interrupted by another task that needs to read the entire address. If this condition can occur in your application, you must provide for synchronization to prevent it from

occurring.

If you supply your own version of TQSETERH, you should do the same for TQGETERH.

RTNULL VERSION

The version of TQSETERH in RTNULL.LIB returns without actually establishing an exception handler. Refer also to the null version of TQGETERH.

TQGETERH

DESCRIPTION

This procedure is called by any procedure in the run-time system that has an exception condition to report. TQGETERH delivers the address of the current exception handler so that the calling procedure can formulate an indirect call to that error handler. You must supply your own version of TQGETERH if your system does not have the UDI procedure DQGETEXCEPTION$HANDLER.

PROCEDURE INTERFACE

```
TQ$GET$ERH:
    PROCEDURE (handler$addr$p) PUBLIC REENTRANT;
    DECLARE handler$addr$p POINTER;
    DECLARE handler$addr BASED handler$addr$p POINTER;
    ;
    END;
```

INPUT PARAMETERS

handler$addr$p contains the address of the output item handler$addr.

OUTPUT

Fill handler$addr with the address of the current exception handler (as provided earlier by TQSETERH).

REQUIRED EXCEPTION CODES

(None.)

COMMENTS

The pointer fetched by TQGETERH has two primary uses:

1. It is used to formulate an indirect call to the current exception handler. The run-time system always calls TQGETERH before calling the exception handler to ensure that the address used in the call refers to the most recently established handler.

2. It can be saved to be used later in a call to TQSETERH to restore the current exception handler after another handler has been temporarily substituted.

If you provide your own version to TQGETERH, you should do the same for TQSETERH.

RTNULL VERSION

The version of TQGETERH in RTNULL.LIB causes the processor to halt, since the null version of TQSETERH does not establish an exception handler.

Memory Management Procedures

Several data structures of the run-time system have dynamic memory requirements; namely:

- Preconnection list
- File descriptors
- File buffers
- Pascal-86/88 heap

The run-time system takes two different approaches to allocation of space for these structures.

1. The first three of these structures (the preconnection list, file descriptors, and file buffers) are managed by LRS procedures. When you provide your own versions of the procedures that deal with these structures, you also take on the responsibility for allocating memory for those structures.

2. The Pascal-86/88 heap is not managed by LRS procedures but rather by procedures at higher levels of the run-time system. The heap-management procedures call on LRS procedures for allocation of memory space for the heap. Therefore, while you cannot provide your own procedures for heap management, you can provide your own procedures for allocation of space for the heap.

The Pascal-86/88 heap is used for storage of dynamic variables. You allocate space for a dynamic variable by using the Pascal-86/88 procedure NEW; you free dynamic variable space by using DISPOSE.

The run-time system provides two memory managers to control the heap. The Pascal-86/88 program automatically determines which memory manager to use depending on the model of segmentation you select for the module being compiled. Pointer types defined under the SMALL model (with -CONST IN DATA-) are 16-bit pointers and require that storage be allocated in the data group DGROUP. Pointer types defined under other models use 32-bit pointers and have no restriction on which segment or group they point to. Both memory managers can be used in one program if the program contains modules compiled both under SMALL (-CONST IN DATA-) and under other models.

The SMALL model memory manager calls the LRS procedure TQGETSMALL$HEAP to determine the size and location of the small heap. The other memory manager (called the large model memory manager) calls the LRS procedures TQ$ALLOCATE and TQ$FREE to dynamically control space for the heap. These LRS procedures are explained in the following sections.

Note that neither of the memory managers is reentrant. This means that no two tasks in a multitasking environment may invoke the same memory manager concurrently. One way to avoid this problem is to provide synchronization in your Pascal-86/88 program for the calls to NEW and DISPOSE, to ensure that no two calls to either of these procedures are active at the same time. This method works for either the large or the SMALL-model memory manager.

TQ$ALLOCATE

DESCRIPTION

This procedure is invoked only by the large model memory manager to expand the Pascal-86/88 heap. TQ$ALLOCATE allocates an additional area of memory for the heap. You need to supply your own version of TQ$ALLOCATE if your system does not include the UDI procedure DQ$ALLOCATE.

PROCEDURE INTERFACE

```
TQ$ALLOCATE:
    PROCEDURE (size, err$code$p) SELECTOR PUBLIC REENTRANT;
    DECLARE size         WORD,
            err$code$p   POINTER;
    DECLARE err$code BASED err$code$p WORD;
    DECLARE seg$addr     SELECTOR;
    ;
    RETURN seg$addr;
    END;
```

INPUT PARAMETERS

size specifies the number of contiguous bytes of memory that the run-time system needs to obtain.

err$code$p contains the address for the output item err$code.

OUTPUT

Fill err$code with a code that indicates the result of executing this procedure.

This is a typed procedure (function). The value of the procedure is a SELECTOR that identifies a block of memory for the run-time system to add to the heap.

REQUIRED EXCEPTION CODES

EOK, EMEM if there is not enough memory

COMMENTS

The large model memory manager performs buffering between NEW requests for heap space and calls to TQ$ALLOCATE. TQ$ALLOCATE is called only to satisfy NEW requests for large dynamic structures and when no more heap space is

available for smaller NEW requests.

If you supply your own version of TQ$ALLOCATE, you should also supply your own version of TQ$FREE.

RTNULL VERSION

The version of TQ$ALLOCATE in RTNULL.LIB returns E$MEM.

TQ$FREE

DESCRIPTION

This procedure is invoked only by the large model memory manager to return a memory segment when Pascal-86/88 DISPOSE requests have eliminated all the dynamic variables previously stored in that segment. TQ$FREE returns memory from the heap to the system. You need to supply your own version of TQ$FREE only if your system does not have the UDI procedure DQ$FREE.

