



WILLIAM AHEARN

LITERATURE

In addition to the product line handbooks listed below, the INTEL PRODUCT GUIDE (no charge, Order No. 210846-003) provides an overview of Intel's complete product lines and customer services.

Consult the INTEL LITERATURE GUIDE (Order No. 210620) for a listing of Intel literature. TO ORDER literature in the U.S., write or call the INTEL LITERATURE DEPARTMENT, 3065 Bowers Avenue, Santa Clara, CA 95051, (800) 538-1876, or (800) 672-1833 (California only). TO ORDER literature from international locations, contact the nearest Intel sales office or distributor (see listings in the back of most any Intel literature).

Use the order blank on the facing page or call our TOLL FREE number listed above to order literature. Remember to add your local sales tax.

1985 HANDBOOKS

Product line handbooks contain data sheets, application notes, article reprints and other design information.

	*U.S. PRICE
QUALITY/RELIABILITY HANDBOOK (Order No. 210997-001) Contains technical details of both quality and reliability programs and principles.	\$15.00
CHMOS HANDBOOK (Order No. 290005-001) Contains data sheets only on all microprocessor, peripheral, microcontroller and memory CHMOS components.	\$12.00
MEMORY COMPONENTS HANDBOOK (Order No. 210830-004)	\$18.00
TELECOMMUNICATION PRODUCTS HANDBOOK (Order No. 230730-003)	\$12.00
MICROCONTROLLER HANDBOOK (Order No. 210918-003)	\$18.00
MICROSYSTEM COMPONENTS HANDBOOK (Order No. 230843-002) Microprocessors and peripherals—2 Volume Set	\$25.00
DEVELOPMENT SYSTEMS HANDBOOK (Order No. 210940-003)	\$15.00
OEM SYSTEMS HANDBOOK (Order No. 210941-003)	\$18.00
SOFTWARE HANDBOOK (Order No. 230786-002)	\$12.00
MILITARY HANDBOOK (Order No. 210461-003) Not available until June.	\$15.00
COMPLETE SET OF HANDBOOKS (Order No. 231003-002) Get a 25% discount off the retail price of \$160.	\$120.00

*U.S. Price Only



U.S. LITERATURE ORDER FORM

NAME: _____ TITLE: _____

COMPANY: _____

ADDRESS: _____

CITY: _____ STATE: _____ ZIP: _____

COUNTRY: _____

PHONE NO.: () _____

ORDER NO.	TITLE	QTY.	PRICE	TOTAL
□□□□□□-□□□□	_____	_____	× _____	= _____
□□□□□□-□□□□	_____	_____	× _____	= _____
□□□□□□-□□□□	_____	_____	× _____	= _____
□□□□□□-□□□□	_____	_____	× _____	= _____
□□□□□□-□□□□	_____	_____	× _____	= _____
□□□□□□-□□□□	_____	_____	× _____	= _____

POSTAGE AND HANDLING:
 Add appropriate postage
 and handling to subtotal
 10% U.S.
 20% Canada

Subtotal _____
 Your Local Sales Tax _____

Allow 4-6 weeks for delivery

Total _____

Pay by Visa, MasterCard, Check or Money Order, payable to Intel Literature. Purchase Orders have a \$50.00 minimum.

Visa Account No. _____ Expiration _____
 MasterCard Date

Signature: _____

Mail To: Intel Literature Distribution
 Mail Stop SC6-714
 3065 Bowers Avenue
 Santa Clara, CA 95051.

Customers outside the U.S. and Canada should contact the local Intel Sales Office or Distributor listed in the back of this book.

For information on quantity discounts, call the 800 number below:

TOLL-FREE NUMBER: (800) 548-4725

Prices good until 12/31/85.

Source HB

Cut Along Dotted

Mail To: Intel Literature Distribution
Mail Stop SC6-714
3065 Bowers Avenue
Santa Clara, CA 95051.



SOFTWARE HANDBOOK

1985

*About Our Cover:
The design on our front cover is an abstract portrayal of the role Intel software plays in systems development. The heart of systems development is surrounded by a sphere of peripheral development through which Intel software can guide the designer to reaching development goals.*

Intel Corporation makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.

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

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

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

BITBUS, COMMputer, CREDIT, Data Pipeline, GENIUS, i, i, ICE, iCS, iDBP, iDIS, i²ICE, iLBX, i_m, iMDDX, iMMX, Insite, Intel, int_el, int_elBOS, Intelvision, int_el_igent Identifier, int_el_igent Programming, Intellec, Intellink, iOSP, iPDS, iSBC, iSBX, iSDM, iSXM, KEPROM, Library Manager, MCS, Megachassis, MICRO-MAINFRAME, MULTIBUS, MULTICHANNEL, MULTIMODULE, OpeNet, Plug-A-Bubble, PROMPT, Promware, QUEST, QueX, Ripplemode, RMX/80, RUPI, Seamless, SLD, SYSTEM 2000, and UPI, and the combination of ICE, iCS, iRMX, iSBC, iSBX, MCS, or UPI and a numerical suffix.

The following are trademarks of the companies indicated and may only be used to identify products of the owners.

CP/M is a trademark of Digital Research, Inc.

DEC, DEC-10, DEC-20, PDP-11, DECnet, DECwriter, RSTS, and VAX are trademarks of Digital Equipment Corporation.

MDS is an ordering code only and is not used as a product name or trademark.

MDS® is a registered trademark of Mohawk Data Sciences Corporation.

Microsoft is a trademark of Microsoft, Inc.

Table of Contents

CHAPTER 1

OVERVIEW

Introduction	1-1
--------------------	-----

CHAPTER 2

OPERATING SYSTEMS

Introduction	2-1
--------------------	-----

8080/8085 Microprocessor Family

DATA SHEET

Digital Research Inc. CP/M 2.2 Operating System	2-2
---	-----

8086/8088/80286 Microprocessor Family

DATA SHEETS

iRMX 86 Operating System	2-5
--------------------------------	-----

iOSP 86, iAPX 86/30, iAPX 88/30, iAPX 186/30 and iAPX 188/30 Support Package	2-24
--	------

iRMX 86 MULTIBUS II Support Package	2-28
---	------

FACT SHEET

iRMX Operating Systems	2-31
------------------------------	------

XENIX 3.0 Operating Systems	2-37
-----------------------------------	------

APPLICATION NOTES

AP-130 Using Operating Systems Processor's to Simplify Microcomputer Designs	2-44
--	------

AP-174 Optimizing the iRMX 86 Operating System Performance on System 86/310 and System 86-330	2-95
--	------

AP-184 Writing Device Drivers for XENIX 86 and 286 - Task or Trivia?	2-123
--	-------

AP-221 An Introduction to Task Management in the iRMX 86 Operating System	2-193
---	-------

ARTICLE REPRINTS

AR-286 Software That Resides in Silicon	2-234
---	-------

AR-287 Putting Real-Time Operating Systems to Work	2-240
--	-------

AR-288 Intel's Matchmaking Strategy: Marry iRMX Operating System with Hardware	2-252
---	-------

AR-337 Industrial PC Systems Demand Real Time Operating Systems	2-255
---	-------

CHAPTER 3

TRANSLATORS AND UTILITIES FOR PROGRAM DEVELOPMENT

Introduction	3-1
--------------------	-----

MCS-80/85 Microprocessor Family

DATA SHEETS

PL/M 80 High Level Programming Language	3-3
---	-----

FORTRAN 80 8080/8085 ANS FORTRAN 77 Intellec Resident Compiler	3-6
--	-----

Microsoft, Inc. BASIC-80 Interpreter Software Package	3-10
---	------

Microsoft, Inc. BASIC-80 Compiler Software Package	3-13
--	------

Microsoft Multiplan Spreadsheet	3-15
---------------------------------------	------

PASCAL-80 Software Package	3-23
----------------------------------	------

WORDSTAR Word Processing Software	3-28
---	------

iAPX 86/88/186/188/286 Microprocessor Family

DATA SHEETS

iAPX 86, 88 Software Development Packages for Series II/PDS	3-35
---	------

86/88/186/188 Software Packages	3-45
---------------------------------------	------

FORTRAN 86/88 Software Package	3-46
--------------------------------------	------

Pascal 86/88 Software Package	3-50
-------------------------------------	------

PL/M 86/88/186/188 Software Package	3-53
---	------

iC-86 C Compiler for the 8086	3-57
-------------------------------------	------

8087 Support Library	3-61
----------------------------	------

8087 Software Support Package	3-65
-------------------------------------	------

8089 iOP Software Support Package #407200	3-68
---	------

iAPX 286 Software Development Package	3-71
---	------

PL/M 286 Software Package	3-76
---------------------------------	------

iSDM 286 iAPX 286 System Debug Monitor	3-80
--	------

80287 Support Library	3-84
-----------------------------	------

Pascal-286 Software Package	3-87
-----------------------------------	------

VAX/VMS Resident Software Development Packages for iAPX 286	3-90
---	------

VAX/VMS Resident iAPX 86/88/186 Software Development Packages	3-96
iSDM 86 System Debug Monitor	3-103
iVDI 720 Graphics Virtual Device Interpreter	3-108
iPLP 720 NAPLPS Interpreter	3-112
FACT SHEETS	
iRMX Languages	3-116
XENIX Languages	3-121
Single Chip Microcontroller Software	
DATA SHEETS	
2920 Software Support Package	3-125
MCS-48 Diskette-Based Software Support Package	3-136
8051 Software Package	3-138
iRMX 51 Real-Time Multitasking Executive	3-147
MCS-96 Software Development Packages	3-153
CHAPTER 4	
DEVELOPMENT PRODUCTIVITY TOOLS	
Introduction	4-1
Program Development and Management Tools	
DATA SHEETS	
PSCOPE High-Level Program Debugger	4-2
Program Management Tools	4-7
ISIS-II Software Toolbox	4-10
8086 Software Toolbox	4-12
AEDIT Text Editor	4-14
CHAPTER 5	
COMMUNICATION SOFTWARE	
Introduction	5-1
DATA SHEETS	
Mainframe Link for Distributed Development	5-2
Intel Asynchronous Communications Link	5-5
iNA 960 Network Software	5-8
NDS II Electronic Mail	5-20
iNA 955 iRMX NDS-II Link	5-22
iRMX 510 iDCM Support Package	5-26
CHAPTER 6	
SYSTEM AND APPLICATIONS SOFTWARE	
Introduction	6-1
FACT SHEETS	
XENIX Productivity Software Tools	6-2
Third Party Software for Intel Products	6-10
Database Information System iDIS 715	6-13
CHAPTER 7	
COMPONENT SOFTWARE	
DATA SHEETS	
80130/80130-2 iAPX 86/30, 88/30, 186/30, 188/30 iRMX 86	
Operating System Processors	7-1
80150/80150-2 iAPX 86/50, iAPX 88/50, 186/50, 188/50 CP/M 86	
Operating System Processors	7-23

CHAPTER 8

USER LIBRARY

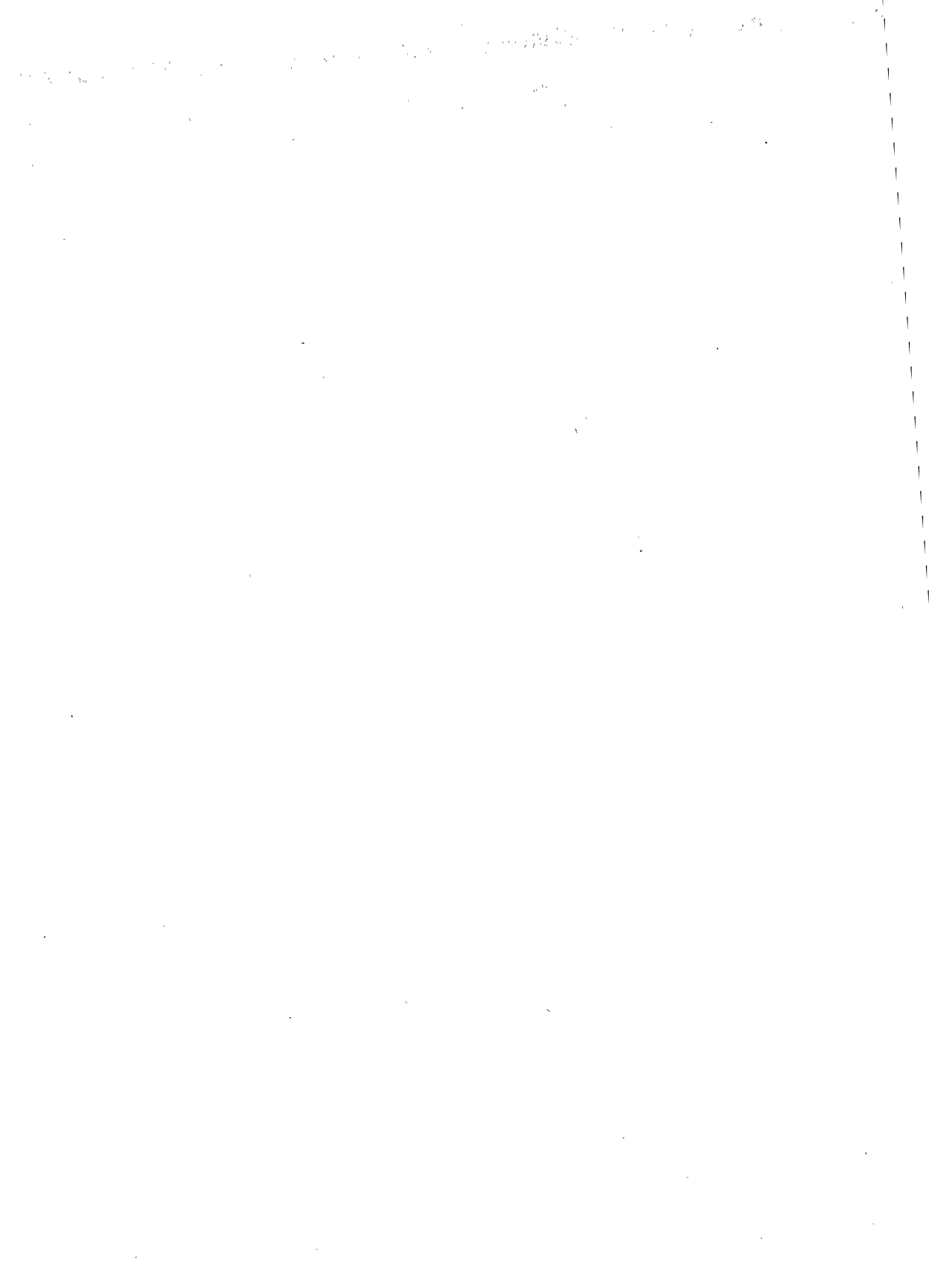
Introduction	8-1
User Library	
Insite User's Program Library	8-2
Insite Submittal Requirements	8-3
Insite Index of Program	8-5

APPENDIX A

Software Standards	A-1
Software Support Services	A-3
IRUG	A-5

Overview

1



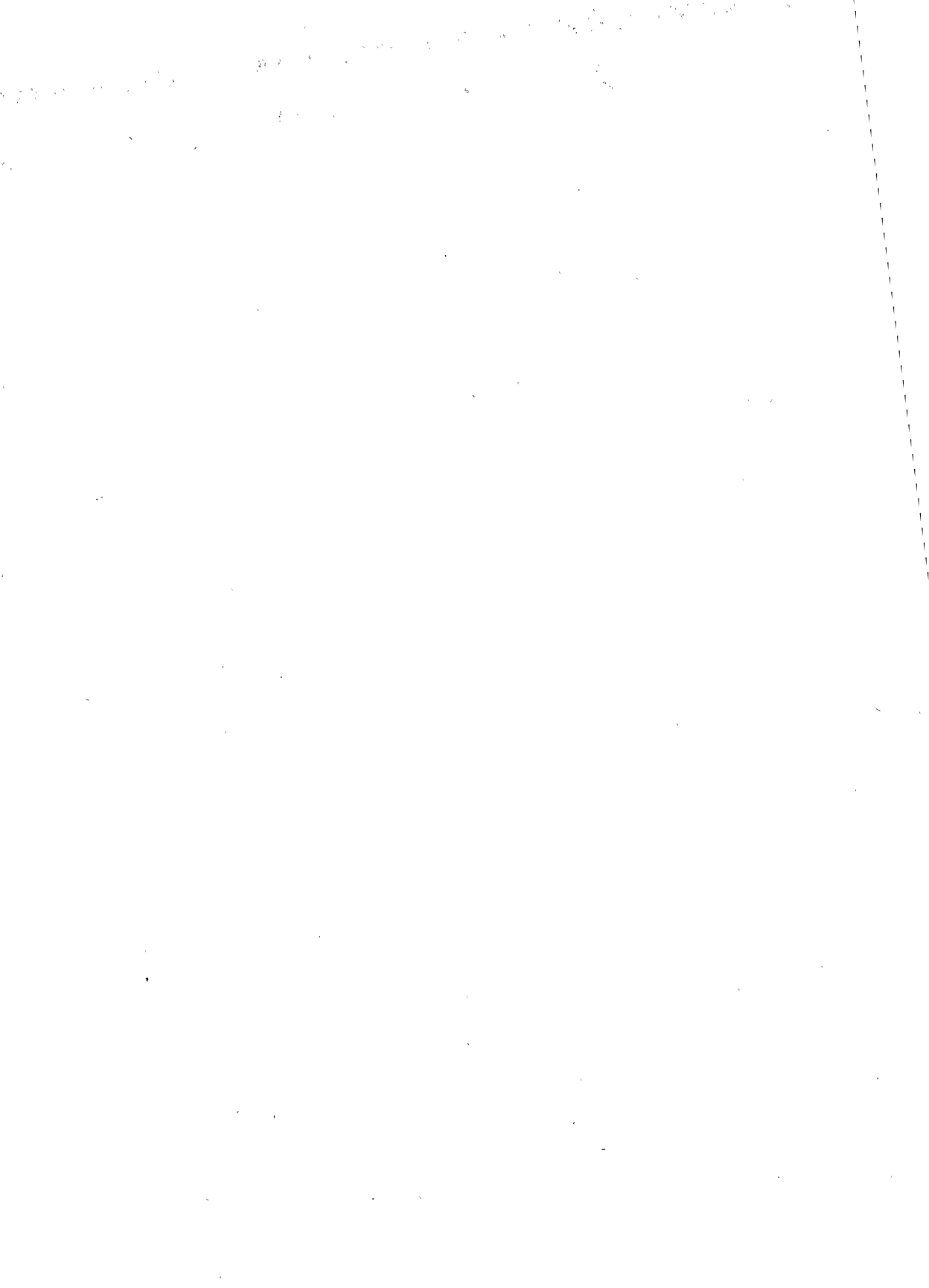


SOFTWARE HANDBOOK OVERVIEW

Welcome to the Intel Software Handbook. This handbook is a complete guide to the software products and services offered by Intel.

Intel's software products follow the open systems strategy that allows Intel products to be purchased at the customers' desired level of integration. Hence these products are available for component, board, or systems applications. This open systems philosophy is backed by software standards that insure that the software can operate at numerous levels of integration. These software standards are described in the appendix.

Software for Intel's products is available both from Intel and from Independent Software Vendors (ISVs). For a complete listing of software available from ISVs, see the Intel Yellow Pages which is published annually by Intel. This handbook describes software products that are available through Intel, consisting of Intel-developed and ISV-developed products. Products that are offered by Intel have all been evaluated and tested to meet Intel's quality standard. They are backed by an extensive support organization described in the appendix.



OPERATING SYSTEMS

INTRODUCTION

The ability to convert advanced microprocessor technology into solutions for modern day problems begins with effective and efficient designs for new hardware products and architecture. However, a most critical element in the success of any microcomputer solution is the availability of a high quality, reliable operating system. Without this software counterpart, the technological advances cannot be fully implemented, nor their benefits fully realized.

The classic role of the microcomputer operating system can be outlined by viewing its major functions and purposes. The functions of the microcomputer operating system are threefold: 1) to manage system resources and the allocation of these resources to users; 2) to provide automatic functions such as initialization and start-up procedures; and 3) to provide an efficient, straightforward and consistent method for user programs to interface with the hardware subsystems, including a simple and friendly human interface. Typically, the operating systems have one of two main purposes. First, they can be used to develop a new software system that runs on another machine. These systems are usually large and fairly sophisticated. ISIS and *XENIX are examples of such developmental operating systems. The second purpose for microcomputer operating systems is directed toward the execution of software programs for targeted application. The largest number of operating systems are of this type, including the RMX systems. The most critical requirement is for these systems to be effective and efficient since they are usually small, fast systems dedicated to a specific real-time application.

This rather neat and simple categorization of microcomputer operating systems, which has been useful in the past, is quickly becoming blurred. The rapid developments in microcomputer technology have increased the power and decreased the cost of microcomputers, allowing them to become applicable to the solution of a broader variety and more sophisticated set of problems. Microcomputer systems must increasingly provide such capabilities as multiprogramming, multitasking, multiprocessing, networking, as well as scheduling and priority determination. As systems become more complex, they must still remain responsive to real-time applications. Operating systems must be able to capitalize on the trends toward placing more and more software into silicon. This trend is blurring the distinction between the hardware and software subsystems. Microcomputer systems are also evolving to encompass both the developmental and target application purposes into one system.

These dramatic changes in technology place additional demands on operating systems. We see operating systems undergoing changes to consider the need for: 1) modularity and ease of configurability; 2) evolutionary, not revolutionary, path of growth; and 3) standardization in languages, networks and the operating system itself. The first need is required to allow the system to be a powerful development tool yet configurable to more specialized applications. The last two items are needed to provide protection of a firm's software investment, including the option to move toward silicon software.

The operating systems and executives in this section are state-of-the-art microcomputer systems that have taken to task the challenges posed by advancing microprocessor technology. These operating systems offer the widest range of solutions with the highest quality and most future-oriented software available today. Consequently, our customers can select the appropriately optimized option to achieve their price/performance goals and give them time-to-market advantage over their competitors.

*XENIX is a trademark of Microsoft Corp.



DIGITAL RESEARCH INC. CP/M* 2.2 OPERATING SYSTEM

- High-performance, single-console operating system
- Simple, reliable file system matched to microcomputer resources
- Table-driven architecture allows field reconfiguration to match a wide variety of disk capacities and needs
- Extensive documentation covers all facts of CP/M applications
- More than 1,000 commercially available compatible software products
- General-purpose subroutines and table-driven data-access algorithms provide a truly universal data management system
- Upward compatibility from all previous versions

CP/M 2.2 is a monitor control program for microcomputer system and application uses on Intel 8080/8085-based microcomputer. CP/M provides a general environment for program execution, construction, storage, and editing, along with the program assembly and check-out facilities.

The CP/M monitor provides rapid access to programs through a comprehensive file management package. The file subsystem supports a named file structure, allowing dynamic allocation of file space as well as sequential and random file access. Using this system, a large number of distinct programs can be stored in both source- and machine-executable form.

CP/M also supports a powerful context editor, Intel-compatible assembler, and debugger subsystems. Nearly all personal software programs can be bought configured to run under CP/M, several of which are available from Intel.

FEATURES

CP/M is logically divided into four distinct modules:

BIOS—Basic I/O System

- Provides primitive operations for access to disk drives and interface to standard peripherals (teletype, CRT, paper tape reader/punch, bubble memory, and user-defined peripherals)
- Allows user modification for tailoring to a particular hardware environment

BDOS—Basic Disk Operating System

- Provides disk management for one to sixteen disk drives containing independent file directories
- Implements disk allocation strategies for fully dynamic file construction and minimization of head movement across the disk

—Uses less than 4K of memory allowing plenty of memory space for applications programs

—Uses less than 4K of memory

—Makes programs transportable from system to system

—Entry points include the following primitive operations which can be programmatically accessed:

SEARCH	Look for a particular disk file by name
OPEN	Open a file for further operations
CLOSE	Close a file after processing
RENAME	Change the name of a particular file
READ	Read a record from a particular file
WRITE	Write a record to a particular file
SELECT	Select a particular disk drive for further operations

CCP—Console Command Processor

- Provides primary user interface by reading and interpreting commands entered through the console
- Loads and transfers control to transient programs, such as assemblers, editors, and debuggers
- Processes built-in standard commands including:

ERA	Erase specified files
DIR	List file names in the directory
REN	Rename the specified file
SAVE	Save memory contents in a file
TYPE	Display the contents of a file on the console

TPA—Transient Program Area

- Holds programs which are loaded from the disk under command of the CCP
- Programs created under CP/M can be checked out by loading and executing these programs in the TPA
- User programs, loaded into the TPA, may use the CCP area for the program's data area
- Transient commands are specified in the same manner as built-in commands
- Additional commands can be easily defined by the user
- Defined transient commands include:

PIP	Peripheral Interchange Program —implements the basic media transfer operations necessary to load, print, punch, copy, and combine disk files, PIP also performs various reformatting and concatenation functions. Formatting options include parity-bit removal, case conversion, Intel hex file validation, subfile extraction, tab expansion, line number generation, and pagination
ED	Text Editor—allows creation and modification of ASCII files using extensive context editing commands: string substitution, string search, insert, delete and block move; ED allows text to be located by context, line number, or relative position with a macro command for making extensive text changes with a single command line

- | | |
|--------|--|
| ASM | Fast 8080 Assembler—uses standard Intel mnemonics and pseudo operations with free-format input, and conditional assembly features |
| DDT | Dynamic Debugging Tool—contains an integral assembler/disassembler module that lets the user patch and display memory in either assembler mnemonic or hexadecimal form and trace program execution with full register and status display; instructions can be executed between breakpoints in real-time, or run fully monitored, one instruction at a time |
| SUBMIT | Allows a group of CP/M commands to be batched together and submitted to the operating system by a single command |
| STAT | Lists the number of bytes of storage remaining on the currently logged disks, provides statistical information about particular files, and displays or alters device assignments |
| LOAD | Converts Intel hex format to absolute binary, ready for direct load and execution in the CP/M environment |
| SYSGEN | Creates new CP/M system disks for back-up purposes |
| MOVCPM | Provides regeneration of CP/M systems for various memory configurations and works in conjunction with SYSGEN to provide additional copies of CP/M |

BENEFITS

- Easy implementation on any computer configuration which uses an Intel 8080/8085 Central Processing Unit (see the CP/M-86 data sheet for CP/M applications on the iAPX86 CPU)
- iPDS version supports bubble memory option as an additional diskette drive. Also allows diskette duplication with a single drive
- Extensive selection of CP/M-compatible programs allows production and support of a comprehensive software package at low cost
- Field programmability for special-purpose operating system requirements
- Upward compatibility from previous versions of CP/M release 1

- Provides field specification of one to sixteen logical drives, each containing up to eight megabytes
- Files may contain up to 65,536 records of 128 bytes each but may not exceed the size of any single disk
- Each disk is designed for 64 distinct files—more directory entries may be allocated if necessary
- Individual users are physically separated by user numbers, with facilities for file copy operations from one user area to another
- Relative-record random-access functions provide direct access to any of the 65,536 records of an eight-megabyte file

SPECIFICATIONS

Hardware Required

- Model 800 with 720 kit
- DS 235 kit or MDS 225 with 720 kit (integral drive supported except as system boot device)
- iPDS Personal Development System
Optional:
 - RAM up to 64K
 - Additional floppy disk drives
 - Single density via 201 controller
 - Bubble memory and optional Shugart 460 5¼" disk drive for iPDS

Documentation Package

Title

- CP/M 2.2 documentation consisting of 7 manuals:
 - An Introduction to CP/M Features and Facilities
 - CP/M 2.2 User's Guide
 - CP/M Assembler (ASM) User's Guide
 - CP/M Dynamic Debugging Tool (DDT) User's Guide
 - ED: A Context Editor for the CP/M Disk System User's Manual
 - CP/M 2 Interface Guide
 - CP/M 2 Alteration Guide

Shipping Media

(Specify by Alpha Character when ordering.)

- A—single density (IBM 3740/1 compatible)
- B—double density
- F—double-sided, double density 5¼" floppy (iPDS format)

Order Code

Product Description

See Price List

CP/M (Control Program for Microcomputers) is a disk-based operating system for the Intel 8080/8085-based systems. CP/M provides a general environment for program development, test, execution and storage. CP/M storage is available via a comprehensive, named-file structure supporting both sequential and random access. CP/M support tools include a Text Editor, a debugger, and an 8080/8085 assembler.

SUPPORT:

Intel offers several levels of support for this product, depending on the system configuration in which it is used. Please consult the price list for a detailed description of the support options available.

An Intel Software License required.

*CP/M is a registered trademark of Digital Research, Inc.

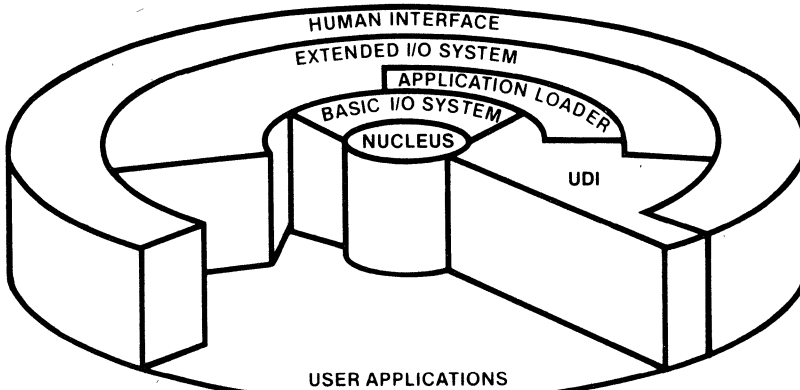
*CP/M-86, MP/M, CP/NET and MP/NET are trademarks of Digital Research, Inc



iRMX™ 86 OPERATING SYSTEM

- Real-time processor management for time-critical iAPX 86, iAPX 88, iAPX 186, iAPX 188, and iAPX 286 (Real Address Mode) applications
- On-target system development with Universal Development Interface (UDI)
- Configurable system size and function for diverse application requirements
- All iRMX™ 86 code can be (P)ROM'ed to support totally solid state designs
- Compatible operating system services for iAPX 86/30, 88/30, 186/30 and 188/30 Operating System Processors (iOSP™ 86)
- Configured systems for the iAPX 86 and iAPX 286 processors in Intel integrated system products (iSYS 86/300 and iSYS 286/300)
- Multi-terminal support with multi-user human interface
- Broad range of device drivers included for industry standard MULTIBUS® peripheral controllers
- Complete support of 8087 and 80287 processor extension
- Powerful utilities for interactive configuration and real-time debugging

The iRMX™ 86 Operating System is an easy-to-use, real-time, multi-tasking and multi-programming software system designed to manage and extend the resources of iSBC® 86, iSBC 88, iSBC 186, iSBC 188, and iSBC 286 Single Board Computers, as well as other iAPX 86, iAPX 88, iAPX 186, iAPX 188, and iAPX 286 (Real Address Mode) based microcomputers. iRMX 86 functions are available in silicon with the iAPX 86/30, 88/30, 186/30 and 188/30 Operating System Processors, in a user configurable software package. iRMX 86 functions are also fully integrated into the SYSTEM 86/300 and SYSTEM 286/300 Family of Microcomputer Systems. The Operating System provides a number of standard interfaces that allow iRMX 86 applications to take advantage of industry standard device controllers, hardware components, and a number of software packages developed by Independent Software Vendors (ISVs). Many high-performance features extend the utility of iRMX 86 Systems into applications such as data collection, transaction processing, and process control where immediate access to advances in VLSI technology is paramount. These systems may deliver real-time performance and explicit control over resources; yet also support applications with multiple users needing to simultaneously access terminals. The configurable layers of the System provide services ranging from interrupt management and standard device drivers for many sophisticated controllers, to data file maintenance commands provided by a comprehensive multi-user human interface. By providing access to the standard Universal Development Interface (UDI) for each user terminal, Original Equipment Manufacturers (OEMs) can pass program development and target application customization capabilities to their users.



iRMX™ VLSI Operating System

The following are trademarks of Intel Corporation and may be used only to describe Intel products: Intel, ICE, iMMX, iRMX, iSBC, iSBX, iSXM, MULTIBUS, MULTICHANNEL and MULTIMODULE. 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. Information contained herein supersedes previously published specifications on these devices from Intel.

The iRMX 86 Operating System is a complete set of system software modules that provide the resource management functions needed by computer systems. These management functions allow Original Equipment Manufacturers (OEMs) to best use resources available in microcomputer systems while getting their products to market quickly, saving time and money. Engineers are relieved of writing complex system software and can concentrate instead on their application software.

This data sheet describes the major features of the iRMX 86 Operating System. The benefits provided to engineers who write application software and to users who want to take advantage of improving microcomputer price and performance are explained. The first section outlines the system resource management functions of the Operating System and describes several system calls. The second section gives a detailed overview of iRMX 86 features aimed at serving both the iRMX 86 system designer and programmer, as well as the end users of the product into which the Operating System is incorporated.

FUNCTIONAL DESCRIPTION

To take best advantage of iAPX 86, 88, 186, 188, and 286 (Real Address Mode) microprocessors in applications where the computer is required to perform many functions simultaneously, the iRMX 86 Operating System provides a multiprogramming environment in which many independent, multi-tasking application programs may run. The flexibility of independent environments allows application programmers to separately manage each application's resources during both the development and test phases.

The resource management functions of the iRMX 86 System are supported by a number of configurable software layers. While many of the functions supplied by the innermost layer, the Nucleus, are required by all systems, all other functions are optional. The I/O systems, for example, may be omitted in systems having no secondary storage requirement. Each layer provides functions that encourage application programmers to use modular design techniques for quick development of easily maintainable programs.

The components of the iRMX 86 Operating System provide both implicit and explicit management of system resources. These resources include processor scheduling, up to one megabyte of system memory, up to 57 independent interrupt sources, all input and output devices, as well as directory and data files contained on mass storage devices and accessed by a number of independent users. Management of these system resources and methods for sharing resources between multiple processors and users is discussed in the following sections.

Process Management

To implement multi-tasking application systems, programmers require a method of managing the different processes of their application, and for allowing the processes to communicate with each other. The Nucleus layer of the iRMX 86 System provides a number of facilities to efficiently manage these processes, and to effectively communicate between them. These facilities are provided by system calls that manipulate data structures called tasks, jobs, regions, semaphores and mailboxes. The iRMX 86 System refers to these structures as "objects".

Tasks are the basic element of all applications built on the iRMX 86 Operating System. Each task is an entity capable of executing CPU instructions and issuing system calls in order to perform a function. Tasks are characterized by their register values (including those of an optional 8087 or 80287 Numeric Processor Extension), a priority between 0 and 255, and the resources associated with them.

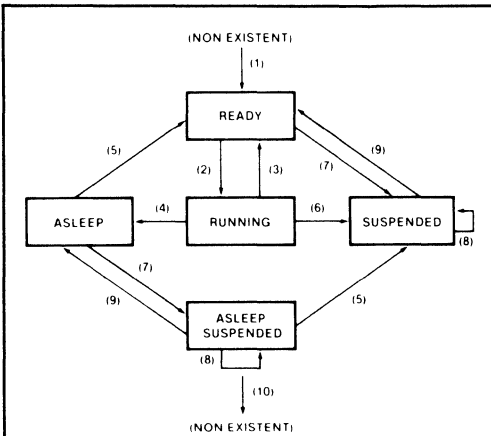
Each iRMX 86 task in the system is scheduled for operation by the iRMX 86 Nucleus. Figure 1 shows the five states in which each task may be placed, and some examples of how a task may move from one state to another. The iRMX 86 Nucleus ensures that each task is placed in the correct state, defined by the events in its external environment and by the task issuing system calls. Each task has a priority to indicate its relative importance and need to respond to its environment. The Nucleus guarantees that the highest priority ready-to-run task is the task that runs.

Jobs are used to define the operating environment of a group of tasks. Jobs effectively limit the scope of an application by collecting all of its tasks and other objects into one group. Because the environment for execution of an application is defined by an iRMX 86 job, separate applications can be efficiently developed by separate development teams.

The iRMX 86 Operating System provides two primary techniques for real-time event synchronization in multi-task applications: regions and semaphores.

Regions are used to restrict access to critical sections of code and data. Once the iRMX 86 Operating System gives a task access to resources guarded by a region, no other tasks may make use of the resources, and the task is given protection against deletion and suspension. Regions are typically used to protect data structures from being simultaneously updated by multiple tasks.

Semaphores are used to provide mutual exclusion between tasks. They contain abstract "units" that are sent between the tasks, and can be used to implement the cooperative sharing of resources.



NOTES:

- (1) Task is created
- (2) Task becomes highest priority ready task
- (3) Task gets pre-empted by one with higher priority
- (4) Task calls SLEEP or task waits at an exchange
- (5) Task sleep period has ended, message was sent to waiting task or wait has ended
- (6) Task calls SUSPEND on self
- (7) Task suspended by other than self
- (8) Task suspended by other than self or a resume that did not bring suspension depth to zero
- (9) Task was resumed by other task
- (10) Task is deleted

Figure 1. Task State Diagram

Multi-tasking applications must communicate information and share system resources among cooperating tasks. The iRMX 86 Operating System assigns a unique 16-bit number, called a token, to each object created in the System. Any task in possession of this token is able to access the object. The iRMX 86 Nucleus allows tasks to gain access to objects, and hence system resources, at run-time with two additional mechanisms: mailboxes and object directories.

Mailboxes are used by tasks wishing to share objects with other tasks. A task may share an object by sending the object token via a mailbox. The receiving task can check to see if a token is there, or can wait at the mailbox until a token is present.

Object Directories are also used to make an object available to other tasks. An object is made public by cataloging its token and name in a directory. In this manner, any task can gain access to the object by knowing its name, and job environment that contains the directory.

Two example jobs are shown in Figure 2 to demonstrate how two tasks can share an object that was not

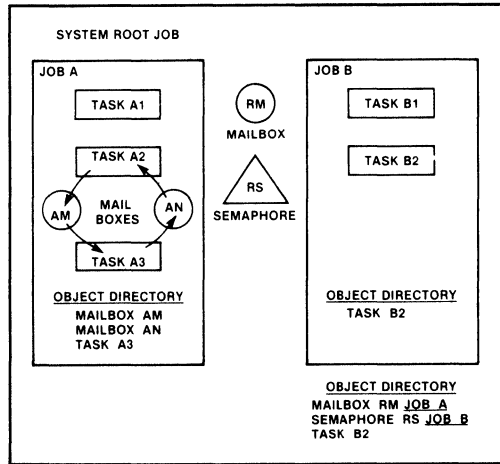


Figure 2. Multiple Jobs Example

known to the programmer at the time the tasks were developed. Both Job 'A' and Job 'B' exist within the environment of the 'Root Job' that forms the foundation of all iRMX 86 systems. Each job possesses a directory in which tasks may catalog the name of an object. Semaphore 'RS', for example, is accessible by all tasks in the system, because its name is cataloged in the directory of the Root Job. Mailbox 'AN' can be used to transfer objects between Tasks 'A2' and 'A3' because its token is accessible in the object directory for Job 'A'.

Table 1 lists the major functions of the iRMX 86 Nucleus that manage system processes.

Memory Management

Each job in an iRMX 86 System defines the amount of the one megabyte of addressable memory to be used by its tasks. The iRMX 86 Operating System manages system memory and allows jobs to share this critical resource by providing another object type: segments.

Segments are contiguous pieces of memory between 16 Bytes and 64K Bytes in length, that exist within the environment of the job in which they were created. Segments form the fundamental piece of system memory used for task stacks, data storage, system buffers, loading programs from secondary storage, passing information between tasks, etc.

The example in Figure 2 also demonstrates when information is shared between Tasks 'A2' and 'A3'; 'A2' only needs to create a segment, put the information in the memory allocated, and send it via the Mailbox 'AM' using the RQ\$SEND\$MESSAGE system call (see Table 1). Task 'A3' would get the message by using the RQ\$RECEIVE\$MESSAGE system call. The Figure also shows how the receiving task could signal the sending task by sending an acknowledgement via the second Mailbox 'AN'.

Table 1. Process Management System Calls

System Call	Function Performed
RQ\$CREATE\$JOB	Creates an environment for a number of tasks and other objects, as well as creating an initial task and its stack.
RQ\$DELETE\$JOB	Deletes a job and all the objects currently defined within its bounds. All memory used is returned to the job from which the deleted job was created.
RQ\$OFFSPRING	Provides a list of all the current jobs created by the specified job.
RQ\$CATALOG\$OBJECT	Enters a name and token for an object into the object directory of a job.
RQ\$UNCATALOG\$OBJECT	Removes an object's token and its name from a job's object directory
RQ\$LOOKUP\$OBJECT	Returns a token for the object with the specified name found in the object directory of the specified job.
RQ\$GET\$TYPE	Returns a code for the type of object referred to by the specified token.
RQ\$CREATE\$MAILBOX	Creates a mailbox with queues for waiting tasks and objects with FIFO or PRIORITY discipline.
RQ\$DELETE\$MAILBOX	Deletes a mailbox.
RQ\$SEND\$MESSAGE	Sends an object to a specified mailbox. If a task is waiting, the object is passed to the appropriate task according to the queuing discipline. If no task is waiting, the object is queued at the mailbox.
RQ\$RECEIVE\$MESSAGE	Attempts to receive an object token from a specified mailbox. The calling task may choose to wait for a specified number of system time units if no token is available.
RQ\$DISABLE\$DELETION	Prevents the deletion of a specified object by increasing its disable count by one.
RQ\$ENABLE\$DELETION	Reduces the disable count of an object by one, and if zero, enables deletion of that object.
RQ\$FORCE\$DELETE	Forces the deletion of a specified object if the disable count is either 0 or 1.
RQ\$CREATE\$TASK	Creates a task with the specified priority and stack area.
RQ\$DELETE\$TASK	Deletes a task from the system, and removes it from any queues in which it may be waiting.
RQ\$SUSPEND\$TASK	Suspends the operation of a task. If the task is already suspended, its suspension depth is increased by one.
RQ\$RESUME\$TASK	Resumes a task. If the task had been suspended multiple times, the suspension depth is reduced by one, and it remains suspended.
RQ\$SLEEP	Causes a task to enter the ASLEEP state for a specified number of system time units.
RQ\$GET\$TASK\$TOKENS	Gets the token for the calling task or associated objects within its environment.
RQ\$SET\$PRIORITY	Dynamically alters the priority of the specified task.
RQ\$GET\$PRIORITY	Obtains the current priority of a specified task.
RQ\$CREATE\$REGION	Creates a region, with an associated queue of FIFO or PRIORITY ordering discipline.
RQ\$DELETE\$REGION	Deletes the specified region if it is not currently in use.
RQ\$ACCEPT\$CONTROL	Gains control of a region only if the region is immediately available.
RQ\$RECEIVE\$CONTROL	Gains control of a region. The calling task may specify the number of system time units it wishes to wait if the region is not immediately available.
RQ\$SEND\$CONTROL	Relinquishes control of a region.
RQ\$CREATE\$SEMAPHORE	Creates a semaphore.
RQ\$DELETE\$SEMAPHORE	Deletes a semaphore.
RQ\$SEND\$UNITS	Increases a semaphore counter by the specified number of units.
RQ\$RECEIVE\$UNITS	Attempts to gain a specified number of units from a semaphore. If the units are not immediately available, the calling task may choose to wait.

Each job is created with both maximum and minimum limits set for its memory pool. Memory required by all objects and resources created in the job is taken from this pool. If more memory is required, a job may be allowed to borrow memory from the pool of its containing job (the job from which it was created). In this manner, initial jobs may efficiently allocate memory to jobs they subsequently create, without knowing their exact requirements.

The iRMX 86 Operating System supplies other memory management functions to search specific address ranges for available memory. The System performs this search at system initialization, and can be configured to ignore non-existent memory and addresses reserved for I/O devices and other application requirements.

Table 2 lists the major system calls used to manage the system memory.

Interrupt Management

Real-time systems, by their nature, must respond to asynchronous and unpredictable events quickly. The iRMX 86 Operating System uses interrupts and the event-driven Nucleus described earlier to give real-time response to events. Use of a pre-emptive scheduling technique ensures that the servicing of high priority

events always takes precedence over other system activities.

The iRMX 86 Operating System gives applications the flexibility to optimize either interrupt response time or interrupt response capability by providing two tiers of Interrupt Management. These two distinct tiers are managed by Interrupt Handlers and Interrupt Tasks.

Interrupt Handlers are the first tier of interrupt service. For small simple functions, interrupt handlers are often the most efficient means of responding to an event. They provide faster response than interrupt tasks, but must be kept simple since interrupts (except the iAPX 86, 88, 186, 188, and 286 non-maskable interrupt) are masked during their execution. When extended service is required, interrupt handlers "signal" a waiting interrupt task that, in turn, performs more complicated functions.

Interrupt Tasks are distinct tasks whose priority is associated with a hardware interrupt level. They are permitted to make any iRMX 86 system call. While an interrupt task is servicing an interrupt, interrupts of lower priority are not allowed to pre-empt the system.

Table 3 shows the iRMX 86 System Calls provided to manage interrupts.