PROCEDURE INTERFACE

```
TQ$FREE:
    PROCEDURE (seg$addr, err$code$p) PUBLIC REENTRANT;
    DECLARE seg$addr    SELECTOR,
            err$code$p  POINTER;
    DECLARE err$code BASED err$code$p WORD;
    ;
    END;
```

INPUT PARAMETERS

seg$addr identifies the memory block to be freed. This can only be a block that was previously allocated via TQ$ALLOCATE.

err$code$p contains the address for the output item err$code.

OUTPUT

Fill err$code with a code indicating the result of executing this procedure.

REQUIRED EXCEPTION CODES

E$OK

COMMENTS

If you supply your own version of TQ$FREE, you should also supply a TQ$ALLOCATE.

RTNULL VERSION

The version of TQ$FREE in RTNULL.LIB returns E$OK without doing anything.

TQGETSMALL$HEAP

DESCRIPTION

Pascal-86/88 programs compiled according to the SMALL model of segmentation (-CONST IN DATA-) do not use TQ$ALLOCATE and TQ$FREE to dynamically allocate space for the heap. Instead, space is allocated from a static area in the program's data group (DGROUP). This procedure fetches the size and location of that memory area. The run-time system calls this procedure the first time an allocation request is made via the Pascal-86/88 built-in procedure NEW. (Not used for FORTRAN-86/88 support.)

PROCEDURE INTERFACE

```
TQ$GET$SMALL$HEAP:
    PROCEDURE (offset$p, size$p, err$code$p) PUBLIC REENTRANT;
    DECLARE offset$p    POINTER,
            size$p      POINTER,
            err$code$p  POINTER;
    DECLARE offset   BASED offset$p   WORD,
            size     BASED size$p     WORD,
            err$code BASED err$code$p WORD;
    ;
    END;
```

INPUT PARAMETERS

offset$p, size$p, and err$code$p all point to areas used for output (offset, size, and err$code, respectively).

OUTPUT

Fill offset with the offset within DGROUP of the beginning of the heap.

Fill size with the size of the heap in bytes.

Fill err$code with a code indicating the result of executing this procedure.

REQUIRED EXCEPTION CODES

EOK, ESUPPORT

COMMENTS

For the heap in SMALL-model programs, Pascal-86/88 creates a segment called

MEMORY, which is located within DGROUP. The size of this segment can be adjusted at link time. The size at run time must be at least 13 bytes.

The default version of TQGETSMALL$HEAP works with run-time locatable (RTL) object modules. It calls the UDI procedure DQGETSIZE to find the size of DGROUP and calculates the size of the heap. This is possible since the relocatable loader calls the UDI procedure DQ$ALLOCATE to get space for the segments. This approach does not work if the UDI procedures DQ$ALLOCATE and DQ$GET$SIZE are not present or if the object module has been absolutely located by LOC86. In these cases you must code your own version of TQGETSMALL$HEAP.

In your version of TQGETSMALL$HEAP, the heap does not necessarily have to be located in the MEMORY segment, but it does have to be at least 13 bytes long. Your version should be linked before the run-time libraries.

RTNULL VERSION

(None.)

EXAMPLE USAGE

```
            NAME    DQGETSMALLHEAP
;
;This procedure demonstrates overriding the default
;TQGETSMALLHEAP in the Pascal-86/88 Logical Record System
;
;Parameters:
;   OFFSET_PTR     : POINTER to the WORD in which the offset
;                    of the heap from the start of DGROUP is
;                    returned.
;   SIZE_PTR       : POINTER to the word in which the heap
;                    size is returned.
;   EXCEPTION_PTR  : POINTER to the WORD in which the
;                    exception status is returned.
;
DGROUP      GROUP  CONST,DATA,MEMORY,MYHEAP
;
; Declare DGROUP segments
CONST       SEGMENT PUBLIC 'DATA'
CONST       ENDS
DATA        SEGMENT PUBLIC 'DATA'
DATA        ENDS
MEMORY      SEGMENT MEMORY 'MEMORY'
MEMORY      ENDS
;
; Define small heap
SMALLHEAPSIZE
&           EQU     2000H
MYHEAP      SEGMENT PUBLIC 'DATA'
SMALLHEAP   DB      SMALLHEAPSIZE DUP(?)
MYHEAP      ENDS
;
; Define parameters
```

```
OFFSET_PTR EQU     DWORD PTR [BP+14]
SIZE_PTR   EQU     DWORD PTR [BP+1Ø]
EXCEPTION_PTR
&          EQU     DWORD PTR [BP+6]
PARAM_SIZE EQU     12
;
; Begin procedure
           ASSUME CS:MYCODE,SS:DGROUP
           PUBLIC TQGETSMALLHEAP
MYCODE     SEGMENT PUBLIC 'CODE'
TQGETSMALLHEAP
&          PROC   FAR
           PUSH   BP              ; Save old BP value
           MOV    BP,SP
;
; Store heap offset
           LES    SI,OFFSET_PTR
           MOV    WORD PTR ES:[SI],OFFSET DGROUP:SMALLHEAP
;
; Store heap size
           LES    SI,SIZE_PTR
           MOV    WORD PTR ES:[SI],SMALLHEAPSIZE
;
; Set exception flag to Ø
           LES    SI,EXCEPTION_PTR
           MOV    WORD PTR ES:[SI],Ø
;
; Return
           POP    BP
           RET    PARAM_SIZE
TQGETSMALLHEAP
&          ENDP
MYCODE     ENDS
           END
```

EXAMPLE DEVICE DRIVERS

The remaining pages of this chapter contain an example of a set of user-written device drivers. This set of drivers is intended for use with a receive-only CRT display. The code in these driver routines does not make any calls to UDI or system procedures; therefore, the drivers are suitable for use without an operating system.

In the configuration for which this example is designed, the CRT is connected to the serial port of the RPB-86 board of an Intellec Series III Microcomputer Development System. The RPB-86 board is similar to an iSBC 86/12A Single Board Computer. The CRT driver procedures control the CRT with the aid of an Intel 8251 Universal Synchronous/Asynchronous Receiver/Transmitter (USART) and an Intel 8253 Programmable Interval Timer, both of which reside on the board. Refer to the iSBC 86/12A Single Board Computer Hardware Reference Manual and the Peripheral Design Handbook for more information on using these devices.

Following the drivers is a simple Pascal-86/88 program that exercises the drivers. Normally, device drivers are implemented at the LRS level only in cases where the target environment does not include an operating system. For testing purposes, however, this example uses console I/O as implemented by the operating system via the default LRS. Note that, except for its use of the special device name :T1:, the Pascal-86/88 program does not "know" about the special device drivers.

An example of LINK86 commands to create an executable module on an Intellec Series III Microcomputer Development System is:

```
RUN LINK86 SHORT1.OBJ, &
  P86RNØ.LIB, &
  P86RN1.LIB, &
  TQNEW.OBJ,  &
  P86RN2.LIB, &
  P86RN3.LIB, &
  NULL87.LIB, &
  LARGE.LIB TO SHORT1.86 BIND
```