Table 2. Memory Management System Calls

System Call	Function Performed
RQ\$CREATE\$SEGMENT	Dynamically allocates a memory segment of the specified size.
RQ\$DELETE\$SEGMENT	Deletes the specified segment by deallocating the memory.
RQ\$GET\$POOL\$ATTRIBUTES	Returns attributes such as the minimum and maximum, as well as current size of the memory in the environment of the calling task's job.
RQ\$GET\$SIZE	Returns the size (in bytes) of a segment.
RQ\$SET\$POOL\$MIN	Dynamically changes the minimum memory requirements of the job environment containing the calling task.

Table 3. Interrupt Management System Calls

System Call	Function Performed
RQ\$SET\$INTERRUPT	Assigns an interrupt handler and, if desired, an interrupt task to the specified interrupt level. Usually the calling task becomes the interrupt task.
RQ\$RESET\$INTERRUPT	Disables an interrupt level, and cancels the assignment of the interrupt handler for that level. If an interrupt task was assigned, it is deleted.
RQ\$GET\$LEVEL	Returns the number of the highest priority interrupt level currently being processed.
RQ\$SIGNAL\$INTERRUPT	Used by an interrupt handler to signal the associated interrupt task that an interrupt has occurred.
RQ\$WAIT\$INTERRUPT	Used by an interrupt task to SLEEP until the associated interrupt handler signals the occurrence of an interrupt.
RQ\$EXIT\$INTERRUPT	Used by an interrupt handler to relinquish control of the System.
RQ\$ENABLE	Enables the hardware to accept interrupts from a specified level.
RQ\$DISABLE	Disables the hardware from accepting interrupts at or below a specified level.

INTERRUPT MANAGEMENT EXAMPLE

Figure 3 illustrates how the iRMX 86 Interrupt System may be used to output strings of characters to a printer. In the example, a mailbox named 'PRINT' is used by all tasks in the system to queue messages to be printed. Application tasks put the characters in segments that are transmitted to the printer interrupt task via the PRINT Mailbox. Once printing is complete, the same interrupt task passes the messages on to another application via the FINISHED Mailbox so that an operator message can be displayed.

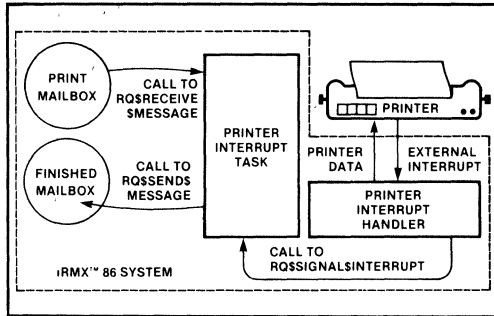


Figure 3. Interrupt Management Example

Basic I/O System

The Basic I/O System (BIOS) provides the direct access to I/O devices needed by real-time applications. The BIOS allows I/O functions to overlap other system functions. In this manner, application tasks make asynchronous calls to the iRMX 86 BIOS, and proceed to perform other activities. When the I/O request must be completed before an application can continue, the task waits at a mailbox for the result of the operation.

Some system calls provided by the BIOS are listed in Table 4.

The Basic I/O System communicates with peripheral devices through device drivers. These device drivers provide the System with four basic functions needed to control and communicate with devices: Initialize I/O, Finish I/O, Queue I/O, and Cancel I/O. Using the device driver interface, users of non-standard devices may write custom drivers compatible with the I/O System.

The iRMX 86 Operating System includes a number of device drivers to allow applications to use standard USART serial communications devices, multiple CRTs and keyboards, bubble memories, diskettes, disks, a Centronics-type parallel printer, and many of Intel's iSBC and iSBX™ device controllers (see Table 8). If an application requires use of a non-standard device, users need only write a device driver to be included with the BIOS, and access it as if it were part of the standard system. For most common random-access devices, this job is further simplified by using standard routines provided with the System. Use of this technique ensures that applications can remain device independent.

Multi-Terminal Support

The iRMX 86 Terminal Support provides line editing and terminal control capabilities. The Terminal Support communicates with devices through simple drivers that do only character I/O functions. Dynamic terminal re-configuration is provided so that attributes such as terminal type and line speed may be changed without modifying the application or the Operating System. Dynamic configuration may be typed in, generated programmatically or stored in a file and copied to a terminal I/O connection.

Table 4. Key BIOS I/O Management System Calls

System Call	Function Performed
RQ\$A\$ATTACH\$FILE	Creates a Connection to an existing file
RQ\$A\$CHANGE\$ACCESS	Changes the types of accesses permitted to the specified user(s) for a specific file
RQ\$A\$CLOSE	Closes the Connection to the specified file so that it may be used again, or so that the type of access may be changed.
RQ\$A\$CREATE\$DIRECTORY	Creates a Named File used to store the names and locations of other Named Files
RQ\$A\$CREATE\$FILE	Creates a data file with the specified access rights
RQ\$A\$DELETE\$CONNECTION	Deletes the Connection to the specified file
RQ\$A\$GET\$FILE\$STATUS	Returns the current status of a specified file.
RQ\$A\$OPEN	Opens a file for either read, write, or update access.
RQ\$A\$READ	Reads a number of bytes from the current position in a specified file
RQ\$A\$SEEK	Moves the current data pointer of a Named or Physical file
RQ\$A\$WRITE	Writes a number of bytes at the current position in a file
RQ\$WAITSIO	Synchronizes a task with the I/O System by causing it to wait for I/O operation results

The iRMX 86 Terminal Support provides automatic translation of control characters to specific control sequences for each terminal. This translation enables applications using standard control characters to function with non-standard terminals. The translation requirements for each terminal can be stored in terminal description files and copied to a connection, as described above.

Disk I/O Performance

Figure 4 shows iRMX 86 performance obtained using the iSBC 215 Winchester Disk and iSBX 218A Diskette Controllers under the specified conditions. The vertical axis is a linear scale of throughput in units of 10,000 bytes per second. The horizontal axis is a logarithmic scale showing the transfer size for the reads and writes. Each data point on the graph indicates the time required for a read/write request of 64K bytes. Therefore each transfer size on the horizontal scale less than 64K was repeated until a total request of 64K was read or written.

Each device driver can be used to interface to a number of separate and, in some cases, different devices

(see Figure 5). The iSBC 215 Device Driver, supplied with the system, is capable of supporting the iSBC 215 Winchester Disk Controller, the iSBC 220 SMD Disk Controller, and the iSBX 218A Flexible Disk Controller (when mounted on an iSBC 215 board). Each device controller may, in turn, control a number of separate device units. In addition, each driver may control a number of like device controllers. This capability allows the use of large storage systems with a minimum of I/O system code to write or maintain.

Extended I/O System

The iRMX 86 Extended I/O System (EIOS) adds a number of I/O management capabilities to simplify access to files. Whereas the BIOS provides users with the basic system calls needed for direct management of I/O resources, many users prefer to have the system perform all the buffering and synchronization of I/O requests automatically. The EIOS allows users to access I/O devices without having to write procedures for buffering data, or to specify particular devices with constant device names.

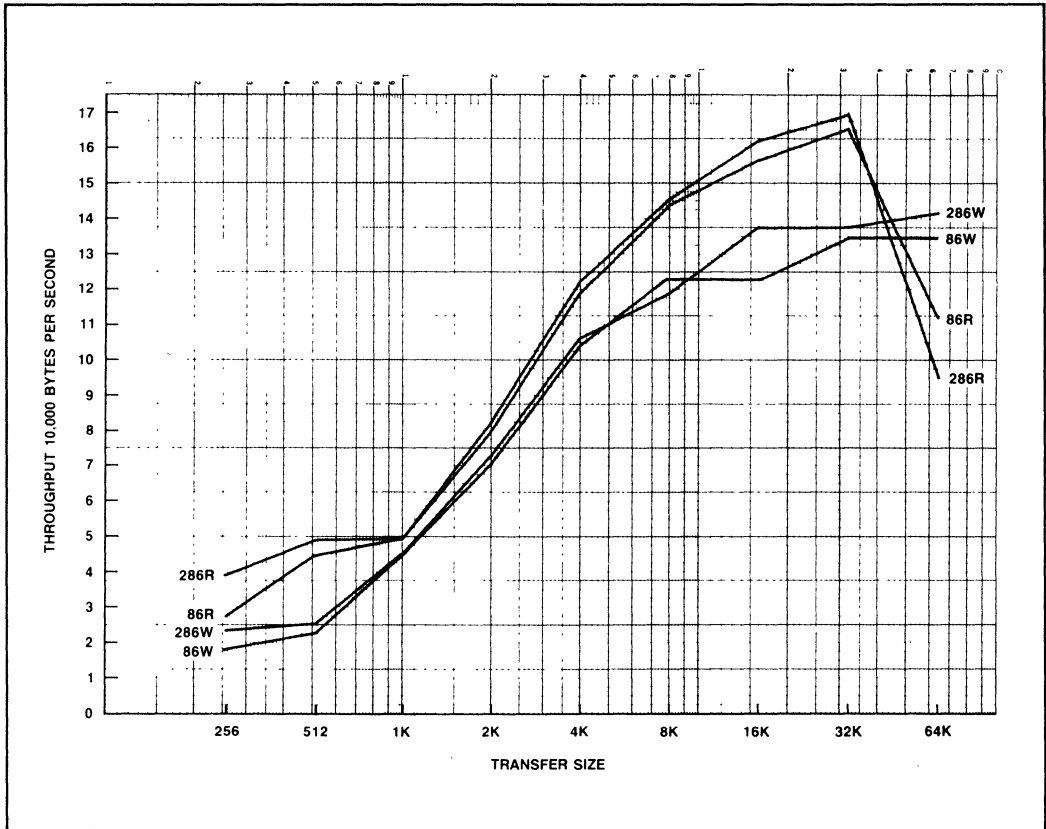


Figure 4. iRMX™ 86 Disk I/O Performance

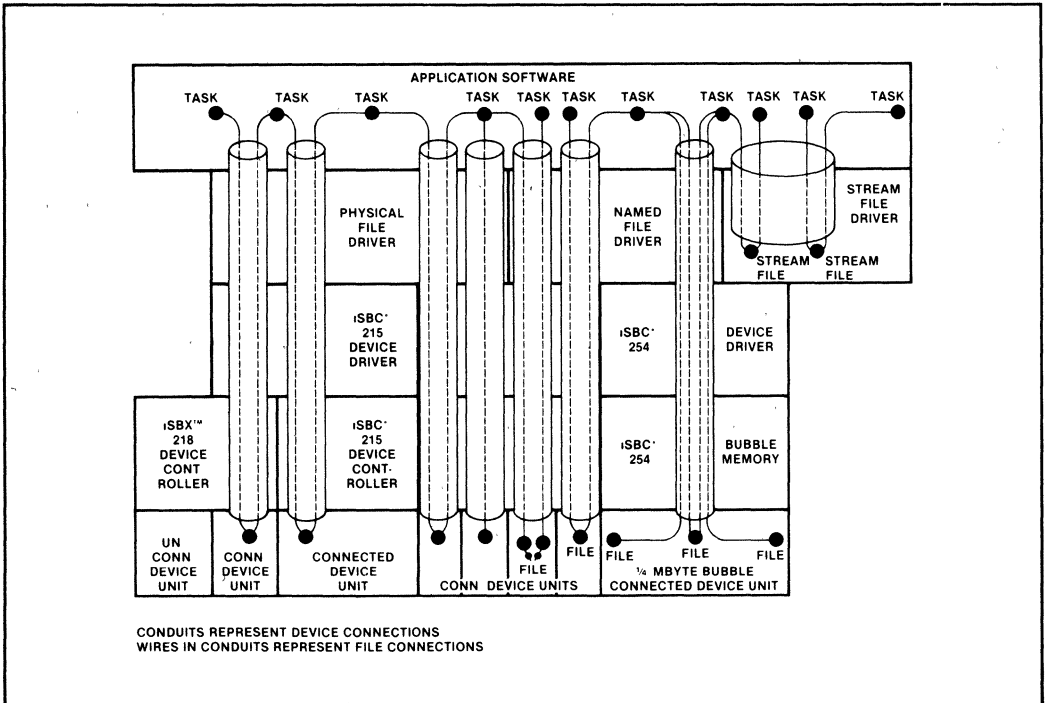


Figure 5. Device Driver and Controller Relationships

By performing device buffering automatically, the iRMX 86 EIOS optimizes accesses to disks and other devices. Often, when an application task asks the System to READ a portion of a file, the System is able to respond immediately with the data it has read in advance of the request. Similarly, the EIOS will not delay a task for writing data to a device unless it is specifically told to, or if its output buffers are filled.

Logical file and device names are provided by the EIOS to give applications complete file and device independence. Applications may send data to the 'line printer' (:LP:) without needing to know which specific device will be used as the printer. This logical name may, in fact, not be a printer at all, but it could be a disk file that is later scheduled for printing.

The EIOS uses the functions provided by the BIOS to synchronize individual I/O requests with results returned by device drivers. Most EIOS system calls are similar to the BIOS calls, except that they appear to suspend the operation of the calling task until the I/O requests are completed.

Two new primitives have been added to the EIOS. These are: RQ\$HYBRID\$DETACH\$DEVICE and RQ\$GET\$LOGICAL\$DEVICE\$STATUS.

RQ\$HYBRID\$DETACH\$DEVICE allows a programmer to temporarily detach a device physically so it can be temporarily attached another way.

RQ\$GET\$LOGICAL\$DEVICE\$STATUS provides information about a logical device: the physical device name, file driver, number of connections to the device, and the owner of the device.

File Management

The iRMX 86 Operating System provides three distinct types of files to ensure efficient management of both program and data files: Named Files, Physical Files, and Stream Files. Each file type provides access to I/O devices through the standard device drivers mentioned earlier. The same device driver is used to access physical and named files for a given device.

NAMED FILES

Named files allow users to access information on secondary storage by referring to a file with its ASCII name. The names of files stored on a device are stored in special files called directories. As directories are themselves named files, the iRMX 86 File System allows directories to contain the names of other directories. Figure 6 illustrates the resulting hierarchical file structure. This structure is useful for isolating file names to particular user applications, and for tailoring system data to the requirements of users and applications sharing storage devices. Using different branches on the directory tree, different users do not have to coordinate in naming their files to ensure unique names.

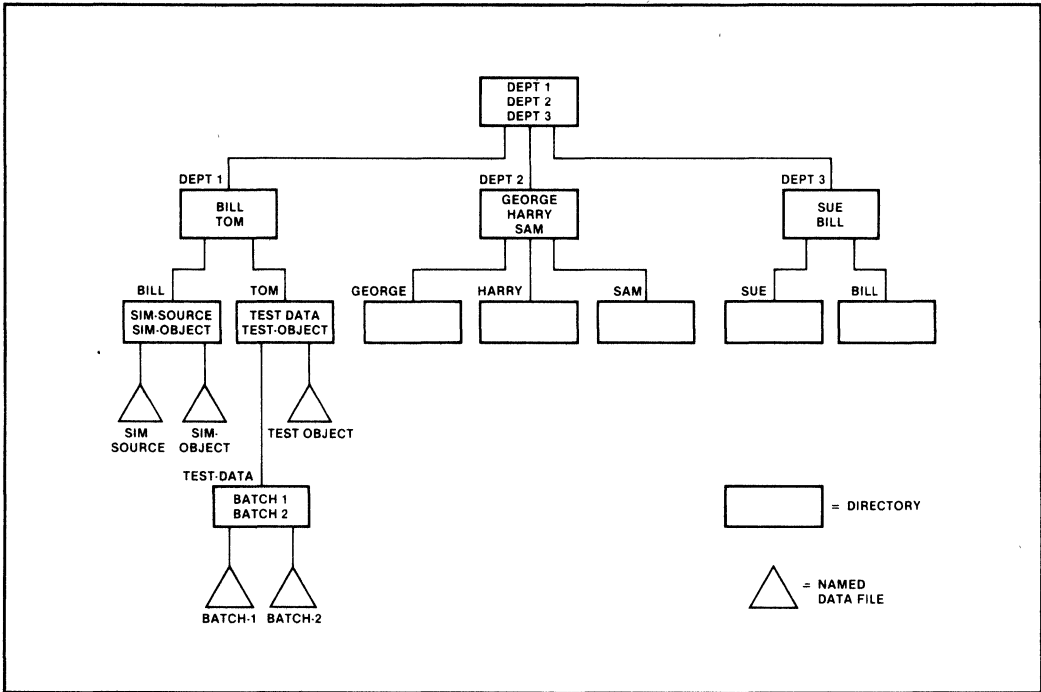


Figure 6. Hierarchical Named File Structure

Whenever a request is made involving a file name, the System will search the appropriate directory in order to find the necessary information about the file's size, access rights, and specific location on the storage device.

The iRMX 86 BIOS uses an efficient format for writing the directory and data information into secondary storage. This standard iRMX 86 format is fully compatible with the ISO Media standard, and other Intel systems such as the iRMX 88 Operating System. This structure enables the system to directly access any byte in a file, often without having to do additional I/O to access space allocation information. The maximum size of an individual file is 4.3 billion bytes.

EASE OF ACCESS

The hierarchical file structure is provided to isolate and organize collections of named files. To give operators fast and simple access to any level within the file tree, an ATTACHFILE command is provided. This command allows operators to create a logical name to a point in the tree so that a long sequence of characters need not be typed each time a file is referred to.

ACCESS PROTECTION

Access to each Named File is protected by the rights assigned to each user by the owner of the file. Rights to read, append, update, and delete may be selectively

granted to other users of the system. In general, users of Named Files are classified into one of two categories: User and World. Users are used when different programmers and programs need to share information stored in a file. The World classification is used when rights are to be granted to all who can use the system.

PHYSICAL FILES

Physical Files allow more direct device access than Named Files. Each Physical File occupies an entire device, treated as a single stream of individually accessible bytes. No access control is provided for Physical Files as they are typically used for such applications as driving a printing device, translating from one device format to another, driving a paper tape device, real-time data acquisition, and controlling analog mechanisms.

STREAM FILES

Stream Files provide applications with a method of using iRMX 86 file management methods for data that does not need to go into secondary storage. Stream Files act as direct channels, through system memory, from one task to another. These channels are very useful to programs, for example, wishing to preserve file and device independence allowing data sent to a printer one time, to a disk file another time, and to another program on a different occasion.

BOOTSTRAP AND APPLICATION LOADERS

Two utilities are supplied with the System to load programs and data into system memory from secondary storage devices:

The iRMX 86 Bootstrap Loader can be configured to a size of less than 1K Bytes of P(ROM), and is typically used to load the initial system from the system disk into memory, and begin its execution. Error reporting and debug switch features have been added to the Bootstrap Loader. When the Bootstrap Loader detects errors such as: file does not exist or device not ready, an error message is reported back to the user. The debug switch will cause the Bootstrap Loader to load the system but not begin its execution. Instead the Bootstrap Loader will pass control to the monitor at the first instruction to be executed by the system.

The Application Loader is typically used by application programs already running in the system to load additional programs and data from any secondary storage device. The Human Interface layer, for example, uses the Application Loader to load the non-resident Human Interface Commands. The Application Loader is capable of loading both relocatable and absolute code as well as program overlays.

Human Interface

The flexibility of the interface between computer controlled machines and their users often determines the usability and ultimate success of the machines. Table 11 lists iRMX 86 Human Interface functions giving users and applications simple access to the file and system management capabilities described earlier. The process, interrupt, and memory management functions described earlier, are performed automatically for Human Interface users.

MULTI-USER ACCESS

Using the multi-terminal support provided by the BIOS, the iRMX 86 Human Interface can support several simultaneous users. The real-time nature of the system is maintained by providing a priority for each user, and using the event-driven iRMX 86 Nucleus to schedule tasks. High-performance interrupt response is guaranteed even while users interact with various application packages. For example, multi-terminal support allows one person to be using the iRMX 86 Editor, while another compiles a FORTRAN 86 or PASCAL 86 program, while several others load and access applications.

Each terminal attached to the iRMX 86 multi-user Human Interface is automatically associated with a user, a memory pool, and an initial program to run when the terminal is connected. This association is made using a file that may be changed at any time. Changes are effective the next time the system is initialized.

The initial program specified for each terminal can be a special application program, a custom Human Inter-

face, or the standard iRMX 86 Command Line Interpreter (CLI). For example, you may choose to use the Microsoft Basic Interpreter as this initial program. After system start-up, each terminal user would be able to run the interpreter without asking for it to be loaded. From the BASIC interpreter, an operator, for example, could run a data collection program, written in BASIC, that communicates with several laboratory instruments, and prints charts and reports based on certain test results. When finished entering, changing, or running a BASIC program, the terminal would remain in BASIC for the next user.

Specifying an application program as a terminal's initial program makes the interface between operators and the computer system much simpler. Each operator need only be aware of the function of a particular application; not needing to interact with any unfamiliar functions also available on the application system.

Specifying the standard iRMX 86 Human Interface CLI as the initial program enables users of the terminals to access all iRMX 86 functions. This CLI makes it easy to manage iRMX 86 files, load and execute Intel-supplied and custom programs, and submit command files for execution.

FEATURE OVERVIEW

The iRMX 86 Operating System is well suited to serve the demanding needs of real-time applications executing on complex microprocessor systems. The iRMX 86 System also provides many tools and features needed by real-time system developers and programmers. The following sections describe features useful in both the development and execution environments. The description of each feature outlines the advantages given to hardware and software engineers concerned with overall system cost, expandability with custom and industry standard options, and long-term maintenance of iRMX 86-based systems. The development environment features also describe the ease with which the iRMX 86 Operating System can be incorporated into overall system designs.

Execution Environment Features

REAL-TIME PERFORMANCE

The iRMX 86 Operating System is designed to offer the high performance, multi-tasking functions required by real-time systems. Designers can make use of the latest VLSI devices such as the 8087 or 80287 Numeric Processor Extension, and the 80130 Operating System Firmware Component to improve their system cost/performance ratio or the iMMX™ 800 MULTIBUS® Message Exchange software package to divide and coordinate various system activities among multiple processors. Typical iRMX 86 system performance characteristics are shown in Table 5.

Many real-time systems require high performance operation. To meet this requirement, all of iRMX 86 can be put into zero wait-state P(ROM). This approach eliminates the possibility of disk access times slowing down performance, while allowing system designers to take advantage of high performance memory devices.

CONFIGURABILITY

The iRMX 86 Operating System is configurable by system layer, and by system call within each layer. In addition all the I/O port addresses used by the System are configurable by the user. This flexibility gives designers the freedom to choose configurations of hardware and software that best suit their size and functional requirements. Two example configurations are shown in Figure 7.

Table 5. iRMX™ Real-Time Performance Using iSBC® 86/30 and iSBC® 286/10 Single Board Computers

Real-Time Function	iSBC® 86/30 Execution Time (msec)	iSBC® 286/10 Execution Time (msec)
Suspend Task	1.02	0.83
Interrupt Latency (to handler)	0.29 (Max)	0.20 (Max)
Interrupt Latency (to handler)	0.02 (Typical)	0.03 (Typical)
Context Switch Caused By Interrupt	0.84 (Max)	0.78 (Max)
Send Message (no context switch)	0.32	0.25
Send Message (with context switch)	0.58	0.49
Send Control (no context switch)	0.21	0.16
Send Control (with context switch)	0.64	0.54
Receive Control (no waiting)	0.26	0.19

Context switch time is the time between executing in the context of a task, and the first instruction to execute in the context of another task.

The execution times shown in Column 2 were measured using an 8MHz iSBC Single Board Computer, 256K on-board RAM, and all program and data stored in on-board RAM.

The execution times shown in Column 3 were measured using a 5MHz iSBC 286/10 Single Board Computer, no on-board RAM, and all program and data stored in LBX RAM.

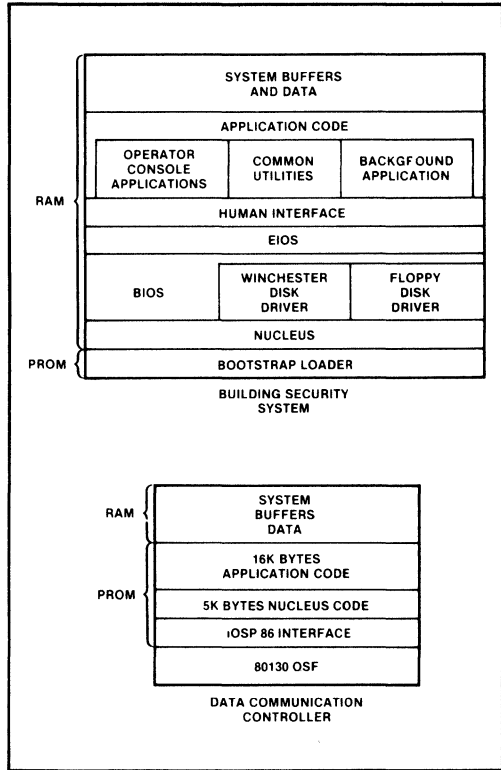


Figure 7. Typical iRMX™ 86 Configurations

Most configuration options are selected during system design stages. Others may be selected during system operation. For example, the amount of memory devoted to queues within a Mailbox can be specified at the time the Mailbox is created. Devoting more memory to the Mailbox allows more messages to be transmitted to other tasks without having to degrade system performance to allocate additional memory dynamically.

The chart shown in Table 6 indicates the actual memory size required to support these different configurations of the iRMX 86 System. Systems requiring only Nucleus level functions may require no more than 13K bytes for the Operating System. (Use of the iAPX 86/30 requires only 4K bytes of RAM, 7K bytes of initialization code in EPROM and the 16K bytes of code in the 80130.) Other applications, needing I/O management functions, may select portions of additional layers that fit their needs and size constraints.

This configurability also applies to the Terminal Handler, Dynamic Debugger, and System Debugger. The Terminal Handler provides a serial terminal interface in a system that otherwise doesn't need an I/O system. Either one of the debuggers need to be included only as debugging tools (usually only during system development).

Table 6. iRMX™ 86 Configuration Size Chart

System Layer	Min. ROMable Size	Max. Size	Data Size
Bootstrap Loader	1K	1.5K	6K*
Nucleus	10.5K	24K	2K
BIOS	26K	78K	1K
Application Loader	4K	10K	2K
EIOS	10.5K	12.5K	1K
Human Interface	22K	22K	15K
UDI	8K	8K	0
Terminal Handler	3K	3K	0.3K
System Debugger	20K	20K	1K
Dynamic Debugger	28.5K	28.5K	1K
Human Interface Commands			116K
Interactive Configuration Utility			308K

* Usable by System after bootloading

MULTI-PROCESSING

The resources provided by a single processor are often not enough to perform certain functions. With the standard interfaces provided by the iMMX 800 MULTIBUS Message Exchange package, the iRMX 86 Operating System supports a loosely-coupled multi-processing environment. Task running on one processor may communicate with tasks running on other processors, even if they operate under different operating systems. The iMMX 800 software is capable of sending messages over the MULTIBUS to tasks operating under either the iRMX 88 Executive, or the iRMX 86 Operating System. Using this message exchange mechanism, applications may increase their system performance quite easily, improve overall interrupt response, gain access to the iSBC® 550 Ethernet Controller, and leave room for future product enhancements.

MULTI-USER ACCESS

Many real-time systems must provide a variety of users access to system control functions and collected data. The iRMX 86 System provides easy-to-use support for applications to access multiple terminals. It also enables multiple and different users to access different applications concurrently.

Figure 8 illustrates a typical iRMX 86 application simultaneously supporting multi-terminal data collection and real-time environments. Shown is a group of terminals used by machinists on a shop floor to communicate with a job management program, a building security system that constantly monitors energy usage requirements, a system operator console capable of accessing all system functions, and a group of terminals in the Production Engineering department used to monitor job costs while developing new device control specifications instructions. The iSBC 544 Intelligent Terminal Interface supports multiple user terminals without degrading system performance to handle character I/O.

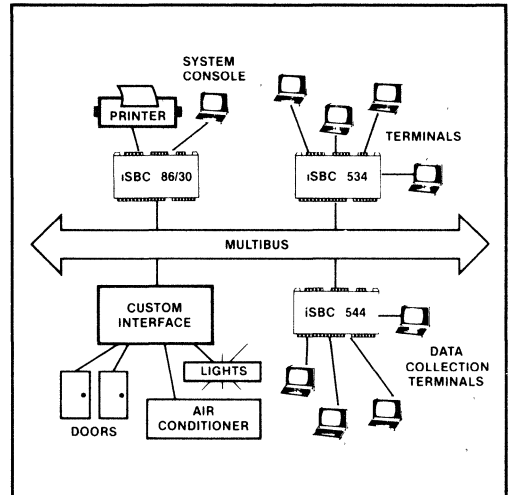


Figure 8. Multi-Terminal and Multi-User Real-Time System

EXTENDABILITY

The iRMX 86 Operating System provides three means of extensions. This extendability is essential for support of OEM and volume end user value added features. This ability is provided by: user-defined operating system calls, user-defined objects (similar to Jobs, Tasks, etc.), and the ability to add functions later in the product life cycle. The modular, layered structure of the System easily facilitates later additions to iRMX 86 applications. User-defined objects are supported by the functions listed in Table 7.

Using standard iRMX 86 system calls, users may define custom objects, enabling applications to easily manipulate commonly used structures as if they were part of the original operating system.

Table 7. User Extension System Calls

System Call	Function Performed
RQ\$CREATE\$COMPOSITE	Creates a custom object built of previously defined objects.
RQ\$DELETE\$COMPOSITE	Deletes the custom object, but not the various objects from which it was built
RQ\$INSPECT\$COMPOSITE	Returns a list of Token Identifiers for the component objects from which the specified composite object is built.
RQ\$ALTER\$COMPOSITE	Replaces a component object of a composite object.
RQ\$CREATE\$EXTENSION	Creates a new type of object and assigns a mailbox used for collecting these objects when they are deleted.
RQ\$DELETE\$EXTENSION	Deletes an extension definition.

EXCEPTION HANDLING

The System includes predefined exception handlers for typical I/O and parameter error conditions. The error handling mechanism is both configurable and extendable.

SUPPORT OF STANDARDS

The iRMX 86 Operating System supports the many hardware and software standards needed by most application systems to ensure that commonly available hardware and software packages may be interfaced with a minimum of cost and effort. The iRMX 86 System supports the iSBC family of products built on the Intel MULTIBUS (IEEE Standard 796), and a number of standard software interfaces such as the UDI and the common device driver interface (See Figure 9). The procedural interfaces of the UDI are listed in Table 9.

The Operating System includes support for the proposed IEEE 80-bit extended real-variable format of the 8087 Numeric Data Processor, and the IEEE 796 (MULTIBUS) hardware interface. Other standards such

as the iRMX 800 MULTIBUS Message Exchange, and an Ethernet communication interface are supported by optional software packages available to run on the iRMX 86 System.

SPECTRUM OF CPU PERFORMANCE

The iRMX 86 Operating System supports a broad range of Intel processors. In addition to support for iAPX 86 and 88 based systems, the iRMX 86 system has been enhanced to support iAPX 186, 188, and 286 (Real Address Mode)-based Systems. This new support enables the user to take advantage of the faster speed and higher performance of Intel's 286 based microprocessors such as the iSBC 286/10 single board computer. By choosing the appropriate CPU, designers can choose from a wide range of performance options, without having to change application software.

COMPONENT LEVEL SUPPORT

The iRMX 86 System may be tailored to support specific hardware configurations. In addition to system memory,

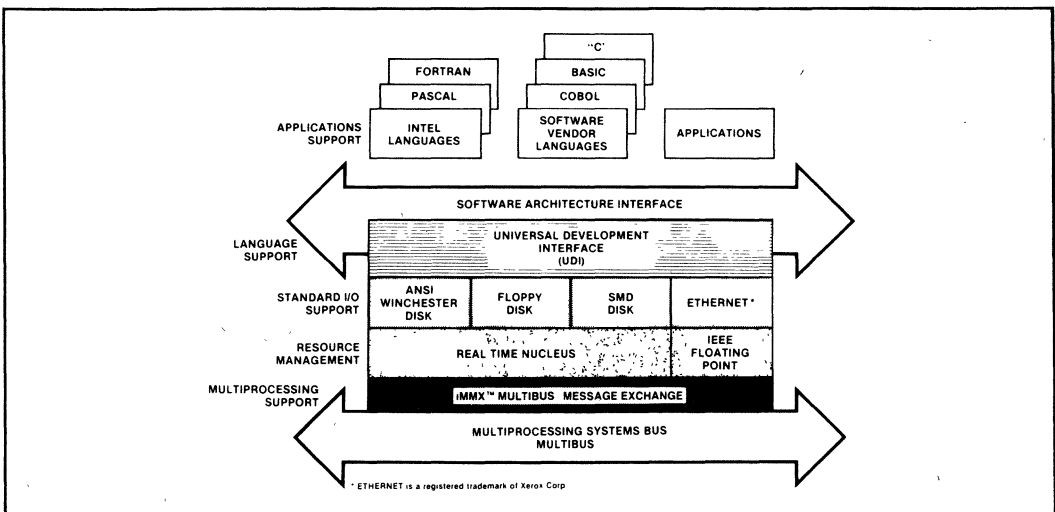


Figure 9. iRMX™ 86 Standard Interfaces

only an iAPX 86, iAPX 88, iAPX 186, iAPX 188, or iAPX 286 microprocessor, an 8259A Programmable Interrupt Controller (PIC), and either an 8253, 8274, or 82530 Programmable Interval Timer (PIT) are required as follows:

- iAPX 86 and iAPX 88 systems need either:
 - 8253 PIT and 8259A PIC (master) or
 - 80130 firmware (PIC is master)
- iAPX 186 and iAPX 188 systems where 186 PIC is slave, needs either:
 - 8253 PIT and 8259A PIC (master) or
 - 80130 firmware (PIC is master)

where 186 PIC is master:

- Uses 186 PIT for the system clock; no external PIT is needed
- Can use either
 - 186 PIC (master) only or
 - 8259A/80130 PIC (slave)
- iAPX 286 systems need
 - 8253 PIT and 8259A PIC.

Alternatively, the iRMX 86 Operating System may be used in conjunction with the 80130 Operating System Firmware Component that not only provides these hardware functions, but eliminates the need for approximately 16K bytes of the iRMX 86 Nucleus code (see Figure 7). For systems requiring extended mathematics capability, an 8087 or 80287 Numeric Data Processor may be added to perform these functions up to 100 times faster than equivalent software. For applications servicing more than 8 interrupt sources, additional 8259A's may be configured as slave controllers.

BOARD LEVEL SUPPORT

The iRMX 86 Operating System includes device drivers to support a broad range of MULTIBUS device controllers. The particular boards and types of devices supported are listed in Table 8. The device controllers all adhere to industry standard electrical and functional interfaces.

In addition to the on-CPU board terminal drivers, the iRMX 86 BIOS includes two iSBC board-level device drivers to support multiple terminal interfaces:

The iSBC 544 Intelligent Four-Channel Terminal Interface Device Driver provides support for multiple controllers each supporting up to four standard RS232 terminals. The iSBC 544 driver takes advantage of an on-board 8085 processor to greatly reduce the system processor time required for terminal I/O by locally managing input and output buffers. The iSBC 544 firmware provided with the operating system can off-load the system CPU by as much as 75% when doing character outputting.

The iSBC 534 Four-Channel USART Controller Device Driver also provides support for multiple controller

boards each supporting up to four standard RS232 terminals.

The new RAM disk feature in iRMX 86 makes a portion of the memory address space look like a disk drive to the I/O system.

Table 8. Supported Devices

iSBC® Device Controller	Description
iSBC® 86,88	Serial Port to CRT, Parallel Port to Centronics-type Printer, Interval Timer and Interrupt Controller.
iSBC® 186/03	Small Computer System Interface (SCSI) Supporting All Random Access "Extended Standard" SCSI/SASI hard disk controllers.
iSBC® 204	Single Density Diskette.
iSBC® 206	Cartridge-Type Hard Disk.
iSBC® 208	Single & Double Density, Single & Double Sided, 8" & 5.25" Diskettes.
iSBC® 215(G)	Standard Winchester Disks.
iSBX® 218	Single or Double density, Single or double sided, 8-inch diskettes (when used on an iSBC 215(G)).
iSBX® 218A	Single or Double Density, Single or Double Sided, 8" & 5.25" Diskette (when used on an iSBC 215G Winchester Controller).
iSBC® 220	Standard Storage Module Board.
iSBX® 251	Bubble Memory Multimodule Board.
iSBC® 254(S)	Bubble Memory Board.
iSBX® 351	1-Channel Serial Port to CRTs, Modems.
iSBC® 534,544	4-Channel Serial Ports to CRTs, Modems.
iSBX™ 270	Black and White CRTs and full ASCII keyboards.
NOTE: (G) = optional iSBC 215, iSBC 215B, or iSBC 215G (S) = optional iSBC 254 or iSBC 254S	

Development Environment Features

The iRMX 86 Operating System supports the efficient utilization of programming time by providing important tools for program development. Some of the tools necessary to develop and debug real-time systems are included with the Operating System. Others, such as language compilers, are available from Intel and from leading Independent Software Vendors.

LANGUAGES

The iRMX 86 Operating System supports 31 standard system calls known as the Universal Development Interface (UDI). Figure 9 shows the iRMX 86 standard interfaces to many compilers and language translators, including the iAPX 86 and 88 Macro Assembler; the PASCAL 86/88, PL/M 86/88, FORTRAN 86/88 and C86 compilers available from Intel. Also included are other

Intel development tools, language translators and utilities available from other vendors. Any application that ran on the iRMX 86 Release 5 Universal Runtime Interface (URI) will run on the iRMX 86 Release 6 UDI. The full set of UDI calls (which includes the URI system calls) is required to run a compiler.

These standard software interfaces (the UDI) ensure that users of the iRMX 86 Operating System may transport their applications to future releases of the iRMX 86 Operating System and other Intel and independent vendor software products. The calls available in the UDI are shown in Table 9.

Table 9. UDI System Calls

System Call	Function Performed
Memory Management: DQ\$ALLOCATE DQ\$FREE DQ\$GET\$SIZE* DQ\$RESERVE\$I\$O\$MEMORY*	Creates a Segment of a specified size Returns the specified segment to the System Returns the size of the specified Segment. Reserves memory to OPEN and ATTACH files
File Management: DQ\$ATTACH DQ\$CHANGE\$ACCESS* DQ\$CHANGE\$EXTENSION DQ\$CLOSE DQ\$CREATE DQ\$DELETE DQ\$DETACH DQ\$OPEN DQ\$GET\$CONNECTION\$STATUS* DQ\$FILE\$INFO* DQ\$READ DQ\$RENAME* DQ\$SEEK DQ\$TRUNCATE DQ\$WRITE	Creates a Connection to a specified file. Changes the user access rights associated with a file or directory. Changes the extension of a file name in memory. Closes the specified file Connection. Creates a Named File. Deletes a Named File. Closes a Named File and deletes its Connection Opens a file for a particular type of access. Returns the current status of the specified file Connection Returns data about a file Connection. Reads the next sequence of bytes from a file. Renames the specified Named File. Moves the position pointer of a file. Truncates a file. Writes a sequence of bytes to a file.
Process Management: DQ\$EXIT DQ\$OVERLAY* DQ\$SPECIAL DQ\$TRAP\$CC	Exits from the current application job. Causes the specified overlay to be loaded Performs special I/O related functions on terminals with special control features. Captures control when CNTRL/C is typed.
Exception Handling: DQ\$GET\$EXCEPTION\$HANDLER DQ\$DECODE\$EXCEPTION DQ\$TRAP\$EXCEPTION	Returns a pointer to the program currently being used to process errors. Returns a short description of the specified error code. Identifies a custom exception processing program for a particular type of error.
Application Assistance: DQ\$DECODE\$TIME DQ\$GET\$ARGUMENT* DQ\$GET\$SYSTEM\$ID* DQ\$GET\$TIME* DQ\$SWITCH\$BUFFER	Returns system time and date in binary and ASCII character format. Returns the next argument from the character string used to invoke the application program. Returns the name of the underlying operating system supporting the UDI. Returns the current time of day as kept by the underlying operating system Selects a new buffer from which to process commands.

* Calls available only through the UDI

The high performance of the iRMX 86 Operating System enhances the throughput of compilers and other development utilities. Table 10 indicates the average performance of typical development environment functions operating in the same configuration described in Figure 4.

Table 10. Development Environment Performance

Function	Average Execution Time
Directory Command (S Format with 25 files)	5.3 sec
Load the COPY Command	1.2 sec
Copy a 1K Byte File (Winchester to Winchester)	1.0 sec
Copy a 16K Byte File	1.7 sec
Copy a 64K Byte File	3.9 sec
Copy a 1K Byte File (Winchester to Diskette)	1.4 sec
Compile PL/M 86	393 lpm
Compile PASCAL 86 Program	453 lpm

TOOLS

Certain tools are necessary for the development of microcomputer applications. The iRMX 86 Human Interface includes many of these tools as non-resident commands. They can be included on the system disk of a application system, and brought into memory when needed to perform functions as listed in Table 11.

Table 11. Major Human Interface Utilities

Command	Function
BACKUP	Copy directories and files from one device to another.
COPY	Copy one or more files to one or more destination files.
CREATEDIR	Create a directory file to store the names of other files
DIR	List the names, sizes, owners, etc. of the files contained in a directory.
ATTACHFILE	Give a logical name to a specified location in a file directory tree.
PERMIT	Grant or rescind user access to a file.
RENAME	Change the name of a file
SUBMIT	Start the processing of a series of commands stored in a file.
SUPER	Change operator's ID to that of the System Manager with global access rights and privileges.

Table 11. Major Human Interface Utilities (Con.t.)

Command	Function
TIME	Set the system time-of-day clock.
VERIFY	Verify the structure of an iRMX™ 86 Named File volume, and check for possible disk data errors.

INTERACTIVE CONFIGURATION UTILITY

The iRMX 86 Operating System is designed to provide OEMs the ability to configure for specific system hardware and software requirements. The Interactive Configuration Utility (ICU) builds iRMX 86 configurations by asking appropriate questions and making reasonable assumptions. It runs on either an Intellec® Series III development system or iRMX 86 development system that includes a hard disk and the UDI. Table 12 lists the hardware and support software requirements of different iRMX 86 development system environments.

Table 12. iRMX™ Development Environment

Intellec® Series III: MDS 313 PL/M 86/88 Compiler One hard disk and one diskette drive
iRMX™ 86 Development System iRMX™ 860 ASM 86 Assembler and Utilities iRMX™863 PL/M 86/88 Compiler iSDM 86 or 286 System Debug Monitor 512K Bytes of RAM 5M Byte On-Line Storage and one double-density diskette drive
SYSTEM 86/300 or 286/300 Series Microcomputer System Basic configuration

Figure 10 shows one of the many screens displayed during the process of defining a configuration. It shows the abbreviations for each choice on the left, a more complete description with the range of possible answers in the center, and the current (sometimes default) choice on the right. The bottom of the screen shows three changes made by the operator (lower case lettering), and a request for help on the Exception Mode question. In response to a request for help, the ICU displays an additional screen outlining possible choices and some overall system effects.

The ICU requests only information required as a result of previous choices. For example, if no Extended I/O System functions are required, the ICU will not ask any further questions about the EIOS. Once a configuration session is complete, the operator may save all the information in a file. Later when small changes are necessary, this file can be modified. A completely new session is not required.

Nucleus		
(ASC)	All Sys Calls [Yes/No]	Yes
(PV)	Parameter Validation [Yes/No]	Yes
(ROD)	Root Object Directory Size [0 - OFF0h]	0014H
(MTS)	Minimum Transfer Size [0 - OFFFH]	0040H
(DEH)	Default Exception Handler [Yes/No/Deb/Use]	Yes
(NEH)	Name of Ex Handler Object Module [1 - 32chs]	
(EM)	Exception Mode [Never/Program/Environ/All]	Never
(NR)	Nucleus in ROM [Yes/No]	No
Enter Changes [Abbreviations ? = new-value] ASC = N		
pv = no		
rod = 48		
em ?		

Figure 10. ICU Screen for iRMX™ 86 Nucleus

REAL-TIME DEBUGGING TOOLS

The iRMX 86 Operating System supports three distinct debugging environments: Static, Dynamic, and Post-Mortem. While the iRMX 86 Operating System does support a multi-user Human Interface, these real-time debugging aids are usually most useful in a single-user environment where modifications made to the system cannot affect other users.

System Debugger

The static debugging aid is the iRMX 86 System Debugger. This debugger is an extension of the iSDM 86 and the iSDM 286 System Debug Monitors. The System Debugger provides static debugging facilities when the system hangs or crashes, when the Nucleus is inadvertently overwritten or destroyed, or when synchronization requirements prevent the debugging of certain tasks. The System Debugger stops the system and allow you to examine the state of the system at that instant, and allows you to:

- Identify and interpret iRMX 86 system calls.
- Display information about iRMX 86 objects.
- Examine a task's stack to determine system call history.

iRMX™ 86 Dynamic Debugger

The iRMX 86 Dynamic Debugger runs as part of an iRMX 86 application. It may be used at any time during program development, or may be integrated into an OEM system to aid in the discovery of latent errors. The Dynamic Debugger can be used to search for errors in any task, even while the other tasks in the system are running. The iRMX 86 Dynamic Debugger communicates with the developer via a terminal handler that supports full line editing.

System Crash/Dump Analyzer

The often difficult job of debugging real-time applications is made much simpler with the System Crash/Dump Analyzer. The analyzer allows program developers to record system memory for later analysis even if the system has halted. This analysis lists such vital information as which jobs have active tasks, which system queues contain which tasks, and what segments contain which data.

PARAMETER VALIDATION

Some iRMX 86 System Calls require parameters that may change during the course of developing iRMX 86 applications. The iRMX 86 Operating System includes an optional set of routines to validate these parameters to ensure that correct numeric values are used and that correct object types are used where the System expects to manipulate an object. For systems based only on the iRMX 86 Nucleus, these routines may be removed to improve the performance and code size of the System once the development phase is completed.

START-UP SYSTEMS

Two ready-to-run, multi-user start-up systems are included in the iRMX 86 Operating System package. These iRMX 86 start-up systems are fully configured, multi-user iRMX 86 Operating Systems ready to be loaded into memory by the Bootstrap Loader. Both start-up systems are configured to include all of the system calls for each layer and most of the features provided by iRMX 86. iRMX start-up systems include UDI support so that users may run languages such as PL/M-86, Pascal, FORTRAN, and software packages from independent vendors.

The start-up system for the iAPX 86 processor is configured for Intel SYSTEM 86/300 Series microcomputers with a minimum of 384K bytes of RAM. The following devices are supported.

- iSBC 215/iSBX 218 or iSBC 215G/iSBX 218A
- iSBC 254(S)
- Line Printer
- 8251A Terminal Driver
- iSBC 544 Terminal Driver

The start-up system for the iAPX 286 processor is configured for Intel SYSTEM 286/300 Series microcomputers with a minimum of 512K bytes and a maximum of 896K bytes of RAM. The following devices are supported.

- iSBC 208
- iSBC 215/iSBX 218 or iSBC 215G/iSBX 218A
- iSBC 254(S)
- Line Printer for iSBC 286/10
- 8274 Terminal Driver
- iSBC 544 Terminal Driver

Either system will run without hardware or software configuration changes and can be reconfigured on a standard system with at least 512K bytes of RAM. Definition files are also included for iSBC 186/03, 186/51 and 188/48 configurations.

This start-up system may be used to run the ICU (if a Winchester disk is attached to the system) to develop custom configurations such as those pictured in Figure 8. As shipped, the Human Interface supports a single user terminal. However, the Start-up System terminal configuration file may be altered easily to support from two to five users.

SPECIFICATIONS

Supported Software Products

iRMX 860	iRMX 86 Development Utilities Package, including the iAPX 86 and 88 Linker, Locator, Macro Assembler, Librarian, and the iRMX 86 Editor.
iRMX 861	PASCAL 86/88 Compiler
iRMX 862	FORTRAN 86/88 Compiler
iRMX 863	PL/M 86/88 Compiler
iRMX 864	TX Screen-oriented Editor
iMMX 800	MULTIBUS Message Exchange software package for iRMX 86, and 88 application systems
iOSP 86	Support Package for iAPX 86/30, 88/30, 186/30, and 188/30 Operating System Processors
iRMX PSCOPE 86	High Level Language Debugger

Supported Hardware Products

COMPONENTS

iAPX 86 and 88 Microprocessors
 iAPX 186 and 188 Microprocessors
 iAPX 286 Microprocessors (Real Address Mode only)
 8087 Numeric Data Processor Extension
 80287 Numeric Data Processor Extension
 iAPX 86/30 (80130) Operating System Firmware Component
 8253 and 8254 Programmable Interval Timers
 8259A Programmable Interrupt Controller
 8251A USART Terminal Controller
 8255 Programmable Parallel Interface
 8274 Terminal Controller
 82530 Serial Communications Controller

iSBC® MULTIBUS BOARD AND SYSTEM PRODUCTS

iSBC 86/12A, 86/05, 86/14, 86/30, 86/35, 88/25, and 88/40 Single Board Computers
 iSBC 186/03 Single Board Computer
 iSBC 186/51 Ethernet Controller
 iSBC 188/48 Communications Controller
 iSBC 286/10 Single Board Computer(Real Address Mode only)
 iSBC 204 Diskette Controller
 iSBC 206 Hard Disk Controller
 iSBC 208 Diskette Controller

iSBC 215(G) Winchester Disk Controller
 iSBX 218(A) Flexible Diskette Multi-Module Controller
 iSBC 220 SMD Disk Hard Controller
 iSBC 254(S) Bubble Memory System
 iSBC 534 4-Channel Terminal Interface
 iSBC 544 Intelligent 4-Channel Terminal Interface and Controller
 iSBX 251 Bubble Memory Multi-Module
 iSBX 350 Parallel Port (Centronics-type Printer Interface)
 iSBX 351 Serial Communications Port
 iSBX 270 CRT Light Pen and Keyboard Interface
 SYSTEM 86/300 Family
 SYSTEM 286/300 Family

AVAILABLE LITERATURE

The iRMX 86 Documentation Set is comprised of the following four volumes of reference manuals. Order numbers are associated with these four volumes only.

iRMX 86 INTRODUCTION AND OPERATOR'S REFERENCE MANUAL FOR RELEASE 6
 Order Number: 146545-001

Introduction to the iRMX 86 Operating System

iRMX 86 Operator's Manual

iRMX 86 Disk Verification Utility Reference Manual

iRMX 86 PROGRAMMERS REFERENCE MANUAL FOR RELEASE 6, PART I

Order Number: 146546-001

iRMX 86 Nucleus Reference Manual

iRMX 86 Basic I/O System Reference Manual

iRMX 86 Extended I/O System Reference Manual

iRMX 86 PROGRAMMERS'S REFERENCE MANUAL FOR RELEASE 6, PART II

Order Number: 146547-001

iRMX 86 Application Loader Reference Manual

iRMX 86 Human Interface Reference Manual

iRMX 86 Universal Development Interface Reference Manual

Guide to Writing Device Drivers for iRMX 86 and iRMX 88 I/O Systems

iRMX 86 Programming Techniques

iRMX 86 Terminal Handler Reference Manual

iRMX 86 Debugger Reference Manual

iRMX 86 System Debugger Reference Manual

iRMX 86 Crash Analyzer Reference Manual

iRMX 86 Bootstrap Loader Reference Manual

iRMX 86 INSTALLATION AND CONFIGURATION GUIDE FOR RELEASE 6
Order Number: 146548-001

iRMX 86 Installation Guide

iRMX 86 Configuration Guide

Master Index for Release 6 of the iRMX 86 Operating System

Application Notes

Ap Note 130 — Using Operating System Processors to Simplify Microcomputer Designs. (Order Number: 230786-001)

Ap Note 174 — Optimizing the iRMX 86 Operating System Performance on System 86/310 and System 86/330 (Order Number: 230990-001)

Training Courses

The iRMX 86 Operating System

Customer Seminars

Contact local Intel Sales Office for details on available video-tape and slide presentations.

ORDERING INFORMATION

The iRMX 86 Operating System is available under a number of different licensing options as noted here. Source listings are available on microfiche. Reconfigurable object libraries are provided on double density ISIS-formatted diskettes or on either double density, single sided iRMX 86-formatted 8" diskettes, or double density, double sided, 5.25" diskettes. ISIS-format diskettes may be used on Intel Intellec Development Systems. The iRMX 86-format may be used on any iRMX 86-based system supporting the appropriate compilers and development environment.

The OEM license options listed here allow users to incorporate the iRMX 86 Operating System into their applications. Each use requires payment of an Incorporation Fee.

ORDER CODE	DESCRIPTION
iRMX 86 KIT BRO:	Double density, single-sided 8" ISIS format OEM license
iRMX 86 KIT ERO:	Double density, single sided 8" iRMX 86-Format OEM license for use on iRMX 86-based environments.

iRMX 86 KIT JRO: Double density, double sided 5.25" iRMX 86-Format OEM license for use on iRMX 86-based environments.

Other licensing options include prepayment of all future incorporation fees, single use rights for a single machine, use at a second development site, one year update service extensions, the right to make copies for additional development systems, and source listing materials.

Each option includes 90 days of support service that provides the quarterly iRMX 86 Technical Report, Software Problem Report Service, and copies of System Updates that occur during this period. Except for source listings, all initial licenses include a complete set of iRMX 86 Documentation.

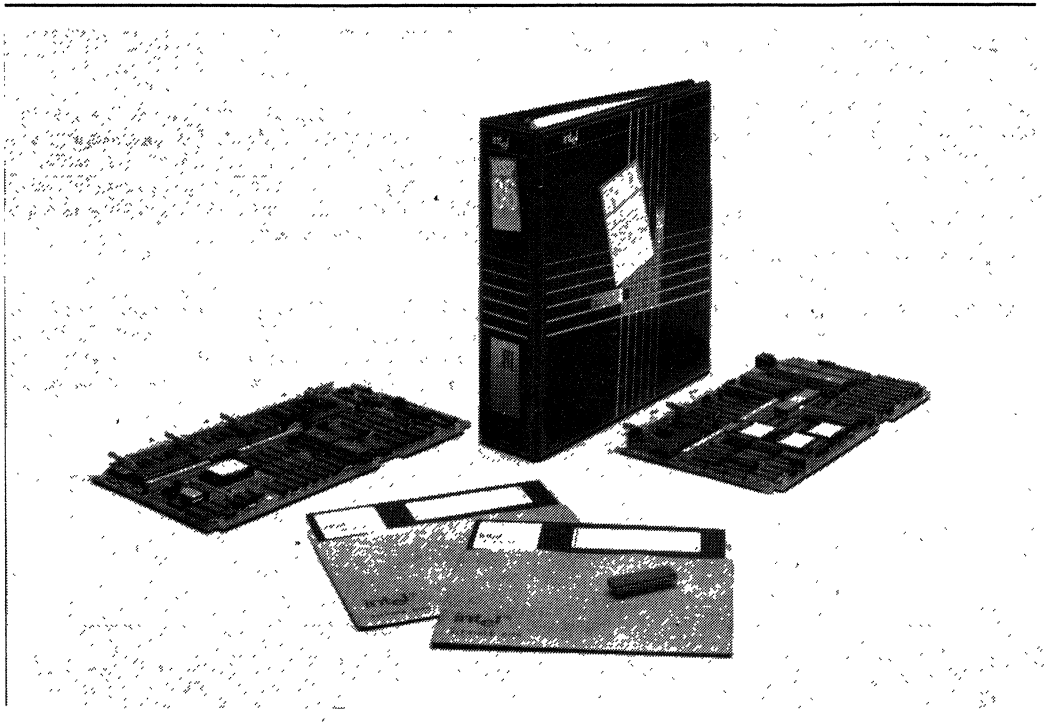
As with all Intel software, purchase of any of these options requires the execution of a standard Intel Master Software License. The specific rights granted to users depends on the specific option and the License signed.



iOSP™ 86 iAPX 86/30, iAPX 88/30, iAPX 186/30 and iAPX 188/30 SUPPORT PACKAGE

- Development and run-time support for iAPX 86/30, 88/30, 186/30, and 188/30 Operating System Processors
- Total iRMX™ 86 Operating System software compatibility
- Extendable with iRMX™ 86 Operating System calls
- Compatible with Intel® PL/M 86, PASCAL 86, FORTRAN 86, and ASM86 MACRO ASSEMBLER
- Supports (P)ROM or RAM based system
- Supports custom system initialization
- Interactive Configuration Utility

The Intel iOSP™ 86 Support Package for the iAPX 86/30, 88/30, 186/30, and 188/30 Operating System Processors contains a comprehensive set of easy-to-use tools needed to develop (P)ROM or RAM-based applications that use the 80130 Operating System Firmware component. This Support Package is compatible with all versions of the 80130 component. All of the system initialization and run-time facilities are provided in libraries that may be configured to specific requirements, and linked to application programs written in either ASM86 MACRO ASSEMBLER or a high level programming language such as PASCAL 86, FORTRAN 86, and PL/M 86. The iOSP 86 Package provides users with the basic initialization and interface routines needed to build application software based on the fundamental operating system functions of the iAPX 86/30, 88/30, 186/30, and 188/30 Operating System Processors. The iOSP 86 Package also enables users to add higher level I/O functions from the fully compatible iRMX™ 86 Operating System, or to form custom, real-time systems.



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.

FUNCTIONAL DESCRIPTION

The iAPX 86/30, 88/30, 186/30, and 188/30 Operating System Processors (OSPs) provide an easy-to-use foundation on which many real-time applications may be built. They provide the functions and system support needed to implement both simple and complex applications that require multiple tasks to run concurrently (see Figure 1). These services are made possible by the addition of the five new data types integrated into the 80130 Operating System Firmware (OSF) component. The 80130 OSF extends the basic data types of the CPU (integer, byte, character, etc.) by adding new system data types (JOB, TASK, MAIL-BOX, SEGMENT, and REGION), and extensive timer, interrupt, memory, and error management designed to give real-time response to multitasking and multiprogramming applications. As shown in the second half of the figure, other operating system functions such as mass storage I/O services and an easy-to-use Human Interface can be added easily, by using modules from the iRMX 86 Operating System. The iOSP 86 Support Package provides both an interface between application software and the Operating System Processors, and development tools designed to make the implementation and initialization of real-time, multitasking systems much easier.

The iOSP 86 Support package provides system developers with the configuration options necessary to tailor the iAPX 86/30, 88/30, 186/30, and 188/30 Operating System Processors to custom applications. Central to the entire configuration process is the Interactive Configuration Utility (ICU86). This utility is an easy-to-use tool which allows you to make configuration decisions by responding to screen-oriented displays. Using the ICU, users can build the necessary support code. The interface libraries form a sim-

ple interface between application software and the operating system primitives of the 80130 OSF component.

Memory and I/O Addressing

The 80130 OSF requires that a 16K byte block of memory address space be reserved for accessing internal functions. The ICU is used to specify the base address of the 80130 and the beginning of the initialization support code.

All Interrupt and Timer management of the OSF is controlled via a reserved 16 byte I/O address block that may be selected by the user. In addition, from 1 to 7 slave 8259A interrupt controllers can be specified in order to provide the system with up to 57 priority interrupt sources. The 80130 baud rate generator may also be configured to support an optional terminal interface.

Extending the 80130 OSF

The 80130 OSF allows users to add their own operating system extensions. These extensions may take advantage of the detailed and efficient intertask communication and synchronization primitives already provided by the 80130, and/or may utilize custom functions tailored to specific applications. The Support Package also enables users to extend the OSF with the extensive services of Intel's iRMX 86 Operating System, thereby allowing applications to grow without having to change or alter application software already written, or having to write other operating system software.

Use of the 80130 OSF with the iRMX 86 Operating System reduces the amount of memory needed for the iRMX 86 Nucleus layer by 14K bytes, and enables applications to take advantage of the increased

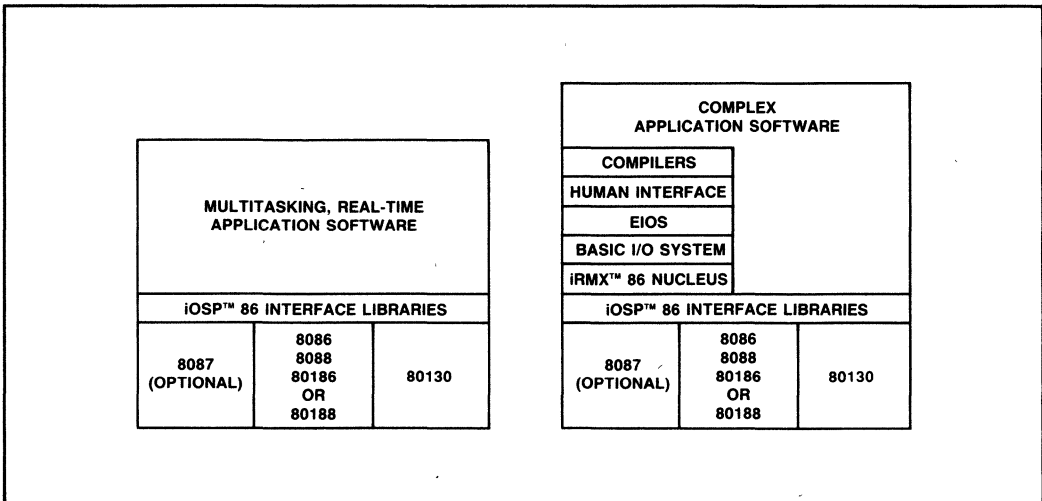


Figure 1. Structure of Typical Systems

performance and reduced size requirements inherent in the iAPX 86/30, 88/30, 186/30, and 188/30 Operating System Processors. Since each of the services provided by the 80130 component is totally compatible with iRMX 86, applications have an automatic upward path to support complete file systems and multiple processor environments.

Application Interfaces

Two interface libraries are included in the iOSP 86 Support Package. The first allows programmers to write application software modules in the Compact Model of computation supported by Intel's compilers. The second provides an interface to program segments written in either the Medium or Large Models. The iOSP 86 Support Package does not support program segments written in the Small Model.

The interface libraries provide the means of accessing all of the primitives supported by the Operating System Processors. With this interface, and all the memory management primitives of the OSPs, applications have full access to 1M byte of memory, and all of the addressing modes of the CPU.

These libraries are fully compatible with object modules produced by the ASM86 MACRO ASSEMBLER, and the PASCAL 86, FORTRAN 86, and PL/M 86 Compilers.

Application Initialization

The iOSP 86 Support Package provides, via the ICU, for the configuration of the system ROOT JOB, and all user application JOBS that require initialization when the system is started. The user also specifies the configuration of the interrupt system (including the optional iAPX 186/188 interrupt controller in either master or slave modes and any slave 8259A interrupt controllers) and the clock rate used for system timing. These choices are automatically programmed into the various devices when the system is initialized.

Parameter Validation

Parameter validation is a configuration option of an OSP-based system. The OSP can check the parameters of the primitive that you invoke either on a systemwide basis or on a per job basis.

Operating System Calls

The 80130 OSF performs a total of 38 operating system primitives all of which are completely compatible with the equivalent iRMX 86 Operating System calls. The iOSP 86 Support Package provides user-level interfaces to these primitives to enable applications to create, delete, control, and exchange the new data types provided by the 80130 OSF. In general, these interfaces allow application software to manage all of the resources of an iAPX 86/30, 88/30, 186/30, or 188/30 OSP (and an optional 8087 Numeric Processor Extension) system via any of the 38 system calls shown in Figure 2.

Required Development Hardware

Use of the iOSP 86 Support Package requires a Series III Intel Development System with double density flexible diskette drives or any iRMX 86 system supporting a standard 5.25 inch or 8 inch flexible diskette drive and the iRMX 860 Assembler and Utilities Package. Use of the 80130 requires only a minimal system including either the iAPX 86/30, 88/30, 186/30 or 188/30 Operating System Processor, and enough system memory to contain the application programs and initialization and interface software provided in the iOSP 86 Package.

Board Level Product Support

Intel microcomputer boards which use the 80130 OSF include the iSBC 186/03 and the iSBC 186/51 Single Board Computers. An iOSP 86 application may be written specifically to run on these boards.

JOB GROUP	SEGMENT GROUP	INTERRUPT MANAGEMENT GROUP
CREATE JOB	CREATE SEGMENT	SET OS EXTENSION
END INIT TASK	DELETE SEGMENT	SET INTERRUPT
		ENTER INTERRUPT
TASK GROUP	REGION GROUP	EXIT INTERRUPT
CREATE TASK	CREATE REGION	WAIT INTERRUPT
DELETE TASK	DELETE REGION	SIGNAL INTERRUPT
SUSPEND TASK	SEND CONTROL	RESET INTERRUPT
RESUME TASK	RECEIVE CONTROL	ENABLE
SLEEP	ACCEPT CONTROL	DISABLE
GET TASK TOKENS		GET LEVEL
SET PRIORITY	OBJECT MANAGEMENT GROUP	
MAILBOX GROUP	CATALOG OBJECT	ERROR CONTROL GROUP
CREATE MAILBOX	LOOKUP OBJECT	SET EXCEPTION
DELETE MAILBOX	DISABLE DELETION	SIGNAL EXCEPTION
SEND MESSAGE	ENABLE DELETION	GET EXCEPTION
RECEIVE MESSAGE	GET TYPE	

Figure 2. Operating System Primitives

Part Number	Description		
		OSP 86 E	iOSP 86 Support Package contained on an iRMX 86 format, single-sided, double density 8 inch diskette.
OSP 86 B	iOSP 86 Support Package contained on an ISIS-II compatible, single-sided, double density 8 inch diskette.		
		OSP 86 J	iOSP 86 Support Package contained on an iRMX 86 format double-sided double density, 5.25 inch, 48 tracks-per-inch diskette.

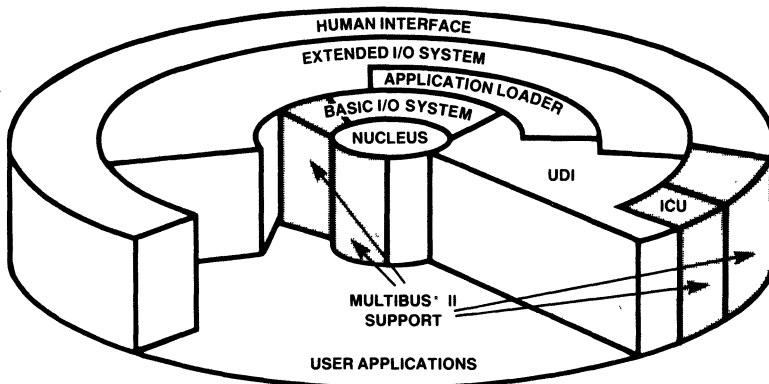
ORDERING INFORMATION

Each of the ordering options listed below include all the necessary initialization and interface procedures needed to use the iAPX 86/30, 88/30, 186/30, and 188/30 Operating System processors. Purchase of the iOSP 86 Package requires verification of an Intel Master Software License. Each package also includes an iOSP 86 User's Manual (Document Number 146798-001), and a 90 day update service.

iRMX™ 86-MULTIBUS® II SUPPORT PACKAGE

- MULTIBUS® II support for iSBC® 286/100 applications in Real Address Mode, including support for the SCSI peripheral interface and up to 1 megabyte addressability
- Functions in conjunction with the iRMX™ 86 Release 6 Operating System
- Interprocessor Signal Support
- Automatic software configuration of memory boards
- Support for battery backed-up, global time-of-day clock
- Extendable to allow addition of custom device drivers

The iRMX™ 86-MULTIBUS® II Support Package, functioning with the iRMX 86 Release 6 Operating System software, provides the ability to execute all configurable layers of the iRMX 86 software in the MULTIBUS II environment (iRMX 86-MULTIBUS II Operating System). Applications in Real Address Mode are supported for the iSBC® 286/100 board, including support for the SCSI peripheral interface and all iSBX™ boards supported by iRMX 86 Release 6, as well as support for iAPX 286 component applications.



NEW IN iRMX™ 86
MULTIBUS II
OPERATING SYSTEM

iRMX™ VLSI Operating System

The following are trademarks of Intel Corporation and may be used only to describe Intel products: Intel, ICE, iMMX, iRMX, iSBC, iSBX, iSXM, MULTIBUS, MULTICHANNEL and MULTIMODULE. 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. Information contained herein supercedes previously published specifications on these devices from Intel.

FUNCTIONAL DESCRIPTION

Overview

The iRMX 86 MULTIBUS II package contains system modules that replace portions of the iRMX 86 Release 6 Operating System, allowing the iRMX 86 Operating System to execute in a MULTIBUS II environment. All the functions available in the iRMX 86 Operating System are available in the iRMX 86-MULTIBUS II Operating System. For a complete description of these functions, their value, and performance, please refer to the Release 6 iRMX 86 Operating System Data Sheet (order number 210885-002).

This functional description section describes the new features provided by the iRMX 86 MULTIBUS II package. These new features add the new capabilities required for OEMs to execute the iRMX 86 Operating System in a MULTIBUS II environment for iSBC 286/100 or iAPX 286 applications in Real Address Mode.

Interprocessor Signal Support

In a MULTIBUS II system, interprocessor communication and synchronization is done via messages over the bus. This communication includes data-less messages to signal that an event has occurred. The iRMX 86 MULTIBUS II package supports signal messages using the Message Interrupt Controller (MIC) Component. The major advantage of signal message support is the ability for a host cpu board to send or receive signal messages from up to 254 distinct sources, with the priorities of each message being based on the sending or receiving task's priority. Sig-

nal messages are not tied to hardware interrupt levels and priorities as external interrupts were in the MULTIBUS I environment.

Automatic Software Configuration of Memory Boards

The iRMX 86-MULTIBUS II Operating System has the option of automatically configuring memory boards. The addresses for each board are defined sequentially in relation to the physical placement of each board in the card cage. This feature allows for the swapping, adding, and deleting of memory boards in the system on a dynamic basis.

Accurate Time-of-Day Clock Support

Resident in every MULTIBUS II system is a Central Services Module (iSBC CSM/001 board). The CSM board contains a battery backed-up, global time-of-day clock. The iRMX 86-MULTIBUS II Operating System uses this clock to automatically initialize the time-of-day clock maintained by the operating system.

Custom Device Driver Support

Like the iRMX 86 Operating System, the iRMX 86-MULTIBUS II Operating System is extendable to support user value-added custom device drivers. This feature allows the system to be more closely tailored to meet a specific application requirement and expands the list of supported hardware products. The user need not purchase source code to write a custom driver and can configure the driver into the system at configuration time. Custom drivers can use the Message Interrupt Controller (MIC) to pass signal messages.

SPECIFICATIONS

Below is the list of supported products for the iRMX 86 MULTIBUS II Support Package.

Supported Software Products

iRMX 86 Release 6 Operating System

Supported Hardware Products

Components:

iAPX 286 Microprocessor (Real Address Mode only)
80287 Numeric Data Processor Extension
8253 and 8254 Programmable Interval Timers
8259A Programmable Interrupt Controller (PIC)
8255 Programmable Parallel Interface (PPI)
82530 Serial Communications Controller (SCC)
82258 Advanced DMA Controller (ADMA)
Bus Arbiter Controller (BAC)
Message Interrupt Controller (MIC)

iSBC® MULTIBUS® II Board Products:

iSBC 286/100 Single Board Computer (Real Address Mode only)
iSBC CSM/001 Central Services Module
iSBC MEM/312, 310, 320, 340 cache-based memory
iSBX 218(A) Flexible Diskette Multi-Module Controller
iSBX 251 Bubble Memory Multi-Module
iSBX 270 CRT Light Pen and Keyboard Interface
iSBX 350 Parallel Port (Centronics-type Printer Interface)
iSBX 351 Serial Communications Port

AVAILABLE LITERATURE

iRMX 86-MULTIBUS II Support Package Reference Manual (order number 147127)

There are four manual kits supplied with the iRMX 86 Release 6 Operating System and are available under the order numbers shown in the iRMX 86 Operating System Data Sheet (order number 210885-002)

ORDERING INFORMATION

The iRMX 86 MULTIBUS II Package is available under a number of different licensing options. Obtaining a license for the iRMX 86 Release 6 Operating System is a pre-requisite to licensing the iRMX 86 MULTIBUS II Package. Reconfigurable object libraries are provided on: 1) Double-density single-sided ISIS-formatted 8" diskettes; 2) Double-density, single-sided iRMX 86-formatted 8" diskettes; 3) Double-density, double-sided, iRMX 86-formatted 5.25" diskettes. ISIS-format diskettes may be used on Series III Development Systems. The iRMX 86-format may be used on Series IV Development Systems (5.25" diskettes) or any iRMX 86-based system supporting the appropriate disk drivers, compilers and development environment.

The OEM license options listed here allow users to incorporate the iRMX 86 MULTIBUS II package into their applications. Each use requires payment of an Incorporation Fee.

ORDER CODE	DESCRIPTION
iRMX 86 II BRO:	Double-density, single-sided 8" ISIS-format OEM license.
iRMX 86 II ERO:	Double-density, single-sided 8" iRMX 86-format OEM license.

iRMX 86 II JRO:	Double-density, double-sided 5.25" iRMX 86-format OEM license.
iRMX 86 II KIT BRO:	Includes iRMX 86 Release 6. Double-density, single-sided 8" ISIS format OEM license.
iRMX 86 II KIT ERO:	Includes iRMX 86 Release 6. Double-density, single-sided 8" iRMX 86-format OEM license.
iRMX 86 II KIT JRO:	Includes iRMX 86 Release 6. Double-density, double-sided 5.25" iRMX 86-format OEM license.

Other licensing options include prepayment of all future incorporation fees and single use rights for a single machine.

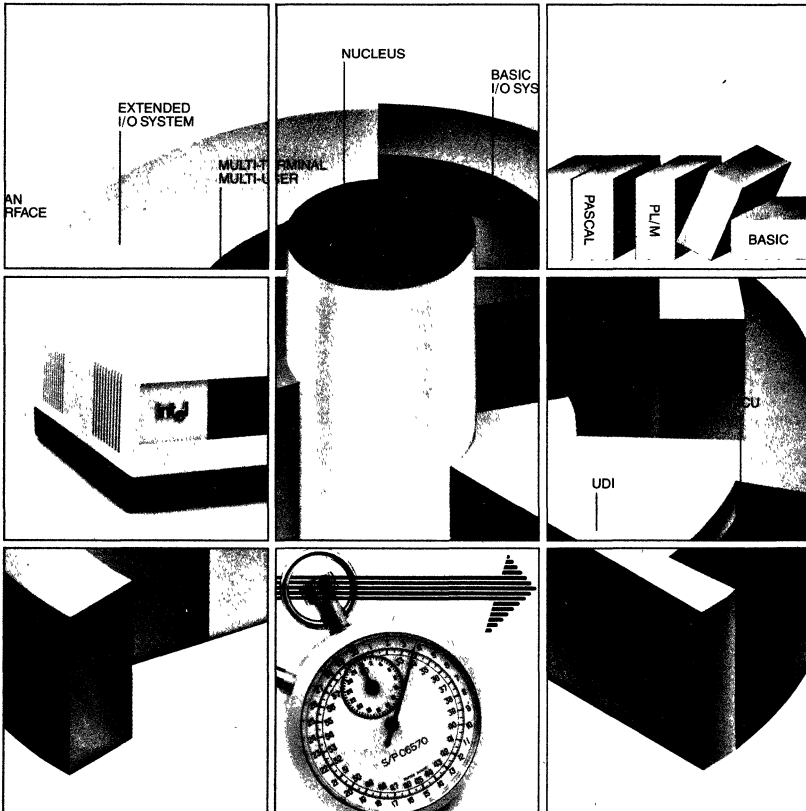
Each option includes 90 days of support service that provides Software Problem Report Service and copies of System Updates that occur during this period.

As with all Intel software, purchase of any of these options requires the execution of a standard Intel Master Software License. The specific rights granted to users depend on the specific option and the license signed.



iRMX™ OPERATING SYSTEM

- High-performance, real-time, multi-tasking operating system for Intel's 86/300 and 286/300 microcomputer systems
- Highly configurable, modular structure for easy system expansion
- Wealth of design facilities and industry-standard languages to support fast, easy development
- Application software portable to next generation of Intel VLSI
- Supported by Intel's post-sales software support organization



The Total Solution for the Real-Time Application OEM

Intel's iRMX™ 86 Operating System is a real-time, multi-tasking, multiuser, multiprogramming operating system designed to support high performance, time-critical applications such as factory automation, industrial control and communications networks. The iRMX operating system serves as an optimized event-driven executive for managing and extending the resources of Intel's 86/300 and 286/300 systems in real-time applications where high speed and low interrupt latency are required. Added performance for demanding numeric-intensive tasks comes from support of Intel's floating point math coprocessors.

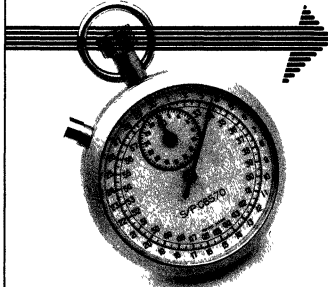
Comprised of modular layers, Intel's iRMX operating system is highly configurable, allowing the OEM to easily customize the system to meet the needs of target applications. In addition, the iRMX operating system provides OEMs with complete development capabilities. It has systems debuggers, crash analyzers, screen editors, utilities, and an Interactive Configuration Utility (ICU)—everything the development engineer needs to design and configure efficiently.

To further reduce development time, a complete set of industry-standard languages enables OEMs to take advantage of existing application software. This shaves months off development time and is a key advantage to the competitive OEM.

Speed, the Name of the Real-Time Game

In a real-time system the computer must respond to interrupts instantly; time is always at a premium. Intel's iRMX Operating System delivers superior real-time performance, thanks to ultra-fast context switching, task synchronization and memory-based message passing.

The iRMX 86 Operating System manages the resources of the 286/300 systems in real-address mode. iRMX 86 makes possible the utilization of the high-



performance capabilities of Intel's iAPX 286 microprocessor for those demanding high-speed applications.

Further accelerating processing power in number-crunching and floating point math applications is iRMX operating system's support of Intel's math coprocessors.

Our 8087 numeric data processor in our iRMX 86-based systems can perform floating point operations four times faster than competitive minicomputers with hardware math processors. For even greater performance, OEMs can select the iAPX 286 and the 80287 coprocessor working in tandem in the iRMX 86 system.

The superior price/performance ratio that results from combining Intel's iRMX operating systems and the System 300 family makes the choice clear: a more competitive Intel micro-based system over a more expensive minicomputer-based system.

Add More Processors for More Power, More Speed

Need still more micro-muscle in your application? In an iRMX-based system, additional intelligent boards can be added to enhance system throughput.

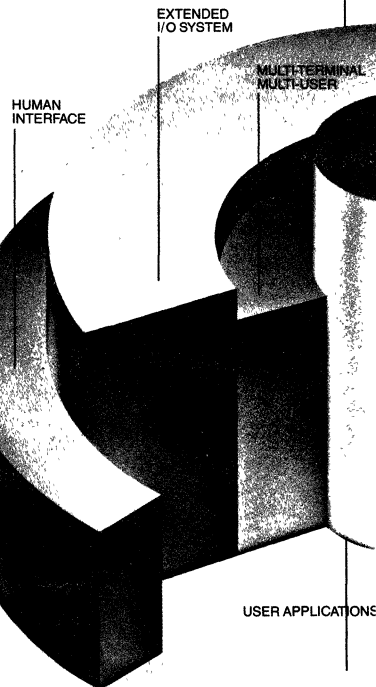
With the iMMX™ 800 (MULTIBUS® Message Exchange) software package, the iRMX 86 Operating System supports a loosely-coupled multiprocessing environment. Tasks running on one board may communicate with tasks running on

other boards, even if they operate under different Intel operating systems or microprocessors.

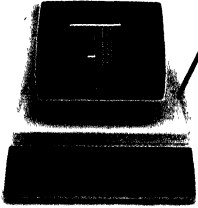
Multiprocessing is possible due to the hardware capabilities of Intel's System 300 MULTIBUS System Bus and the software support provided by iMMX™ 800. Overall system performance and flexibility can be greatly enhanced by off-loading the main CPU with such intelligent I/O boards as Intel's quad serial communication controller, digital controller or Ethernet communications controller.

Modular Software for Versatile, Easy Configuration

The iRMX operating systems shipped with Intel's 86/300 and 286/300 hardware systems are preconfigured at the factory to support a standard board set; however, the OEM can additionally configure or



extend the operating system to meet specific needs.



Intel's iRMX operating systems are configurable by system layer and by system call within each layer. Such flexibility gives designers the ability to choose software features that best suit their application's size and functional requirements. The iRMX Operating System also includes I/O drivers for many of Intel's MULTIBUS boards and industry-

BASIC
I/O SYSTEM

APPLICATION
LOADER

UDI

ICU

standard peripherals. You simply select the ones you need.

The Interactive Configuration Utility (ICU) is a built-in facility for assisting the OEM in the configuration process. The ICU prompts the user for system parameters and requirements, then builds a command file to compile, assemble, link, and locate necessary files.

The net results for the OEM: fast, easy system configuration with quick time-to-market benefits.

For customizing and extending your iRMX system, Intel has provided all the "hooks" necessary to make the job easy. The iRMX 86 Operating System contains extendability features that enable the OEM to add custom operating system calls, custom features, and custom functionality to his application—at any time in the application's life. The ability to add functions late in a product's life is key to an OEM's competitive edge in a fast-changing market.

iRMX™ Operating System Has All the Fundamentals, Too!

In addition to multiprocessing, Intel's iRMX operating systems have all the basics you would expect to find in a minicomputer operating system... capabilities such as multitasking, multiprogramming, and multiterminal support.

Multitasking requires a method of managing the different processes of an application and for allowing these processes to communicate with each other. The iRMX Nucleus provides these facilities plus task scheduling. The Basic I/O System provides users with the system calls for direct management of I/O devices needed for real-time applications. The Extended I/O System adds a number of I/O management capabilities to simplify access to files, such as automatic buffering and synchronization of I/O requests.

The Human Interface functions give users and applications simple access to the file and system management capabilities. Using the multiterminal support provided by the Basic I/O system, the Human Interface can support several simultaneous users. For example, multi-terminal support allows one person to use the iRMX Editor, while another compiles a FORTRAN or Pascal program, while several others load and access applications.

On-Target Development: One System Does It All

The beauty of Intel systems lies in their flexibility. Engineers developing an iRMX-based target system can use the same iRMX-based system in the development process; the development and target systems are one in the same. The bottom-line benefit is low entry-level costs for the OEM.

On-target development contributes immeasurably to a shorter development curve and decreased time-to-market, since it isn't necessary to purchase and learn separate development systems. With Intel's iRMX-based system, one system does it all.



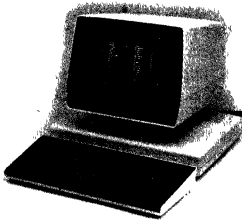
Tap into a Wide Range of Languages and Utilities

An Intel iRMX-based system supports many industry-standard and widely available languages: FORTRAN 77, Pascal (ISO Draft Standard) and PL/M compilers; Intel Assemblers, and popular independent vendor products, such as Microsoft's BASIC and Mark Williams' C compiler.

iRMX operating systems also have a menu-driven, screen-oriented text editor and a variety of utilities for manipulating

object code to facilitate the development process.

Multiple-language support is made possible by a set of systems calls known



as the Universal Development Interface (UDI) which enables the iRMX systems to interface with many compilers and language translators. UDI ensures that users will be able to transport applications to future releases of iRMX operating systems as well as use language and utilities of other software vendors that support UDI. (For more information on Intel iRMX languages, see the iRMX Language Fact Sheet)

As an option, a commercial extension package iCEX is available. It provides such useful utilities as: a Shared I/O System (SIOS) that allows multiple tasks to access mass storage data through shared buffers in main storage; a Re-entrant Program Manager (RPM) that eliminates the need to have multiple copies of the same program in memory to support concurrent applications; a File Printer; Multi-user LOG ON facilities; and many more.

Intel's Open Systems Approach Means Freedom to Grow

At Intel, we believe that systems need to expand in order to meet the needs of a changing market; and that is how we design our products.

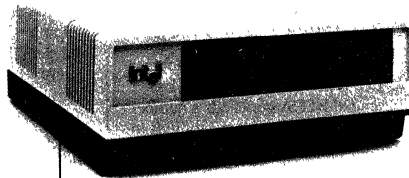
Standards are the key to systems that are open to future expansion, future technology and future markets.

Intel's iRMX operating systems are built from the inside-out with industry standards: UDI (Universal Development Interface), RTI (Runtime Interface), MULTIBUS System Bus (IEEE 796), iMMX 800 Package (MULTIBUS multi processing), Ethernet (IEEE 802.3), extended math format (IEEE P754), and industry-standard peripheral device interfaces.

An OEM who builds his product around one of Intel's RMX-board systems is assured of multi-vendor hardware/software alternatives and a future upgrade path. In today's highly competitive markets, that is the only kind of system to build.

Today, you'll have the ability to tap into readily available application software packages, languages, and utilities, MULTIBUS boards, and peripherals. Tomorrow, you will be able to tap into the latest, high-performance VLSI without sacrificing today's software investment. Applications written on iRMX 86 will run on Intel's iAPX 86, iAPX 88, iAPX 186, iAPX 188 and iAPX 286-based systems.

Not to be forgotten are the advantages of starting from the systems level to begin with. Intel has invested hundreds of man-years in software and hardware development for its systems products. For the OEM trying to meet a market window, time-to-market is much faster when starting with a system instead of boards or components. It makes good business sense to let Intel provide the "micro-engine"; so you can concentrate on your area of expertise and get to market sooner!



Worldwide Service and Support

The iRMX 86 Operating System is a mature proven product with thousands of installations at the component, board and systems levels. Post-sales software support is available to Intel iRMX 86 Operating System OEMs in the form of software updates and routine systems software maintenance. Software support is extendable in one-year increments after the initial 90-day warranty. Hotline service is available separately to customers needing quick regional software support. All software is completely documented, and users receive monthly technical reports, newsletters and access to the iRMX users group and software libraries.

iRMX users can also take advantage of Intel's worldwide staff of trained hardware and software engineers for application design assistance. We offer complete training for operating system software and associated system hardware, bringing OEM's up to speed and helping get their products to market quickly.

Intel, the Technology Leader ... With the Total Solution

Intel started the microprocessor revolution with the 4004 and has been the market leader with every generation of advanced microprocessor VLSI since. We not only invented the microprocessor but MULTIBUS single board computers, as well.

Intel's technology leadership has, by necessity, extended from microprocessors into operating system software. iRMX is recognized as the industry standard real-time VLSI operating system.

OEMs can enhance their product's marketability by leveraging their value-added on top of the solid foundation of an iRMX-based Intel 300 microcomputer system. Intel's solution offers the most price/performance with the least risk to progressive OEMs... because we know the real-time game from the inside out.

RMX™ OPERATING SYSTEM



Specifications

Supported Software Products

iRMX 860	iRMX 86 Development Utilities Package including the iAPX 86 and 88 Linker, Locator, Macro Assembler, Librarian, and the iRMX 86 Editor
iRMX 861	Pascal 86/88 Compiler
iRMX 862	FORTRAN 86/88 Compiler
iRMX 863	PL/M 86/88 Compiler
iRMX 864	TX-Screen-Oriented Editor
iRMX 865	BASIC Interpreter
iRMX 866	C Compiler
iMMX 800	MULTIBUS® Message Exchange software package for iRMX 80, 86, 88, and 286 application systems

Supported Hardware Products

iSBC* MULTIBUS* Products

iSBC 86/12A, 86/05, 86/14, 86/30, 86/35, 88/25, 88/40, and 286/10	Single Board Computers
iSBC 186/03	Single Board Computer
iSBC 186/51	Ethernet Controller
iSBC 188/48	Communications Controller
iSBC 286/10	Single Board Computer (Real Address Mode only)
iSBC 204	Flexible Disk Controller
iSBC 206	Hard Disk Controller
iSBC 208	Flexible Disk Controller
iSBC 215	Winchester Disk Controller
iSBC 220	SMD Disk Controller
iSBX 251	Bubble Memory System

iSBC 254	Bubble Memory System
iSBC 534	4-Channel Terminal Interface
iSBC 544	Intelligent 4-Channel Terminal Interface and Controller
iSBX 218	Flexible Disk Controller
iSBX 350	Parallel Port (Centronix-type Printer Interface)
iSBX 351	Serial Communications Port
iSBX 270	CRT, Light Pen and Keyboard Interface
System 86/300 Family	
System 286/300 Family	

Available Literature

The iRMX 86 Documentation Set is comprised of the following four volumes of reference manuals. Order numbers are associated with these four volumes only.

iRMX 86 Introduction and Operator's Reference Manual for Release 6

Order Number: 146545-001

Introduction to the iRMX 86 Operating System

iRMX 86 Operator's Manual

iRMX 86 Disk Verification Utility Reference Manual

iRMX 86 Programmers Reference Manual for Release 6, Part 1

Order Number: 146546-001

iRMX 86 Nucleus Reference Manual

iRMX 86 Basic I/O System Reference Manual

iRMX 86 Extended I/O System Reference Manual

iRMX 86 Programmer's Reference Manual for Release 6, Part II

Order Number: 146547-001

iRMX 86 Application Loader Reference Manual

iRMX 86 Human Interface Reference Manual

iRMX 86 Universal Development Interface Reference Manual

Guide to Writing Device Drivers for iRMX 86 and iRMX 88 I/O Systems

iRMX 86 Programming Techniques

iRMX 86 Terminal Handler Reference Manual

iRMX 86 Debugger Reference Manual

iRMX 86 System Debugger Reference Manual

iRMX 86 Crash Analyzer Reference Manual

iRMX 86 Bootstrap Loader Reference Manual

iRMX 86 Installation and Configuration Guide for Release 6

Order Number: 146548-001

iRMX 86 Installation Guide

iRMX 86 Configuration Guide

Master Index for Release 6 of the iRMX 86 Operating System

iRMX™ 86 Configuration Size Chart

System Layer	Min. ROMable Size	Max. Size	Data Size
Bootstrap Loader	1K	1.5K	6K*
Nucleus	10.5K	24K	2K
BIOS	26K	78K	1K
Application Loader	4K	10K	2K
EIOS	10.5K	12.5K	1K
Human Interface	22K	22K	15K
UDI	11K	11K	0
Terminal Handler	3K	3K	0.3K
Debugger	28.5K	28.5K	1K
Human Interface Commands			116K
Interactive Configuration Utility			308K
System 86/300 Memory:	348KB		
Maximum Addressable Memory:	1MB		
Minimum Memory Required with ICU Loaded:	448KB		

*Usable by System after Bootloading



Ordering Information

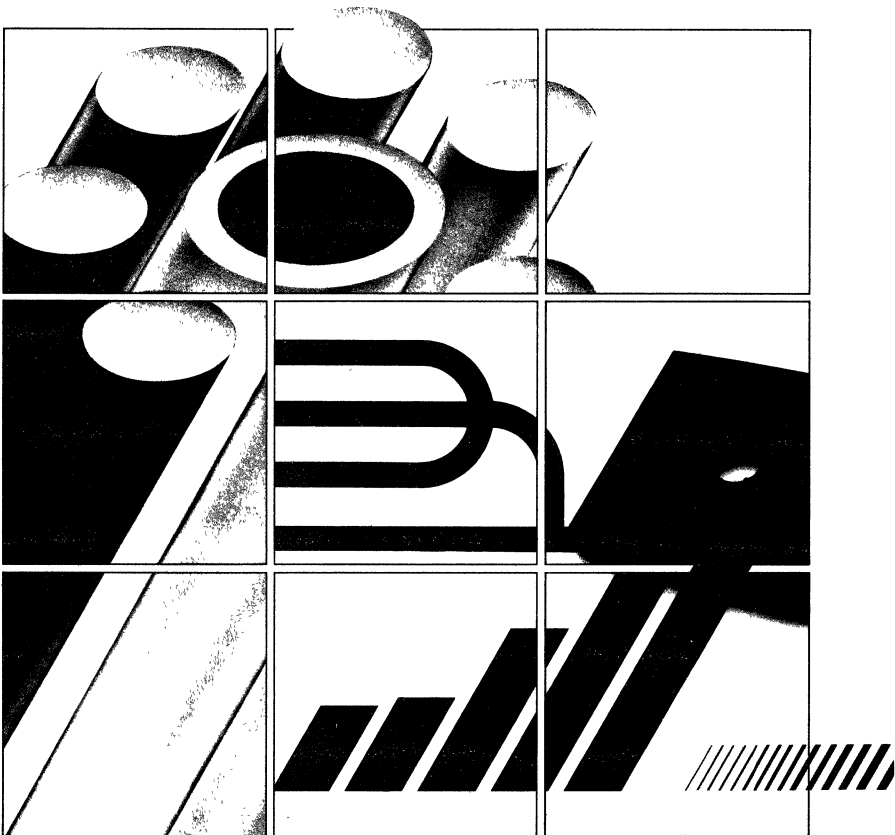
Each iRMX operating system includes two startup systems supporting Intel's System 300 standard hardware and Intel processor boards. Intel System customers also receive the iRMX 860 (Assembler, Linker, Locator, Libraries, Editor, Utilities) and iRMX 863 (PL/M Language) products and are entitled to one prepaid incorporation fee. Also included: Software Problem Reporting Service (SPR), and a 90 day System Software Subscription (new s/w release updates). Also includes System Software documentation.