Since TQ$NEW.OBJ appears before the default LRS in P86RN2.LIB and P86RN3.LIB, LINK86 links in the new version of TQ$DEVICE instead of the default version. Both the new CRT driver and the default drivers are linked; the default drivers are still needed to support console I/O for the example Pascal-86/88 program.

```
ISIS-II PL/M-86 V2.1 COMPILATION OF MODULE TQCRTMODULE
OBJECT MODULE PLACED IN :F1:TQNEW.OBJ
COMPILER INVOKED BY:  PLM86 :F1:TQNEW.SRC


                $PAGEWIDTH(73)
                $LARGE
    1           TQ$CRT$MODULE: DO;

    2   1        DECLARE TRUE  LITERALLY '0FFH',
                         FALSE LITERALLY '0',
                         E$OK  LITERALLY '0';

    3   1         TQDEVICE:
                    PROCEDURE(NAME$PTR,NAME$LENGTH,
                              DRIVER$TABLE$PTR) WORD PUBLIC;
    4   2             DECLARE (NAME$PTR,DRIVER$TABLE$PTR) POINTER;
    5   2             DECLARE NAME$LENGTH BYTE;
    6   2             DECLARE (MATCH$FLAG,I) BYTE;
    7   2             DECLARE DRIVER$BASE BASED DRIVER$TABLE$PTR POINTER;
    8   2             DECLARE DRIVER$NAME BASED NAME$PTR(1) BYTE;
    9   2             DECLARE NAME(4) BYTE DATA(':T1:');
   10   2             DECLARE DRIVER$TABLE(10) POINTER;

                   /*  IF SPECIAL LINE PRINTER DEVICE
                           THEN REASSIGN DEVICE DRIVER TABLE  */

   11   2             MATCH$FLAG = TRUE;
   12   2             IF NAME$LENGTH <> 4 THEN MATCH$FLAG = FALSE;
   14   2             IF MATCH$FLAG = TRUE THEN DO I=0 TO 3;
   16   3               IF NAME(I) <> DRIVER$NAME(I) THEN MATCH$FLAG = FALSE;
   18   3             END;
   19   2             IF MATCH$FLAG = TRUE THEN DO;
   21   3               DRIVER$BASE = @DRIVER$TABLE;
   22   3               DRIVER$TABLE(0) = @OPEN$NEW$DEVICE;
   23   3               DRIVER$TABLE(1) = @CLOSE$NEW$DEVICE;
   24   3               DRIVER$TABLE(2) = @READ$NEW$DEVICE;
   25   3               DRIVER$TABLE(3) = @WRITE$NEW$DEVICE;
   26   3               DRIVER$TABLE(4) = @SEEK$NEW$DEVICE;
   27   3               DRIVER$TABLE(5) = @NEW$MOVE;
   28   3               DRIVER$TABLE(6) = @NEW$MARK$END;
   29   3               DRIVER$TABLE(7) = @NEW$REWIND;
   30   3               DRIVER$TABLE(8) = @NEW$BACKSPACE;
   31   3               DRIVER$TABLE(9) = @NEW$END$FILE;
   32   3             END;
   33   2             RETURN E$OK;
   34   2           END TQDEVICE;

                   /*************************************************/
                   /*************************************************/
                   /*          D E V I C E   D R I V E R S          */
                   /*************************************************/
                   /*************************************************/

                   /*************************************************/
                   /*      OPEN A FILE                              */
```

```
                /*************************************************************/
 35    1         OPEN$NEW$DEVICE:
                  PROCEDURE(FD$SEG,NAME$PTR,NAME$LENGTH,
                            ATTRIB,REC$LENGTH) WORD PUBLIC;
 36    2          DECLARE (FD$SEG,ATTRIB,REC$LENGTH) WORD;
 37    2          DECLARE NAME$PTR POINTER;
 38    2          DECLARE NAME$LENGTH BYTE;
 39    2          DECLARE CNTL$PORT$8253 LITERALLY '0D6H' ;
                            /* 3-4 OF 86/12A HRM */

 40    2          DECLARE COUNT$REG$8253 LITERALLY '0D4H' ;
 41    2          DECLARE STAT$PORT$8251 LITERALLY '0DAH' ;
 42    2          DECLARE CONTROL$BYTE$8253 LITERALLY '0B6H' ;
                            /* COUNTER 2 */
                            /* READ/LOAD LEAST SIG. BYTE FIRST */
                            /* MODE 3 - SQUARE WAVE RATE GENERATOR */
                            /* 16 BIT BINARY COUNTER */

 43    2          DECLARE RESET$8251$TO$MODE LITERALLY '40H' ;
 44    2          DECLARE CONTROL$BYTE$8251  LITERALLY '4FH' ;
                            /* BAUD RATE - ASYN X64  */
                            /* 8 BIT CHARACTER LENGTH       */
                            /* NO PARITY                    */
                            /* ONE STOP BIT                 */

 45    2          DECLARE BAUD$CODE   LITERALLY '0040H' ;
 46    2          DECLARE ENABLE$8251 LITERALLY '27H';
                            /* TRANSMIT ENABLE      */
                            /* RECEIVE ENABLE       */
                            /* DTR BAR = 0   (READY) */
                            /* RTS BAR = 0   (READY) */

                /* FILE DESCRIPTOR HAS 16 BYTES TO STORE NECESSARY
                   INFORMATION.  VAR WILL CONTAIN THE POINTER TO THE
                   FILE DESCRIPTOR.  FD$SEG FORMS THE BASE PORTION OF
                   THE ADDRESS POINTING TO THE FILE DESC.
                   STORE INFO REGARDING TYPE OF FILE TO BE USED WHEN
                   MARKING END OF RECORD.                              */

 47    2          DECLARE VAR POINTER;
 48    2          DECLARE VAR$DESCRIPTOR STRUCTURE (OFFSET WORD, BASE WORD)
                      AT (@VAR);
 49    2          DECLARE FILE$DESCRIPTOR BASED VAR STRUCTURE(
                                                 AVAILABLE(16) BYTE,
                                                 RESERVE(32) BYTE);

 50    2          VAR$DESCRIPTOR.OFFSET = 0;
 51    2          VAR$DESCRIPTOR.BASE = FD$SEG;
 52    2          FILE$DESCRIPTOR.AVAILABLE(0)=(ATTRIB AND 08H);
                            /* FORM OF FILE */


                /* INITIALIZE THE INTERVAL TIMER (8253)  */

 53    2          OUTPUT (CNTL$PORT$8253) = CONTROL$BYTE$8253;
 54    2          OUTPUT (COUNT$REG$8253) = LOW(BAUD$CODE);
 55    2          OUTPUT (COUNT$REG$8253) = HIGH(BAUD$CODE);
```

```
                /* INITIALIZE THE USART (8251).
                   SEND 3 ZERO'S TO CLEAR FIRST.  */

 56   2           OUTPUT (STAT$PORT$8251) = ØH;
 57   2           OUTPUT (STAT$PORT$8251) = ØH;
 58   2           OUTPUT (STAT$PORT$8251) = ØH;

 59   2           OUTPUT (STAT$PORT$8251) = RESET$8251$TO$MODE;
 6Ø   2           OUTPUT (STAT$PORT$8251) = CONTROL$BYTE$8251;
 61   2           OUTPUT (STAT$PORT$8251) = ENABLE$8251;

                /* RETURN STATUS OF ZERO TO INDICATE SUCCESSFUL OPEN */
 62   2           RETURN E$OK;
 63   2           END OPEN$NEW$DEVICE;

                /*********************************************************/
                /*     CLOSE A FILE                                      */
                /*********************************************************/
 64   1           CLOSE$NEW$DEVICE: PROCEDURE(FD$SEG,DISPOSE) WORD PUBLIC;
 65   2           DECLARE FD$SEG WORD;
 66   2           DECLARE DISPOSE BYTE;
 67   2           RETURN E$OK;
 68   2           END CLOSE$NEW$DEVICE;

                /*********************************************************/
                /*      READ A BLOCK                                     */
                /*********************************************************/
 69   1           READ$NEW$DEVICE:
                  PROCEDURE(FD$SEG,BUFFER,COUNT,ACTUAL$PTR) WORD PUBLIC;
 7Ø   2           DECLARE (FD$SEG,COUNT) WORD;
 71   2           DECLARE (BUFFER,ACTUAL$PTR) POINTER;
 72   2           RETURN E$OK;
 73   2           END READ$NEW$DEVICE;

                /*********************************************************/
                /*      WRITE A BLOCK                                    */
                /*********************************************************/
 74   1           WRITE$NEW$DEVICE:
                  PROCEDURE (FD$SEG,BUFFER,COUNT) WORD PUBLIC;
 75   2           DECLARE (FD$SEG,COUNT) WORD;
 76   2           DECLARE BUFFER POINTER;
 77   2           DECLARE STATUS BYTE;
 78   2           DECLARE I WORD;
 79   2           DECLARE BUF BASED BUFFER(1) BYTE;
 8Ø   2           DECLARE STAT$PORT$8251 LITERALLY 'ØDAH';
 81   2           DECLARE DATA$PORT$8251 LITERALLY 'ØD8H';

 82   2           DO I=Ø TO COUNT-1;
 83   3            DO WHILE ((INPUT(STAT$PORT$8251)  AND 4H) = Ø);
                             /* CHECK FOR TxEMPTY  */
 84   4            END;
 85   3            OUTPUT(DATA$PORT$8251) = BUF(I);
 86   3            END;
 87   2           RETURN E$OK;
 88   2           END WRITE$NEW$DEVICE;
```

```
                /*************************************************************/
                /*      RECORD SEEK                                          */
                /*************************************************************/
 89    1          SEEK$NEW$DEVICE:
                  PROCEDURE(FD$SEG,MODE,HIGH$OFFSET,
                                        LOW$OFFSET) WORD PUBLIC;
 90    2          DECLARE (FD$SEG,HIGH$OFFSET,LOW$OFFSET) WORD;
 91    2          DECLARE MODE BYTE;
 92    2          RETURN E$OK;
 93    2          END SEEK$NEW$DEVICE;

                /*************************************************************/
                /*      MOVE FORWARD                                         */
                /*************************************************************/
 94    1          NEW$MOVE: PROCEDURE(FD$SEG) WORD PUBLIC;
 95    2          DECLARE FD$SEG WORD;
 96    2          RETURN E$OK;
 97    2          END NEW$MOVE;

                /*************************************************************/
                /*      MARK RECORD END                                      */
                /*************************************************************/
 98    1          NEW$MARK$END: PROCEDURE(FD$SEG) WORD PUBLIC;
 99    2          DECLARE FD$SEG WORD;
100    2          DECLARE DATA$PORT$8251 LITERALLY '0D8H';
101    2          DECLARE STAT$PORT$8251 LITERALLY '0DAH';
102    2          DECLARE CR LITERALLY '0DH';
103    2          DECLARE LF LITERALLY '0AH';

                /* FILE DESCRIPTOR HAS 16 BYTES TO STORE NECESSARY
                   INFORMATION.  VAR WILL CONTAIN THE POINTER TO THE FILE
                   DESCRIPTOR. FD$SEG FORMS THE BASE PORTION OF THE ADDRESS
                   POINTING TO THE FILE DESC.  RETRIEVE INFO REGARDING TYPE
                   OF FILE TO KNOW HOW TO END A RECORD.                    */

104    2          DECLARE VAR POINTER;
105    2          DECLARE VAR$DESCRIPTOR
                            STRUCTURE (OFFSET WORD, BASE WORD) AT (@VAR);
106    2          DECLARE FILE$DESCRIPTOR BASED VAR STRUCTURE(
                                                AVAILABLE(16) BYTE,
                                                RESERVE(32) BYTE);

107    2          VAR$DESCRIPTOR.OFFSET = 0;
108    2          VAR$DESCRIPTOR.BASE = FD$SEG;
109    2          IF (FILE$DESCRIPTOR.AVAILABLE(0) = 08H)  THEN DO;

                  /* WRITE CARRIAGE RETURN AND LINE FEED
                     FOR FORMATTED FILES.                  */

111    3            DO WHILE ((INPUT(STAT$PORT$8251) AND 4) = 0);
112    4            END;
113    3            OUTPUT(DATA$PORT$8251) = CR;
114    3            DO WHILE ((INPUT(STAT$PORT$8251) AND 4) = 0);
115    4            END;
116    3            OUTPUT(DATA$PORT$8251) = LF;
117    3          END;
118    2          RETURN E$OK;
```

```
 119   2          END NEW$MARK$END;

                /*************************************************************/
                /*      REWIND A FILE                                        */
                /*************************************************************/
 120   1          NEW$REWIND: PROCEDURE(FD$SEG,MODE) WORD PUBLIC;
 121   2          DECLARE FD$SEG WORD;
 122   2          DECLARE MODE BYTE;
 123   2          RETURN E$OK;
 124   2          END NEW$REWIND;

                /*************************************************************/
                /*      BACKSPACE                                            */
                /*************************************************************/
 125   1          NEW$BACKSPACE: PROCEDURE(FD$SEG) WORD PUBLIC;
 126   2          DECLARE FD$SEG WORD;
 127   2          RETURN E$OK;
 128   2          END NEW$BACKSPACE;

                /*************************************************************/
                /*       END FILE                                            */
                /*************************************************************/
 129   1          NEW$END$FILE: PROCEDURE(FD$SEG) WORD PUBLIC;
 130   2          DECLARE FD$SEG WORD;
 131   2          RETURN E$OK;
 132   2          END NEW$END$FILE;

 133   1        END TQ$CRT$MODULE;


MODULE INFORMATION:

     CODE AREA SIZE      = 0258H    600D
     CONSTANT AREA SIZE  = 0000H      0D
     VARIABLE AREA SIZE  = 0035H     53D
     MAXIMUM STACK SIZE  = 000EH     14D
     250 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