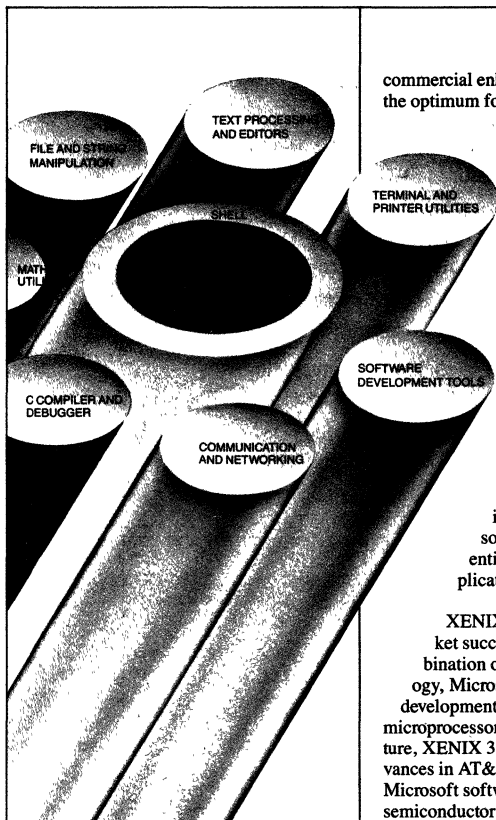
Refer to Intel's OEM price list, OEM Microcomputer System section, for ordering information.



XENIX* 3.0 OPERATING SYSTEM

- **XENIX 3.0 Industry Standard Multiuser Operating System**
- **Fully licensed version of the UNIX† operating system optimized for the Intel 80286 processor**
- **Leading edge microprocessor implementation of UNIX, fastest floating point performance on a microprocessor**
- **Important commercial OEM enhancements**
- **Supports multiple levels of integration: components, boards and systems**
- **Supported by Intel's worldwide post-sales service and support organizations**





XENIX 3.0—Industry Standard Multiuser Operating System

Intel's XENIX 3.0 Operating System for the 80286 is a fully-licensed derivation of Bell Laboratories' UNIX System III. XENIX 3.0 includes not only all the functionality of UNIX System III, but also powerful enhancements from Microsoft and Intel that meet the needs of the commercial OEM.

The Best Foundation for Building OEM Solutions

XENIX 3.0 provides the OEM with a complete software base on which to build value-added functionality. It includes the operating system, the C language, text processors, development tools, system accounting and security features, and

commercial enhancements that make it the optimum foundation for OEM application software solutions.

XENIX: Portable, Flexible, Powerful

XENIX has become the industry-standard microcomputer operating system for interactive, multi-user applications. It has gained wide popularity in applications such as distributed data processing, business data processing, word processing, software development, scientific and engineering applications, and graphics.

XENIX has achieved this market success through a solid combination of UNIX system technology, Microsoft value-added product development, and Intel's experience in microprocessor technology. In the future, XENIX 3.0 will benefit from advances in AT&T UNIX technology, Microsoft software technology and Intel semiconductor and system technology.

XENIX is also an extremely powerful operating system, providing the applications programmer with a wealth of development tools and utilities for bringing OEM products to market quickly.

XENIX 3.0: Leading Edge UNIX Performance on a Micro

As the first UNIX operating system derivative optimized for the iAPX 286, XENIX 3.0 alone can take full advantage of the 80286's unique features:

On-chip memory management and protection provides two key advantages for XENIX 3.0 over other microprocessor UNIX implementations. First, on-chip memory management and protection drastically reduces the overhead in accessing system memory as compared to the usual separate memory management unit. With this functionality right on the chip, the operating system works more smoothly and efficiently.

Second, on-chip memory management and protection circuitry ensures that each version of XENIX 3.0 will be very compatible with every other version. This heretofore impossible level of compatibility aids OEM, software developers, and end users due to the wider availability of compatible software.

Advanced microprocessor architecture provides pipeline processing, wherein a continual flow of instructions is kept in the CPU queue, results in throughput several times faster than the fastest competing microprocessor.

1.75x

INTEL 286/310

1.5x

CONVERGENT TECHNOLOGY MINIFRAME
ALTOS 986

1.25x

SUN @MODEL 100

1.0x

NCR TOWER

Fast floating point processing

is due to XENIX 3.0 support of the Intel iAPX 287 math coprocessor. Floating point processing delivers throughput that is an order of magnitude faster than non-floating point processing. Extra high processing speeds are needed in applications such as data base processing, commercial data reduction and graphics.

*XENIX is a trademark of Microsoft Corporation

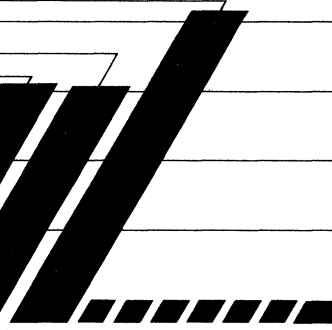
**Faster, More Reliable Still
When Teamed with Other
Intel Systems Components**

The throughput enhancements in the XENIX 3.0 software are pushed to even greater speeds by special hardware architecture in Intel's systems and board products.

MULTIBUS® System Architecture is the industry-standard system bus. It accommodates any of the special-purpose Intel iSBC® boards, as well as a multitude of third party Multibus boards and standard peripherals, for easy system expansion.

iLBX™ (Local Bus Exchange) is an Intel hardware innovation that increases the amount of local memory accessible by the operating system to significantly improve system throughput.

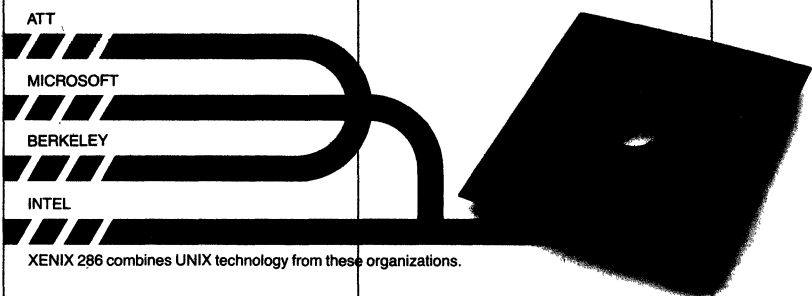
Error Correction Circuitry (ECC) automatically detects and corrects soft errors in RAM. This on-board, self-correction facility reduces errors and further underscores data integrity.



See Intel benchmark series

**A Faster Operating System
Means Market Leadership**

The combination of the industry's most widely accepted operating system for multi-user, interactive applications with the industry's fastest and most advanced microprocessor gives the OEM a far superior price/performance ratio than is



XENIX 286 combines UNIX technology from these organizations.

available through other options. The result for the OEM: market leadership due to the ability to more attractively price products based on superior performance.

**XENIX 3.0: The Best of
Everything**

The XENIX 3.0 Operating System contains the best of many vendors' UNIX/XENIX development efforts during the last ten years (see Fig. above). We have taken the best features of many UNIX versions—ease of use, flexibility, performance, security, reliability—and added our own enhancements (not the least of which is compatibility with the iAPX 286) to make XENIX 3.0 the optimum software foundation for the commercial OEM.

**Superior Date Reliability and
Integrity**

XENIX 3.0 contains enhancements to provide extremely high data reliability and integrity, particularly important to the OEM who is adding value to a system product. The following enhancements in XENIX 3.0 contribute to uniformly reliable data at all stages of application development.

Automatic disk recovery is an improvement of the UNIX file system that allows automatic recovery of the file system in the event of unexpected system shutdown.

Record and file locks arbitrate multiple-access requests to the same record or file, allowing the programmer to extend locks to a single record, group of records or the entire file. This is important in multi-user applications to prevent two or more users accessing and updating the same information simultaneously.

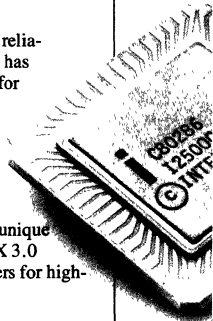
XENIX System Analysis Test (XSAT) is a complete hardware-software diagnostic package included with all Intel integrated system products. XSAT provides a total analysis of a XENIX-based system, ensuring reliability even after the OEM configures new drivers into the system.

**Tools for Easy System
Configuration**

In addition to increased data reliability measures, XENIX 3.0 has been functionally enhanced for easier system configuration. An interactive configuration utility allows the user to specify device drivers, disk buffers, memory size, etc., making it easy for the OEM to meet unique design requirements. XENIX 3.0 includes over 12 device drivers for high-speed controllers.

Friendlier Interface

The standard UNIX human interface has been enhanced in XENIX 3.0, with the



addition of vi, a full-screen editor, for easier and faster application development.

The XENIX C shell augments the capabilities of the standard UNIX shell with the ability to maintain histories of invoked processes and provide the alias feature, saving re-keying of often-used commands. XENIX 3.0 also provides the visual shell, a menu driven command interpreter which makes full use of the screen to display status and environmental information to the user. It has a built-in HELP facility and allows users to add new applications to the menu.

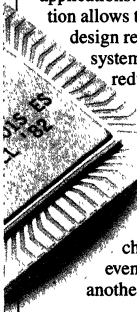
Intel's Open Systems Approach

Intel believes that system components — hardware or software — should be fully compatible with other family members at any level of integration and open to future VLSI advancements. XENIX 3.0 was designed to be part of the Open Systems concept.

Portability from Chip to Board to System

Intel's XENIX 3.0 Operating System is available for and fully compatible across Intel component, board and system designs, something that no other XENIX version offers.

Such portability gives OEMs the flexibility to choose the most appropriate and profitable level of integration for their applications. Component-level integration allows the OEM to meet unique design requirements; board and system-level integration afford reduced time to market.



There is no loss in software development investment as your needs change, since you can port XENIX-based applications from the chip to the system level or even from one Intel processor to another.



Open to Still Greater Configurability through Third-Party Software and Hardware

XENIX 3.0 users can tap into an extensive base of existing third-party languages and application packages for almost endless versatility in system configurability. There are hundreds of such packages available today with many more on the way. To assure the availability and quality of these packages on our systems, we have the Independent Software Vendor Program. Through this activity, software vendors are given Intel systems as well as technical assistance to aid them in porting their packages. The resulting product is thoroughly evaluated by Intel prior to certification for operation on our system products.

Superior Documentation

In line with the OEM orientation of the Intel hardware and software combination, the documentation for Intel's XENIX 286 product provides excellent support for system builders. In addition to the mature UNIX documentation from AT&T and the value-added feature documentation by Microsoft, Intel adds a wealth of publications aimed at helping the OEM to successfully launch XENIX 3.0 based products.

Worldwide Support and Service

XENIX 3.0 customers can take advantage of Intel's worldwide staff of trained hardware and software engineers in contracting for application design assistance. A liberal warranty, including software updates and a technical newsletter, follows the sale. Once the war-

ranty expires customers can choose from a variety of support contracts.

Intel offers complete training on the XENIX 3.0 Operating System as well as the iAPX 286 processor and associated hardware.

Intel, The Technological Leader...

Intel is committed to pushing the frontiers of VLSI design to their ultimate limits. In the process, we move our customers along the technology curve without interruptions in application development or expensive mid-stream architecture changes.

Intel started the micro revolution with the 4004 and has been the market leader with every generation of advanced processors since.

Systems and system software are a natural for us: who better knows the pieces and how to make them work together?

...In Total Solutions

The XENIX 3.0 Operating System fully exploits the iAPX 286, the fastest and most sophisticated microprocessor on the market. No other processor/operating system combination will give OEMs a faster and more economical path to getting systems and applications on the market.

Intel has always been first with the latest and most advanced VLSI and now with system software tailor-made for Intel VLSI. Because we're there first, our customers are first in their respective markets with state-of-the-art OEM and end-user products.

XENIX* 3.0

XENIX 3.0 includes support for the following Intel Systems, single board computers and processors.

- System 286/310
- System 286/380

- iSBC® 286/10 Processor Board
 - 16mb of addressing
 - On-chip memory protection
- CX Series RAM board
 - ECC (Error Correction Circuitry)
 - iLBX™ (Local Bus Extension)
- iSBC 215 Winchester Controller
- iSBX 218 Floppy Controller
- iSBC 534 Serial I/O Expansion Board
- iSBC 544 Intelligent Serial I/O Expansion Board
- iSBC 188/48 8-channel Serial I/O Expansion Board
- iSBC 552 Ethernet Controller Board
- iSBX 217 Tape Controller Board

- 80286 Central Processor
- 80287 Fast Floating Point Processor

Documentation

Documentation Includes:

- Overview of the XENIX 286 Operating System
- XENIX 286 Installation and Configuration Guide
- XENIX 286 System Administrator's Guide
- XENIX 286 Communications Guide
- XENIX 286 Visual Shell User's Guide
- XENIX 286 User's Guide
- XENIX 286 Reference Manual
- XENIX 286 C Library Guide
- XENIX 286 Programmer's Guide
- XENIX 286 Device Driver Guide
- XENIX 286 Text Formatting Guide

Text Books

- The UNIX Book—Banahan & Rutter
- The UNIX System—Bourne
- The UNIX Operating System—Kaare
- Understanding UNIX: A Conceptual Guide—Groff & Weinberg
- The UNIX Programming Environment—Kernighan & Pike
- Introducing the UNIX System—McGilton & Morgan
- A Practical Guide to the UNIX System—Sobell
- A User Guide to the UNIX System—Yates & Thomas
- A Business Guide to the UNIX System—Yates and Emerson



Ordering Information

XXN 286 HRO	XENIX Object Software (8" double side, double density) plus license rights
XXN 286 KRO	XENIX Object Software (5¼" double-sided, double density) plus license rights
XXN 286 RF	Software Incorporation Fee
SYS 310-17X	System Kit including System 310-17 and XENIX Software
SYS 310-17MX	System Kit including System 310-17, XENIX Software, 6 user support
SYX 286 RO	License rights extension for system customers
SYX 286 RF	System incorporation fee



Ordering Information

XNX 286 H	XENIX Object Software (8" double side, double density)
XNX 286K	XENIX Object Software (5¼" double-sided, double density)
XNX 286 RO	Software License Rights Extension
XNX 286 RF	Software Incorporation Fee
I73258	XENIX Documentation Package
CTW 14PP	XENIX Customer Training
SPRTECHREP	XENIX Support Subscription Services
HOTLINE	XENIX Hotline Phone Service
SP86 330 XINSTALL	XENIX Software Installation
CONSULT-FIELD	XENIX Onsite Field Consulting
CONSULT-LT	XENIX Onsite Field Consulting for extended time periods.

March 1982

**Using Operating System
Firmware Components
to Simplify Hardware
and Software Design**

John Wharton
Microprocessor Applications

INTRODUCTION

Intel recently introduced a new set of extensions to its microprocessor product line. The iAPX 86/30 and iAPX88/30 Operating System Processors (OSPs) augment the general-purpose instruction set of the well-known 8086/8088 architecture to include common, real-time, operating system capabilities. A single device, the 80130 Operating System Firmware component (OSF), now provides hardware support for functions previously relegated to software.

The 80130 introduces new concepts in the areas of both hardware and software. At first glance, traditional component-level hardware designers could feel somewhat intimidated by the esoteric concepts and unfamiliar buzzwords encountered in the software world. Even the experts in conventional operating system (OS) design may initially find it strange that what used to be "soft" software routines are now cast in silicon.

This application note is intended for readers at both levels. The first section reviews the development of processor extensions in general and operating system firmware in particular. Later sections should help you understand what a real-time operating system can do, how the 80130 provides these capabilities, and how to

design system hardware and software to take advantage of such features.

The note also documents a complete (albeit simple) system, including schematics and listings. The reader may wish to reconstruct this system to get started with OSFs. Finally, a step-by-step description of the so-called "configuration" process shows how physical system parameters are incorporated into the software as the software is "installed" in memory. Throughout the note are a number of "exercises"—questions relating to concepts just presented. Please take a few moments to think about these questions before reading on.

The reader need not have worked with operating systems previously, though such background would be helpful. The reader should also know something about microprocessor hardware—at a minimum, how the 8086 or 8088 devices operate. For simplicity, most of the software examples are written in PL/M-86, so the reader should be familiar with PL/M-80 or some other block-structured language. Finally, be forewarned that the configuration steps make use of several ISIS utility programs, including EDIT, SUBMIT, ASM86, LINK86, and LOC86. Readers who wish to brush up on any of the above should consult the appropriate Intel reference manuals.

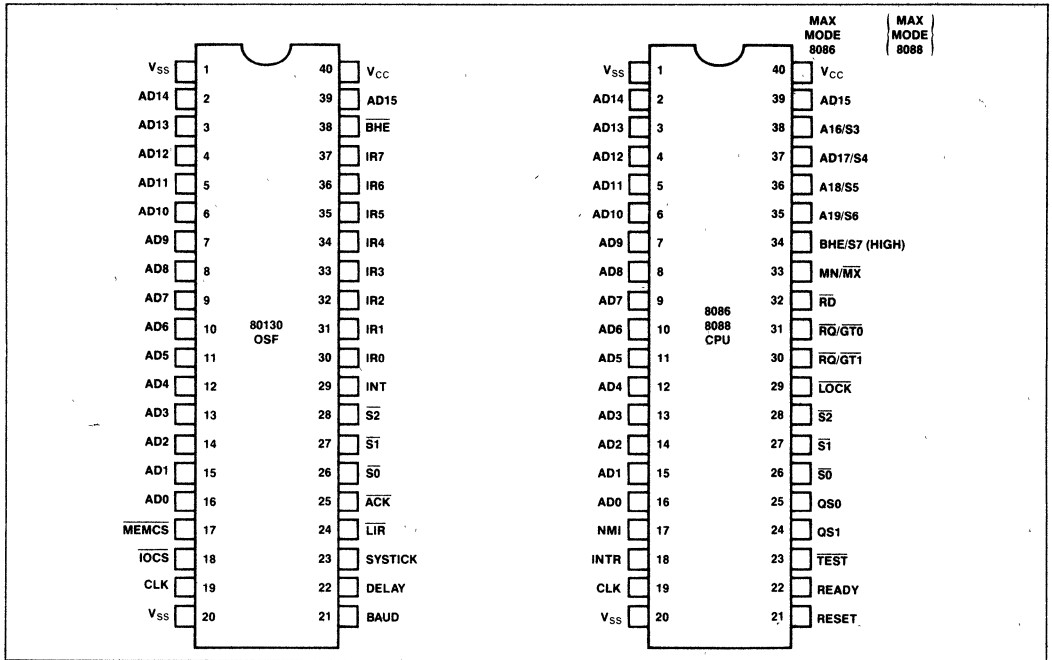


Figure 1. 8086 and 80130 Pinout Diagrams

EVOLUTION OF PROCESSOR EXTENSIONS

In the early days of microcomputing (circa 1974), things were simple. The first microprocessors comprised just the central processing unit of a simple computer. Systems built up from these processors were generally small, dedicated-purpose device controllers—often replacing the random logic of an earlier design. The system designer had responsibility for the development of the hardware and all application software.

Semiconductor technology has progressed rapidly since then. Devices have become more sophisticated, as have the applications in which they are used. System functions today are more complex than they used to be, and are demanding more in the way of both system hardware and software.

To help designers cope with this complexity, semiconductor vendors are building increasingly more “functionality” into their standard product lines. Whereas the general arithmetic functions of the 8080 and 8085 were limited to addition and subtraction of eight-bit unsigned (ordinal) values, for example, the Intel® 8088 and 8086 now add, subtract, multiply, or divide eight- or 16-bit, signed or unsigned variables—an obvious improvement.

The evolution of floating-point arithmetic provides another example of technology growth. Initially, designers of numeric and process-control systems each developed the floating-point arithmetic routines they needed. Intel eased this task considerably in 1977 when it introduced a standard floating-point format and a floating-point arithmetic software library, FPAL-80. In 1978, the iSBC 310 High-Speed Mathematics Unit implemented these same functions with dedicated hardware and executed them an order-of-magnitude faster.

The 8231A Arithmetic Processor Unit (introduced in 1979) provided similar functionality in one chip at much lower cost. To accommodate the needs of today's world, the Intel RealMath™ software standard and the 8087 numeric coprocessor perform 80-bit floating-point arithmetic for high-performance 8088 and 8086 systems.

This evolution of floating-point hardware illustrates two recurring themes in the microcomputer industry. First, there is a natural trend toward componentization:

1. New applications reveal a need for new types of functionality (in this case, floating-point arithmetic).
2. As common requirements become evident, vendors develop software to serve these needs.

3. Specialized hardware is developed to support the established functions more simply and effectively than software alone.

In time, everything ends up in silicon.

The second theme is this: different functions should be implemented in different ways to fit the customer's needs. “Universal” requirements—like 16-bit multiplication—are best incorporated into the CPU. Functions needed only by certain applications—like high-speed, extended-precision square roots—should be provided as optional Processor Extensions so that their expense is incurred only by those who need them. In keeping with this philosophy, Intel currently offers several processor extension products (see “What's in a Name?”).

What's in a Name?

The 80130 Operating System Firmware (OSF) device is only the latest member of an extremely flexible family of Intel microprocessors. Its siblings include the 8086 and 8088 Central Processing Units (CPUs), the 8089 I/O Processor (IOP), and a floating-point math coprocessor, the 8087 Numeric Processor Extension (NPX). These individual standard components may be mixed and matched in numerous ways to create combinations optimized for widely varying applications.

To make it easier to discuss the most common configurations, Intel has defined an “Advanced Processor Series” (iAPX) numbering scheme, something akin to those used in the minicomputer and mainframe worlds. The 8086 CPU by itself, for instance, is called the iAPX 86/10. The 8086/8087 combination is dubbed the iAPX 86/20. An 8086/80130 pair has the name iAPX 86/30. The 8086, 8087, and 80130 together would form an iAPX 86/40.

When each of these combinations uses an 8088 in lieu of the 8086, each of the numbers above substitutes “88” for the “86”. An 8088 teamed with an 80130 is therefore called the iAPX 88/30. Finally, adding an 8089 to any system changes the final zero to a one. So, an iAPX 88/41 system would be one using the 8088/8087/8089/80130 chip set.

Real-Time Operating Systems

Let's turn our attention now to the subject of microcomputer operating system software—an area steadily growing in importance. The trends toward standardized functions with specialized implementations will become evident.

But first, what is an operating system? The phrase means different things to different people. In 20 words or less: An OS is a tool, a set of programs or routines which reduce and simplify the problem of managing system resources. (Well, 21, actually . . .)

Most microcomputer programmers have encountered single-user diskette operating systems, Intel's ISIS-II®, and CP/M® and CP/M-86® from Digital Research Incorporated among them. In essence, an OS of this sort is a collection of run-time subroutines which perform device I/O operations and give application programs access to a disk-based file system. Along with these are routines to supervise the loading and execution of application programs. Historically, this type of OS is oriented toward user-interactive applications: software development, business computing, and the like.

In the mainframe world, the goal of an operating system is to use expensive equipment as efficiently as possible. Batch processing systems ensure that programs waste as little CPU time as possible, though each monopolizes the CPU until it has completed. A time-sharing OS allots short periodic "slices" of time to each of several independent users, during which each has access to the CPU, memory, and other system resources.

A step above the traditional time-sliced OS are "real-time, multitasking operating systems." But what is a "real-time" application? ("Don't all programs execute in real time?")

A real-time system is one in which the CPU must do many different things (tasks), all more-or-less simulta-

neously. Unlike the sequential time-sharing of mainframe OSs, though, the tasks are prioritized. Low-priority tasks are preempted if any of higher priority have work to do. The higher-priority task then runs until it must wait for some external event to occur or no longer needs the CPU for some other reason. Thus, the CPU services tasks in their order of importance.

A computer controlling factory machinery, for instance, might perform five separate tasks:

1. Monitor input switches to detect emergency conditions, determine intended operating mode, or update indicator lights showing machine status;
2. Drive a stepper motor to position a tool;
3. Keep track of the time of day;
4. Send output to the console (e.g., CRT), either in response to explicit commands or as part of some other task;
5. Read and process characters entered from a console keyboard.

These tasks seem largely unrelated, though the first few may be more important to system operation than the others. Let's consider some alternate ways to accomplish these functions with today's microcomputers.

Conceptually, the most straightforward approach might be to dedicate a separate computer to each. The program for each would then be quite simple: an initialization phase followed by an endless loop performing the dedicated function. Algorithms for the first four tasks are flowcharted in Figure 2.

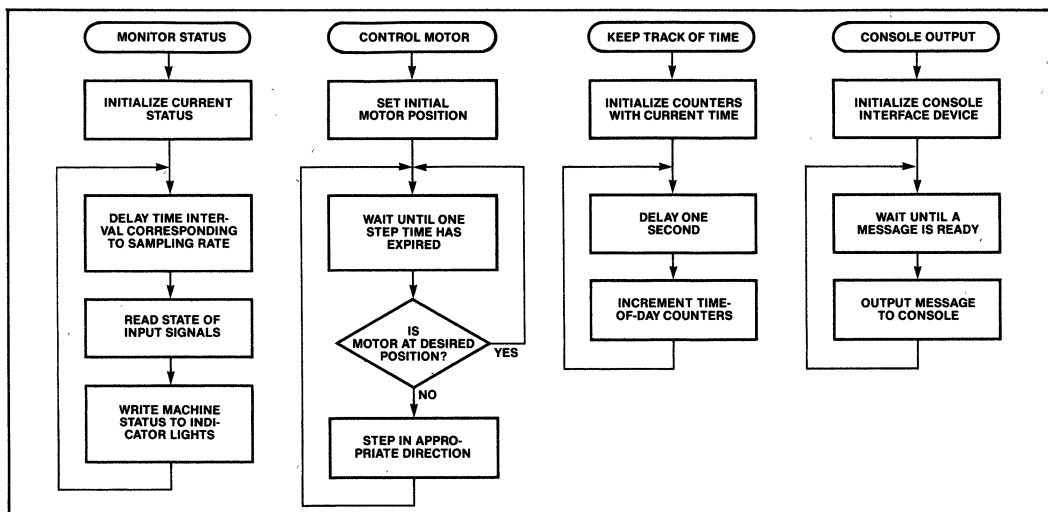


Figure 2. Flowcharts for Concurrent Machine-Tool Tasks

What's wrong with this approach? Ignoring cost, the need for multiple CPUs becomes physically unrealistic for more than a few tasks—60, say, or 600. And tasks are rarely fully independent; note that the switches monitored by task 1 could affect task 2, and that tasks 4 and 5 interact with the rest of the system in as yet undefined ways. So, some sort of communications would have to be set up between the micros.

Exercise 1. Suppose five tasks are all interrelated. How many communications channels would have to be set up between different processors? If each channel requires two dedicated communication

chips, how would the number of peripheral devices compare with the number of CPUs?

In each task, the CPU spends most of its time waiting for time to pass or for something to happen. One CPU would be able to implement all five tasks if its time were properly divided among them. An alternate approach, then, might be for a single processor to attend to each task in turn, performing the actions called for by each. Figure 3 shows a flowchart for this scheme. Only one CPU is required and the tasks can communicate between themselves and share physical resources like the console.

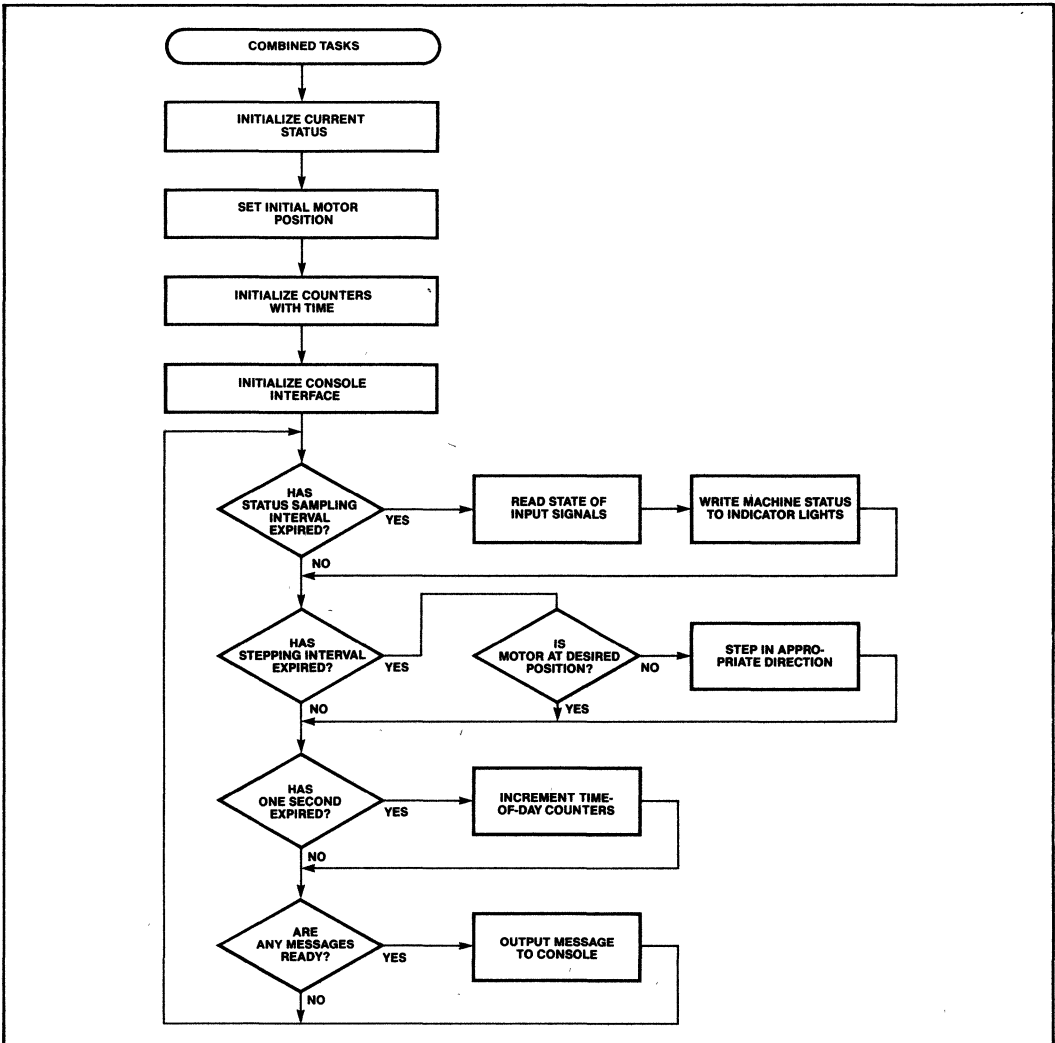


Figure 3. Machine-Tool Tasks Implemented Via Polling Scheme

The problem here is the heavy interaction between tasks. Before it can be serviced, an important task may have to wait for many other less critical tasks to complete. This imposes a constraint that each task release the CPU as quickly as possible. Also, lumping tasks together obscures the boundaries between them. Initialization sequences must be grouped with each other, rather than with the sections of code affected. Adding to or deleting any task may affect the others. It's not clear how to structure the program such that programmers could cooperate on such a program.

Moreover, the various tasks can interfere with each other. Suppose on a given pass through the processor loop, three tasks each send one new character of a message to the console display screen. The resulting output would be most interesting.

The third, and optimal approach, would be one which combined the advantages of the first two approaches, while avoiding the pitfalls. Each function of the overall system could be designed, written, and tested separately, as in the first approach, yet all the software would run on a single computer system as in the second. Tasks could therefore communicate with each other easily, and share peripherals such as CRTs. This multitask control and communication function could be performed largely through software.

The key is finding a way to properly budget CPU time between the various tasks. Early pioneers of complex, real-time, control system design found that they needed special routines, apart from the application tasks themselves, to supervise the execution of application tasks. It was (at best) an inconvenience for so many engineers to independently define, design, document, test and debug software with the same general purpose. At worst, schedules slipped or projects were cancelled for the lack of reliable executive software.

To help avoid these hazards and free up the designers to concentrate on more immediate goals, Intel developed iRMX 80, the first real-time, multitasking, executive operating system for microprocessors. iRMX 86 was introduced to the 16-bit world two years later in 1980.

Because of the critical real-time nature of such operating systems, they require certain hardware capabilities in the host system, such as special timer logic clocked at certain frequencies to measure the passing of time, and interrupt controllers to monitor assorted asynchronous events. Combine all this with a handful of memory chips to house just the OS software, and the address decode and control logic needed by all of the above, and you'll find you need the equivalent of a single-board computer system just to support a multitasking environment.

Until now, that is. The current trend is to integrate OS software and hardware functions into silicon. Intel's iAPX 432 32-bit MicroMainframe™ system does this within the CPU. For the 16-bit world, however, Intel provides a separate chip, the 80130, which contains operating system firmware as well as timer and interrupt control functions.

What is the 80130 OSF? It is an extremely sophisticated integrated circuit, fabricated using Intel's high-performance HMOS technology, which contains over 160,000 devices. In one 40-pin package (Figure 4), the 80130 combines several timers, multiple-mode interrupt control logic, and a large control store memory—plus buffers, decoders and the like—to form the integrated heart of a multitasking operating system. Compared with the iRMX 86 Nucleus, for example, the 80130 replaces an 8259A PIC, an 8253 PIT, a special oscillator, 16K bytes' worth of memory, and associated control logic.

The 80130 operates in conjunction with the 8086 CPU. Together, the two chips are called the iAPX 86/30 OSP. The same device may be paired just as easily with an 8088 forming the iAPX 88/30. From here on, though, references to the 8086 or "host processor" apply to both CPUs. Due to the high speed of HMOS, the 80130 currently runs at system clock rates up to 8 MHz without inserting any wait states. Firmware in the 80130 supports the 35 primitive functions listed in Table 1. Many of these are discussed in Chapter IV.

SYSTEM HARDWARE DESIGN

The 80130 supports a wide range of system architectures, from compact to quite complex. Most, however, have in common the functional blocks represented in Figure 5. After a brief review of iAPX 86/30 systems in general, we'll examine 80130 requirements in greater detail.

Basic Functional Blocks

In addition to the 80130, the central processing "core" of a typical OSP system would include an 8088 or 8086 operating in maximum mode, an 8284A clock generator, and an 8288 system controller, all connected according to the standard rules. More on the 80130-specific interconnects later.

Address latches (e.g., 8282s or 8283s) are generally needed to demultiplex the processor address bus for standard memory devices and for memory and I/O device-select logic. The number (from zero to three octal latches) depends on the host processor, memories, and the addressing scheme employed. Data

Table 1. Operating System Primitives Supported by 80130

<p>Task Management</p> <ul style="list-style-type: none"> Suspend Task Resume Task Sleep Create Task Delete Task Set Priority Get Task Tokens 	<p>Interrupt Management</p> <ul style="list-style-type: none"> Set Interrupt Signal Interrupt Reset Interrupt Enter Interrupt Wait Interrupt Exit Interrupt Enable Disable Get Level
<p>Intertask Communications and Synchronization</p> <ul style="list-style-type: none"> Send Message Receive Message Create Mailbox Delete Mailbox 	<p>Free Memory Management/System Partitioning</p> <ul style="list-style-type: none"> Create Segment Delete Segment Create Job
<p>Mutual Exclusion Control</p> <ul style="list-style-type: none"> Receive Control Accept Control Send Control Create Region Delete Region 	<p>Misc. Support</p> <ul style="list-style-type: none"> Signal Exception Get Type Disable Deletion Enable Deletion Set O.S. Extension Get Exception Handler Set Exception Handler

transceivers (8286s or 8287s) may also be needed for increased bus buffering.

Any complete microprocessor system must also have some combination of I/O peripherals and memory, collectively indicated by the box labeled "Local Resources." As we shall see, some of the system RAM and ROM (or EPROM) must be reserved for OSP itself. Additional logic decodes the latched address lines to generate chip-select signals for the memory and I/O devices.

This note only discusses simple, single-processor systems. More sophisticated architectures may incorporate a multimaster system bus, in addition to a local processor bus. This would require additional system controllers, address latches, and bus transceivers for bus isolation, and address mapping logic (not shown) to select between the various busses, enable the respective transceivers, generate a System Ready signal, and so forth. For design information on such techniques, refer to application note AP-67 in the *iAPX 86,88 User's Manual*.

80130 Pin Functions

Back to the 80130. Certain pins on the 80130 (in particular, AD15-AD0) attach directly to the CPU. The AD pins are bidirectional, accepting addresses from the host and returning instructions or data. By monitoring the system clock and status signals, $\overline{S2}$ - $\overline{S0}$, the 80130 can decode the processor status internally and respond automatically to the appropriate bus cycles. The \overline{BHE} input lets the 80130 determine the width of data transfers and distinguishes an 8088 host from an 8086. If you refer back to Figure 1, you'll notice that these 80130 pin assignments were selected to simplify P.C. board layout.

Because of the 80130's location on the CPU side of any latches or data transceivers (on what is sometimes called the "pin bus"), the transceivers (if used) must be disabled when the 80130 is driving the processor bus. Whenever the 80130 is responding to any type of bus cycle, it generates an \overline{ACK} signal. As Figure 4 suggests, one way to avoid contention is to simply disable the transceivers when \overline{ACK} is active. \overline{ACK} can also be used to prevent the insertion of wait states.

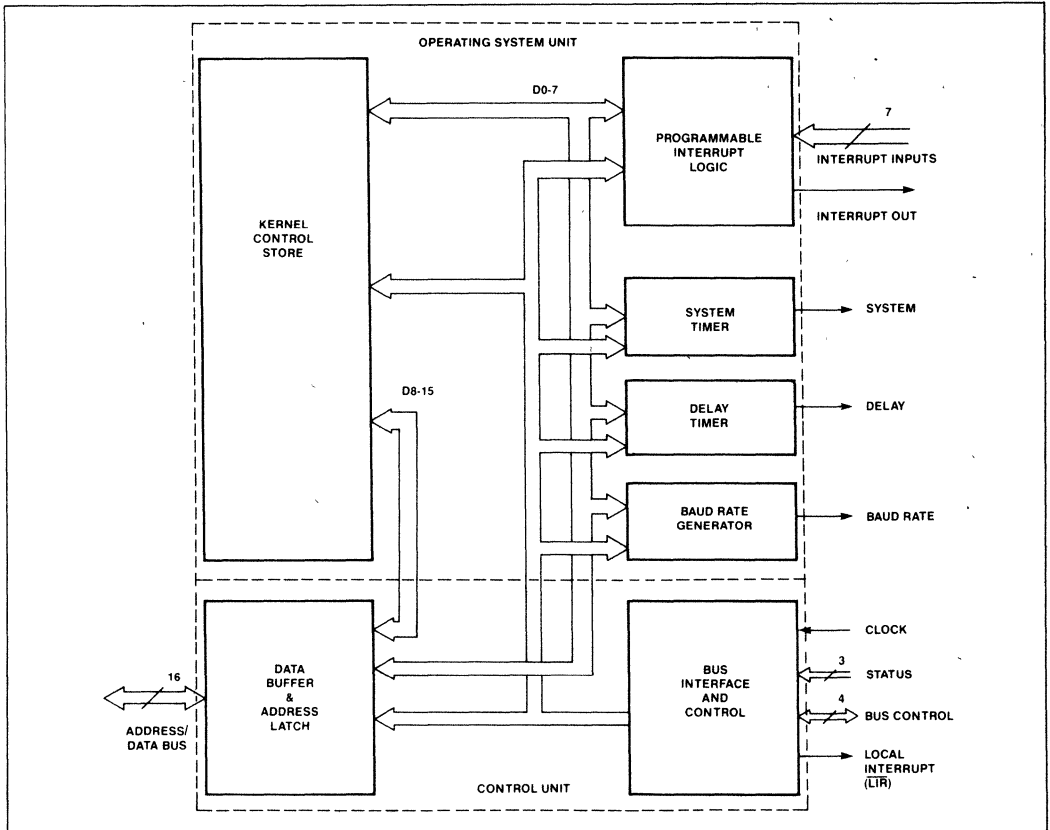


Figure 4. 80130 Internal Block Diagram

Additional pins on the 80130 include eight interrupt-request inputs. Internal interrupt control logic provides many of the functions of the 8259A. During system configuration (Chapter V), each of the eight may be individually defined as a direct level-sensitive or edge-triggered interrupt request, or each may be cascaded with a standard 8259A in slave mode.

The INT output must be connected to the host CPU to inform it of an enabled interrupt request. In very large systems with multiple, cascaded interrupt controllers, Local Interrupt Request ($\overline{\text{LIR}}$) indicates to the bus contention logic whether a requesting slave is local, or must be accessed via a multimaster bus.

The 80130 also contains dedicated timer logic to provide the OS time base, which is output on SYSTICK. Software operating in conjunction with the 81030 assumes one of the interrupt inputs (INT2 in this case) is

driven by SYSTICK, so this connection must be made externally. Routines within the 80130 initialize and perform all bit-level control of the interrupt and timer logic, according to options and parameters specified during the configuration process. Freeing the programmers from this tedium allows them to devote more thought to solving their own unique problems.

An additional, independent timer generates a user-programmable, square-wave output signal called BAUD to clock an off-chip USART.

Since the 80130 displays some of the characteristics of both memory and I/O, it requires chip-select signals for both the memory (MEMCS) and I/O (IOCS) address spaces. These are discussed at length below. Finally, Intel has reserved one output pin (called "DELAY") for use in future designs. Leave it unconnected in iAPX 86/30 systems.

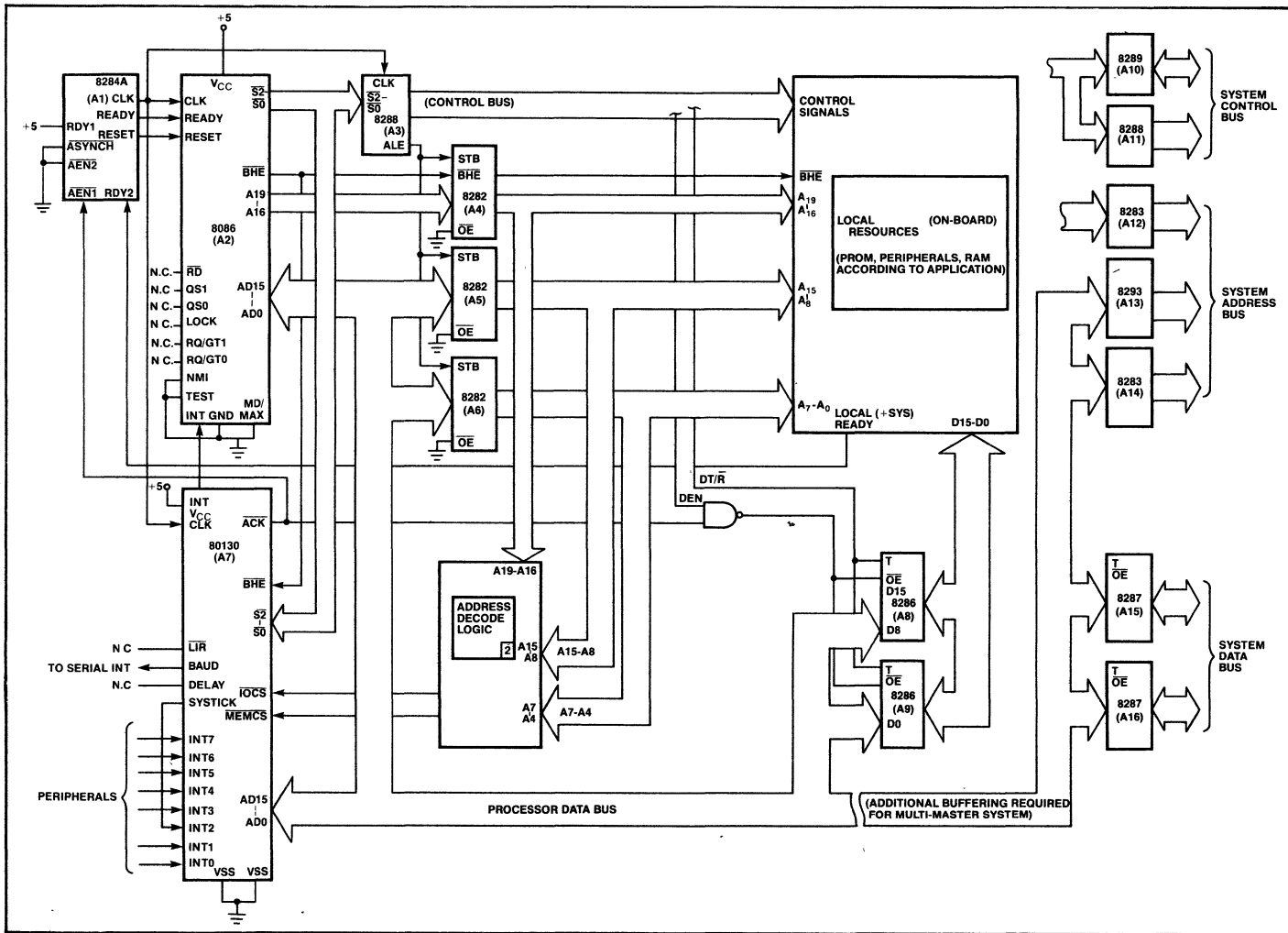


Figure 5. Basic iAPX 86/30 Microcomputer System Block Diagram

Additional System Requirements

The OSP requires a certain amount of off-chip memory for its own operation. The system must provide at least 1K bytes of RAM at address 00000H for the CPU interrupt vectors, plus another 1500₁₀ bytes for OSP system variables, data structures, stacks, and the like. This RAM may reside anywhere in the 8086 megabyte address space, although it is often contiguous with the interrupt vector up front. Application tasks must each have their own stack, so allow at least an additional 300 bytes of RAM for each.

Any iAPX 86 system must have ROM or EPROM at the upper end of memory to hold the CPU restart vector. About 3400 more bytes are consumed by code to initialize and access the OSP. This code is generated automatically from libraries on a diskette provided with a product called the iAPX 86/30 and iAPX 88/30 Operating System Processor Support Package (iOSP 86). Space left in the initialization EPROMs is available for application tasks.

As code is being written, the system designer should count on another 1500 bytes of code from the support

libraries being added to his application during the linking and system configuration steps. These memory requirements are shown in Figure 6. In practice, the separate blocks in this figure would be grouped together for more efficient use of RAM and EPROM chips.

The 80130 occupies a 16K-byte block of addresses in the host-processor memory space, so external logic should decode address bits A₁₉-A₁₄ to generate $\overline{\text{MEMCS}}$. Similarly, the timer and interrupt control logic occupy a 16-byte block of addresses in the I/O space; at least some of the bits A₁₅-A₄ must be decoded to generate $\overline{\text{IOCS}}$. The 80130 decodes all the lower-order address bits (14 for memory, four for I/O internally).

Firmware in the 80130 leaves a great deal of flexibility in decoding the chip-select signals, to be compatible with whatever decode logic is already present in the system. The I/O starting address may be on any 16-byte boundary in the full CPU I/O space. The memory block has only two restrictions: the off-chip initialization and interface code memory must be placed immediately above the MEMCS block, so the 80130 may not occupy the extreme top of memory, nor may the 80130 reside at address 00000H since this area is reserved for interrupt vectors.

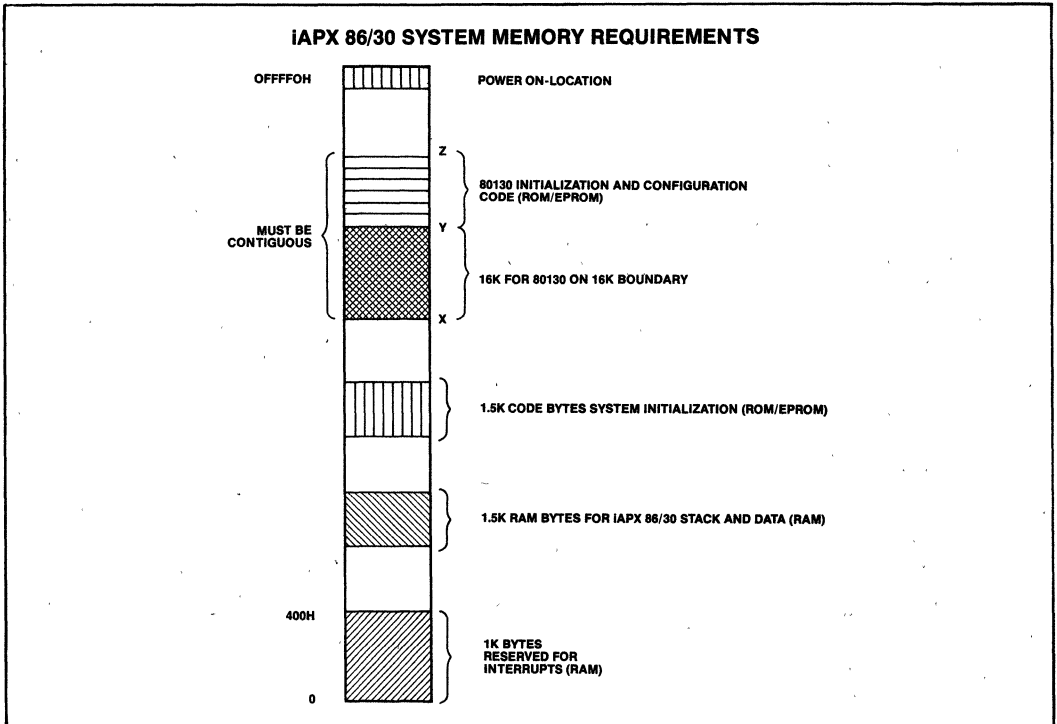


Figure 6. Operating System Processor System Memory Requirements

The propagation delay numbers plugged into the equation are worst-case values from the appropriate Intel data sheets. The CPU is an 8086-2 operating at 8 MHz. This means the address decode logic must produce stable CS outputs within 140 nanoseconds.

Exercise 2. Using standard, low-power Schottky TTL, does it make sense for a circuit to take longer than 140 nsec. to decode 6 program or 12 I/O address bits? Even if the rather liberal setup specs are not met, the 80130 would still work fine. Wait states would be needed until the chip-select signal was active, however, so performance would degrade some.

The second point of concern relates to ready signal timing. The 80130's acknowledge output signal, \overline{ACK} , can be used to control the CPU's ready signal. For this case, the chip-select signal must be active early in a memory or I/O cycle to allow activation of \overline{ACK} early enough to prevent wait states. There are two schemes for implementing ready signals; "normally ready" and "normally not ready." (For more details, refer to AP-67, "8086 System Design.") Chip-select timing is more critical in some "normally not ready" systems.

In a "normally not ready" design, acknowledge signals are generated when each resource is accessed. The individual acknowledgements are combined to form a system-wide ready signal which is synchronized by the 8284A clock generator via the RDY and AEN inputs. The 8284A can be strapped to accept asynchronous ready signals (asynchronous operation) or to accept synchronous ready signals (synchronous operation). Synchronous 8284A operation provides more time for address latch propagation and chip-select decoding. In addition, inverting ACK off chip produces an active-high ready signal compatible with the 8284A RDY inputs, which have shorter set-up requirements than AEN inputs. (As a side benefit, a NAND gate used like this can combine ACK with the active-low acknowledge signals from other parts of the system.) Based on these assumptions, the time available for address latch propagation and chip-select decoding at 8 MHz is:

$$\begin{aligned}
 T_{CLAV} + T_{OVCS} + T_{CSAK} + R_{R1VCL} &\leq T_{CLCL} + T_{CLCL} \\
 T_{OVCS} &\leq 2 T_{CLCL} - T_{CLAV} - T_{CSAK} - T_{R1VCL} \\
 &\leq 250 - 60 - 110 - 35 \\
 &\leq 45 \text{ nsec.}
 \end{aligned}$$

The circuit in Figure 8 which uses Schottky TTL components leaves about 15 nsec. to produce \overline{MEMCS} from

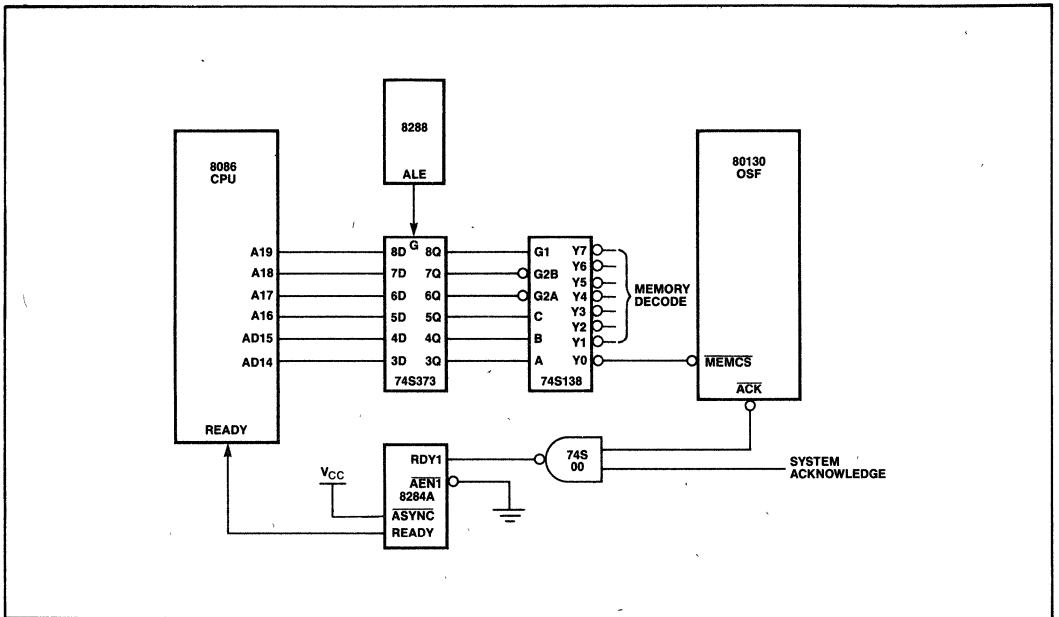


Figure 8. High-Speed Address Decoding Circuit

the high-order address bits—more than enough for the 74S138 one-of-eight decoder shown.

Granted, this does not leave much leeway to fully decode the I/O address bits. A 12-input NAND gate on AD15–AD4 could be used, introducing only a single propagation delay but forcing the I/O register block to start at 0FFF0H. Incomplete decoding is also legal: it is safe to drive $\overline{\text{IOCS}}$ with the (latched) AD15 signal directly, provided all other ports in the system are disabled when this bit is low. In this case, the effective address of the I/O block (which must be specified during the system configuration step) could be 0000H, or any other multiple of 16 between 0000H and 7FF0H.

Again, the OSP system will still operate even if the memory or I/O decoding is slow. The acknowledge signal returned to the host CPU would just be delayed accordingly, so unnecessary wait states would be inserted in access cycles, but the 80130 would *not* malfunction. Only rarely does the OSP access resources in its I/O space. Even if slow decode logic were to insert several wait states into every I/O cycle, the overall effect on system performance would be insignificant.

A few words of caution, though. If the 8284A is strapped for synchronous operation, external circuitry must guarantee that ready-input transitions don't violate the latch set-up requirements. Also, the chip-select signal must *not* remain low so long after the address changes that the 80130 could respond to a non-80130 access cycle.

Exercise 3. Suppose the typical timing values for a particular decoder would easily meet the ready-input set-up requirements presented above for asynchronous 8284A operation, but pathological worst-case figures were just a little slow. Could that circuit still be used safely in most applications? What would happen if the worst-case combination of worst-case conditions ever actually did occur? These occasional extra wait states would probably not cause a hard system failure.

Exercise 4. Earlier it was mentioned that the acknowledge signal could also be used to avoid bus contention. Prove that with any decode logic which meets the above requirements, $\overline{\text{ACK}}$ would disable the bus transceivers before the host CPU samples the bus.

Example System Design

Appendix A includes full schematics for a complete iAPX 86/30 system providing considerable functionality with only 27 chips. In addition to the OSP, the

system has 4K bytes of 2114 RAM (with sockets for another 4K), from 8K to 32K bytes of 2732A or 2764 EPROM, an 8251A USART operating at 9600 baud, and an 8255A Programmable Peripheral Interface with 24 parallel I/O lines. Eight of the inputs read logic values off DIP switches; eight outputs drive small LEDs. Four more outputs connect to the coil drivers of a four-phase stepper motor. A layout diagram of the prototype appears in Figure 9.

The system is even simpler than the discussion of "typical" requirements implied. The 8086 direct-bus drive capability is adequate to make the data transceivers unnecessary. (To equalize the bus loading, the 8255A is connected to the upper half of the bus.) Address decoding logic was minimized by making the high-order address bits "don't-cares." Moreover, the part count could have been reduced to 16 using an 8088 and multiplexed-bus 8185 RAMs and 8755A EPROMs. (The reader may be surprised to learn that, except for wire-wrapping mistakes, the prototype system hardware worked when it was first powered up. The author certainly was!)

APPLICATION SOFTWARE DEVELOPMENT

Like other well-structured programs, application software to run on the iAPX 86/30 is written as a number of separate procedures or subroutines. In conventional programs, though, execution begins with a section of code (the *program body*) at the outermost level. The program calls application procedures, which may call other procedures, but which eventually run to completion and return to the program body.

In an OSP application, though, there is no "outermost level" in the traditional sense; rather, the procedures are started, suspended, and resumed as situations warrant under the control of the OSP. The term "task" refers to the execution of such a procedure in this way. While an instruction stream is suspended, the OSP keeps track of the task state (instruction counter, CPU register contents, etc.) so that it may be resumed later.

Each task is assigned a relative priority by the programmer, on a scale of 0 (high priority) to 255 (low). Tasks with higher (numerically lower) priority are given preferential treatment by the OSP; the task actually controlling the CPU at any given instant will be the one with the highest priority which is not waiting for some event to occur. (If all this sounds confusing, examples coming later may help.)

A task which operates independent of other tasks can be written without knowing anything about the others.

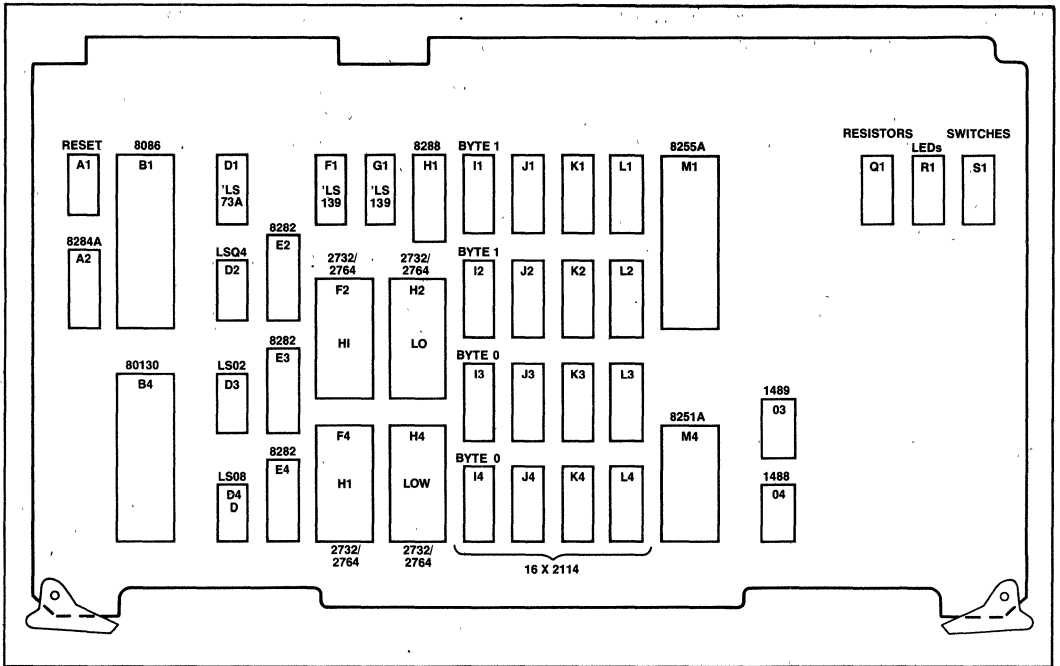


Figure 9. Example System Prototype Layout

This makes it easy to divide a very large programming job among a team of programmers, each writing the code for some of the tasks. Moreover, a task need not even know if other tasks exist. They may be tested and debugged before others have even been written. As an application evolves, new tasks may be added or unnecessary ones removed without affecting the rest.

The number of tasks in an application may need to be quite large. The number of tasks allowed in one application is essentially unlimited, as is the number of other objects—regions, mailboxes, segments, and the like. (The term “object” relates to different types of data structures maintained internally by the OSP.) Each object is internally identified by a unique 16-bit “token,” which means the theoretical maximum total is over 65,000. The more pragmatic issue of physical memory consumption limits the number of simultaneous concurrent tasks to “only” several thousand.

(When a number of tasks cooperate to accomplish some common goal, the collection of tasks is referred to as an application “job.” The OSP also allows for an unlimited number of application jobs, though only one is illustrated in the example discussed here. A second similar machine, with different status switches, a differ-

ent motor, and a different console might make up a second job.)

All OSP application jobs must have one special initialization task (often called INIT\$TASK) just to get started; this one may, in turn, create other tasks as it executes. The initialization task for this example is discussed at the end of this chapter.

Hardware Initialization

The life of any task can be broken into three phases: start-up, execution, and termination. The start-up phase initializes variables, data structures, and other objects needed by the task. During the execution phase the task performs its useful work. Depending on the application, this may be a single sequence of actions, or a loop executed repeatedly. When the task completes, it must terminate itself so as not to use any more CPU time. One or more phases may be omitted. For example, some tasks are intended to execute “forever,” in which case the termination phase is not required.

This life cycle is suggested by Example 1, a segment of code called HARDWARE\$INIT\$TASK. This task first

programs the 80130 internal timer logic to generate a square-wave cycle on the BAUD pin every 52 system clock cycles, which corresponds to a system console data rate of 9600 baud. The task then sets the system's 8255A PPI and 8251A USART devices to operate in the desired modes, and outputs a short sign-on message to the CRT. For the sake of reader's unfamiliar with the protocol for interfacing with the 8251A, simple input and output routines (C\$IN and C\$OUT) are reproduced in Example 2.

```
HARDWARE$INIT$TASK PROCEDURE.
DECLARE HARD$INIT$EXCEPT$CODE WORD.
DECLARE PARAM$51 (*) BYTE DATA (40H,8DH,00H,40H,4EH,27H).
DECLARE PARAM$51$INDEX BYTE.
DECLARE SIGNON$MESSAGE (*) BYTE DATA
  ('CR,LF','APX 86/30 HARDWARE INITIALIZED',CR,LF).
DECLARE SIGNON$INDEX BYTE.

OUTPUT(PPI$CMD)=90H.
OUTPUT(TIMER$CMD)=0B6H.
OUTPUT(BAUD$TIMER)=33, /*GENERATES 9600 BAUD FROM 5 MHZ*/
OUTPUT(BAUD$TIMER)=0.
DO PARAM$51$INDEX=0 TO (SIZE(PARAM$51)-1),
  OUTPUT(CMD$51)=PARAM$51(PARAM$51$INDEX),
  END, /*OF USART INITIALIZATION DO-LOOP*/
DO SIGNON$INDEX=0 TO (SIZE(SIGNON$MESSAGE)-1),
  CALL C$OUT(SIGNON$MESSAGE(SIGNON$INDEX)),
  END, /*OF SIGN-ON DO-LOOP*/
CALL RQ$RESUME$TASK(INIT$TASK$TOKEN,@HARD$INIT$EXCEPT$CODE),
CALL RQ$DELETE$TASK(0,@HARD$INIT$EXCEPT$CODE),
END HARDWARE$INIT$TASK.
```

Example 1. System Hardware Initialization Task

```
C$OUT PROCEDURE (CHAR),
  DECLARE CHAR BYTE,
  DO WHILE (INPUT(STAT$51) AND 01H)=0,
    /* NOTHING */
  END,
  OUTPUT(CHAR$51)=CHAR,
  END C$OUT.

C$IN PROCEDURE BYTE,
  DO WHILE (INPUT(STAT$51) AND 02H)=0,
    /* NOTHING */
  END,
  RETURN INPUT(CHAR$51),
  END C$IN.
```

Example 2. Simple 8251A Input and Output Routines

The baud timer should be initialized by a code sequence like that shown here. The 80130 logic is actually compatible with the initialization sequence which would be needed to configure timer 2 of an 8253A as a programmable rate generator. The baud rate parameter loaded into the timer is simply the system clock frequency divided by the desired output frequency. No other timers should be affected by user programs.

When the hardware has been initialized, the task calls an operating system procedure called RQ\$RESUME\$TASK. This signals the OSP that the task's start-up phase has completed, and that the initialization task (which in this case suspended itself after creating HARD\$INIT\$TASK) may continue. Since its function is hardware initialization only, HARD\$INIT\$TASK has no execution phase *per se*. It terminates by calling

the procedure RQ\$DELETE\$TASK, suicidally specifying itself as the task to be deleted.

Exercise 5. Beginners may make two common programming errors when developing OSP tasks. The first is when a task deletes itself without ever resuming the suspended task that created it. The second is to not terminate a task properly, with the result that the processor executes a return instruction when the task's work is done. (However, execution of the task did not originate with a call from the OS.) As with all computers, an OSP will do exactly what it is told. How do you suppose the system would react in each case? (Hint: only one of the two failure modes is predictable.)

You may have noticed three things from this short example and Table 1. First, every OSP call begins with the letters RQ. (PL/M compilers totally ignore dollar signs within symbols; they serve only to split long symbol names to make them easier for humans to read.) The letters RQ don't mean anything in particular; their purpose is to make sure OSP routine names don't conflict with any user symbols. These particular letters were chosen to be compatible with the historical naming convention used by iRMX 86. It may be useful, though, to think of RQ as an abbreviation for REQUEST, implying that the OSP provides useful services at the bidding of application code.

The second thing to notice is that the OSP routine names imply pretty well what each routine does. On the one hand, long procedure names take a little longer to type; on the other, they make code listings much easier to read and understand. In effect, the long names help make OSP code self-documenting. The long names shouldn't hinder code development; rarely can programmers think faster than they can type. If they could, programmer productivity would be measured in thousands of lines per day.

The third thing is that the last parameter in every OSP system call points to a word in which the OSP procedure will return an exception code to the application task. The procedure will return a non-zero exception code in this word if it cannot do its job correctly. This does not always imply that an error occurred; sometimes it just means another task isn't ready to cooperate yet. Sometimes an exception value indicates whether the OSP request was processed immediately or delayed for some reason. In fact, some OSP routines are guaranteed never to return a non-zero exception code, yet the pointer is still required for the sake of consistency. For a full explanation of the other parameters for the OSP procedures and details on what the different exception codes mean, consult the *iAPX 86/30, 88/30 User's Manual*.

To illustrate how the OSP procedures are used, the following code examples implement the machine controller tasks introduced earlier. Appendix B puts all the code examples together, though not in the exact order discussed. *Be Forewarned:* the examples border on trivial. They are in this note to demonstrate how to call system routines with as few lines of code as possible, not to tax the capabilities of the OSP. In fact, none of the tasks even check for exception codes returned by the OSP, under the naive assumption that nothing will go wrong in a debugged program. If you're interested in more elaborate software examples, consult application notes AP-86 and AP-110. These notes focus specifically on iRMX 86, but their methods and much of the code apply equally to the OSP systems.

Simple Time Delays

The STATUS\$TASK routine simply monitors eight switches through an input port, and updates eight LEDs with a pattern determined by the switch settings and task status. Specifically, the LEDs display the bit-wise Exclusive-OR function of the inputs and an eight-bit software counter maintained by the task. This action will repeat twice per second. The task does nothing between iterations.

The RQ\$SLEEP routine gives application tasks a way to release the CPU when it is not needed. Any task calling this routine is "put to sleep" for the amount of time it specifies (from 1 to 65,000 SYSTICK intervals), releasing the CPU to service other tasks in the meantime. After the requested time has transpired, the OSP task will reawaken the task and resume its execution, provided a more important task is not then executing.

The 80130 timer logic generates the fundamental System Tick by dividing the system clock frequency by two, then subdividing that frequency by a 16-bit value specified during the configuration process. The period used here is 5 msec., which would result in an 5 MHz system by dividing the 2.5 MHz internal frequency by 12,500.

Exercise 6: At this rate, what's the longest gap that would result from a single call to RQ\$SLEEP? How could this duration be extended?

PL/M listings for the complete STATUS\$TASK routine appear in Example 3.

```
STATUS$TASK PROCEDURE,
DECLARE STATUS$COUNTER BYTE,
DECLARE STATUS$EXCEPT$CODE WORD,

STATUS$COUNTER=0,
CALL RQ$RESUME$TASK(INIT$TASK$TOKEN, @STATUS$EXCEPT$CODE),
DO FOREVER,
  OUTPUT(PP1$B)=INPUT(PP1$A) XOR STATUS$COUNTER,
  STATUS$COUNTER=STATUS$COUNTER+1,
  CALL RQ$SLEEP(100, @STATUS$EXCEPT$CODE),
END,
END STATUS$TASK,
```

Example 3. Status Polling and Reporting Task

Stepper Motor Control

Conceptually, a stepper motor consists of four coils spaced evenly around a rotating permanent magnet. By energizing the coils in various combinations, the magnet can be induced to align itself with the coils, individually or in pairs. A microcomputer can make a stepper motor rotate, step-by-step, in either direction, by emitting appropriate coil control signal patterns at intervals corresponding to the step rate.

The stepper-motor sequencer (Example 4) is an embellished version of STATUS\$TASK. The OSP calls are intermixed with a few more statements of application code, and the task uses global variables as delay parameters. The reader may wish to adapt the command interpreter task at the end of this chapter to let the operator modify (read: "play with") these parameters to adjust the motor speed as the program runs.

```
DECLARE CW$STEP$DELAY BYTE,
CW$STEP$DELAY BYTE,
CW$PAUSE$DELAY BYTE,
CW$PAUSE$DELAY BYTE,

MOTOR$TASK PROCEDURE,
DECLARE MOTOR$EXCEPT$CODE WORD,
DECLARE MOTOR$POSITION BYTE,
MOTOR$PHASE BYTE,
DECLARE PHASE$CODE (4) BYTE
DATA (00000101B, 00000110B, 00001010B, 00001001B),

CW$STEP$DELAY=50, /*INITIAL STEP DELAYS = 1/4 SECOND*/
CW$STEP$DELAY=50,
CW$PAUSE$DELAY=200, /*PAUSES AFTER ROTATION = 1 SECOND*/
CW$PAUSE$DELAY=200,
CALL RQ$RESUME$TASK(INIT$TASK$TOKEN, @MOTOR$EXCEPT$CODE),
DO FOREVER,
  DO MOTOR$POSITION=0 TO 100,
  MOTOR$PHASE=MOTOR$POSITION AND 0003H,
  OUTPUT(PP1$C)=PHASE$CODE(MOTOR$PHASE),
  CALL RQ$SLEEP(CW$STEP$DELAY, @MOTOR$EXCEPT$CODE),
  END,
  CALL RQ$SLEEP(CW$PAUSE$DELAY, @MOTOR$EXCEPT$CODE),
  DO MOTOR$POSITION=0 TO 100,
  MOTOR$PHASE=(100-MOTOR$POSITION) AND 0003H,
  OUTPUT(PP1$C)=PHASE$CODE(MOTOR$PHASE),
  CALL RQ$SLEEP(CW$STEP$DELAY, @MOTOR$EXCEPT$CODE),
  END,
  CALL RQ$SLEEP(CW$PAUSE$DELAY, @MOTOR$EXCEPT$CODE),
  END,
END MOTOR$TASK,
```

Example 4. Stepper-Motor Controller Task

Real-Time Interrupt Processing

The 80130 supports a two-tiered hierarchy of interrupt processing. The lower-level tier corresponds to the

traditional concept of hardware interrupt servicing; a routine called an "Interrupt Handler" is invoked by the 80130 internal interrupt control logic for immediate response to asynchronous external events. A short routine like this might, for example, move one character from a USART to a buffer. Interrupt handlers operate with lower-priority interrupts disabled, so it is a good idea to keep these routines as quick as possible.

"Interrupt Tasks," on the other hand, are higher-level tasks which sit idle until "released" by an interrupt handler. The task then executes along with other active tasks, under the control of the OSP. Such a task should be used to perform slower but less time-critical processing when occasions warrant, such as when the aforementioned buffer is full. Moving such additional processing outside the hardware-invoked interrupt handler reduces the worst-case interrupt processing time.

This hierarchy also decreases interrupt latency. Most OSP primitives execute in their own, private "environment" (e.g., with their own stack and data segments) rather than that of the calling task. Interrupt handlers, on the other hand, run in the same environment as the interrupted task. (In fact, the 80130 primitives may themselves be interrupted!) Leaving the CPU segment registers unchanged minimizes software overhead and interrupt response time, but also means that interrupt handlers may not call certain OS routines. An interrupt task, on the other hand, is initiated and suspended by the OSP itself, with no such restrictions.

Let's see how these capabilities would be used. The time delays introduced by the RQ\$SLEEP call are only as accurate as the crystal frequency from which they are ultimately derived. This may not be exact enough for critical time-keeping applications, since oscillators vary slightly with temperature and power fluctuation.

To keep track of the time of day, the example system uses a 60-Hz A.C. signal as its time base. (Most power utility companies carefully regulate line frequency to *exactly* 60 Hz, averaged over time.) A signal from the power supply is made TTL-compatible to drive one of the 80130 interrupt request pins. An interrupt handler responds to the interrupts, keeping track of one second's worth of A.C. cycles. An interrupt task counts the seconds by incrementing a series of variables.

Example 5 illustrates the former routine. AC\$HANDLER simply increments a variable on each 60-Hz interrupt. Upon reaching 60, it clears the counter and signals TIME\$TASK (Example 6).

```
DECLARE AC$CYCLE$COUNT BYTE,
AC$HANDLER PROCEDURE INTERRUPT 59, /*VECTOR FOR 80130 INT3*/
DECLARE AC$EXCEPT$CODE WORD,
CALL RQ$ENTER$INTERRUPT(AC$INTERRUPT$LEVEL,@AC$EXCEPT$CODE),
AC$CYCLE$COUNT=AC$CYCLE$COUNT+1,
IF AC$CYCLE$COUNT >= 60
THEN DO,
AC$CYCLE$COUNT=0,
CALL RQ$SIGNAL$INTERRUPT(AC$INTERRUPT$LEVEL,
@AC$EXCEPT$CODE),
END,
ELSE CALL RQ$EXIT$INTERRUPT(AC$INTERRUPT$LEVEL,
@AC$EXCEPT$CODE),
END AC$HANDLER,
```

Example 5. 60-Hz A.C. Interrupt Handler

In its initialization phase, TIME\$TASK sets up the interrupt handler by calling the RQ\$SET\$INTERRUPT routine. The body of TIME\$TASK (the execution phase) is just a series of nested loops counting hours, minutes, and seconds. When TIME\$TASK calls RQ\$WAIT\$INTERRUPT inside its inner-most loop, the OSP suspends execution of the task until AC\$HANDLER signals that another second's worth of A.C. cycles has elapsed. Thus, interrupt handlers can serve to "pace" interrupt tasks. After a day, TIME\$TASK completes and deletes itself.

```
DECLARE SECOND$COUNT BYTE,
MINUTE$COUNT BYTE,
HOUR$COUNT BYTE,
TIME$TASK PROCEDURE,
DECLARE TIME$EXCEPT$CODE WORD,
AC$CYCLE$COUNT=0,
CALL RQ$SET$INTERRUPT(AC$INTERRUPT$LEVEL,01H,
INTERRUPT$PTR(AC$HANDLER),DATA$SEG$ADDR BASE,
@TIME$EXCEPT$CODE),
CALL RQ$RESUME$TASK(INIT$TASK$TOKEN,@TIME$EXCEPT$CODE),
DO HOUR$COUNT=0 TO 23,
DO MINUTE$COUNT=0 TO 59,
DO SECOND$COUNT=0 TO 59,
CALL RQ$WAIT$INTERRUPT(AC$INTERRUPT$LEVEL,
@TIME$EXCEPT$CODE),
IF SECOND$COUNT MOD 5 = 0
THEN CALL PROTECTED$CRT$OUT(BEL),
END, /* SECOND LOOP */
END, /* MINUTE LOOP */
END, /* HOUR LOOP */
CALL RQ$SET$INTERRUPT(AC$INTERRUPT$LEVEL,
@TIME$EXCEPT$CODE),
CALL RQ$DELETE$TASK(0,@TIME$EXCEPT$CODE),
END TIME$TASK,
```

Example 6. Interrupt Task to Maintain Time of Day

Exercise 7: The time maintained by TIME\$TASK is consistently wrong, unless the system resets at midnight. Aside from that, how much error would accumulate per month had TIME\$TASK paced its inner loop by calling RQ\$SLEEP if the system oscillator was 00.01% off? How does this compare with a cheap digital watch? How much error will accumulate from the 60-Hz time base described?

TIME\$TASK incorporates another gimmick: every five seconds it sends an ASCII "BEL" character (07H) to the console to make it beep, by calling a routine called PROTECTED\$OUTPUT. This lead-in gives us a chance to discuss OSP provisions for task synchronization and mutual exclusion.

Mutual Exclusion

Whenever system resources (e.g., the console) are shared among multiple concurrent tasks, the software designer must be aware of the potential for conflicts. In single-threaded (as opposed to multitasking) programs, the easiest way to transmit characters is by calling a console output routine (written by the user or supplied by the OS) which outputs the character code. (Remember the examples following the hardware initialization routine?)

This approach presents two problems in a multitasking system. One is efficiency: a high-priority task could hang up the whole system while it waits for a printer solenoid to energize, induce a magnetic field, accelerate the hammer, contact a daisy-wheel spoke, move it up to the ribbon, and press them both against the paper. This waste of time is termed "busy waiting," and should always be avoided. By OSP standards, even 1/30 of a second can seem interminable; if the printer is otherwise occupied, the whole system could shut down indefinitely.

Aside from efficiency, though, there is a more serious synchronization problem here. Assume Task A has a higher priority than Task B. Task A is asleep. Task B calls a subroutine to poll the USART and transmit a character. The USART becomes ready. When this is detected, the subroutine prepares to output the character to the USART

Time out! Task A just woke up and starts running. Task A wants to transmit its own character. It calls its own output routine, checks the USART, finds it available, sends it a new character, and goes back to sleep (or suspends itself, or awaits another interrupt—whatever).

Now Task B continues. It "knows" the USART is available, having dutifully monitored it earlier. Task B's character goes out to the USART. The USART goes out to lunch. (In practice, the USART will probably just transmit corrupted data; still, its operating requirements have been violated.)

In Task B's output routine, the sequence of statements from when the peripheral is found to be ready to when the next character is written constitutes a "critical region" (a.k.a. "critical section" or "non-interruptable sequence"). Recognizing such regions and handling them correctly is an important concern in any multitasking system, so the OSP provides several facilities—interrupt control, regions and mailboxes—to help handle general synchronization and mutual exclusion problems. Which one to choose depends on the circumstance.

Exercise 8: In this example, would it be better if Tasks A and B shared a single output routine, so that only one section of code sent data to the USART? Convince yourself that the same (or worse!) problems could still arise.

Sometimes critical sections can be protected by just disabling interrupts at appropriate points in the application software. To maintain the integrity of an iAPX 86/30 system, *application code must never execute the STI, CLI, or HLT instructions* (ENABLE, DISABLE, or HALT statements in PL/M), nor can it access the interrupt control logic directly. Instead, the interrupt status should be controlled with the OSP RQ\$ENABLE and RQ\$DISABLE procedures; routines should be halted via RQ\$SUSPEND or RQ\$WAIT\$INTERRUPT.

Back to TIME\$TASK: we want to transmit BELs to the console every five seconds. The console output task will be transmitting other characters. A "clever" programmer may recognize that this will lead to a critical section and analyze the situation as follows:

1. A hazard would arise if TIME\$TASK sends out a beep when CONSOLE\$OUT\$TASK is using the USART;
2. TIME\$TASK will only execute after being signaled by ASC\$HANDLER;
3. ASC\$HANDLER only reponds to an external interrupt.

"Therefore, all CONSOLE\$OUT\$TASK has to do to be safe is disable the 60-Hz interrupt around its output routine."

Not quite. There are still potential hazards. Suppose CRT\$OUT\$TASK has the same priority as TIME\$TASK. TIME\$TASK may already have been signaled by ASC\$HANDLER and be ready to run when CRT\$OUT\$TASK completes. An otherwise unrelated event—another interrupt, for instance—could momentarily suspend CRT\$OUT\$TASK during the critical region with A.C. interrupts disabled. When the OSP returns to that level, it might resume with TIME\$TASK, *not* CRT\$OUT\$TASK. This could lead to the same malfunctions as before, so disabling 60-Hz interrupts didn't help. This series of worst-case assumptions is admittedly convoluted, but the resulting sporadic errors are among the hardest of all bugs to squash.

The problem is that this attempted solution involves too much interaction between tasks, making it confusing and error-prone. Even if some scheme of priority-level assignments and task interactions could be made to work, later modifications or simple additions to the job

could cause bugs to reappear. (The analogy of an unexploded time bomb comes to mind.)

A simpler solution would be one corresponding more closely with the problem. Accordingly, the OSP supports several primitives just to supervise and control access to critical regions.

One of the OSP "data types" is a data structure called a "Region," which can be used by application code to control access to a shared port or some other resource. A task wishing access to the resource should call the OSP procedure `RQ$RECEIVE$CONTROL` before trying to access that resource; when done it must call `RQ$SEND$CONTROL`.

The OSP keeps track of which regions are in use. As long as a region is busy (i.e., has been entered but not yet exited), the OSP will prevent other tasks from entering the region by putting them to sleep. The OSP keeps a queue of all tasks waiting for the busy region. When the region later becomes available (i.e., when the task controlling the region calls `RQ$SEND$CONTROL`), one of the sleeping tasks—either the highest priority or the most patient—will be awakened, granted control of the region, and sent on its way. (When a region is created, the OSP is told whether to awaken tasks waiting for the region based on their priority or how long they have been waiting.) Effectively, a call to `RQ$RECEIVE$CONTROL` will not return to the application task until the resource in question becomes available.

The `PROTECTEDCRTOUTPUT` (Example 7) demonstrates this protocol. The routine is declared reentrant which means (by definition) the routine may be interrupted and restarted safely. A reentrant routine may be shared by a number of tasks, instead of replicating the same code throughout the application.

```
PROTECTED$CRT$OUT PROCEDURE (CHAR) REENTRANT,
  DECLARE CHAR BYTE,
  DECLARE CRT$EXCEPT$CODE WORD,
  CALL RG$RECEIVE$CONTROL (CRT$REGION$TOKEN, @CRT$EXCEPT$CODE),
  DO WHILE (INPUT (STAT$S1) AND O1H)=0,
    /* NOTHING */
  END,
  OUTPUT (CHAR$S1)=CHAR,
  CALL RG$SEND$CONTROL (@CRT$EXCEPT$CODE),
  END PROTECTED$CRT$OUT.
```

Example 7. CRT Output Routine Protected by Region Protocol

As a concession to simplicity, `PROTECTEDCRTOUTPUT` does use a form of the busy waiting method described earlier. The maximum delay at 9600

baud is only one millisecond, however, much shorter than a system tick. Besides, tasks performing character I/O will all have low priority levels, so the OSP would just delay them if anything more urgent comes up.

Exercise 9: Decide whether this explanation is a feeble attempt at rationalization, or a well-justified engineering trade-off.

Inter-Task Communication

But what if a high priority task must output a string of characters, or the peripheral response time is too long? Busy-waiting may not be acceptable. Alternatively, the output routine could buffer the data and service the USART within an interrupt routine. Another would be to simply pass the data off to a special (low-priority) output task and continue.

Tasks pass information to each other via something called a "message." A message may be the token for any type of OSP object, but the most common and most flexible type is called a "memory segment." In our example, segments will be used to carry strings of ASCII characters between tasks, so we'll examine segments first. Message formats are defined by the individual application programmer—make sure the sending and receiving tasks assume the same format!

A memory segment is just a section of contiguous-system RAM allocated (set aside) by the OSP at the request of an executing task. The OSP keeps track of a free memory "pool," which is initially all unused RAM in the system. When a task needs some RAM, it tells the `RQ$CREATE$SEGMENT` procedure how much it wants. The OSP finds a suitable memory block in the pool, and returns a 16-bit token defining its location. (If not enough memory is available, the procedure returns an exception code.)

The token is the base portion of pointer to the first usable byte of the segment, with the offset portion assumed to be zero. (The token values for all other objects have no physical significance.) Knowing this, it's possible to access elements of the segment as the application warrants.

The subroutine in Example 8 shows how to request a segment and construct a message. `PRINT$TIME` sends the ASCII values of the time-of-day counters (maintained in `TIME$TASK`) to the CRT output task described later. The message format adopted for these examples will consist of a byte giving the message

RECEIVE\$MESSAGE later, the OSP will give it the tokens one at a time.

What happens if a task tries to receive a message when the mailbox is empty? (This is quite possible, since tasks do run asynchronously.) What token would the OSP return?

In the simple case . . . it doesn't! Instead of returning right away with no data, the OSP will wait until data is available. In the meantime, the OSP puts the receiving task to sleep, remembering that it is waiting for a message at that mailbox. The next time a message is sent to that mailbox, the OSP will awaken the receiving task, give it the token, and—if its priority is high enough—resume its execution. Alternatively, receiving tasks may elect to not wait if the mailbox is empty, or to wait only a specified time.

Many tasks may actually send and receive messages through a single mailbox, with messages being queued in the order that the RQ\$SEND\$MESSAGE calls are executed. The OSP also maintains a list of tasks waiting to receive messages from an empty mailbox, analogous to the queued tasks waiting for region control. As each message is sent to the mailbox, it is passed immediately to a waiting task, either the one waiting the longest or the one with the highest priority (likewise determined by a parameter specified when the mailbox is created).

Exercise 12: Under what conditions could a mailbox's message queue contain messages waiting to be received, while the task queue contains tasks waiting for messages? Ignore the possibility that this may happen momentarily during the implementation of either routine. If you think of any such circumstances, please contact the author.

Example 10 shows a task which prints the messages sent above. Upon receiving a message token, CRT\$OUT\$TASK determines the message length from the first two bytes, and sequentially prints each element of the string through the PROTECTED\$CRT\$OUTPUT routine explained earlier. When done, the segment containing the message is deleted, returning its RAM to the free-memory pool.