```
Source File: SHORT1.PAS
Object File: SHORT1.OBJ
Controls Specified:  DEBUG.

STMT LINE NESTING       SOURCE TEXT: SHORT1.PAS
   1    1  Ø  Ø         PROGRAM SHORT(INPUT,OUTPUT);
   2    2  Ø  Ø         VAR I:INTEGER;
   3    3  Ø  Ø             A: CHAR;
   4    4  Ø  Ø             NEWDEVICE:TEXT;
   5    5  Ø  Ø
                        BEGIN
   5    7  Ø  1
                            REWRITE(NEWDEVICE,':T1:');
   6    9  Ø  1
                            WRITE('INPUT AN INTEGER ');
   7   11  Ø  1             READLN(I);
   8   12  Ø  1
                            WRITE('INPUT A CHARACTER ');
   9   14  Ø  1             READLN(A);
  1Ø   15  Ø  1
                        WRITELN;
  11   17  Ø  1         WRITELN('THE CHARACTER YOU INPUT IS ',A);
  12   18  Ø  1         WRITELN(NEWDEVICE,'THE CHARACTER YOU INPUT IS ',A);
  13   19  Ø  1         WRITELN('THE INTEGER YOU INPUT IS ',I);
  14   2Ø  Ø  1         WRITELN(NEWDEVICE,'THE INTEGER YOU INPUT IS ',I)
                        END.
```


Summary Information:

PROCEDURE	OFFSET	CODE SIZE		DATA SIZE		STACK SIZE	
SHORT	Ø06CH	Ø183H	387D	Ø01BH	27D	Ø00EH	14D
Total		Ø1EFH	495D	Ø01BH	27D	Ø042H	66D