A few words are in order about the segment accessing techniques demonstrated here. PL/M-86 has a special data type, called a "pointer," used to indirectly access other PL/M variables. OSP application programs must be compiled with the "compact" or "large" model specified. This tells the compiler to implement pointers as 32-bit double words corresponding to the two parts (base:offset) of the 8086 machine-segmented addressing scheme. PL/M-86 tries to shield the programmer

```
CRT$OUT$TASK PROCEDURE,
  DECLARE MESSAGE$LENGTH BYTE,
  DECLARE MESSAGE$TOKEN WORD,
  DECLARE RESPONSE$TOKEN WORD,
  DECLARE MESSAGE$EXCEPT$CODE WORD,
  DECLARE MESSAGE$SEGMENT$OFFSET WORD,
  MESSAGE$SEGMENT$BASE WORD;
  DECLARE MESSAGE$SEGMENT$PNTR POINTER AT
    (MESSAGE$SEGMENT$OFFSET);
  DECLARE MESSAGE$STRING$CHAR BASED MESSAGE$SEGMENT$PNTR BYTE.

  CALL RQ$RESUME$TASK (INIT$TASK$TOKEN, @MESSAGE$EXCEPT$CODE),
  DO FOREVER;
    MESSAGE$TOKEN=RQ$RECEIVE$MESSAGE (CRT$MAILBOX$TOKEN, OFFFFH,
    @RESPONSE$TOKEN, @MESSAGE$EXCEPT$CODE);
    MESSAGE$SEGMENT$OFFSET=0;
    MESSAGE$SEGMENT$BASE=MESSAGE$TOKEN;
    MESSAGE$LENGTH=MESSAGE$STRING$CHAR;
    DO MESSAGE$SEGMENT$OFFSET=1 TO MESSAGE$LENGTH;
      CALL PROTECTED$CRT$OUT (MESSAGE$STRING$CHAR);
    END;
  CALL RQ$DELETE$SEGMENT (MESSAGE$TOKEN, @MESSAGE$EXCEPT$CODE);
  END. /* OF FOREVER-LOOP */
END CRT$OUT$TASK;
```

Example 10. Task to Transmit Messages to the CRT

from the details, yet at times the two parts must be manipulated separately (for instance, to access data in an OSP segment knowing only the segment token/base value).

To get around this, these examples assign a pair of word variables to the same address as a PL/M pointer variable. Each representation is then an alias for the other. To determine the base or offset value of an item of data, load the pointer variable with a pointer to the item and then reference the appropriate field of the overlaid pair of word variables. To "build" an arbitrary pointer, assign computed values to the base and offset fields and then access the data item via the composite pointer.

Exercise 13: PL/M 86 does not have built-in functions to separate the high and low-order words of a pointer variable. Does this seem to be a weakness in the language? Bear in mind that the machine representation for pointers varies depending on which programming model is specified at compilation time. When the "small" model is selected, the compilers take advantage of a 16-bit pointer representation for faster and more compact code.

Console Command Interpreter

If a system has a console keyboard, it's probably used to accept and interpret operator commands. For this demonstration system, the lowest priority of all tasks is a simple-minded routine which polls the USART until a character has been received, and immediately echoes it by calling—you guessed it!—PROTECTED\$CRT\$OUTPUT. Thus, the keyboard is "alive"; it responds immediately to keystrokes, so the operator can type whatever nonsense he desires while everything else is going on.

Ten of the keys (digits 0 through 9), invoke special commands which illustrate interactions between the

multiple tasks. Commands 0 and 1 print out the time and status messages; the rest suspend and resume various tasks, as shown by Table 2. The code for COMMAND\$TASK appears in Example 11.

Initialization Task

Now that the application tasks have been written, we can write the initialization task.

All applications require a special type of task to initialize system variables and peripherals and create tasks and other objects used by the application. It, too, is written as a PL/M procedure, and can thus be divided conceptually into the same three phases.

Example 12 shows such a task for the demonstration system. The first thing INIT\$TASK does is determine the base address of the job data segment by assigning pointer DATA\$SEG\$PTR with its own address. Next it calls the RQ\$GET\$TASK\$TOKENS routine, which tells the task what token value the OSP assigned it at run time. It then initializes the system peripherals by creating the hardware initialization task discussed above; this code could have been integrated into INIT\$TASK itself just as easily. During its own "execution" phase, INIT\$TASK calls routines to create the OSP data structures shared by the application tasks: the REGION controlling access to the USART, and the MAILBOX repository for output messages. INIT\$TASK creates the application tasks themselves by calling RQ\$CREATE\$TASK.

Though not always required, it is common practice for the overall initialization task to suspend itself after creating each offspring, to let the newborn task get started. Under this convention, each offspring task must resume the initialization task by calling the

```

COMMAND$TASK PROCEDURE,
DECLARE CONSOLE$CHAR BYTE,
DECLARE COMMAND$EXCEPT$CODE WORD,

CALL RQ$RESUME$TASK (INIT$TASK$TOKEN, @COMMAND$EXCEPT$CODE),
DO FOREVER,
  CONSOLE$CHAR=C$IN AND 7FH,
  CALL PROTECTED$CRT$OUT (CONSOLE$CHAR),
  IF CONSOLE$CHAR=C'R
    THEN CALL PROTECTED$CRT$OUT (LF);
  IF (CONSOLE$CHAR >= '0') AND (CONSOLE$CHAR <= '9')
    THEN DO,
      CALL PROTECTED$CRT$OUT (CR),
      CALL PROTECTED$CRT$OUT (LF),
      DO CASE (CONSOLE$CHAR-'0'),
        CALL PRINT$TOD,
        CALL PRINT$STATUS,
        CALL RQ$SUSPEND$TASK (CRT$OUT$TASK$TOKEN,
          @COMMAND$EXCEPT$CODE),
        CALL RQ$RESUME$TASK (CRT$OUT$TASK$TOKEN,
          @COMMAND$EXCEPT$CODE),
        CALL RQ$DISABLE (AC$INTERRUPT$LEVEL,
          @COMMAND$EXCEPT$CODE),
        CALL RQ$ENABLE (AC$INTERRUPT$LEVEL,
          @COMMAND$EXCEPT$CODE),
        CALL RQ$SUSPEND$TASK (MOTOR$TASK$TOKEN,
          @COMMAND$EXCEPT$CODE),
        CALL RQ$RESUME$TASK (MOTOR$TASK$TOKEN,
          @COMMAND$EXCEPT$CODE),
        CALL RQ$SUSPEND$TASK (STATUS$TASK$TOKEN,
          @COMMAND$EXCEPT$CODE),
        CALL RQ$RESUME$TASK (STATUS$TASK$TOKEN,
          @COMMAND$EXCEPT$CODE),
        END, /* OF CASE-LIST */
      END, /* OF COMMAND PROCESSING */
    END,
  END,
END COMMAND$TASK,

```

Example 11. Task to Accept and Process Keyboard Commands

```

INIT$TASK PROCEDURE PUBLIC,
DECLARE INIT$EXCEPT$CODE WORD,

DATA$SEG$PTR=@INIT$TASK$TOKEN, /*LOAD DATA SEGMENT BASE*/
CRT$MAILBOX$TOKEN=RQ$CREATE$MAILBOX (0, @INIT$EXCEPT$CODE),
CRT$REGION$TOKEN=RQ$CREATE$REGION (0, @INIT$EXCEPT$CODE),
INIT$TASK$TOKEN=RQ$GET$TASK$TOKENS (0, @INIT$EXCEPT$CODE),
HARDWARE$INIT$TASK$TOKEN=RQ$CREATE$TASK
  (110, @HARDWARE$INIT$TASK, DATA$SEG$ADDR BASE, 0, 300,
  0, @INIT$EXCEPT$CODE),
CALL RQ$SUSPEND$TASK (0, @INIT$EXCEPT$CODE),
STATUS$TASK$TOKEN=RQ$CREATE$TASK (110, @STATUS$TASK,
  DATA$SEG$ADDR BASE, 0, 300, 0, @INIT$EXCEPT$CODE),
CALL RQ$SUSPEND$TASK (0, @INIT$EXCEPT$CODE),
MOTOR$TASK$TOKEN=RQ$CREATE$TASK (110, @MOTOR$TASK,
  DATA$SEG$ADDR BASE, 0, 300, 0, @INIT$EXCEPT$CODE),
CALL RQ$SUSPEND$TASK (0, @INIT$EXCEPT$CODE),
TIME$TASK$TOKEN=RQ$CREATE$TASK (120, @TIME$TASK,
  DATA$SEG$ADDR BASE, 0, 300, 0, @INIT$EXCEPT$CODE),
CALL RQ$SUSPEND$TASK (0, @INIT$EXCEPT$CODE),
CRT$OUT$TASK$TOKEN=RQ$CREATE$TASK (120, @CRT$OUT$TASK,
  DATA$SEG$ADDR BASE, 0, 300, 0, @INIT$EXCEPT$CODE),
CALL RQ$SUSPEND$TASK (0, @INIT$EXCEPT$CODE),
COMMAND$TASK$TOKEN=RQ$CREATE$TASK (130, @COMMAND$TASK,
  DATA$SEG$ADDR BASE, 0, 300, 0, @INIT$EXCEPT$CODE),
CALL RQ$SUSPEND$TASK (0, @INIT$EXCEPT$CODE),
CALL RQ$END$INIT$TASK,
CALL RQ$DELETE$TASK (0, @INIT$EXCEPT$CODE),
END INIT$TASK,

```

Example 12. Task to Initialize System Software

Table 2. Special Console Commands

Key	Function
0	Send Time-of-day message to CRT.
1	Send status update message to CRT.
2	Suspend CRT output task. The OSP will automatically save messages to the task in the CRT mailbox queue.
3	Resume CRT output task. Queued messages will be displayed.
4	Disable 60-Hz interrupt-driven time base. Time-of-day clock will stop.
5	Enable 60-Hz time base to resume clock execution.
6	Suspend motor control task. Motor will stop.
7	Resume motor control task. Note that if task was suspended 17 times, it must be resumed 17 times.
8	Suspend status polling task. Lights indicating system status will freeze in current state.
9	Resume status polling task.

RQ\$RESUME\$TASK routine when its own local initialization is complete. This convention is called synchronous initialization; its purpose is to ensure that each task is allowed to complete its own start-up phase before the next task is created. Otherwise, there's a risk that higher-priority tasks created later could start executing before earlier tasks were ready for them, with (at best) unpredictable results.

When all the tasks have been created, INIT\$TASK has served its purpose. It must then call RQ\$SEND\$INIT\$TASK. This short procedure (actually self-contained in an OSP Support Package interface library, not built into the 80130) tells the OSP that all the off-spring tasks have been created for a given job. At this point, INIT\$TASK could continue with non-initialization activities. The code for KEYBOARD\$TASK might have been implemented here, for example. Since this example has nothing more to do, INIT\$TASK deletes itself with a final call to RQ\$DELETE\$TASK.

Code Translation

That's all, folks. Mix together the above code fragments, declare literals and global variables, and compile until done (about four minutes). The source file name selected for this example is AP130.PLM. The compiler will produce two files: an annotated source listing (named AP130.LST) reproduced *in toto* in Appendix B, and a relocatable object file (AP130.OBJ) which will be used in the installation procedure discussed next.

High-Level Parameter Passing Conventions

Well-designed programs generally rely on subprograms ("procedures" in PL/M terminology) for often-repeated instruction sequences, or to perform machine-level operations within High-Level Language programs. PL/M-86 and other Intel high-level languages use a standard set of conventions to pass parameters and results between procedures; assembly language programmers are advised to adhere to these conventions for software compatibility.

Before calling a subroutine or function, input parameters must be pushed sequentially onto the stack, in the order (left-to-right) they appear in the procedure parameter list. When eight-bit parameters are pushed, the high-order byte associated with them is undefined. Thirty-two-bit pointer values are pushed in two steps, offset word before base word. The stack "grows" down, so the left-most parameter will have highest-numbered address.

Functions which return a byte or word value (i.e., typed procedures) do so in the CPU AL or AX registers. Pointers are returned through the ES:AX register pair. The *PL/M Programming Manual* explains these conventions more fully.

One way to see how an assembly language routine would interface with PL/M is to first write a dummy PL/M procedure using the same parameter sequence as the desired assembly language routine. Compile this procedure with the compiler CODE switch set. The listing will then include the appropriate assembly language instruction sequence, and may be followed as a pattern for the final routine.

SOFTWARE CONFIGURATIONS & INTEGRATION

When the application code has been written and compiled, the hardest part of program development is over. Before the code may be executed, though, the OSP must be told various things about the system hardware environment, desired software options, application job characteristics, and so forth.

This information is conveyed during a multi-phase sequence of steps collectively called the Configuration process. Though the process is somewhat lengthy and time-consuming, it is also very "mechanical"; the person doing the work does not need to understand any of the application code or even know what it does. Normally, configuration would be performed by a technician or a single member of the programming team, aided by appropriate SUBMIT command files. This chapter shows the full configuration and installation process for the demonstration system. For more details, refer to the *OSP User's Manual*.

The three phases of the configuration are:

1. Generating, linking, and locating OSP support code required for the EPROM immediately above the 80130 address space;
2. Linking and locating the object file for the application job developed in Section IV;
3. Creating, linking, and locating a short module (called the Root Job) which initializes the OSP and application jobs when system is reset.

Finally, of course, the absolute code resulting from each phase must be programmed into EPROMs or loaded into a test system before it can be executed.

Before starting, though, it is beneficial to draw up a memory map for host system hardware, to determine what sections of memory are available. This map will be filled in as each module is linked and located.

The prototype system memory space has two areas of interest: addresses 00000H through 01FFFFH contain RAM, while 0FC000H through 0FFFFFFH contain EPROM. Since the CPU uses the first 1K bytes of RAM for the CPU interrupt pointers, and the last 16 bytes for the restart sequence, these areas should be recorded on the map. For reference purposes, Figure 11 also indicates that addresses 0F8000H through 0FBFFFH enable the 80130 firmware. All this is shown in Figure 11.

Generating the OSP Support Code

The OSP support code "customizes" the OSP firmware for a particular hardware environment, initializes the system, and supports extended software capabilities.

To define the hardware environment, the user creates a source file which invokes a series of Intel-supplied macros. Parameters for these macros specify the 80130 I/O base address, SYSTICK interval (in system clock cycles), and how the interrupt request pins will be used.

For instance, the code example in Figure 12 defines the prototype system hardware. This source file must be assembled, linked with several libraries from the OSP support disk, and located to produce the actual OSP support code. Figure 13 shows the actual sequence of commands needed. The DATA starting address specified within the LOC86 parameter list (00400H) is the first free byte of system RAM (see Figure 11); the CODE address (0F8000H) is simply the 80130 firmware starting address.

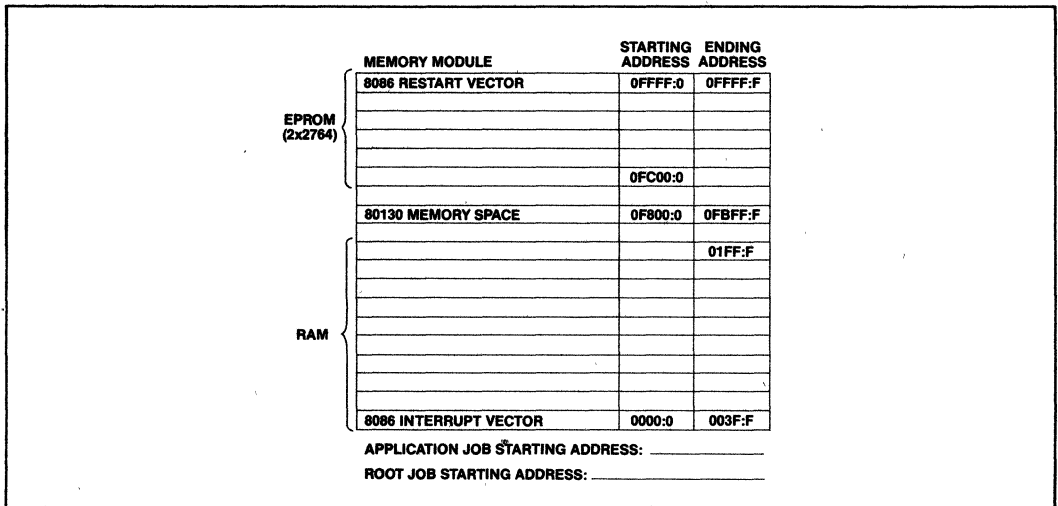


Figure 11. Example System Memory Map

```

*TITLE(80130 DEVICE CONFIGURATION TABLE)
NAMEDEVCF
*INCLUDE( F1 NDEVCF MAC)
%MASTER_PIC(80130,2000H,0,0)
,SLAVE_PIC( SLAVE_TYPE, BASE_PORT, EDGE_VS_LEVEL, MASTER_LEVEL )
%TIMER(80130,2008H,28H,12500)
,NDP_SUPPORT( ENCODED_LEVEL )
END
    
```

Figure 12. 80130 Device Configuration Table

```

FO ASM86 F1 SUP130 A86 PRINT( F1 SUP130 LST) ERRORPRINT %
MACRO(B0) PAGEWIDTH(132)

FO LINK86 %
F1 OSX LIB(OSXB6, OSXCNF), %
F1 NUC1 LIB(NBEGIN), %
F1 ODEVCF OBJ, %
F1 OSX LIB, %
F1 NUC1 LIB, %
F1 OSX LIB, %
F1 NUC2 LIB, %
F1 OSX LIB, %
F1 NUC4 LIB, %
F1 OSX LIB, %
F1 NURSLV LIB, %
F1 OSX LIB %
TO F1 SUP130 LNK MAP PRINT( F1 SUP130 MP1) NAME(MINIMAL_B0130)

FO LOC86 %
F1 SUP130 LNK TO F1 SUP130 MAP PRINT( F1 SUP130 MP2) SC(3) &
SEGSIZE(STACK(0)) %
ADDRESSES(CLASSES(CODE(0FB000H), DATA(004000H))) %
ORDER(CLASSES(DATA, STACK)) %
OBJECTCONTROLS(NOLINES, NOCOMMENTS, NOSYMBOLS)

```

Figure 13. Support Code Configuration Commands

A reliable and relatively straightforward way to perform this step is to create a file containing the exact command sequence shown in Figure 13 and execute this file using the SUBMIT utility program. Of course, the example assumes SUBMIT, ASM86, LINK86, and LOC86 are all on drive :F0:, and that the various libraries have been copied from the support disk to drive :F1:.

(An alternate, support-code configuration scheme lets the user modify the OSP software characteristics in special situations. A programmer working with iRMX 86, for instance, may wish to augment the OSP firmware to support all the iRMX Nucleus primitives. This would be done by editing and assembling file 0TABLE.A86 to select from a menu of software options, and modifying the linkage step slightly to include one of the iRMX 86 libraries. The OSP built-in features are more than sufficient for the purposes of this note, though, so only the first approach is illustrated.)

Appendix D reproduces the Locate map file produced during this phase. Near the end of file SUP130.MP2 is a table of memory usage, showing that the last bytes of RAM and ROM consumed are 00A6: FH and 0FC61: FH, respectively. Update Figure 11 with this information. (The final version of the demonstration-system memory map appears in Appendix C.) This phase needn't be repeated unless the system hardware characteristics change.

Application Code Configuration

After compiling the application job, it must be linked with a library of interface routines from the support diskette, and located within available memory. Use RPIFC.LIB or RPIFL.LIB, depending on whether the job was compiled with the Compact or Large software model. Figure 14 is a command sequence file suggested for this purpose. Again, the starting addresses specified for LOC86 are taken from the system memory map.

Whenever the support code is reconfigured, check SUP130.MP2 to see if its memory needs have changed. If so, the application-job-configuration command file will need to be edited. This is still a lot simpler (not to mention more reliable) than retyping the whole sequence each time application jobs are revised. Readers familiar with the capabilities of the SUBMIT program may prefer to represent these variables by parameters, such that they may be easily specified each time the command file is invoked.

As in the first phase, examine the locate map ("AP130.MP2", reproduced in Appendix E) after the application code has been configured and update the memory map. Also, note the segment and offset values assigned to the initialization task. These will be needed later.

Creating the Root Job

By now, all of the code needed to execute the application program has been prepared and is ready to run—except it has no way to get it started! The OSP hardware and system data structures must be initialized before INIT\$TASK can be created. A short module called the Root Job performs this function.

The process closely resembles the one which produced the OSP support code. First, determine various system characteristics. Then create a file defining these characteristics as macro input parameters. Finally, assemble, link, and locate the file to produce the final code.

Figure 15 is the Root Job source file for the demonstration system, dubbed RJB130.A86. It consists of just five macro calls. The %JOB macro defines certain characteristics of the application job; for a full description see the *OSP User's Manual*. One of these parameters is the initialization-task starting address (noted in the last step), which will likely change with each iteration of the application software.

The two %SAB macros define "System Address Blocks"—sections of the overall memory space which the OSP should not consider "free space." Note that the first invocation blocks off the RAM addresses consumed so far in the memory map, plus an extra 140H bytes reserved for the Root Job initialization stack.

```

,
, SUBMIT FILE TO LINK APPLICATION JOB TO INTERFACE LIBRARY
, AND LOCATE RESULTING OUTPUT.
, REVISED 10/23/81 - JHW
LINK86 F1 AP130.DBJ, :F1 RPIFC LIB TO F1 AP130 LNK      &
MAP PRINT (:F1, AP130 MP1)

LOC86 F1 AP130.LNK TO F1 AP130                          &
ORDER (CLASSES (DATA, STACK, MEMORY))                  &
SEGSIZE (STACK (0))                                     &
ADDRESSES (CLASSES (DATA (00A70H),                     &
                     CODE (0FC620H)))                   &
MAP PRINT (:F1, AP130 MP2)                               &
OBJECTCONTROLS (NOLINES, NOCOMMENTS, NOPUBLICS, NOSYMBOLS)

OHB6 F1 AP130 TO F1 AP130 HB6

COPY .F1, AP130 MP1 TO LP

COPY F1 AP130 MP2 TO LP

```

Figure 14. Job Configuration Commands

```

, SOURCE PROGRAM DEFINING CHARACTERISTICS OF ROOT JOB FOR
, AP-130 DEMONSTRATION PROGRAM (JHW - 10/25/81)
%INCLUDE (:F1, CTABLE MAC)

%SAB (0, 00C0, U)
%SAB (0200, FFFF, U)
%JOB (0, 0C0H, 100H, OFFFFH, OFFFFH, 1, 0, 0, 1, 0, 100, 0FC62, 06B5, 0, 0, 0, 200H, 0)
%DSX (OFB000H, N)
%SYSTEM (FB00, 0, 4, N, N, 1)

END

```

Figure 15. Root Job Configuration File

(After completing this phase, examine RJB130.MP2 to confirm that 140H is the correct number.) The second %SAB invocation excludes addresses 02000H through 0FFFFFFH, all of which is non-RAM, either EPROM, 80130 firmware, or non-existent. The %SYSTEM macro defines system-wide software parameters.

Figure 16 is a command file to translate, link, and locate the root job. Once again, the LOC86 parameters come from Figure 11. The listings produced during this phase are reproduced in Appendix F. The final memory map appears in Appendix C.

EPROM Programming

We are now ready to program EPROMs with the program modules linked and located above. Intel's Universal PROM Programmer (UPP) and a control program called the Universal Prom Mapper (UPM) will be used in this step. Particular commands to the UPM will vary with program size, memory location, and EPROM type, but the general sequence should resemble that shown here.

The first step is to invoke UPM and initialize the programming system, following a command sequence similar to that in Figure 17. The example system incorporates two 2764 devices, so 16K bytes of memory buffer are cleared.

Next, all the final code modules produced above (e.g., SUP130, AP130, and RJB130) must be loaded into the

UPM memory buffer. The three COMB commands in Figure 18 perform this function.

When the final system is reset, execution must branch into the root job initialization sequence. When the absolute code modules have finished loading, manually patch a jump instruction into the buffer area corresponding to the CPU reset vector. The opcode for the 8086 or 8088 intersegment jump is OEAH; the instruction's address field must contain the address assigned to label RQ\$START\$ADDRESS (read from the root job locate map), the 16-bit segment offset (low byte first) followed by the segment base address (ditto). The UPM CHANGE command should be used to make this patch, as illustrated in Figure 19.

The UPM memory buffer now contains a complete image of the code needed for the system EPROMs. Up until now, all software-related steps—source code preparation, translation, linking and locating—have been the same for 8086- or 8088-based systems. At this point, however, the software installation procedures diverge slightly.

Recall that the 8086 fetches instructions 16 bits at a time, from coordinated pairs of EPROMs. One contains only even-numbered program bytes, the other, odd. To separate the linear UPM buffer into high- and low-order bytes for iAPX 86/30 designs, use the UPM STRIP command as shown in Figure 20.

Now "burn" the EPROMs with the PROGRAM command in Figure 21.

```

, LINK AND LOCATE THE iRMX 86 ROOT JOB
,
, MODIFIED FOR TWO-DRIVE OPERATION
, REVISED 10/25 - JHW
,
ASMB6 F1 RJB130 AB6 MACRO(75)
,
LINK86                                     &
      F1 croot lib(root),                 &
      F1 RJB130 obj,                      &
      F1 croot lib                        &
      TO F1 RJB130 lnk                    &
      MAP PRINT( F1 RJB130 mp1)
,
LOC86 F1 RJB130 lnk                       &
      TO F1 RJB130                       &
      MAP PRINT( F1 RJB130 mp2) &
      OC(noll, nopl, nocm, nosb) &
      PC(noll, pl, nocm, nosb) &
      SEGSIZE(stack(0)) &
      ORDER(classes(data, stack, memory)) &
      ADDRESSES(classes(code(0FD180H), &
                        data(00AD0H)))
,
OHB6 F1 RJB130 TO F1 RJB130 HB6
,
COPY F1 RJB130 LST TO LP
,
COPY F1 RJB130 MP1 TO LP
,
COPY F1 RJB130 MP2 TO LP
,

```

Figure 16. Root Job Configuration Commands

```
fill from 0 to 3fffh with 0fffh
```

Figure 17. UPM Initialization Sequence

```
read 86hex file : f1: sup130. h86 from 0 to 3fffh start 0fc000h
read 86hex file : f1: ap130. h86 from 0 to 3fffh start 0fc000h
read 86hex file : f1 : rjb130. h86 from 0 to 3fffh start 0fc000h
```

Figure 18. UPM Commands to Load Hex Files

```
change 3ff0h=0eah, 11h, 00h, 18h, 0fdh
```

Figure 19. UPM Command to Patch Restart Vector

```
strip low from 0 to 3fffh into 4000h
strip hi from 0 to 3fffh into 6000h
```

Figure 20. UPM Commands to Strip High and Low Bytes

```
program from 4000h to 5fffh start 0
program from 6000h to 7fffh start 0
exit
```

Figure 21. UPM Commands to Program EPROMs

To save some trouble, the UPM invocation and all commands except the manual patch can be combined into a SUBMIT command file. Replace the CHANGE command with a control-E character so the operator can adjust the starting address for the iteration. Also place control-Es before each PROGRAM step to give the operator time to socket the next memory device.

SUMMARY

The development of the 80130 marks a major milestone in the evolution of microcomputer systems. For the first time, a single VLSI device integrates the hardware facilities and operating system firmware needed by real-time multitasking applications. The 80130 offers the system hardware designer the advantages of higher integration—reduced device count, smaller boards, greater reliability—along with faster design cycles and optimal system performance.

The 80130 gives the software engineer built-in support for 35 standard operating system primitives. Application problems may now be solved at a higher level than

before. It is now possible for concurrent tasks to be dispatched, memory segments allocated, and messages relayed through mailboxes nearly as easily as subroutines, dynamic variables, and I/O ports were used in the past. In effect, Jobs, Tasks, Segments, Mailboxes, and Regions become new OSP data types, manipulated entirely by firmware in the 80130.

Yet despite standardizing these functions, the OSP does not restrict the user's flexibility. The device can accommodate a variety of hardware environments, and both the hardware and software capabilities are desired.

ACKNOWLEDGEMENTS

The author would like to thank Peter Pederson for designing and implementing the demonstration system breadboard discussed in this note, Pam Johnson for her assistance in typing the manuscript, and Hal Kop, Lionel Smith, George Alexy, Chuck McMinn, and Sandy Wharton for their help in reviewing the drafts and providing many thoughtful comments and criticisms.

APPENDIX A
EXAMPLE SYSTEM SCHEMATICS

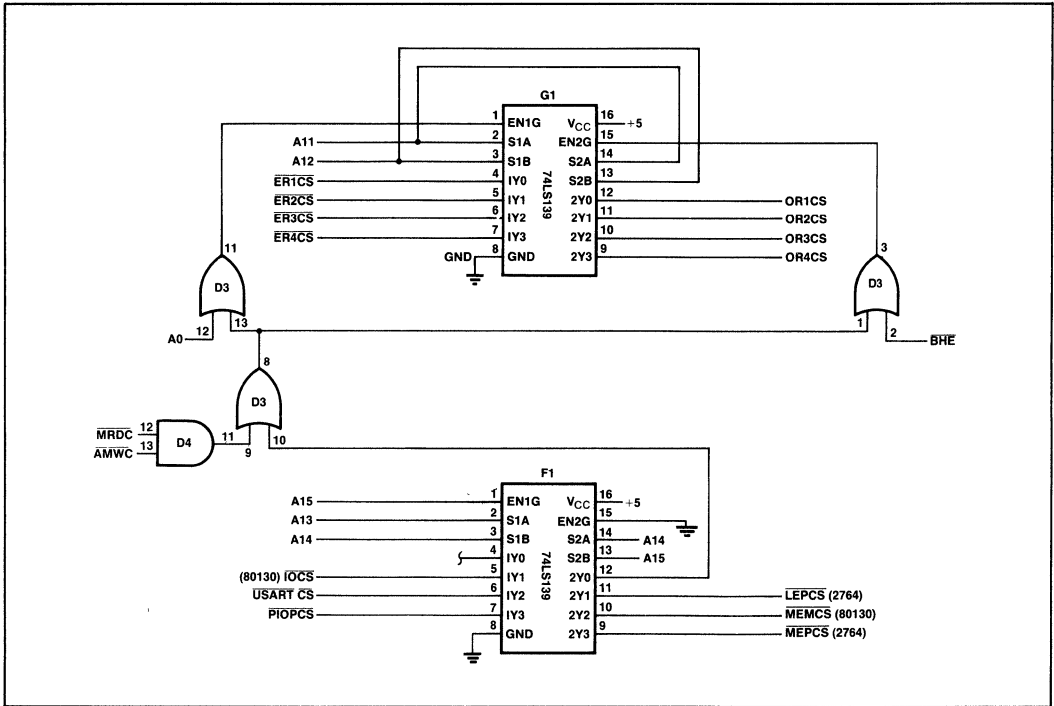


Figure A-1. Example System Schematics (continued)

APPENDIX B
SOURCE CODE LISTINGS

ISIS-II PL/M-86 V2 0 COMPILATION OF MODULE DEMO130
 OBJECT MODULE PLACED IN F1:AP130.OBJ
 COMPILER INVOKED BY: PLM86 .F1:AP130.PLM DATE(12/21)

```

$DEBUG COMPACT ROM TITLE('AP-130 APPENDIX B - 12/21/81')

1      DEMO$130.  DO,

      /* SYSTEM-WIDE LITERAL DECLARATIONS:  */

2      I      DECLARE FOREVER LITERALLY 'WHILE 01H',

      /* I/O PORT DEFINITIONS.  */

3      I      DECLARE CHAR$51 LITERALLY '4000H',
      CMD$51 LITERALLY '4002H',
      STAT$51 LITERALLY '4002H',

4      I      DECLARE PPI$A LITERALLY '6001H',
      PPI$B LITERALLY '6003H',
      PPI$C LITERALLY '6005H',
      PPI$CMD LITERALLY '6007H',
      PPI$STAT LITERALLY '6007H',

5      I      DECLARE TIMER$CMD LITERALLY '200EH',
      BAUD$TIMER LITERALLY '200CH';

6      I      DECLARE AC$INTERRUPT$LEVEL LITERALLY '00111000B',

7      I      DECLARE CR LITERALLY 'ODH',
      LF LITERALLY 'OAH',
      BEL LITERALLY 'O7H',

8      I      DECLARE ASCII$CODE (16) BYTE DATA ('0123456789ABCDEF');

$EJECT

=      $INCLUDE (.F1 NUCLUS.EXT)
=      $SAVE NOLIST
=      $INCLUDE (.F1 NEXCEP.LIT)
=      $save nolist

      /* GLOBAL VARIABLE DECLARATIONS.  */

299    I      DECLARE DATA$SEG$PTR POINTER,
      DATA$SEG$ADDR STRUCTURE (OFFSET WORD,BASE WORD)
      AT (@DATA$SEG$PTR),

300    I      DECLARE HARDWARE$INIT$TASK$TOKEN WORD,
      STATUS$TASK$TOKEN WORD,
      MOTOR$TASK$TOKEN WORD,
      TIME$TASK$TOKEN WORD,
      AC$HANDLER$TOKEN WORD,
      CRT$OUT$TASK$TOKEN WORD,
      COMMAND$TASK$TOKEN WORD,
      INIT$TASK$TOKEN WORD,

301    I      DECLARE CRT$MAILBOX$TOKEN WORD,
      CRT$REGION$TOKEN WORD;

```

```
$EJECT,
```

```
/* CODE EXAMPLE 2 SIMPLE CRT INPUT AND OUTPUT ROUTINES. */
```

```
302 1 C$OUT: PROCEDURE (CHAR);
303 2 DECLARE CHAR BYTE;
304 2 DO WHILE (INPUT(STAT$51) AND 01H)=0;
      /* NOTHING */
305 3 END;
306 2 OUTPUT(CHAR$51)=CHAR;
307 2 END C$OUT;
```

```
308 1 C$IN: PROCEDURE BYTE;
309 2 DO WHILE (INPUT(STAT$51) AND 02H)=0;
      /* NOTHING */
310 3 END;
311 2 RETURN INPUT(CHAR$51);
312 2 END C$IN;
```

```
$EJECT
```

```
/* CODE EXAMPLE 1. HARDWARE INITIALIZATION TASK. */
```

```
313 1 HARDWARE$INIT$TASK. PROCEDURE;
314 2 DECLARE HARD$INIT$EXCEPT$CODE WORD;
315 2 DECLARE PARAM$51 (*) BYTE DATA (40H, 8DH, 00H, 40H, 4EH, 27H);
316 2 DECLARE PARAM$51$INDEX BYTE;
317 2 DECLARE SIGN$ON$MESSAGE (*) BYTE DATA
      (CR, LF, 'IAPX 86/30 HARDWARE INITIALIZED', CR, LF);
318 2 DECLARE SIGN$ON$INDEX BYTE;

319 2 OUTPUT(PPI$CMD)=90H,
320 2 OUTPUT(TIMER$CMD)=0B6H;
321 2 OUTPUT(BAUD$TIMER)=33; /*GENERATES 9600 BAUD FROM 5 MHZ*/
322 2 OUTPUT(BAUD$TIMER)=0;
323 2 DO PARAM$51$INDEX=0 TO (SIZE(PARAM$51)-1);
324 3 OUTPUT(CMD$51)=PARAM$51(PARAM$51$INDEX);
325 3 END; /*OF USART INITIALIZATION DO-LOOP*/
326 2 DO SIGN$ON$INDEX=0 TO (SIZE(SIGN$ON$MESSAGE)-1);
327 3 CALL C$OUT(SIGN$ON$MESSAGE(SIGN$ON$INDEX));
328 3 END; /*OF SIGN-ON DO-LOOP*/
329 2 CALL RG$RESUME$TASK(INIT$TASK$TOKEN, @HARD$INIT$EXCEPT$CODE);
330 2 CALL RG$DELETE$TASK(0, @HARD$INIT$EXCEPT$CODE);
331 2 END HARDWARE$INIT$TASK;
```

```
$EJECT
```

```
/* CODE EXAMPLE 3. STATUS POLLING AND REPORTING TASK. */
```

```
332 1 STATUS$TASK PROCEDURE;
333 2 DECLARE STATUS$COUNTER BYTE;
334 2 DECLARE STATUS$EXCEPT$CODE WORD;

335 2 STATUS$COUNTER=0;
336 2 CALL RG$RESUME$TASK(INIT$TASK$TOKEN, @STATUS$EXCEPT$CODE);
337 2 DO FOREVER;
338 3 OUTPUT(PPI$B)=INPUT(PPI$A) XOR STATUS$COUNTER;
339 3 STATUS$COUNTER=STATUS$COUNTER+1;
340 3 CALL RG$SLEEP(100, @STATUS$EXCEPT$CODE);
341 3 END;
342 2 END STATUS$TASK;
```

\$EJECT

/* CODE EXAMPLE 4. STEPPER MOTOR CONTROL TASK. */

```

343 1  DECLARE CW$STEP$DELAY BYTE,
      CCW$STEP$DELAY BYTE,
      CW$PAUSE$DELAY BYTE,
      CCW$PAUSE$DELAY BYTE;

344 1  MOTOR$TASK: PROCEDURE;
345 2  DECLARE MOTOR$EXCEPT$CODE WORD;
346 2  DECLARE MOTOR$POSITION BYTE,
      MOTOR$PHASE BYTE;
347 2  DECLARE PHASE$CODE (4) BYTE
      DATA (0000101B, 0000110B, 00001010B, 00001001B);

348 2  CW$STEP$DELAY=50; /*INITIAL STEP DELAYS = 1/4 SECOND*/
349 2  CCW$STEP$DELAY=50;
350 2  CW$PAUSE$DELAY=200, /*PAUSES AFTER ROTATION = 1 SECOND*/
351 2  CCW$PAUSE$DELAY=200;
352 2  CALL RQ$RESUME$TASK(INIT$TASK$TOKEN, @MOTOR$EXCEPT$CODE);
353 2  DO FOREVER;
354 3  DO MOTOR$POSITION=0 TO 100;
355 4  MOTOR$PHASE=MOTOR$POSITION AND 0003H;
356 4  OUTPUT(PPI$C)=PHASE$CODE(MOTOR$PHASE);
357 4  CALL RQ$SLEEP(CW$STEP$DELAY, @MOTOR$EXCEPT$CODE);
358 4  END;
359 3  CALL RQ$SLEEP(CW$PAUSE$DELAY, @MOTOR$EXCEPT$CODE);
360 3  DO MOTOR$POSITION=0 TO 100;
361 4  MOTOR$PHASE=(100-MOTOR$POSITION) AND 0003H;
362 4  OUTPUT(PPI$C)=PHASE$CODE(MOTOR$PHASE);
363 4  CALL RQ$SLEEP(CW$STEP$DELAY, @MOTOR$EXCEPT$CODE);
364 4  END;
365 3  CALL RQ$SLEEP(CCW$PAUSE$DELAY, @MOTOR$EXCEPT$CODE);
366 3  END;
367 2  END MOTOR$TASK;

```

\$EJECT

/* CODE EXAMPLE 5. INTERRUPT HANDLER TO TRACK 60 HZ INPUT. */

```

368 1  DECLARE AC$CYCLE$COUNT BYTE;

369 1  AC$HANDLER. PROCEDURE INTERRUPT 59; /*VECTOR FOR 80130 INT3*/
370 2  DECLARE AC$EXCEPT$CODE WORD;

371 2  CALL RQ$ENTER$INTERRUPT(AC$INTERRUPT$LEVEL, @AC$EXCEPT$CODE);
372 2  AC$CYCLE$COUNT=AC$CYCLE$COUNT+1;
373 2  IF AC$CYCLE$COUNT >= 60
      THEN DO;
375 3  AC$CYCLE$COUNT=0;
376 3  CALL RQ$SIGNAL$INTERRUPT(AC$INTERRUPT$LEVEL,
      @AC$EXCEPT$CODE);
377 3  END;
378 2  ELSE CALL RQ$EXIT$INTERRUPT(AC$INTERRUPT$LEVEL,
      @AC$EXCEPT$CODE);
379 2  END AC$HANDLER;

```

```
$EJECT
```

```
/* CODE EXAMPLE 7. PROTECTED CRT OUTPUT SUBROUTINE. */
```

```
380 1  PROTECTED$CRT$OUT. PROCEDURE (CHAR) REENTRANT;
381 2  DECLARE CHAR BYTE;
382 2  DECLARE CRT$EXCEPT$CODE WORD;
383 2  CALL RQ$RECEIVE$CONTROL(CRT$REGION$TOKEN, @CRT$EXCEPT$CODE);
384 2  DO WHILE (INPUT(STAT$51) AND 01H)=0;
      /* NOTHING */
385 3  END;
386 2  OUTPUT(CHAR$51)=CHAR;
387 2  CALL RQ$SEND$CONTROL(@CRT$EXCEPT$CODE);
388 2  /END PROTECTED$CRT$OUT,
```

```
$EJECT
```

```
/* CODE EXAMPLE 6. INTERRUPT TASK TO MONITOR CLOCK TIME. */
```

```
389 1  DECLARE SECOND$COUNT BYTE,
      MINUTE$COUNT BYTE,
      HOUR$COUNT BYTE;

390 1  TIME$TASK: PROCEDURE;
391 2  DECLARE TIME$EXCEPT$CODE WORD;

392 2  AC$CYCLE$COUNT=0;
393 2  CALL RQ$SET$INTERRUPT(AC$INTERRUPT$LEVEL, 01H,
      INTERRUPT$PTR(AC$HANDLER), DATA$SEQ$ADDR. BASE,
      @TIME$EXCEPT$CODE);
394 2  CALL RQ$RESUME$TASK(INIT$TASK$TOKEN, @TIME$EXCEPT$CODE);
395 2  DO HOUR$COUNT=0 TO 23;
396 3  DO MINUTE$COUNT=0 TO 59;
397 4  DO SECOND$COUNT=0 TO 59;
398 5  CALL RQ$WAIT$INTERRUPT(AC$INTERRUPT$LEVEL,
      @TIME$EXCEPT$CODE);
399 5  IF SECOND$COUNT MOD 5 = 0
      THEN CALL PROTECTED$CRT$OUT(BEL);
401 5  END; /* SECOND LOOP */
402 4  END; /* MINUTE LOOP */
403 3  END; /* HOUR LOOP */
404 2  CALL RQ$RESET$INTERRUPT(AC$INTERRUPT$LEVEL,
      @TIME$EXCEPT$CODE);
405 2  CALL RQ$DELETE$TASK(0, @TIME$EXCEPT$CODE);
406 2  END TIME$TASK;
```

*EJECT

/* CODE EXAMPLE 8. SUBROUTINE TO CREATE TIME-OF-DAY MESSAGE. */

```

407 1 PRINT$TOD. PROCEDURE;
408 2 DECLARE TOD$MESSAGE$TOKEN WORD;
409 2 DECLARE TOD$EXCEPT$CODE WORD;
410 2 DECLARE TOD$SEGMENT$OFFSET WORD,
    TOD$SEGMENT$BASE WORD;
411 2 DECLARE TOD$SEGMENT$PNTR POINTER AT (@TOD$SEGMENT$OFFSET);
412 2 DECLARE TOD$TEMPLATE (28) BYTE
    DATA (27, 'THE TIME IS NOW hh:mm:ss.', CR, LF);
413 2 DECLARE TOD$STRING BASED TOD$SEGMENT$PNTR (28) BYTE;
414 2 DECLARE TOD$STRING$INDEX BYTE;

415 2 TOD$MESSAGE$TOKEN=RQ$CREATE$SEGMENT(28, @TOD$EXCEPT$CODE);
416 2 TOD$SEGMENT$BASE=TOD$MESSAGE$TOKEN;
417 2 TOD$SEGMENT$OFFSET=0;
418 2 DO TOD$STRING$INDEX=0 TO 27;
419 3 TOD$STRING(TOD$STRING$INDEX)=
    TOD$TEMPLATE(TOD$STRING$INDEX);
420 3 END;
421 2 TOD$STRING(17)=ASCII$CODE(HOUR$COUNT/10);
422 2 TOD$STRING(18)=ASCII$CODE(HOUR$COUNT MOD 10);
423 2 TOD$STRING(20)=ASCII$CODE(MINUTE$COUNT/10);
424 2 TOD$STRING(21)=ASCII$CODE(MINUTE$COUNT MOD 10);
425 2 TOD$STRING(23)=ASCII$CODE(SECOND$COUNT/10);
426 2 TOD$STRING(24)=ASCII$CODE(SECOND$COUNT MOD 10);
427 2 CALL RQ$SEND$MESSAGE(CRT$MAILBOX$TOKEN,
    TOD$MESSAGE$TOKEN, 0, @TOD$EXCEPT$CODE);
428 2 RETURN;
429 2 END PRINT$TOD;

```

*EJECT

/* CODE EXAMPLE 9. SUBROUTINE TO CREATE SWITCH STATUS MESSAGE. */

```

430 1 PRINT$STATUS: PROCEDURE,
431 2 DECLARE STATUS$MESSAGE$TOKEN WORD;
432 2 DECLARE STATUS$EXCEPT$CODE WORD;
433 2 DECLARE STATUS$SEGMENT$OFFSET WORD,
    STATUS$SEGMENT$BASE WORD;
434 2 DECLARE STATUS$SEGMENT$PNTR POINTER
    AT (@STATUS$SEGMENT$OFFSET);
435 2 DECLARE STATUS$TEMPLATE (40) BYTE DATA
    (39, 'THE SWITCHES ARE NOW SET TO . . . . .B', CR, LF);
436 2 DECLARE STATUS$STRING BASED STATUS$SEGMENT$PNTR (40) BYTE;
437 2 DECLARE STATUS$STRING$INDEX BYTE;
438 2 DECLARE BIT$PATTERN BYTE;

439 2 STATUS$MESSAGE$TOKEN=RQ$CREATE$SEGMENT(40,
    @STATUS$EXCEPT$CODE);
440 2 STATUS$SEGMENT$BASE=STATUS$MESSAGE$TOKEN;
441 2 STATUS$SEGMENT$OFFSET=0;
442 2 DO STATUS$STRING$INDEX=0 TO 39;
443 3 STATUS$STRING(STATUS$STRING$INDEX)=
    STATUS$TEMPLATE(STATUS$STRING$INDEX);
444 3 END;
445 2 BIT$PATTERN=INPUT(PPI$A);
446 2 DO STATUS$STRING$INDEX=29 TO 36;
447 3 STATUS$STRING(STATUS$STRING$INDEX)=
    ASCII$CODE(BIT$PATTERN AND 01H);
448 3 BIT$PATTERN=ROR(BIT$PATTERN, 1);
449 3 END;
450 2 CALL RQ$SEND$MESSAGE(CRT$MAILBOX$TOKEN,
    STATUS$MESSAGE$TOKEN, 0, @STATUS$EXCEPT$CODE);
451 2 END PRINT$STATUS.