```
  21 Lines Read.
   Ø Errors Detected.
 39% Utilization of Memory.
```

INDEX

%VAL, 1-6
:BB:, 5-2, A-3, A-34
:CI:, 5-2, 6-8, 8-3, 9-3, A-2, A-33, A-34, A-38, A-39, B-13
:CO:, 6-8, 8-3, 9-3, A-2, A-22, A-33, A-34, B-13
:LP:, 5-2, 6-4, 6-8, 7-3, A-3, A-22, A-34
:TI:, A-3, A-33
:TO:, A-3, A-33
:WORK:, 5-2, 6-3, 8-3, A-3, A-22
8Ø85, 5-1
8Ø87 Emulator, 3-1, 3-2, 3-6, 3-7, 3-14, 6-2
8Ø87 Numerics Data Processor, 1-7, 2-2, 2-4, 3-1, 3-2, 3-5 thru 3-1Ø, 5-4 thru 5-6, 6-2, 9-1
8Ø87 Support Library, 3-1, 3-2, 3-5 thru 3-1Ø, 3-12 thru 3-14, 4-2, 8-2, 9-1
8Ø87.LIB, 3-7 thru 3-1Ø
8251 Universal Synchronous/Asynchronous Receiver/Transmitter (USART), B-49
8253 Programmable Interval Timer, B-49
8259A Programmable Interrupt Controller, 5-4, 5-6, 6-6

allocate (See memory management)
American National Standards Institute (ANSI), 1-1
ASM-86, Chapter 1, 3-6, 3-7, 3-1Ø, 3-14, 4-4
avoidable exception conditions, A-5, A-46, A-48

backspace, B-2, B-6, B-36
buffers and buffering, A-2, A-3, A-33, A-37, A-39, A-42, B-18, B-19, B-23, B-27, B-42, B-43

cancel (See termination)
CEL87.LIB, 3-7 thru 3-1Ø
close, 8-3, A-24, A-43, B-2, B-6, B-9, B-2Ø, B-23, B-24
command tail, 8-2, 8-3, A-1, A-12, A-16
command line, 4-1, 9-2, A-1, A-12, A-15, B-4
command line interpreter (CLI), A-13, A-16
COMMON, 1-6
COMPAC.LIB, 5-2
compiling, 5-7, 6-8, 6-9, 7-4
condition code (See exception code)
configuration
  iRMX 86 Operating System, 6-6
  iRMX 88 application, 7-4
  run-time system, 3-1
connection, 4-1, 5-2, 5-3, 6-3, 6-4, 7-2, 7-3, 8-3, A-1, A-2, A-7, A-22 thru A-25, A-33, A-37, A-38, A-4Ø, A-41, A-43, B-19, B-21
console, 3-3, 4-1, 5-3, 5-7, 6-7, 8-3, A-2, A-38, A-39, B-3, B-21, B-49
contention, B-38
control-C, 5-3, A-39
control-D, 5-3, A-39
control-Z, A-39
CREDIT text editor, 2-1
CRT display, A-39, B-49
custom system, 2-1, 2-2, 2-4

data types, 1-4, 1-5, 3-7, A-7, B-5
DCON87.LIB, 3-7, 3-9, 3-10
DEBUG control, 2-4, 2-5
DEBUG-86, 2-1, 2-3, 2-4
debuggers and debugging, Chapter 2, A-39
  (See also DEBUG-86, iAPX 86,88 Monitor Program,
  iRMX 86 System Debug Monitor, iRMX 86 System Debugger,
  and symbolic debugging)
delimiters, A-13, A-31
denormalized operands, 3-8
device driver, 3-5, B-2, B-5 thru B-8, B-10, B-11, B-13,
    B-18 thru B-37, B-49
DGROUP, B-42, B-46, B-47
disk addressing, 8-3
DISPOSE, B-42, B-45
DQ$ALLOCATE, 6-3, 6-5, 7-1, 8-2, A-17, B-8, B-19, B-43, B-47
DQ$ATTACH, 5-2, 6-3, 7-1, 7-2, 8-2, A-22, B-19
DQ$CHANGE$ACCESS, 5-2, A-32
DQ$CHANGE$EXTENSION, 7-1, A-31
DQ$CLOSE, 7-1, 8-2, A-42
DQ$CREATE, 5-2, 7-1, 8-2, A-23, B-19
DQ$DECODE$EXCEPTION, 6-3, 7-1, 7-2,A-48
DQ$DECODE$TIME, 5-3, A-10
DQ$DELETE, 5-2, 6-3, 7-1, A-25
DQ$DETACH, 7-1, 8-2, A-24, B-19
DQ$EXIT, 5-2, 7-1, 8-2, A-43, B-17
DQ$FILE$INFO, 5-3, A-28, A-29
DQ$FREE, 7-1, 8-2, A-18, B-19, B-45
DQGETARGUMENT, 7-1, 7-3, 8-2, A-7, A-12 thru A-14
DQGETCONNECTION$STATUS, 6-3, 7-1, 8-2, A-26, A-27
DQGETEXCEPTION$HANDLER, 7-1, 8-2, A-46,A-47, B-40
DQGETSIZE, 6-3, 7-1, 8-2, A-19, B-47
DQGETSYSTEM$ID, 7-1, A-11
DQGETTIME, 5-3, 6-3, A-9
DQ$OPEN, 5-3, 7-1, 7-3, 8-2, A-33, A-34, B-19
DQ$OVERLAY, 7-1, A-44
DQ$READ, 5-3, 7-1, 8-2, A-37
DQ$RENAME, 5-3, 6-3, 7-1, 7-3, A-30
DQ$RESERVE$IO$MEMORY, 5-3, A-20, A-21
DQ$SEEK, 5-3, 7-1, 8-2, A-35, A-36
DQ$SPECIAL, 7-1, 7-3, 8-2, A-38, A-39, B-19
DQ$SWITCH$BUFFER, 7-1, A-15, A-16
DQ$TRAP$CC, A-49
DQ$TRAP$EXCEPTION, 7-1, 7-3, 8-2, A-45, B-13, B-38
DQ$TRUNCATE, 5-3, 7-1, 8-2, 8-3, A-41
DQ$WRITE, 7-1, 8-2, A-40
dynamic variables, B-42, B-43, B-45

E8087, 3-7 thru 3-11, 3-14
E8087.LIB, 3-7 thru 3-10
editor (See text editor)
EH87.LIB, 3-7 thru 3-10
end of record, B-2, B-6, B-18, B-25 thru B-27, B-32, B-33
end of file, 5-3, A-38, B-2, B-4, B-6, B-18, B-25, B-30, B-37
error (See exception)
ESCAPE, A-46

exception codes, 3-15, 4-2, 4-3, 5-3, 5-4, 6-3, 6-6, 6-7, 7-2 thru 7-4, A-3, A-4, A-6, B-3 thru B-5
  for UDI procedures, A-9 thru A-49
  for LRS procedures, B-8 thru B-49
exception conditions, A-2 thru A-4, B-4, B-17, B-38, B-40
exception handler and exception handling, 2-3, 3-3, 3-4, 3-7, 3-8, 3-15, 4-3, 5-5, 7-3, 9-1, A-3, A-4, A-7, A-45 thru A-49, B-1 thru B-5, B-12 thru B-14, B-38 thru B-41
extension, A-31
EXTERNAL, 1-3, 1-4, 1-6
external labels, 1-3
EXTRN, 1-4

F86RNx.LIB, 3-5, 3-8, 3-9, 3-11, 3-13, 3-14
file descriptors, B-5, B-8, B-9, B-19, B-23 thru B-25, B-27, B-29, B-31, B-32, B-34, B-36, B-37, B-42
file/device driver (See device driver)
file markers, B-18
file name, 4-1, A-7, A-25, A-30 thru A-32, B-4, B-15, B-16
  (See also path name)
file position pointer, A-26, A-33, A-36, A-37, A-40, A-41, B-18, B-19, B-26, B-29 thru B-31, B-36, B-37
FILTER, 3-7, 9-1, 9-2
floating point, 1-7, 2-1, 3-1, 3-5, 3-7, 4-3
formatted files, B-6, B-20
FORTRAN-86/88, Chapter 1, 3-1, 3-5 thru 3-9, 3-11 thru 3-15, 4-4, 8-1, 9-2, B-1, B-2, B-13, B-15, B-16, B-19, B-21, B-22, B-31, B-32, B-46
free (See memory management)
FUNCTION, 1-3

group, B-42, B-46

heap, 3-11, B-42, B-43, B-46

I/O, 2-3, 3-3 thru 3-5, 5-7, 6-1, 6-7, 7-1, 8-3, A-35, B-1, B-18, B-49
iAPX 86,88 Monitor Program, 2-2
ICE In-Circuit Emulator, 2-2 thru 2-4
IEEE proposed floating-point standard, 3-7, 9-1
initialization
  of 8087, 5-5, A-6
  of run-time system, 3-11, 3-14, 9-2, B-12, B-13
INITFP, 3-14
Intellec Microcomputer Development System, 2-4, 7-4
  (See also Intellec Series III Microcomputer Development System)
Intellec Series III Microcomputer Development System, 2-1 thru 2-3, 5-1, 5-6, 5-7, B-49
Interactive Configuration Utility (ICU), 7-4
interactive files and programs, A-35, A-45, B-21
interrupt, 1-2, 3-3, 4-3, 4-4, 5-4 thru 5-6, 6-6, 7-4, 7-5, A-4 thru A-6
iRMX 86 Real-Time Multitasking Executive, 2-2, 2-3, 5-7, 5-9, Chapter 6
iRMX 86 System Debug Monitor, 2-2, 2-3
iRMX 86 System Debugger, 2-2, 2-3, 6-1
iRMX 88 Real-Time Multitasking Executive, Chapter 7
iSBC 337 Multimodule Numeric Data Processor, 2-1, 5-4