```

#EJECT

/* CODE EXAMPLE 10. TASK TO RECEIVE MESSAGES AND TRANSMIT THEM TO CRT. */

```

452 1 CRT$OUT$TASK. PROCEDURE;
453 2   DECLARE MESSAGE$LENGTH BYTE;
454 2   DECLARE MESSAGE$TOKEN WORD;
455 2   DECLARE RESPONSE$TOKEN WORD;
456 2   DECLARE MESSAGE$EXCEPT$CODE WORD;
457 2   DECLARE MESSAGE$SEGMENT$OFFSET WORD;
      MESSAGE$SEGMENT$BASE WORD;
458 2   DECLARE MESSAGE$SEGMENT$PNTR POINTER AT (@MESSAGE$SEGMENT$OFFSET);
459 2   DECLARE MESSAGE$STRING$CHAR BASED MESSAGE$SEGMENT$PNTR BYTE;

460 2   CALL RQ$RESUME$TASK(INIT$TASK$TOKEN, @MESSAGE$EXCEPT$CODE);
461 2   DO FOREVER;
462 3     MESSAGE$TOKEN=RQ$RECEIVE$MESSAGE(CRT$MAILBOX$TOKEN, OFFFHH,
      @RESPONSE$TOKEN, @MESSAGE$EXCEPT$CODE);
463 3     MESSAGE$SEGMENT$OFFSET=0;
464 3     MESSAGE$SEGMENT$BASE=MESSAGE$TOKEN;
465 3     MESSAGE$LENGTH=MESSAGE$STRING$CHAR;
466 3     DO MESSAGE$SEGMENT$OFFSET=1 TO MESSAGE$LENGTH;
467 4       CALL PROTECTED$CRT$OUT(MESSAGE$STRING$CHAR);
468 4     END;
469 3     CALL RQ$DELETE$SEGMENT(MESSAGE$TOKEN, @MESSAGE$EXCEPT$CODE);
470 3     END; /* OF FOREVER-LOOP */
471 2   END CRT$OUT$TASK;

```

#EJECT

/* CODE EXAMPLE 11. TASK TO POLL KEYBOARD AND PROCESS COMMANDS. */

```

472 1 COMMAND$TASK. PROCEDURE;
473 2   DECLARE CONSOLE$CHAR BYTE;
474 2   DECLARE COMMAND$EXCEPT$CODE WORD;

475 2   CALL RQ$RESUME$TASK(INIT$TASK$TOKEN, @COMMAND$EXCEPT$CODE);
476 2   DO FOREVER;
477 3     CONSOLE$CHAR=C$IN AND 7FH;
478 3     CALL PROTECTED$CRT$OUT(CONSOLE$CHAR);
479 3     IF CONSOLE$CHAR=CR
480 4       THEN CALL PROTECTED$CRT$OUT(LF);
481 3     IF (CONSOLE$CHAR >= '0') AND (CONSOLE$CHAR <= '9')
482 4       THEN DO;
483 4         CALL PROTECTED$CRT$OUT(CR);
484 4         CALL PROTECTED$CRT$OUT(LF);
485 4         DO CASE (CONSOLE$CHAR-'0');
486 5           CALL PRINT$TOD;
487 5           CALL PRINT$STATUS;
488 5           CALL RQ$SUSPEND$TASK(CRT$OUT$TASK$TOKEN,
      @COMMAND$EXCEPT$CODE);
489 5           CALL RQ$RESUME$TASK(CRT$OUT$TASK$TOKEN,
      @COMMAND$EXCEPT$CODE);
490 5           CALL RQ$DISABLE(AC$INTERRUPT$LEVEL,
      @COMMAND$EXCEPT$CODE);
491 5           CALL RQ$ENABLE(AC$INTERRUPT$LEVEL,
      @COMMAND$EXCEPT$CODE);
492 5           CALL RQ$SUSPEND$TASK(MOTOR$TASK$TOKEN,
      @COMMAND$EXCEPT$CODE);
493 5           CALL RQ$RESUME$TASK(MOTOR$TASK$TOKEN,
      @COMMAND$EXCEPT$CODE);
494 5           CALL RQ$SUSPEND$TASK(STATUS$TASK$TOKEN,
      @COMMAND$EXCEPT$CODE);
495 5           CALL RQ$RESUME$TASK(STATUS$TASK$TOKEN,
      @COMMAND$EXCEPT$CODE);
496 5         END; /* OF CASE-LIST */
497 4       END; /* OF COMMAND PROCESSING */
498 3     END;
499 2   END COMMAND$TASK;

```


#EJECT

/* CODE EXAMPLE 12. TASK TO INITIALIZE OSP SOFTWARE. */

```

500 1  INIT$TASK: PROCEDURE PUBLIC;
501 2  DECLARE INIT$EXCEPT$CODE WORD;

502 2      DATA$SEG$PTR=@INIT$TASK$TOKEN; /*LOAD DATA SEGMENT BASE*/
503 2      CRT$MAILBOX$TOKEN=RQ$CREATE$MAILBOX(0,@INIT$EXCEPT$CODE);
504 2      CRT$REGION$TOKEN=RQ$CREATE$REGION(0,@INIT$EXCEPT$CODE);
505 2      INIT$TASK$TOKEN=RQ$GET$TASK$TOKENS(0,@INIT$EXCEPT$CODE);
506 2      HARDWARE$INIT$TASK$TOKEN=RQ$CREATE$TASK
          (110,@HARDWARE$INIT$TASK,DATA$SEG$ADDR.BASE,0,300,
          0,@INIT$EXCEPT$CODE);
507 2      CALL RQ$SUSPEND$TASK(0,@INIT$EXCEPT$CODE);
508 2      STATUS$TASK$TOKEN=RQ$CREATE$TASK(110,@STATUS$TASK,
          DATA$SEG$ADDR.BASE,0,300,0,@INIT$EXCEPT$CODE);
509 2      CALL RQ$SUSPEND$TASK(0,@INIT$EXCEPT$CODE);
510 2      MOTOR$TASK$TOKEN=RQ$CREATE$TASK(110,@MOTOR$TASK,
          DATA$SEG$ADDR.BASE,0,300,0,@INIT$EXCEPT$CODE);
511 2      CALL RQ$SUSPEND$TASK(0,@INIT$EXCEPT$CODE);
512 2      TIME$TASK$TOKEN=RQ$CREATE$TASK(120,@TIME$TASK,
          DATA$SEG$ADDR.BASE,0,300,0,@INIT$EXCEPT$CODE);
513 2      CALL RQ$SUSPEND$TASK(0,@INIT$EXCEPT$CODE);
514 2      CRT$OUT$TASK$TOKEN=RQ$CREATE$TASK(120,@CRT$OUT$TASK,
          DATA$SEG$ADDR.BASE,0,300,0,@INIT$EXCEPT$CODE);
515 2      CALL RQ$SUSPEND$TASK(0,@INIT$EXCEPT$CODE);
516 2      COMMAND$TASK$TOKEN=RQ$CREATE$TASK(130,@COMMAND$TASK,
          DATA$SEG$ADDR.BASE,0,300,0,@INIT$EXCEPT$CODE);
517 2      CALL RQ$SUSPEND$TASK(0,@INIT$EXCEPT$CODE);
518 2      CALL RQ$END$INIT$TASK;
519 2      CALL RQ$DELETE$TASK(0,@INIT$EXCEPT$CODE);
520 2      END INIT$TASK;

521 1      END DEMO$130,

```

MODULE INFORMATION

```

CODE AREA SIZE      = 084CH   2124D
CONSTANT AREA SIZE = 0000H    0D
VARIABLE AREA SIZE = 0052H   82D
MAXIMUM STACK SIZE = 0026H   38D
848 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-86 COMPILATION

**APPENDIX C
SYSTEM MEMORY MAP**

EXAMPLE SYSTEM MEMORY MAP

	MEMORY MODULE	STARTING ADDRESS	ENDING ADDRESS
EPROM (2x2764)	8086 RESTART VECTOR	0FFFF:0	0FFF:F
	ROOT JOB CODE AREA	0FD18:0	0FD36:6
	APPLICATION JOB CODE AREA	0FC62:0	0FD17:B
	OSP SUPPORT CODE AREA	0FC00:0	0FC61:F
	80130 MEMORY SPACE	0F800:0	0FBFF:F
	(FREE SYSTEM RAM)	00C0:0	01FF:F
RAM	ROOT JOB DATA AREA	00AD:0	00BF:F
	APPLICATION JOB DATA AREA	00A7:0	00AC:1
	OSP SUPPORT DATA AREA	0040:0	00A6:F
	8086 INTERRUPT VECTOR	0000:0	003F:F

INITIALIZATION TASK STARTING ADDRESS: FC62:06B5

ROOT JOB STARTING ADDRESS: FD18:0011

APPENDIX D
SUPPORT CODE LOCATE MAP

ISIS-II MCS-86 LOCATER, V1 2 INVOKED BY
 FO LOCB4
 F1 SUP130 LNK TO F1 SUP130 MAP PRINT (F1 SUP130 MP2) SC(3) &
 SEGSIZE(STACK(0)) &
 ADDRESSES(CLASSES(CODE(OF8000H),DATA(00400H))) &
 ORDER(CLASSES(DATA,STACK)) &
 OBJECTCONTROLS(NOLINES,NOCOMMENTS,NOSYMBOLS)
 WARNING 26 DECREASING SIZE OF SEGMENT
 SEGMENT STACK

SYMBOL TABLE OF MODULE MINIMAL_BO130
 READ FROM FILE F1 SUP130 LNK
 WRITTEN TO FILE F1 SUP130

BASE	OFFSET	TYPE	SYMBOL	BASE	OFFSET	TYPE	SYMBOL	BASE	OFFSET	TYPE	SYMBOL
0040H	0000H	PUB	INTERRUPTTASKVEC	0040H	0120H	PUB	DEFAULT_HANDLER	0040H	0144H	PUB	READYLISTROOT
0040H	0148H	PUB	INTTERRORENTRY	0040H	014CH	PUB	SYSTEMEXCEPTIONH -ANDLERPTR	0040H	0150H	PUB	DELETIONTASKTOKEN -N
0040H	0152H	PUB	EXTENSIONLISTROO -T	0040H	0154H	PUB	DELETION_OBJECT_ -BASE	0040H	0156H	PUB	SYSTEMPOOLTOKEN
0040H	0156H	PUB	ROOTJOBTOKEN	0040H	015AH	PUB	MINTRANSIZE	0040H	015CH	PUB	LAST_NDP_TASK
0040H	015EH	PUB	NDP_INTERRUPT_LE -VEL_VAR	0040H	0160H	PUB	PARAM_VALIDATION -VECTOR	0040H	0162H	PUB	REGION_FLAGS
0040H	0164H	PUB	TASK_WAITING_FL -GS	0040H	0166H	PUB	REGION_TOKEN_TAB -LE	0040H	0166H	PUB	SIGNAL_Q_INDEX
0040H	0178H	PUB	SIGNAL_Q	0040H	01EBH	PUB	KERNEL_FLAG	0040H	01E9H	PUB	ACTIVATE_SIGNAL_ -Q
0040H	01EAH	PUB	FILLCHAR	0040H	01EBH	PUB	NUM_SLAVES	0040H	01ECH	PUB	OLD_SLAVE_NUM
0040H	01EDH	PUB	INTMASK	0040H	01F6H	PUB	DISABLEMASK	0040H	01FFH	PUB	LEVEL_SET_TABLE
0040H	0208H	PUB	INR_PORT	0040H	021AH	PUB	EDI_PORT	0040H	022CH	PUB	ISR_PORT
0040H	023EH	PUB	PIC_INFO	0040H	0247H	PUB	CLOCK_SPEC_EOI	0040H	0248H	PUB	CLOCK_ON
0040H	0249H	PUB	CLOCK_OFF	0040H	024AH	PUB	CLOCK_LEVEL	0040H	0250H	PUB	END_OF_DATA
F800H	45CCH	PUB	NDP_INTERRUPT_LE -VEL	F800H	45C2H	PUB	VALIDATE_PARAMS_ -BODY_DUMMY	F800H	4542H	PUB	GETDESCRTOKEN
F800H	4556H	PUB	GETDESCRPOINTER	F800H	4567H	PUB	GETPOINTER	F800H	453DH	PUB	SCANMEMORY
F800H	453BH	PUB	OVERFLOW	F800H	4533H	PUB	NENTRY_BODY	F800H	452EH	PUB	KBSPEND
F800H	4529H	PUB	INITIALIZE	F800H	4524H	PUB	KENABLELEVELNS	F800H	451FH	PUB	KENABLELEVEL
F800H	451AH	PUB	KCREATEREGIONNS	F800H	4515H	PUB	KCREATEDOBJECTNS	F800H	4510H	PUB	KCREATEDOBJECT
F800H	4508H	PUB	INITNDP	F800H	4506H	PUB	INITIALIZE	F800H	4501H	PUB	FINISHINITIALIZA -TION
F800H	44FC	PUB	EDI_ROUTINE	F800H	44F7H	PUB	DIVIDEBYZERO	F800H	44F2H	PUB	DECODE_LEVEL
F800H	44EDH	PUB	COMMON_ERROR	F800H	44EBH	PUB	CLOCKENTRY_BODY	F800H	44E3H	PUB	ARRAYBOUNDS
F800H	44D0H	PUB	SYSTEMEXCEPTIONH -ANDLER	F800H	4472H	PUB	INITIALIZE_TIMER	F800H	43AEH	PUB	INITIALIZE_PICS
F800H	435CH	PUB	INIT_INTERNAL_RE -GIONS	F800H	434EH	PUB	NDP_INTERRUPT_HA -NDLER	F800H	433FH	PUB	CLOCKENTRY
F800H	4336H	PUB	NENTRY	F800H	40FEH	PUB	INITIALIZENUCLEU -S	F800H	40B6H	PUB	RQWAITINTERRUPT_ -BODY
F800H	40B1H	PUB	RGSIGNALINTERRUPT -T_BODY	F800H	40ACH	PUB	RQGETLEVEL_BODY	F800H	40A7H	PUB	RGEXITINTERRUPT_ -BODY
F800H	40A2H	PUB	RGENTERINTERRUPT -BODY	F800H	409DH	PUB	RQDISABLE_BODY	F800H	4094H	PUB	RQWAITINTERRUPT
F800H	408AH	PUB	RGSIGNALINTERRUPT -T	F800H	4080H	PUB	RQGETLEVEL	F800H	4076H	PUB	RGENTERINTERRUPT
F800H	406CH	PUB	RGEXITINTERRUPT	F800H	4062H	PUB	RQDISABLE	F800H	405DH	PUB	NUNLOCK_DELETION -OBJECT
F800H	4058H	PUB	NUNLOCKNS	F800H	4053H	PUB	NUNLOCK	F800H	404EH	PUB	NOPEN_DELETION_D -BJECT
F800H	4049H	PUB	NOPENNS	F800H	4044H	PUB	NOPEN	F800H	403FH	PUB	NLOCK_DELETION_D -BJECT
F800H	403AH	PUB	NLOCKNS	F800H	4035H	PUB	NLOCK	F800H	4030H	PUB	NCLOSE_DELETION_ -OBJECT
F800H	402BH	PUB	NCLOSENS	F800H	4026H	PUB	NCLOSE	F800H	4021H	PUB	DELETERUNNINGTAS -K
F800H	401CH	PUB	DELETEDOBJECT	F800H	400AH	PUB	COPYRIGHT	F800H	4000H	PUB	NBEGIN
F800H	4000H	PUB	INIT_NUCLEUS_JUM -P	FC5DH	0004H	PUB	IMR_START	FC5CH	000EH	PUB	
FC5CH	000FH	PUB	INIT_CMD1	FC5CH	0010H	PUB	INIT_CMDS_MASTER	FC5CH	0011H	PUB	
FC5CH	0012H	PUB	INIT_CMD4_MASTER	FC61H	000EH	PUB	SLAVE_TABLE	FC61H	0003H	PUB	
FC61H	0005H	PUB	CLOCK_O_PDRT	FC61H	0007H	PUB	CLOCK_COUNT	FC61H	000AH	PUB	
FC61H	0008H	PUB	C_CLOCK_SPEC_EDI	FC61H	000CH	PUB	C_CLOCK_ON	FC61H	0009H	PUB	
F800H	4576H	PUB	LEVEL7_HANDLER	F800H	4574H	PUB	PARAM_VALIDATION -PATH				

MEMORY MAP OF MODULE MINIMAL_BO130
 READ FROM FILE F1 SUP130 LNK
 WRITTEN TO FILE F1 SUP130

SEGMENT MAP

START	STOP	LENGTH	ALIGN	NAME	CLASS
00000H	003FFH	0400H	A	(ABSOLUTE)	
00400H	009EFH	05F0H	W	DATA	DATA
009F0H	009FFH	0010H	G	INTVEC_REG_SEG	DATA
00A00H	00A0FH	0010H	G	EXT_REG_SEG	DATA
00A10H	00A1FH	0010H	G	JOB_REG_SEG	DATA
00A20H	00A2FH	0010H	G	SEM_REG_SEG	DATA
00A30H	00A3FH	0010H	G	MAIL_REG_SEG	DATA
00A40H	00A4FH	0010H	G	OD_REG_SEG	DATA
00A50H	00A5FH	0010H	G	PDCL_REG_SEG	DATA

00A60H	00A6FH	0010H	G	DELETION_REG_S -EQ	DATA	← LAST RAM BYTE USED
00A70H	00A70H	0000H	W	STACK	STACK	
00A70H	00A70H	0000H	G	??SEG		
F8000H	FC5CDH	45CEH	W	CODE	CODE	
FC5CEH	FC5D2H	0005H	W	PIC_CNF_SEG	CODE	
FC5D4H	FC5E5H	0012H	W	_IMR_PORT	CODE	
FC5E6H	FC5F7H	0012H	W	_EOI_PORT	CODE	
FC5F8H	FC609H	0012H	W	_ISR_READ_PORT	CODE	
FC60AH	FC612H	0009H	B	_PIC_INF0	CODE	
FC613H	FC61CH	000AH	B	TIMER_CNF_SEG	CODE	
FC61EH	FC61EH	0000H	W	CSEG	CODE	
FC61EH	FC61FH	0002H	W	SLAVE_SEG	CODE	← LAST EPROM BYTE USED
FC620H	FC620H	0000H	W	MEMORY	MEMORY	

GROUP MAP

ADDRESS	GROUP OR SEGMENT NAME
00400H	DGROUP
	DATA
	INTVEC_REG_SEG
	EXT_REG_SEG
	JOB_REG_SEG
	SEM_REG_SEG
	MAIL_REG_SEG
	DD_REG_SEG
	POOL_REG_SEG
	DELETION_REG_SEG
F8000H	CGROUP
	CODE
	PIC_CNF_SEG
	_IMR_PORT
	_EOI_PORT
	_ISR_READ_PORT
	_PIC_INF0
	TIMER_CNF_SEG
	CSEG
	SLAVE_SEG

**APPENDIX E
APPLICATION JOB LOCATE MAP**

AP-130

```

ISIS-II MCS-86 LOCATER, V1 2 INVOKED BY
LOC86 F1 AP130 LNK TO F1 AP130
ORDER (CLASSES(DATA, STACK, MEMORY))
SEGSIZE (STACK (0))
ADDRESSES (CLASSES (DATA (00A70H),
                    CODE (0FC620H)))
MAP PRINT ( F1 AP130 MP2)
OBJECTCONTROLS (NOLINES, NOCOMMENTS, NOPUBLICS, NOSYMBOLS)
WARNING 26 DECREASING SIZE OF SEGMENT
SEGMENT STACK
    
```

SYMBOL TABLE OF MODULE DEMO130
 READ FROM FILE F1 AP130 LNK
 WRITTEN TO FILE F1 AP130

BASE	OFFSET	TYPE	SYMBOL	BASE	OFFSET	TYPE	SYMBOL
FC62H	0B3AH	PUB	RQENDINITASK	FC62H	0B1CH	PUB	RG_N_C_RETURN_40
FC62H	0B00H	PUB	RG_N_C_RETURN_20	FC62H	0AE4H	PUB	RG_N_C_RETURN_14
FC62H	0AC9H	PUB	RG_N_C_RETURN_12	FC62H	0AACH	PUB	RG_N_C_RETURN_10
FC62H	0A90H	PUB	RG_N_C_RETURN_8	FC62H	0A74H	PUB	RG_N_C_RETURN_6
FC62H	0A5BH	PUB	RG_N_C_RETURN_4	FC62H	0A3EH	PUB	RGERROR
FC62H	0A2BH	PUB	RQGETLEVEL	FC62H	0A0EH	PUB	RGSIGNALEXCEPTIO
FC62H	09F0H	PUB	RQWAITINTERRUPT	FC62H	09DAH	PUB	-N RGSIGNALINTERRUPT
FC62H	09D4H	PUB	RQDELETESSEMAPHOR	FC62H	09CEH	PUB	-T RQDELETEMAILBOX
FC62H	09BBH	PUB	RQEXITINTERRUPT	FC62H	09B2H	PUB	RGUNCATALOGOBJEC
FC62H	09AC	PUB	RQSENDUNITS	FC62H	09A6H	PUB	-T RQSPENDTASK
FC62H	09A0H	PUB	RQSETPRIORITY	FC62H	099AH	PUB	RQSETPOLMIN
FC62H	0994H	PUB	RQSETOSEXTENSION	FC62H	098EH	PUB	RQSENDMESSAGE
FC62H	098BH	PUB	RQSLEEP	FC62H	0982H	PUB	RQSETINTERRUPT
FC62H	097CH	PUB	RQSETEXCEPTIONHA	FC62H	0976H	PUB	RQSENDCONTROL
FC62H	0970H	PUB	-NDLER RQRECEIVEUNITS	FC62H	096AH	PUB	RQRESUMETASK
FC62H	093BH	PUB	RQRECEIVESSAOE	FC62H	0932H	PUB	RQRESETINTERRUPT
FC62H	092CH	PUB	RQRECEIVECONTROL	FC62H	0926H	PUB	RQGFSPRING
FC62H	0920H	PUB	RGLDOKUOBJECT	FC62H	091AH	PUB	RQINSPECTCOMPOSI
FC62H	0914H	PUB	RQGETTASKTOKENS	FC62H	090EH	PUB	-E RQGETTYPE
FC62H	090BH	PUB	RQGETSIZE	FC62H	0902H	PUB	RQGETPRIORITY
FC62H	08FCH	PUB	RQGETPOOLATTRIB	FC62H	08F6H	PUB	RQSETEXCEPTIONHA
FC62H	08F0H	PUB	RQFORCEDELETE	FC62H	08EAH	PUB	-NDLER RGENABLE
FC62H	08D4H	PUB	RQENTERINTERRUPT	FC62H	08CEH	PUB	RGENABLEDELETION
FC62H	08CBH	PUB	RQDELETETASK	FC62H	08C2H	PUB	RQDELETESegment
FC62H	08ACH	PUB	RQDISABLE	FC62H	08A6H	PUB	RQDELETEREGION
FC62H	08A0H	PUB	RQDELETEJOB	FC62H	089AH	PUB	RQDELETEEXTENSIO
FC62H	0894H	PUB	RQDISABLEDELETDIO	FC62H	088EH	PUB	-N RQDELETECOMPOSIT
FC62H	088BH	PUB	-E RQCREATETASK	FC62H	0882H	PUB	RQCREATESSEMAPHOR
FC62H	087CH	PUB	RQCREATESEGMENT	FC62H	0876H	PUB	-E RQCREATEREGION
FC62H	0870H	PUB	RQCATALOGOBJECT	FC62H	086AH	PUB	RQCREATEMAILBOX
FC62H	0864H	PUB	RQCREATEJOB	FC62H	085EH	PUB	RQCREATEEXTENSIO
FC62H	085BH	PUB	RQCREATECOMPOSIT	FC62H	0852H	PUB	-N RQALTERCOMPOSITE
FC62H	084CH	PUB	-E RQACCEPTCONTROL	FC62H	06B5H	PUB	INITTASK
DEMO130			SYMBOLS AND LINES				
FD17H	000CH	SYM	MEMORY	FC62H	0000H	SYM	ASCII CODE
00A7H	0000H	SYM	DATABASEPTR	00A7H	0000H	SYM	DATABASEGADDR
00A7H	0004H	SYM	HARDWAREINITASK	00A7H	0006H	SYM	STATUSTASKTOKEN
00A7H	0008H	SYM	-TOKEN	00A7H	000AH	SYM	TIMETASKTOKEN
00A7H	000CH	SYM	MOTORTASKTOKEN	00A7H	000EH	SYM	CRTOUPTASKTOKEN
00A7H	0010H	SYM	ACHANDLERTOKEN	00A7H	0012H	SYM	INITTASKTOKEN
00A7H	0014H	SYM	COMMANDTASKTOKEN	00A7H	0016H	SYM	CRTRGIONTOKEN
FC62H	0018H	SYM	CRMAILBOXTOKEN	FC62H	001AH	SYM	CHAR
FC62H	001BH	SYM	OUT	FC62H	001BH	SYM	STATUS
FC62H	00A1H	SYM	CIN	FC62H	00B9H	SYM	HARDWAREINITASK
00A7H	001BH	SYM	HARDINITECEPTCO	FC62H	0010H	SYM	PARAM51
00A7H	0040H	SYM	-DE PARAM51INDEX	FC62H	0016H	SYM	SIGNONMESSAGE
00A7H	0041H	SYM	SIGNONINDEX	FC62H	013BH	SYM	STATUSTASK
00A7H	0042H	SYM	STATUSCOUNTER	00A7H	001AH	SYM	STATUSEXCEPTCODE
00A7H	0043H	SYM	CWSTEPDELAY	00A7H	0044H	SYM	CWSTEPDELAY
00A7H	0045H	SYM	CWPAUSEDELAY	00A7H	0046H	SYM	CCWPAUSEDELAY
FC62H	0172H	SYM	MOTORTASK	00A7H	001CH	SYM	MOTOREXCEPTCODE
00A7H	0047H	SYM	MOTORPOSITION	00A7H	0048H	SYM	MOTORPHASE
FC62H	0029H	SYM	PHASECODE	00A7H	0049H	SYM	ACCVCLECOUNT
FC62H	0256H	SYM	ACHANDLER	00A7H	001EH	SYM	ACEXCEPTCODE
FC62H	029CH	SYM	PROTECTEDCRTOUT	STACK	0006H	SYM	CHAR
STACK	0002H	SYM	CRTEXCEPTCODE	00A7H	004AH	SYM	SECONDCOUNT
00A7H	004BH	SYM	MINUTECOUNT	00A7H	004CH	SYM	HOURLCOUNT
FC62H	02CFH	SYM	TIMETASK	00A7H	0020H	SYM	TIMEXCEPTCODE
FC62H	03BBH	SYM	PRINTTOD	00A7H	0022H	SYM	TODMESSAGECTOKEN
00A7H	0024H	SYM	TODEXCEPTCODE	00A7H	0026H	SYM	TODSEGMENTOFFSET
00A7H	0028H	SYM	TODSEGMENTBASE	00A7H	0026H	SYM	TODSEGMENTPNTR
FC62H	003DH	SYM	TODTEMPLATE	00A7H	0026H	SYM	BAS
00A7H	004BH	SYM	TODSTRINGINDEX	FC62H	04B9H	SYM	TODSTRING
00A7H	002AH	SYM	STATUSMESSAGECTOK	00A7H	002CH	SYM	PRINTSTATUS
			-EN				STATUSEXCEPTCODE

00A7H	002EH	SYM	STATUSSEGMENTOFF	00A7H	0030H	SYM	STATUSSEGMENTBAS
			-SET				-E
00A7H	002EH	SYM	STATUSSEGMENTPNT	FC62H	0059H	SYM	STATUSSTEMPLATE
			-R				
00A7H	002EH	BAS	STATUSSTRING	00A7H	004EH	SYM	STATUSSTRINGINDE
							-X
00A7H	004FH	SYM	BITPATTERN	FC62H	052FH	SYM	CRTOUUTASK
00A7H	0050H	SYM	MESSAGELENGTH	00A7H	0032H	SYM	MESSAGETOKEN
00A7H	0034H	SYM	RESPONSETOKEN	00A7H	0036H	SYM	MESSAGEEXCEPTCOD
							-E
00A7H	003BH	SYM	MESSAGESEGMENTOF	00A7H	003AH	SYM	MESSAGESEGMENTBA
			-FSET				-SE
00A7H	003BH	SYM	MESSAGESEGMENTPN	00A7H	003BH	BAS	MESSAGESTRINGCHA
			-TR				-R
FC62H	05AFH	SYM	COMMANDTASK	00A7H	0051H	SYM	CONSOLECHAR
00A7H	003CH	SYM	COMMANDEXCEPTCOD	FC62H	06B5H	SYM	INITTASK
			-E				
00A7H	003EH	SYM	INITEXCEPTCODE	FC62H	0084H	LIN	302
FC62H	00B7H	LIN	304	FC62H	0093H	LIN	305
FC62H	0096H	LIN	306	FC62H	009DH	LIN	307
FC62H	00A1H	LIN	308	FC62H	00A4H	LIN	309
FC62H	00B0H	LIN	310	FC62H	00B3H	LIN	311
FC62H	00B9H	LIN	312	FC62H	00B9H	LIN	313
FC62H	00BCH	LIN	319	FC62H	00C2H	LIN	320
FC62H	00CBH	LIN	321	FC62H	00CEH	LIN	322
FC62H	00D1H	LIN	323	FC62H	00E4H	LIN	324
FC62H	00D7H	LIN	325	FC62H	00F8H	LIN	326
FC62H	010CH	LIN	327	FC62H	0116H	LIN	328
FC62H	011FH	LIN	329	FC62H	012CH	LIN	330
FC62H	0139H	LIN	331	FC62H	013BH	LIN	332
FC62H	013EH	LIN	335	FC62H	0143H	LIN	336
FC62H	0150H	LIN	337	FC62H	0150H	LIN	338
FC62H	015CH	LIN	339	FC62H	0160H	LIN	340
FC62H	016DH	LIN	341	FC62H	0170H	LIN	342
FC62H	0172H	LIN	344	FC62H	0175H	LIN	348
FC62H	017AH	LIN	349	FC62H	017FH	LIN	350
FC62H	0184H	LIN	351	FC62H	0189H	LIN	352
FC62H	0196H	LIN	353	FC62H	0196H	LIN	354
FC62H	01A5H	LIN	355	FC62H	01B0H	LIN	356
FC62H	01BDH	LIN	357	FC62H	01CDH	LIN	358
FC62H	01D6H	LIN	359	FC62H	01E6H	LIN	360
FC62H	01F5H	LIN	361	FC62H	0202H	LIN	362
FC62H	020FH	LIN	363	FC62H	021FH	LIN	364
FC62H	022BH	LIN	365	FC62H	023BH	LIN	366
FC62H	023BH	LIN	367	FC62H	0256H	LIN	369
FC62H	0259H	LIN	371	FC62H	0266H	LIN	372
FC62H	0270H	LIN	373	FC62H	027BH	LIN	375
FC62H	027DH	LIN	375	FC62H	029AH	LIN	377
FC62H	028DH	LIN	378	FC62H	029AH	LIN	379
FC62H	029CH	LIN	380	FC62H	02A0H	LIN	383
FC62H	02ACH	LIN	384	FC62H	028BH	LIN	385
FC62H	02BBH	LIN	386	FC62H	02C2H	LIN	387
FC62H	02CAH	LIN	388	FC62H	02CFH	LIN	390
FC62H	02D2H	LIN	392	FC62H	02D7H	LIN	393
FC62H	02F3H	LIN	394	FC62H	0300H	LIN	395
FC62H	030FH	LIN	396	FC62H	031EH	LIN	397
FC62H	032DH	LIN	398	FC62H	033AH	LIN	399
FC62H	034EH	LIN	400	FC62H	0354H	LIN	401
FC62H	035DH	LIN	402	FC62H	0366H	LIN	403
FC62H	036FH	LIN	404	FC62H	037CH	LIN	405
FC62H	03B9H	LIN	406	FC62H	038BH	LIN	407
FC62H	03BEH	LIN	415	FC62H	039FH	LIN	416
FC62H	03A7H	LIN	417	FC62H	03ADH	LIN	418
FC62H	03BEH	LIN	419	FC62H	03D0H	LIN	420
FC62H	03D9H	LIN	421	FC62H	03F5H	LIN	422
FC62H	040EH	LIN	423	FC62H	0427H	LIN	424
FC62H	0440H	LIN	425	FC62H	0459H	LIN	426
FC62H	0472H	LIN	427	FC62H	04B7H	LIN	428
FC62H	04B9H	LIN	429	FC62H	04B9H	LIN	430
FC62H	04BCH	LIN	439	FC62H	049DH	LIN	440
FC62H	04A5H	LIN	441	FC62H	04ABH	LIN	442
FC62H	04BCH	LIN	443	FC62H	04CEH	LIN	444
FC62H	04D7H	LIN	445	FC62H	04DFH	LIN	446
FC62H	04EEH	LIN	447	FC62H	050BH	LIN	448
FC62H	050FH	LIN	449	FC62H	051BH	LIN	450
FC62H	052DH	LIN	451	FC62H	052FH	LIN	452
FC62H	0532H	LIN	460	FC62H	053FH	LIN	461
FC62H	053FH	LIN	462	FC62H	055AH	LIN	463
FC62H	0560H	LIN	464	FC62H	056BH	LIN	465
FC62H	0573H	LIN	466	FC62H	058BH	LIN	467
FC62H	0592H	LIN	468	FC62H	059DH	LIN	469
FC62H	05AAH	LIN	470	FC62H	05ADH	LIN	471
FC62H	05AFH	LIN	472	FC62H	05B2H	LIN	475
FC62H	05BFH	LIN	476	FC62H	05BFH	LIN	477
FC62H	05C9H	LIN	478	FC62H	05D0H	LIN	479
FC62H	05DAH	LIN	480	FC62H	05E0H	LIN	481
FC62H	05F4H	LIN	483	FC62H	05FAH	LIN	484
FC62H	0600H	LIN	485	FC62H	0610H	LIN	486
FC62H	0616H	LIN	487	FC62H	061CH	LIN	488
FC62H	062CH	LIN	489	FC62H	063CH	LIN	490
FC62H	064CH	LIN	491	FC62H	065CH	LIN	492
FC62H	066CH	LIN	493	FC62H	067CH	LIN	494
FC62H	06BCH	LIN	495	FC62H	069CH	LIN	496
FC62H	06B0H	LIN	498	FC62H	06B3H	LIN	499
FC62H	06B5H	LIN	500	FC62H	06BBH	LIN	502

FC62H	06C4H	LIN	503	FC62H	06D5H	LIN	504
FC62H	06E6H	LIN	505	FC62H	06F5H	LIN	506
FC62H	071FH	LIN	507	FC62H	072CH	LIN	508
FC62H	0755H	LIN	509	FC62H	0762H	LIN	510
FC62H	07BBH	LIN	511	FC62H	0798H	LIN	512
FC62H	07C1H	LIN	513	FC62H	07CEH	LIN	514
FC62H	07F7H	LIN	515	FC62H	0804H	LIN	516
FC62H	082DH	LIN	517	FC62H	083AH	LIN	518
FC62H	083DH	LIN	519	FC62H	084AH	LIN	520
FC62H	0084H	LIN	521				

MEMORY MAP OF MODULE DEMO130
 READ FROM FILE F1.AP130.LNK
 WRITTEN TO FILE F1.AP130

SEGMENT MAP

START	STOP	LENGTH	ALIGN	NAME	CLASS
-------	------	--------	-------	------	-------

00A70H	00AC1H	0052H	W	DATA	DATA	← LAST DATA BYTE OF APPLICATION JOB
--------	--------	-------	---	------	------	-------------------------------------

00AC2H	00AC2H	0000H	W	STACK	STACK
00AD0H	00AD0H	0000H	G	??SEG	

FC620H	FD17BH	085CH	W	CODE	CODE	← LAST CODE BYTE OF APPLICATION JOB
--------	--------	-------	---	------	------	-------------------------------------

FD17CH	FD17CH	0000H	W	MEMORY	MEMORY
--------	--------	-------	---	--------	--------

GROUP MAP

ADDRESS	GROUP OR SEGMENT NAME
FC620H	CGR0UP
	CODE
00A70H	DGR0UP
	DATA

**APPENDIX F
ROOT JOB LOCATE MAP**

ISIS-II MCS-86 LOCATER, V1.2 INVOKED BY.