iSBC 86/12A Single Board Computer, 2-1, 2-2, B-49
iSBC 957B iAPX 86,88 Interface and Execution Package, 2-1

ISIS-II, 2-1, 5-1
ISO, 1-1

job, 3-11 thru 3-14, A-1, A-17

LARGE.LIB, 5-2, 5-8
LIB86, 1-2, 1-4, 3-11
library, 1-2 thru 1-4, 3-1, 3-5, 3-6, 3-8, 3-11 thru 3-15, 4-1, 4-2,
    5-2, 5-5, 5-7, 6-1, 6-2, 6-6 thru 6-7, 7-1, 7-4, 8-1, 8-2,
    9-1 thru 9-3, B-47
line editing, A-38, A-39
LINK86, 1-2 thru 1-4, 2-5, 3-8, 3-12 thru 3-15, 5-8, 5-9, 6-7, 6-9,
    8-2, A-44, B-44
linkage and linking, 1-3, 2-1, 3-8, 3-11 thru 3-15, 4-1, 5-5, 5-7 thru 5-9,
    6-2, 6-6, 6-9, 7-4, 9-1 thru 9-3, B-5, B-47
LOC86, 2-5, B-47
logical names, 6-8
Logical Record Interface (LRI), 3-4, 3-5, Appendix B
Logical Record System (LRS), 3-4, 3-5, 3-11, 3-13, 9-1, Appendix B

main module, 1-2, 3-11
memory manager and memory management, 3-4, 3-5, 4-1, 6-3, 6-4, 7-1, 8-3,
    A-1, A-2, A-7, A-17 thru A-20, A-42, A-43, B-1, B-2, B-8, B-9, B-13,
    B-42 thru B-48
memory pool (See memory management)
MEMORY segment, B-47
MEMPOOL, 5-9, 6-7
models of segmentation, 3-11, 5-2, 6-2, 7-4, A-7, A-19, A-45, A-46, B-42,
    B-46
MODULE, 1-3
module identification, 1-3
multitasking, 5-6, 6-1, 6-2, B-38, B-42

NAME, 1-3
NaN, 3-8
NEW, B-42 thru B-44, B-46
NOPUBLICS, 3-12
NULL87.LIB, 3-7 thru 3-10, 3-12, 3-14, 5-3, 5-9, 6-12

OEM system, 2-1, 2-2
open, 8-3, A-24, A-33, A-37, A-41, B-2, B-4, B-5, B-6, B-8, B-9, B-10, B-15,
    B-19 thru B-24
operating system, 3-5, 3-12, 3-13, 4-1 thru 4-3, 7-2, Chapter 8,
    A-1 thru A-4, A-9, A-11, A-13, A-25, A-33, A-38 A-45, B-1, B-49
  (See also iRMX 86, iRMX 88, ISIS-II, Series III Operating System)
overlays, 6-9, A-49, B-3

P86RNx.LIB, 3-5, 3-8, 3-10 thru 3-14, 5-8, 5-9, 6-12, B-49
parameter, 1-2, 1-6, 1-7, 3-6, 9-1, 9-2, A-4, A-6, A-8, B-15
  for UDI procedures, A-9 thru A-49
  for LRS procedures, B-8 thru B-49
parsing, 4-1, 9-2, A-7, A-12, A-15, A-16, B-14
Pascal-86/88, Chapter 1, 3-1, 3-5 thru 3-8, 3-10 thru 3-15, 4-4, 5-7, 5-8,
    6-10, 6-11, 8-1, 9-2, B-1, B-2, B-13, B-15, B-16, B-19 thru B-21,
    B-23, B-29, B-31, B-32, B-36, B-37, B-42, B-43, B-45, B-46, B-49
path name, 1-4, 6-3, 8-3, 9-2, A-2, A-22, A-23, A-30, B-10, B-13, B-15,

B-16, B-19, B-21
PE8087, 3-7, 3-14
PL/M-86, Chapter 1, 3-6, 3-7, 3-10, 3-14, 4-4, 6-9, A-8, A-19, A-45, B-3 thru B-5
preconnection, 9-2, 9-3, B-4, B-12 thru B-16, B-42
printer, A-3, A-34, B-20
port, 2-3
PROGRAM, 1-3, 9-2, B-13
PUBLIC, 1-3, 1-4, 1-6, 9-2
public labels, 1-3, 3-12
PURGE, 3-12

read, A-33, A-40, B-2, B-4, B-6, B-21, B-25, B-26, B-29, B-35
reentrancy, 3-5, 3-11 thru 3-13, 4-4, 5-6, 6-6, B-3, B-12, B-15, B-38, B-42
registers, 1-7, 2-2 thru 2-4, A-8, B-4
rewind, B-2, B-6, B-34, B-35
RPB-86, B-49
RTI-88, 7-1, 7-2
RTNULL.LIB, 3-5, 3-8, B-5
RUN, 5-1, 5-7 thru 5-9

scratch file (See workfile)
seek, 6-3, A-27 A-33, A-40, B-2, B-6, B-20, B-29, B-30
segment, B-42, B-46, B-47
Series III (See Intellec Series III Microcomputer Development System)
Series III Operating System, Chapter 5
skip, B-2, B-6, B-31
SMALL.LIB, 5-2
stack, 1-7, 4-3, 4-4, 9-2, A-8, B-3
SUBMIT, 7-4
subprogram, 1-2, 1-7
subroutine (See subprogram)
SUBROUTINE, 1-3
symbolic debugging, 2-3 thru 2-5
synchronization, B-38, B-42
system calls, 2-3, 4-1, 4-2

task, 2-3, 3-11 thru 3-13, 6-6, A-1, A-6
termination, A-4, A-43, B-3, B-17
text editor, 2-1, 2-2
TEXTFILE, B-21
TQ_001, 3-14
TQ_312, 9-1
TQ_999, 3-14
TQ$ALLOCATE, B-2, B-13, B-42 thru B-46
TQ$DEFAULTPL, 9-2, 9-3, B-13, B-14
TQ$DEVICE, B-2, B-6, B-10, B-11, B-49
TQ$ESTART, 9-1, B-3, B-13, B-14
TQ$EXIT, B-2, B-17
TQ$FILE$DESCRIPTOR, B-2, B-5, B-8, B-9, B-14, B-24
TQ$FREE, B-2, B-13, B-42, B-44 thru B-46
TQGETERH, B-2, B-14, B-38 thru B-41
TQGETPRECON, B-2, B-12 thru B-16, B-19
TQGETSMALL$HEAP, 3-11, B-2, B-13, B-42, B-46 thru B-48
TQ$INITIALIZE, B-2, B-3, B-12 thru B-15, B-38
TQ$INITIO, B-13

TQ$INITMM, B-13
TQ$PARSECL, 9-2, B-13, B-14
TQSETERH, 9-1, B-2, B-14, B-38 thru B-41
TQ_TRAP87, 5-5, 5-6
TQ$WHERES$TRAP87, 5-5, 5-6
transparent console input, A-38, A-39
transporting applications programs, 4-1, 4-2, A-2
truncate, 5-3, 8-3, B-37

unavoidable exception conditions, A-3, A-5
unformatted files, B-6, B-20
unit numbers, 9-2, 9-3, B-13, B-15, B-16
Universal Development Interface (UDI), 3-5, 4-1 thru 4-3, 5-2, 5-3, 5-7, 5-9, 6-1 thru 6-3, 6-6 thru 6-10, 7-1, 7-2, Chapter 8, 9-1, Appendix A, B-1, B-3, B-8, B-13, B-17, B-19, B-38, B-40, B-43, B-45, B-47, B-49
update, A-33, A-41, B-21, B-35
URXCOM.LIB, 6-2
URXLRG.LIB, 5-9, 6-2, 6-10, 6-12
URXSML.LIB, 6-2

version, ix, 3-11

workfile, 5-3, 8-3, A-3, B-19, B-20, B-23
write, A-33, A-39, A-41, B-2, B-4, B-6, B-21, B-27 thru B-29, B-35

REQUEST FOR READER'S COMMENTS

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

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this publication. If you have any comments on the product that this publication describes, please contact your Intel representative. If you wish to order publications, contact the Intel Literature Department (see page ii of this manual).

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

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

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

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

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

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

CITY _____ STATE _____ ZIP CODE _____

(COUNTRY)

Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS ...

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

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

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1040 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

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

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

Printed in U.S.A.