```
LOC86 .f1 RJB130 lnk          &
      TO :F1 RJB130          &
      MAP PRINT(.f1.RJB130 mp2) &
      GC(noli, nopl, nocm, nosb) &
      PC(noli, pl, nocm, nosb)  &
      SEGSIZE(stack(0))        &
      ORDER(classes(data, stack, memory)) &
      ADDRESSES(classes(code(0FD180H), &
                        data(0OAD0H))) &
```

WARNING 26 DECREASING SIZE OF SEGMENT
SEGMENT STACK

SYMBOL TABLE OF MODULE ROOT
READ FROM FILE F1 RJB130 LNK
WRITTEN TO FILE F1 RJB130

BASE	OFFSET	TYPE	SYMBOL	BASE	OFFSET	TYPE	SYMBOL
FD18H	0180H	PUB	NUC_INIT_ENTRY	FD18H	0184H	PUB	CODEDATA

FD18H	0011H	PUB	ROSTARTADDRESS	← ROOT JOB STARTING ADDRESS	FD18H	0010H	PUB	INTERROR
-------	-------	-----	----------------	-----------------------------	-------	-------	-----	----------

FD18H	0000H	PUB	CRASH	FD18H	002AH	PUB	ROROOTJOBVERSION
FD18H	0030H	PUB	RODRTASK	FD18H	010CH	PUB	SYSTEMSUICIDE
FD18H	0118H	PUB	RGCREATEJOB	FD18H	011EH	PUB	RGGETTASKTOKENS
FD18H	0124H	PUB	RGSPENDTASK	FD18H	012AH	PUB	RG_N_C_RETURN_6
FD18H	0146H	PUB	RG_N_C_RETURN_40	FD18H	0162H	PUB	RGERROR
0OADH	0000H	PUB	JOBNUMBER	0OADH	0002H	PUB	RODRTASKSTATUS

MEMORY MAP OF MODULE ROOT
READ FROM FILE F1 RJB130 LNK
WRITTEN TO FILE F1 RJB130

MODULE START ADDRESS PARAGRAPH = FD18H OFFSET = 0011H
SEGMENT MAP

START	STOP	LENGTH	ALIGN	NAME	CLASS
0OAD0H	0OAD3H	0004H	W	DATA	DATA

0OAD4H	00BFFH	012CH	W	INIT_STACK	STACK	← LAST DATA BYTE OF ROOT JOB
--------	--------	-------	---	------------	-------	------------------------------

0OC00H	0OC00H	0000H	W	STACK	STACK
0OC00H	0OC00H	0000H	G	??SEQ	
FD180H	FD339H	01BAH	W	CODE	CODE
FD33AH	FD345H	000CH	W	SAB_DESCRIPTOR	CODE
				-S	

FD346H	FD366H	0021H	W	U_V_DESCRIPTOR	CODE	← LAST CODE BYTE OF ROOT JOB
				-S		

FD368H	FD368H	0000H	W	MEMORY	MEMORY
--------	--------	-------	---	--------	--------

GROUP MAP

ADDRESS	GROUP OR SEGMENT NAME
0OAD0H	DGROUP
	DATA
FD180H	CGROUP
	CODE
	SAB_DESCRIPTOR
	U_V_DESCRIPTOR



January 1984

**Optimizing the iRMX™ 86
Operating System
Performance on System
86/310 and System 86/330**

**CATHERINE LUNDBERG
ISO APPLICATIONS MARKETING**

**Optimizing the iRMX™ 86
Operating System
Performance on System
86/310 and System
86/330**

CONTENTS

INTRODUCTION

**OVERVIEW OF THE iRMX™ 86
OPERATING SYSTEM**

PERFORMANCE TUNING

- The Size Of The Loader Buffers
- The Volume Granularity Of Devices
- The Number Of BIOS Buffers
- The Interleave Factor Of Devices

PERFORMANCE TESTS

- RSAT Maximum Transfer Rate
- iRMX™ 86 Operating System Generation
- iRMX™ 86 BIOS Generation
- COPY Test
- DIR Test
- Boot Disk Generation
- Boot Test
- BACKUP And RESTORE Test

EVALUATION SEQUENCE

**SYSTEM 86/330A PERFORMANCE
RESULTS**

**SYSTEM 86/310 PERFORMANCE
RESULTS**

CONCLUSION

**APPENDIX A: SYSTEM
CONFIGURATIONS**

**APPENDIX B: DEFINITION FILE FOR AN
OPTIMUM iRMX™ 86 OPERATING
SYSTEM**

INTRODUCTION

The Intel iRMX™ 86 Operating System is one of the most widely used real-time operating systems. Because it is intended for real-time applications it must respond immediately to the event that has the highest priority. Most users of the iRMX 86 Operating System are application program oriented and their programs use the real-time capabilities fully.

The Release 5 version of the iRMX 86 Operating System has the capability of supporting program development. This means that a system can be used for application code development, and later as the target system. Development costs can be decreased within a company, since there is no need to have separate systems for program development and for target systems. In addition, programs do not have to be downloaded from the development system to the target system.

Since there are many possible software configurations for mixes of real time applications and program development work, the purpose of this application note is to identify system configuration options which will improve the overall performance of the iRMX 86 Operating System from the Human Interface level, which is the level that the user sees while doing development work. The results of this application note could also be applicable to the user who is optimizing a target system configuration, since most of the parameters discussed in this study would affect the performance of the target system. Although the Release 5.0 operating system is faster and more optimized than previous versions of the iRMX 86 Operating System, further optimization has been found possible, especially when a specific hardware configuration is known.

OVERVIEW OF THE iRMX™ 86 OPERATING SYSTEM

The iRMX 86 Operating System is composed of layers. The layers in the iRMX 86 Release 5.0 Operating System are the Nucleus, the Basic I/O System, the Extended I/O System, the Application Loader, the Human Interface, and the Universal Development Interface (See Figure 1.) A layer provides a specific subset of operating system function. For instance, the Nucleus provides management of iRMX 86 objects such as tasks and jobs, the BIOS provides device independent I/O services, and the UDI provides the interface software to allow applications to be operating system independent.

Different layers are added according to the requirements of the application. Lower layers may be used without upper layers, but upper layers must have the lower layers as their groundwork. Using fewer layers of the operating system usually means that the application can execute faster. The UDI layer is required to run the utilities (such as PL/M 86, ASM86, and LINK86) since all the languages products are based on the UDI. The code produced by running these utilities may not need all the layers.

Most layers of the iRMX 86 Operating System have user configurable parameters. The values given to these parameters can be changed to allow the operating system to perform efficiently. Some parameters should be left at their default values, or the operating system will not function properly. For other parameters, the optimum value depends on the application, and the hardware configuration being used.

The System 86/310 and System 86/330A have different requirements because of their hardware implementations. The major difference is their Winchester and flexible disks. Parameters that affect disk performance have the most effect on the systems. This application brief deals with those parameters which can be changed to provide optimal values for disk performance.

PERFORMANCE TUNING

A number of parameters can affect the system performance. The parameters which are configurable in the iRMX 86 Operating System don't change the actual speed at which the processor works. The variables that can improve performance affect how fast information can be given to the processor, or how much time must be spent to maintain system integrity. There are two ways to change the values of these parameters.

One way is to reconfigure the operating system using the ICU. The ICU is an interactive configuration utility which uses a screen oriented list of parameters to allow changes in an operating system. While using the ICU, different values are given to particular parameters to change the system performance. The ICU also allows the

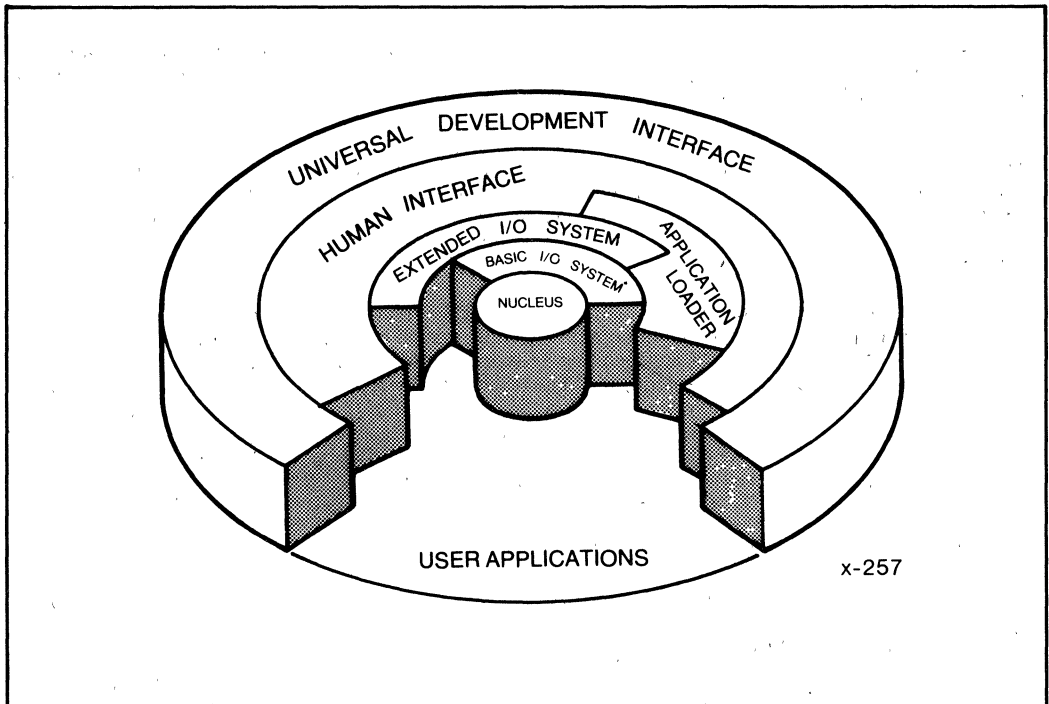


Figure 1 Model of the iRMX™86 Operating System

addition or deletion of individual layers of the operating system. The ICU creates a submit file from the information contained in the definition file. When this file is submitted, it generates a new operating system which uses the new values that were given to the parameters.

The second way parameters can be changed to affect system performance is to format the random access devices differently. Changing the interleave factor while using the FORMAT cusp makes a great difference in the disk I/O performance.

The Size of the Loader Buffers

The iRMX 86 Application Loader has two configurable buffer sizes. The Loader internal buffer is the Loader's working buffer for converting Object Module Formats (OMFs) to executable code. The default value for the Loader internal buffer is 400H, or 1K. Increasing this will not improve performance, but is sometimes necessary. OMFs are usually smaller than 1K, but the Fortran 86 compiler can sometimes generate records that are longer than 1K. Trying to load these records will cause an "ESREC_LENGTH" error, which can be eliminated by increasing the Loader internal buffer size. This parameter is found in the ICU screen titled "Application Loader". The variable is called "Internal Buffer Size (IBS)".

The Loader read buffer is used as a caching buffer when reading data from secondary storage. The Loader read buffer size influences the system performance. The Loader read buffer size is found in the ICU screen labeled "Application Loader". The parameter is labeled "Read Buffer Size (RBS)". The default value of the Loader read buffer size is 400H, or 1K. Increasing the Loader read buffer size will improve system performance.

The Volume Granularity of Devices

File fragmentation happens when a file is written into a space that is not large enough to hold the whole file. Portions of the file must then be stored somewhere else on the disk to finish writing the file. This means that the disk controller must seek to several places on the disk to read the whole file, or to write the file, and the speed of disk I/O will be decreased. There are three different characteristics the user can vary to control file fragmentation on a device: device granularity, volume granularity, and file granularity. Device granularity is both the minimum allocation size, the minimum transfer size and is usually the sector size of a device. Volume granularity is the minimum file allocation size for all files on the device. The volume granularity is a multiple of the device granularity. Finally, file granularity is the minimum allocation size for a particular file and is a multiple of the volume granularity. By increasing the volume granularity and file granularity the files on a device should be forced to be contiguous. If the Winchester is fragmented, the smallest contiguous chunk of data for any one file will be the volume granularity.

Volume granularity on Winchester drives does not appear to affect system performance, so that parameter can be left at its default value of 400H, or 1K. This parameter is found in the screen titled "Intel iSBC® 215/iSBX 218 Device-Unit Information". The field is labeled "Granularity".

The diskette media which is used most frequently on the System 86/330A is a double sided double density disk with a device granularity of 256 bytes (100H). The physical name of the first such device is "w added0" in the standard definition file. With a device granularity of 256 bytes, a volume granularity of 256 bytes is the most effective. Using a device with a device granularity of 1024 bytes will improve the performance of the diskette. The physical name of the first double sided double density diskette with 1 K byte device granularity is "w addedx0". The device tested for this application note was "w added0" the diskette with device granularity of 256 bytes. The diskette device used in the System 86/310 has the physical name of "w addedx0". It is a double sided, double density diskette with a device granularity of 512 bytes.

The Number of BIOS Buffers

The BIOS buffers are internal caching buffers which are used to hold data which is written to or read from secondary storage. The BIOS will only transfer the number of bytes that are a multiple of the device granularity. The device granularity is 1024 for the Winchester, and it varies for flexible disks. The most commonly used values are 512 for the System 86/310's diskette, and 256 for the System 86/330A's standard diskette.

Increasing the number of buffers for each device usually has a positive effect on disk performance, but only up to a certain point. Buffers must be updated, or flushed to the device, after a certain amount of time to ensure a reliable system. There are two parameters that control flushing buffers. They are called "Update Timeout," and "Fixed Update" or "Common Update Timeout". Update timeout is a variable for each device. A value of 64H (100) for update timeout means that when a device has not been accessed for 1 second, the buffers associated with that device will be written to the device. If the device is in constant use, this timeout limit may never be reached, especially if the value given to update timeout is large. For this reason, there is a second update parameter which can also be used.

The second parameter is called "Common Update Timeout" in the BIOS screen, and "Fixed Update" in the individual device screens. Common update timeout has one value for all devices in the operating system, but it is specified on an individual unit basis if it will be invoked. A value of 3E8H (1000) for common update timeout means that buffers associated with devices where common update timeout was invoked will be flushed at the end of 10 seconds. The buffers will be flushed if they have been used since the last time the system wide common update timeout happened. This ensures that even buffers for devices which have been continuously in use will be written to their device.

Increasing the amount of time between updating system buffers would improve performance, but it degrades reliability. If a system failure occurs, for instance, a power failure, the disks would have incorrect data on them if the buffers hadn't been written to disk. Using update timeout and common update timeout values that are appropriate reduces the damage caused by a system failure.

Different values of update timeout and common update timeout were not tested for the systems. These parameters were left at their default values for each device.

The time necessary to update the buffers causes a decrease in performance if there are too many buffers. Buffers also use memory when the device is connected. Careful usage of memory is necessary especially when the amount of memory available is limited, as is frequently the case for the target system.

Since diskettes are usually used only to transfer files between systems or for backups, the performance of the flexible disks is not as important as the performance of the Winchester device. Since increasing the number of buffers for a device increases the amount of memory required to attach the device, adding buffers for diskettes does not usually outweigh the need to conserve memory.

The BIOS number of buffers parameter is found in the ICU screen titled "Intel iSBC 215/iSBX 218 Device Unit Information". Each device has its own Device Unit Information screen. The field is labeled "Number of Buffers". The default number of buffers for Winchester devices is 4. The default number of buffers for flexible disk devices is usually 2.

The Interleave Factor of Devices

One of the parameters of the FORMAT command is "Interleave". If the consecutively-accessed sectors of a disk are staggered (that is, if they are not consecutive physical sectors), disk access time can decrease considerably. The reason for this decrease is that although a controller cannot read a sector and issue another read command in the time it takes for the next sector to be positioned under the head, the controller can perform this operation in less time than it takes for the disk to revolve once. Therefore, if the consecutively-accessed sectors are staggered correctly, the next accessed sector will be positioned under the read head just as the controller becomes ready to read it.

The amount of staggering is called the interleave factor. An interleave factor of two means that as the disk rotates, the controller consecutively accesses every second sector. Note that a properly set interleave factor also implies the number of disk rotations necessary to access all the sectors on a given track. An interleave factor of two implies that it takes two rotations of the disk to access all the sectors on a track.

The interleave factor is important when large transfers of consecutive data take place at speeds that approach the maximum transfer rate of the disk. Most information put on a disk will be stored in sectors that are consecutively accessed, if that is possible. If a disk has been heavily used so that few logically adjoining blocks are available, then the information will be stored in nonconsecutively accessed blocks, wherever there is space available. Naturally, this will slow down data transfer speed, since seeks must be done frequently to find where the next block of data is located. System performance will be best if the most frequently used utilities, programs and data are written onto the disk first, after formatting the disk with the optimum interleave factor.

There are three distinct cases where large amounts of data are transferred.

- 1) When the operating system is bootstrap loaded from disk
- 2) When the Application Loader is used to load an application program from disk
- 3) When programs are invoked that perform large transfers of consecutive data, such as the Human Interface COPY command

Each of these operations does a different amount of processing to the data which is being transferred. This means that the turnaround time between sector accesses is different.

The Bootstrap Loader instructs the disk controller to read one sector at a time. Thus, the turnaround time depends on the execution overhead of the Bootstrap Loader and is comparatively long. A large interleave factor is optimal for flexible disks that are used with the Bootstrap Loader. For hard disks however, the Bootstrap Loader has no effect on the turnaround time because revolution speed is so great that more than one disk revolution occurs between sector reads.

The Application Loader reads several sectors at a time into its internal read buffer. Then it takes a relatively long time to process the object records in this buffer. The ideal interleave factor here is one that optimizes for

the object record processing time between disk accesses. For flexible diskettes, this interleave factor is somewhat smaller than that for the Bootstrap Loader. However, the Application Loader is not affected by the interleave factor on hard disks.

Applications which transfer large amounts of consecutive data (such as the COPY command) can initiate data transfers involving many sequential sectors. Thus, the controller accesses sectors on a given track as fast as possible. Here, the ideal interleave factor is one that optimizes for the turnaround speed of the disk controller.

The ideal interleave factor depends heavily on the application. However, because the revolution speed of hard disks is so high, they should be formatted with interleave factors that are optimized for the turnaround speed of the disk controller.

It is more important to match the interleave factor to the application with diskettes. They are usually smaller devices and are usually used for one major type of access. Flexible disks are much more sensitive to varying interleave factors, since the controllers for flexible devices are not as fast as the Winchester controllers. Different types of flexible disks will have different optimum values for the interleave factor. So optimum values for 8 inch diskettes will not be the same as optimum values for 5¼ inch diskettes.

The default value for the interleave factor in the FORMAT command is 5. The recommended Winchester interleave factor for the iRMX 86 Release 5.0 operating system was 4. This was evaluated to verify if this was the best interleave factor for the Winchester. Then flexible disks were evaluated to find out which interleave factor was best for each type of application.

PERFORMANCE TESTS

Since the goal of this application note was to determine optimum values of parameters when systems were being used for development, benchmarks which measure CPU performance were not used. Instead, all the tests used involved a lot of disk I/O. Some of them also involved building tables in memory. These tables could have been built on disk if there had been insufficient memory available for them, but that would have degraded performance markedly.

The languages and system utilities were used extensively, as well as the DIR, COPY, BACKUP and RESTORE cusps. These are the kinds of things that are done most frequently while using the system for development purposes. Descriptions of each of the tests used follow.

RSAT Maximum Transfer Rate:

The RSAT test (iRMX 86 System Acceptance Test) measures the number of bytes transferred every 60 seconds from a secondary storage device. It also keeps track of the maximum number of bytes transferred in a 60 second time period. If RSAT is invoked with a large buffer size, a good approximation of the maximum transfer rate of a device is obtained. A larger buffer size will increase the transfer rate. The largest buffer size that RSAT allows is 63K. This is a multiple of both the Winchester device granularity and the diskette device granularity, so it allows the maximum transfer rate.

The RSAT test is a composite of reads, writes, seeks, and truncates. The results are a good indication of overall performance during disk I/O. The RSAT test was run for both the Winchester and the diskette evaluations. The RSAT performance data contributed heavily in the analysis of the performance data.

iRMX™ 86 Operating System Generation:

The generation of an iRMX 86 Operating system from an Interactive Configuration Utility created submit file consists of a number of compilations and assemblies, and extensive use of the utilities. The generation of an iRMX 86 Operating System gives a very good indication of the performance that can be expected when using these utilities. The time necessary to complete the iRMX 86 Operating System generation was weighted heavily in the analysis of the performance data. This test involved I/O only on the Winchester.

iRMX™ 86 BIOS Generation

Because the BIOS generation has fewer steps in it than a full iRMX 86 Operating System generation, it was used to zero in on the best configuration quickly. It assembles two modules, and then links two groups of libraries together. This test involved I/O only on the Winchester.

COPY Test:

The Human Interface COPY cusp was invoked to copy a large file (greater than 128K) from secondary storage. This test was used to see what the normal throughput of the system was. This was done for both Winchester and flexible disk devices.

DIR Test:

This test listed a directory with a large number of files in it to the terminal. The short file format was used to display the directory. This test was of some interest but was not weighted heavily in the analysis of the performance data. The time required to access a directory is extremely dependent on the location of the directory on the device. This test was done for the Winchester and flexible disk devices.

Boot Disk Generation:

The boot disk generation test formatted a diskette and then copied the files need for a bootable system onto the diskette. This test gave a good indication of the performance when writing to a diskette. This test was performed to determine diskette performance with different interleave factors.

Boot Test:

The amount of time it took the iRMX 86 Operating System to boot from a diskette was recorded to determine the best interleave factor for booting. This test was performed only on diskettes to determine the optimum interleave factor.

BACKUP And RESTORE Test:

The time to BACKUP and RESTORE from the Winchester to one flexible disk was gathered to determine the best interleave factor for the diskette when it is being used as a backup device. The iRMX 86 Release 5 versions of BACKUP and RESTORE were used in the evaluation. The volume granularity and BIOS buffer sizes were not a factor, since the diskette is formatted physical. BACKUP and RESTORE perform 1K reads and writes to the diskette. BACKUP and RESTORE use both Winchester I/O and diskette I/O, but the diskette was the only device which had the interleave factor changed as part of the evaluation with this test.

EVALUATION SEQUENCE

Since all possible permutations of variables could not be tested, the evaluation was performed by varying one parameter at a time. The research started with the default configurations of the operating system. The first parameter was varied to find the best value, and then its best value was used to determine the next parameter's optimum value. This method continued until all optimum values had been found. The parameters were tested in the following sequence.

- 1) The best Winchester interleave factor was found for the standard system. Values from 1 to 9 were tested for the System 86/330A, and interleave factors from 1 to 8 were tested for the System 86/310.
- 2) The best Application Loader read buffer size was determined. Buffer sizes from 1K to 8K bytes were tested for both systems in 1K byte increments.

- 3) The best number of Winchester BIOS buffers was determined. The systems were tested with 1 to 8 BIOS buffers for the Winchester.
- 4) The best Winchester volume granularity was found. Volume granularities of 1K and 2K bytes were tested for each system.
- 5) The best Winchester interleave factor was determined. Again, Winchester interleaves from 1 to 9 were tested for the System 86/330A, and Winchester interleaves from 1 to 8 were tested for the System 86/310.
- 6) The best number of BIOS buffers was determined for the diskettes. The systems were tested with 1 to 6 BIOS buffers.
- 7) The best diskette volume granularity was determined. Values of 256, 512 and 1024 bytes were tested for the System 86/330A diskette. Values of 512 and 1024 bytes were tested for the System 86/310 diskette.
- 8) The best flexible disk interleave factor was determined for each operation. Interleave factors from 1 to 8 were tested for the System 86/330A diskette. Interleave factors from 1 to 7 were tested for the System 86/310 diskette.

SYSTEM 86/330A PERFORMANCE RESULTS

Performance data was collected on a production System 86/330A using the iRMX 86 Release 5.0 Operating System in its standard configuration. Tests were run to determine the best configuration parameter values. Performance data was again collected with the best configuration of the iRMX 86 Operating System to determine the improvement in performance. The results for both configurations of the operating system using the 8" Priam Winchester using the iSBC 215 controller are shown in Tables 1 and 2 for the 5 MHz system and the 8 MHz system.

**Table 1. System 86/330A Winchester Performance
(5 MHz iSBC® 86/30 Single Board Computer)**

Test	Execution Time		
	Standard (min:sec)	Optimum (min:sec)	Improvement (percent)
iRMX 86 Generation	17:22	16:08	7%
BIOS Generation	4:20	4:05	6%
DIR of 171 files	0:25	0:25	0%
COPY 128 K byte file	0:10	0:10	0%
	Bytes per Second		
	Standard	Optimum	Improvement
RSAT max transfer rate	80,640	90,316	12%

**Table 2. System 86/330A Winchester Performance
(8 MHz iSBC® 86/30 Single Board Computer)**

Test	Execution Time		
	Standard (min:sec)	Optimum (min:sec)	Improvement (percent)
iRMX 86 Generation	13:26	12:29	7%
BIOS Generation	3:21	3:09	6%
DIR of 171 files	0:22	0:19	14%
COPY 128 K byte file	0:10	0:08	20%
	Bytes per Second		
	Standard	Optimum	Improvement
RSAT max transfer rate	92,467	105,370	12%

Note that the granularity of the measurements was 1 second. In a test that takes 10 seconds to run, the real amount of time necessary to run a test could be 10% off of the result shown. This can account for the differences shown between the 5 and 8 MHz systems, especially in the DIR and COPY command tests. The tests which took greater quantities of time are a more accurate reflection of the actual system performance that can be expected.

The optimum values of the parameters are listed below. These were the parameter values which were used in the optimal iRMX 86 operating system configuration.

- The Application Loader read buffer size was increased from 1K to 7K.
- The volume granularity of the Winchester was left at 1K, which is the device granularity for the Winchester.
- The number of Winchester BIOS buffers was increased from 4 to 8.
- A Winchester interleave factor of 3 instead of 4 was used.
- The volume granularity of the diskette was left at 256 bytes. This was the 8" diskette's device granularity.
- The number of diskette BIOS buffers was increased from 2 to 4.

Performance data was collected on the 8" DS/DD diskette drive using the iSBX™ 218 controller after finding the optimum values of the parameters for the rest of the System 86/330A. The best interleave factor for the diskette was determined for each type of use. The results are shown in Table 3.

The fastest boot time was found when the diskette was formatted with an interleave factor of 7. For transferring files between systems by using the COPY command the interleave factor should be 3. This is shown in Table 4 by the results of the Boot Disk Generation test, the DIR test, and the COPY test. When treating an 8" diskette as a physical device, as in BACKUP and RESTORE, the interleave factor should be 2.

SYSTEM 86/310 PERFORMANCE RESULTS

Performance data was collected on a System 86/310 using the iRMX 86 Release 5.1 Operating System in its standard configuration. Tests were run to determine the best configuration parameter values. Performance data was again collected with the best configuration of the iRMX 86 Operating System to determine the improvement in performance. The result for both configurations of the operating system at 5 MHz using the 5¼" CMI Winchester with the iSBC 215 controller is shown in Table 5. The result for the optimum configuration of the operating system at 8 MHz using the 5¼" CMI Winchester is shown in Table 6.

Table 3. System 86/330A DS/DD Diskette Performance

Test	Execution Time (min:sec)		
	7	3	2
Interleave Factor	7	3	2
Boot Time	1:02	2:42	2:36
Boot Disk Generation	6:34	6:18	6:43
DIR of 14 Files	0:22	0:21	0:22
COPY 187, 768 Byte File	1:01	0:45	1:03
BACKUP	4:15	2:43	2:25
RESTORE	4:40	2:43	2:25
	Bytes per Second		
RSAT Max Transfer Rate	5,376	9,677	12,902

Table 4. Optimum Interleave Factors for System 86/330A Diskettes

Type of Application	Optimum Interleave Factor
Using Bootstrap Loader	7
Transferring Named Files	3
BACKUP and RESTORE	2

**Table 5. System 86/310 Winchester Performance
(5MHz ISBC® 86/30 Single Board Computer)**

Test	Execution Time		
	Standard (min:sec)	Optimum (min:sec)	Improvement (percent)
iRMX 86 Generation	19:16	17:57	7%
BIOS Generation	4:26	4:22	1%
DIR of 171 Files	0:27	0:24	11%
COPY 128 K Byte File	0:14	0:13	7%
	Bytes per Second		
	Standard	Optimum	Improvement
RSAT Max Transfer Rate	72,038	69,888	-3%

**Table 6. System 86/310 Winchester Performance
(8 MHz iSBC® 86/30 Single Board Computer)**

Test	Execution Time
	Optimum (min:sec)
iRMX 86 Generation	14:35
BIOS Generation	3:32
DIR of 171 Files	0:20
COPY 128 K Byte File	0:11
	Bytes per Second
	Optimum
RSAT Max Transfer Rate	78,490

The best performance of the System 86/310 was found with the following configuration of the parameters.

- The Application Loader read buffer size was increased from 1K to 7K bytes.
- The volume granularity of the Winchester was left at 1K bytes.
- The number of Winchester BIOS buffers was increased from 4 to 8.
- A Winchester interleave factor of 4 was used.
- The volume granularity of the diskette was left at 256 bytes.
- The number of diskette BIOS buffers was increased from 2 to 4.

After the optimum values were found for the variables affecting Winchester performance, performance data was collected on the 5.25" DS/DD diskette drive using the iSBX 218A controller to determine the best interleave factor for the diskette. Again, different interleave factors were best for different uses of the flexible disk. The results are shown in Table 7.

Table 7. System 86/310 DS/DD Diskette Performance

Test	Execution Time (min:sec)		
	5	2	1
Interleave Factor			
Boot Time	1:20	1:55	1:55
Boot Disk Generation	12:50	10:59	11:20
DIR of 14 Files	0:24	0:27	0:24
COPY 181,784 Byte File	1:17	1:01	1:16
BACKUP	2:02	1:20	1:10
RESTORE	2:04	1:11	1:00
	Bytes per Second.		
RSAT Max Transfer Rate	3,226	6,451	9,677

As the data shows, the best interleave factor for booting was 5. For ordinary use in transferring files between systems with the COPY cusp the interleave factor should be 2. This was demonstrated by the Boot Disk Generation Test, the DIR test and the COPY test. For BACKUP and RESTORE from the Winchester to the flexible disk the best interleave factor was 1. These numbers are shown in Table 8 below.

Table 8. Optimum Interleave Factors for System 86/310 Diskettes

Type of Application	Optimum Interleave Factor
Using Bootstrap Loader	5
Transferring Named Files	2
BACKUP and RESTORE	1

CONCLUSION

The parameters changed generally affected I/O performance the most. The Application Loader read buffer size was changed from 1K to 7K bytes. The number of BIOS buffers was changed from 2 to 4 for flexible diskettes, and from 4 to 8 for Winchester devices. The interleave factor was set to 3 for the System 86/330A Winchester, and to 4 for the System 86/310 Winchester. The optimum interleave factor for each system's flexible diskette varied according to how the diskette was to be used.

By reconfiguring the system with different values for the configuration parameters and no changes to the hardware, a performance improvement of up to 20% may result. Performance may also be improved by changing the interleave factor when formatting random access disk devices. An application that used disk I/O would benefit by using the optimum values found in this application note.

APPENDIX A
SYSTEM CONFIGURATIONS

APPENDIX A

SYSTEM CONFIGURATIONS

Both systems used in the performance testing were production model systems. The software used on them was the iRMX 86 Release 5.0 Operating System for the System 86/330A, and the iRMX 86 Release 5.1 Operating System for the System 86/310. Hardware and software configurations of each system are shown in Table A-1.

Table A-1. System Configurations

System 86/330A	System 86/310
iSBC 86/30 Single Board Computer 384 K Memory iSBC 215/218 Disk Controller iRMX 86 Release 5.0 Operating System	iSBC 86/30 Single Board Computer 640 K Memory iSBC 215/218 Disk Controller iRMX 86 Release 5.1 Operating System

Note: More memory was required to run the ICU than was available in the System 86/330A. The ICU requires 448K bytes of RAM to run.

**APPENDIX B
DEFINITION FILE
OF AN OPTIMIZED iRMX™
86 OPERATING SYSTEM**

APPENDIX B

Definition File of an Optimized iRMX™ 86 Operating System

Hardware

(OSP) 80130 Operating System Extension [Yes/No]	No
(OTU) 80130 Timer Used [Yes/No]	No
(OPU) 80130 PIC used [Yes/No]	No
(OCD) 80130 Coypright = 1981 [Yes/No]	Yes
(BL) 80130 Base Address Location [40h-0FFFFh]	0000H
(BP) 80130 Base Port Address [0-0FFFFH]	0000H
(MP) 8259A Master Port [0-0FFFFH]	00C0H
(MPS) Master PIC Port Separation [0-0FFH]	0002H
(SIL) Slave Interrupt Levels [1-7/None]	None
(LSS) Level Sensitive Slaves [1-7/None]	None
(LSP) Local slave PICS [1-7/None]	None
(TP) 8253 Timer Port [0-0FFFFH]	00D0H
(CIL) Clock Interrupt Level [0-7]	0002H
(CN) Timer Counter Number [0,1,2]	0000H
(CI) Clock Interval [0-0FFFFH msec]	000AH
(CF) Clock Frequency [0-0FFFFH khz]	04CDH
(TPS) Timer Port Separation [0-0FFH]	0002H
(NPX) Numeric Processor Extension [Yes/No]	Yes
(NIL) NPX Interrupt Level [Encoded]	0008H

Memory

Type : RAM = low, high
 Type : ROM = low, high
 Type : RAM = 0104H, 239FH
 Type : RAM = 28CDH, F7FFH

Sub-systems

(UDI) Universal Development Interface [Yes/No]	Yes
(HI) Human Interface [Yes/No]	Req
(AL) Application Loader [Yes/No]	Req
(EIO) Extended I/O System [Yes/No]	Req
(BIO) Basic I/O System [Yes/No]	Req
(DB) Debugger [Yes/No]	No
(TH) Terminal Handler [Yes/No]	No
(CA) Crash Analyzer [Yes/No]	No
(UIR) UDI in ROM [Yes/No]	No
(CAR) Crash Analyzer in ROM [Yes/No]	No
(RIR) Root Job in ROM [Yes/No]	No

Human Interface

(ICL) Initial Command Line Size [0-0FFFFH]	0100H
(CNM) Command Name Length [0-255]	0030H
(SYS) System Directory [1-45 characters]	:SD:SYSTEM
(DRP) Default Resident Initial Program [Yes/No]	Yes
(RIP) Resident Initial Program [1-45 characters]	Default
(CDN) Configuration Device Name [1-14 chars]	:SD:
(PMI) Human Interface Pool Minimum [0-0FFFFH]	0100H
(PMA) Human Interface Pool Maximum [0-0FFFFH]	FFFFH
(HIR) Human Interface in ROM [Yes/No]	No

HI Jobs

(MIN) Jobs Minimum Memory [0-0FFFFH pages] 0100H
 (MAX) Jobs Maximum Memory [0-0FFFFH pages] 0000H
 (NPX) Numeric Processor Extension Used [Yes/No] Yes

Resident User

(TDN) Terminal Device Name [1-12 characters] TO
 (MTP) Maximum Task Priority [0-0FFH] 00A0H
 (UID) User ID Number [0-0FFFFH] FFFFH
 (MIN) Minimum Memory Required [0-0FFFFH] 0100H
 (MAX) Maximum Memory Required [0-0FFFFH] FFFFH
 (IPP) Initial-Program Pathname [RESIDENT/1-45 characters] RESIDENT
 (DEF) Default Directory [1-45 characters] :sd:user/world

Prefixes

Prefix: 1-45 characters
 Prefix: :\$:
 Prefix: :PROG:
 Prefix: :UTILS:
 Prefix: :SYSTEM:
 Prefix: :LANG:
 Prefix:

HI Logical Names

Logical Name: logical_name,path_name [1-12 Chars, 1-45 Chars]
 Logical Name: LANG, :SD:WORK
 Logical Name: WORK, :SD:WORK
 Logical Name: SYSTEM, :SD:SYSTEM
 Logical Name: UTILS, :SD:UTILS

Application Loader

(IBS) Internal Buffer Size [0-0FFFFh] 0400H
 (RBS) Read Buffer Size [0-0FFFFh] 1C00H
 (LJT) Load Job Type [None/Async/Sync] Synchronous and Asynchronous
 (DMP) Default Memory Pool Size [0-0FFFFh] 0100H
 (CT) Code Type [Abs/Pic/Ltl/Ovr] Overlay, LTL, PIC and Abs
 (ALR) Application Loader in ROM [Yes/No] No

EIOS

(ASC) All Sys Calls in EIOS Req
 (ABR) Automatic Boot Device Recognition [Yes/No] Yes
 (DLN) Default System Device Logical Name [1-12 characters] sd
 (DPN) Default System Device Physical Name [1-12 characters] wfd
 (DFD) Default System Device File Driver [Phys/Str/Named] Named
 (DO) Default System Device Owners ID [0-0FFFFH] 0000H
 (EBS) Internal Buffer Size [0-0FFFFh] 0400H
 (DDS) Default IO Job Directory Size [5-0FF0h] 0020H
 (ITP) Internal EIOS Task's Priorities [0-0FFH] 0083H
 (PMI) EIOS Pool Minimum [0-0FFFFH] 0180H
 (PMA) EIOS Pool Maximum [0-0FFFFH] 0180H
 (EIR) Extended I/O System in ROM [Yes/No] No

I/O Users

User: user name,Owner-ID [,ID,ID,ID,ID]

Logical Names

Logical Name: logical_name,device_name,file_driver,owners-id [1-12 Chars ,1-14 Chars,
Physical/Stream/Named, 0-0FFFFH]
 Logical Name: BB, BB, Physical, 0000H
 Logical Name: STREAM,STREAM, Stream, 0000H
 Logical Name: LP,LP, Physical, 0000H

BIOS

(ASC) All Sys Calls in BIOS [Yes/No] Req
 (ADP) Attach Device Task Priority [1-0FFFH] 0081H
 (TF) Timing Facilities Required [Yes/No] Yes
 (TTP) Timer Task Priority [0-0FFFH] 0081H
 (CON) Connection Job Delete Priority [0-0FFFH] 0082H
 (ACE) Ability to Create Existing Files [Yes/No] Yes
 (SMI) System Manager ID [Yes/No] Yes
 (CUT) Common Update Timeout [0-0FFFFH] 03E8H
 (CST) Control-Sequence Translation [Yes/No] Req
 (PMI) BIOS Pool Minimum [0-0FFFFH] 0800H
 (PMA) BOIS Pool Maximum [0-0FFFFH] 0800H
 (BIR) Basic I/O System in ROM [Yes/No] No

Intel Terminal Driver

(IIL) Input Interrupt Level [Encoded] 0068H
 (OIL) Output Interrupt Level [Encoded] 0078H
 (UDP) USART Data Port [0-0FFFFH] 00D8H
 (USP) USART Status Port [0-0FFFFH] 00DAH
 (IRP) 8253 Input Rate Port [0-0FFFFH] 00D4H
 (ICP) 8253 Input Control Port [0-0FFFFH] 00D6H
 (IRC) 8253 Input Counter Number [0-2] 0002H
 (IRM) Input Rate Maximum [0-0FFFFFFFH] 00012C00H
 (ORP) 8253 Output Rate Port [0-0FFFFH] 0000H
 (OCP) 8253 Output Control Port [0-0FFFFH] 0000H
 (ORC) 8253 Output Counter Number [0-2] 0000H
 (ORM) Output Rate Maximum [0-0FFFFFFFH] 00000000H

Intel Terminal Driver Unit Information

(NAM) Unit Info Name [1-17 Chars] t0_info
 (LEM) Line Edit Mode [Trans/Normal/Flush] Normal
 (ECH) Echo Mode [Yes/No] Yes
 (IPC) Input Parity Control [Yes/No] Yes
 (OPC) Output Parity Control [Yes/No] Yes
 (OCC) Output Control in Input [Yes/No] Yes
 (OSC) OSC Controls [Both/In/Out/Neither] Both
 (DUP) Duplex Mode [Full/Half] Full
 (TRM) Terminal Type [CRT/Hard Copy] CRT
 (MC) Modem Control [Yes/No] No
 (RPC) Read Parity Checking [See Help/0-3] 0000H
 (WPC) Write Parity Checking [See Help/0-4] 0000H
 (BR) Baud Rate [0-0FFFFH] 2580H
 (SN) Scroll Number [0-0FFFFH] 0017H

Intel Terminal Driver Device-Unit Information

(NAM) Device-Unit Name [1-13 Chars] TO
 (UN) Unit Number on this Device [0-0FFFH] 0000H
 (UIN) Unit Info Name [1-17 Chars] t0_info
 (MB) Max Buffers [0-0FFFH] 0000H

Intel iSBC® 215/218 Driver

(IL) Interrupt Level [Encoded Level] 0058H
 (ITP) Interrupt Task Priority [0-0FFFH] 0082H
 (WIP) Wakeup I/O Port [0-0FFFFH] 0100H

Intel iSBC® 215/218 Unit Information

(NAM) Unit Info Name [1-17 Chars] uinfo_215gen
 (MR) Maximum Retries [0-0FFFFH] 0009H
 (CS) Cylinder Size [0-0FFFFH] 0000H
 (NC) Number of Cylinders [0-0FFFFH] 0001H
 (NFH) Number of Fixed Platters/Disk [0-0FFFH] 0001H
 (NRH) Number of Remove Platters/Disk [0-0FFFH] 0000H
 (NS) Number of Sectors/Track [0-0FFFFH] 000CH
 (NAC) Number of Aux. Cylinders [0-0FFFH] 0001H
 (SSN) Starting Sector Number [0-0FFFFFFFH] 00000000H
 (BTI) Bad Track Information [Yes/No] Yes

Intel iSBC® 215/218 Unit Information

(NAM) Unit Info Name [1-17 Chars] uinfo_215w5
 (MR) Maximum Retries [0-0FFFFH] 0009H
 (CS) Cylinder Size [0-0FFFFH] 0000H
 (NS) Number of Cylinders [0-0FFFFH] 0132H
 (NFH) Numbers of Fixed Platters/Disk [0-0FFFH] 0004H
 (NRH) Number of Remove Platters/Disk [0-0FFFH] 0000H
 (NS) Number of Sectors/Track [0-0FFFFH] 0009H
 (NAC) Number of Aux. Cylinders [0-0FFFH] 000AH
 (SSN) Starting Sector Number [0-0FFFFFFFH] 00000000H
 (BIT) Bad Track Information [Yes/No] Yes

Intel iSBC® 215/218 Unit Information

(NAM) Unit Info Name [1-17 Chars] uinfo_215w
 (MR) Maximum Retries [0-0FFFFH] 0009H
 (CS) Cylinder Size [0-0FFFFH] 0000H
 (NC) Number of Cylinders [0-0FFFFH] 020DH
 (NFH) Numbers of Fixed Platters/Disk [0-0FFFH] 0005H
 (NRH) Number of Remove Platters/Disk [0-0FFFH] 000H
 (NS) Number of Sectors/Track [0-0FFFFH] 000CH
 (NAC) Number of Aux. Cylinders [0-0FFFH] 000AH
 (SSN) Starting Sector Number [0-0FFFFFFFH] 00000000H
 (BIT) Bad Track Information [Yes/No] Yes

Intel iSBC® 215/218 Unit Information

(NAM) Unit Info Name [1-17 Chars] uinfo_215pt
 (MR) Maximum Retries [0-0FFFFH] 0009H
 (CS) Cylinder Size [0-0FFFFH] 0000H
 (NC) Number of Cylinders [0-0FFFFH] 01D2H
 (NFH) Number of Fixed Platters/Disk [0-0FFFH] 0003H
 (NRH) Number of Remove Platters/Disk [0-0FFFH] 0000H
 (NS) Number of Sectors/Track [0-0FFFFH] 000CH
 (NAC) Number of Aux. Cylinders [0-0FFFH] 0006H
 (SSN) Starting Sector Number [0-0FFFFFFFH] 00000000H
 (BTI) Bad Track Information [Yes/No] Yes

Intel iSBC® 215/218 Unit Information

(NAM) Unit Info Name [1-17 Chars] uinfo_215f
 (MR) Maximum Retries [0-0FFFFH] 0009H

(CS) Cylinder Size [0-0FFFFH]	0000H
(NC) Number of Cylinders [0-0FFFFH]	004DH
(NFH) Number of Fixed Platters/Disk [0-0FFH]	0000H
(NRH) Number of Remove Platters/Disk [0-0FFH]	0001H
(NS) Number of Sectors/Track [0-0FFFFH]	001AH
(NAC) Number of Aux. Cylinders [0-0FFH]	0000H
(SSN) Starting Sector Number [0-0FFFFFFFH]	00000000H
(BTI) Bad Track Information [Yes/No]	Yes

Intel iSBC® 215/218 Unit Information

(NAM) Unit Info Name [1-17 Chars]	uinfo_215fd
(MR) Maximum Retries [0-0FFFFH]	0009H
(CS) Cylinder Size [0-0FFFFH]	0000H
(NC) Number of Cylinders [0-0FFFFH]	004DH
(NFH) Number of Fixed Platters/Disk [0-0FFH]	0000H
(NRH) Number of Remove Platters/Disk [0-0FFH]	0002H
(NS) Number of Sectors/Track [0-0FFFFH]	001AH
(NAC) Number of Aux Cylinders [0-0FFH]	0000H
(SSN) Starting Sector Number [0-0FFFFFFFH]	00000000H
(BTI) Bad Track Information [Yes/No]	Yes

Intel iSBC® 215/218 Unit Information

(NAM) Unit Info Name [1-17 Chars]	uinfo_shugart96
(MR) Maximum Retries [0-0FFFFH]	0009H
(CS) Cylinder Size [0-0FFFFH]	0000H
(NC) Number of Cylinders [0-0FFFFH]	0050H
(NFH) Number of Fixed Platters/Disk [0-0FFH]	0000H
(NRH) Number of Remove Platters/Disk [0-0FFH]	0002H
(NS) Number of Sectors/Track [0-0FFFFH]	0008H
(NAC) Number of Aux. Cylinders [0-0FFH]	0000H
(SSN) Starting Sector Number [0-0FFFFFFFH]	00000000H
(BTI) Bad Track Information [Yes/No]	No

Intel iSBC® 215/218 Unit Information

(NAM) Unit Info Name [1-17 Chars]	uinfo_shugart48
(MR) Maximum Retries [0-0FFFFH]	0009H
(CS) Cylinder Size [0-0FFFFH]	0000H
(NC) Number of Cylinders [0-0FFFFH]	0028H
(NFH) Number of Fixed Platters/Disk [0-0FFH]	0000H
(NRH) Number of Remove Platters/Disk [0-0FFH]	0002H
(NS) Number of Sectors/Track [0-0FFFFH]	0008H
(NAC) Number of Aux. Cylinders [0-0FFH]	0000H
(SSN) Starting Sector Number [0-0FFFFFFFH]	00000000H
(BTI) Bad Track Information [Yes/No]	No

Intel iSBC® 215/iSBX™ 218 Device-Unit Information

(NAM) Device-Unit Name [1-13 Chars]	cm0
(PFD) Physical File Driver Required [Yes/No]	Yes
(NFD) Named File Driver Required [Yes/No]	Yes
(SDD) Single or Double Density Disks [Single/Double]	Single
(SDS) Single or Double Sided Disks [Single/Double]	Single
(EFI) 8 or 5 Inch Disks [8/5]	5
(GRA) Granularity [0-0FFFFH]	0400H
(DSZ) Device Size [0-0FFFFFFFH]	00A68000H
(UN) Unit Number on this Device [0-0FFH]	0000H
(UIN) Unit Info name [1-17 Chars]	uinfo_215w5
(UDT) Update Timeout [0-0FFFFH]	0064H
(NB) Number of Buffers [nonrandom = 0/rand = 1-0FFFFH]	0008H
(FUP) Fixed Update [True/False]	True
(MB) Max Buffers [0-0FFH]	00FFH

Intel iSBC® 215/iSBX™ 218 Device-Unit Information

(NAM) Device-Unit Name [1-13 Chars]	iw0
(PFD) Physical File Driver Required [Yes/No]	Yes
(NFD) Named File Driver Required [Yes/No]	Yes
(SDD) Single or Double Density Disks [Single/Double]	Single
(SDS) Single or Double Sided Disks [Single/Double]	Single
(EFI) 8 or 5 Inch Disks [8/5]	8
(GRA) Granularity [0-0FFFFH]	0400H
(DSZ) Device Size [0-0FFFFFFFH]	01E2D000H
(UN) Unit Number on this Device [0-0FFH]	0000H
(UIN) Unit Info name [1-17 Chars]	uinfo_215w
(UDT) Update Timeout [0-0FFFFH]	0064H
(NB) Number of Buffers [nonrandom = 0/rand = 1-0FFFFH]	0008H
(FUP) Fixed Update [True/False]	True
(MB) Max Buffers [0-0FFH]	00FFH

Intel iSBC® 215/iSBX™ 218 Device-Unit Information

(NAM) Device-Unit Name [1-13 Chars]	wmfdx0
(PFD) Physical File Driver Required [Yes/No]	Yes
(NFD) Named File Driver Required [Yes/No]	Yes
(SDD) Single or Double Density Disks [Single/Double]	Double
(SDS) Single or Double Sided Disks [Single/Double]	Double
(EFI) 8 or 5 Inch Disks [8/5]	5
(GRA) Granularity [0-0FFFFH]	0200H
(DSZ) Device Size [0-0FFFFFFFH]	0004F800H
(UN) Unit Number on this Device [0-0FFH]	0008H
(UIN) Unit Info Name [1-17 Chars]	uinfo_shugart48
(UDT) Update Timeout [0-0FFFFH]	0064H
(NB) Number of Buffers [nonrandom = 0/rand = 1-0FFFFH]	0004H
(FUP) Fixed Update [True/False]	True
(MB) Max Buffers [0-0FFH]	00FFH

Intel iSBC® 215/iSBX™ 218 Device-Unit Information

(NAM) Device-Unit Name [1-13 Chars]	wfdd0
(PFD) Physical File Driver Required [Yes/No]	Yes
(NFD) Named File Driver Required [Yes/No]	Yes
(SDD) Single or Double Density Disks [Single/Double]	Double
(SDS) Single or Double Sided Disks [Single/Double]	Double
(EFI) 8 or 5 Inch Disks [8/5]	8
(GRA) Granularity [0-0FFFFH]	0100H
(DSZ) Device Size [0-0FFFFFFFH]	000F9700H
(UN) Unit Number on this Device [0-0FFH]	0008H
(UIN) Unit Info Name [1-17 Chars]	uinfo_215fd
(UDT) Update Timeout [0-0FFFFH]	0064H
(NB) Number of Buffers [nonrandom = 0/rand = 1-0FFFFH]	0004H
(FUP) Fixed Update [True/False]	True
(MB) Max Buffers [0-0FFH]	00FFH

Intel iSBC® 215/iSBX™ 218 Device-Unit Information

(NAM) Device-Unit Name [1-13 Chars]	wfd0
(PFD) Physical File Driver Required [Yes/No]	Yes
(NFD) Named File Driver Required [Yes/No]	Yes
(SDD) Single or Double Density Disks [Single/Double]	Double
(SDS) Single or Double Sided Disks [Single/Double]	Single
(EFI) 8 or 5 Inch Disks [8/5]	8
(GRA) Granularity [0-0FFFFH]	0100H
(DSZ) Device Size [0-0FFFFFFFH]	0007C500H

(UN) Unit Number on this Device [0-0FFH]	0008H
(UIN) Unit Info Name [1-17 Chars]	uinfo_215f
(UDT) Update Timeout [0-0FFFFH]	0064H
(NB) Number of Buffers [nonrandom = 0/rand = 1-0FFFFH]	0004H
(FUP) Fixed Update [True/False]	True
(MB) Max Buffers [0-0FFH]	00FFH

Intel iSBC® 215/iSBX™ 218 Device-Unit Information

(NAM) Device-Unit Name [1-13 Chars]	wf0
(PFD) Physical File Driver Required [Yes/No]	Yes
(NFD) Named File Driver Required [Yes/No]	Yes
(SDD) Single or Double Density Disks [Single/Double]	Single
(SDS) Single or Double Sided Disks [Single/Double]	Single
(EFI) 8 or 5 Inch Disks [8/5]	8
(GRA) Granularity [0-0FFFFH]	0080H
(DSZ) Device Size [0-0FFFFFFFH]	0003E900H
(UN) Unit Number on this Device [0-0FFH]	0008H
(UIN) Unit Info Name [1-17 Chars]	uinfo_215f
(UDT) Update Timeout [0-0FFFFH]	0064H
(NB) Number of Buffers [nonrandom = 0/rand = 1-0FFFFH]	0004H
(FUP) Fixed Update [True/False]	True
(MB) Max Buffers [0-0FFH]	00FFH

Intel iSBC® 215/iSBX™ 218 Device-Unit Information

(NAM) Device-Unit Name [1-13 Chars]	cm1
(PFD) Physical File Driver Required [Yes/No]	Yes
(NFD) Named File Driver Required [Yes/No]	Yes
(SDD) Single or Double Density Disks [Single/Double]	Single
(SDS) Single or Double Sided Disks [Single/Double]	Single
(EFI) 8 or 5 Inch Disks [8/5]	5
(GRA) Granularity [0-0FFFFH]	0400H
(DSZ) Device Size [0-0FFFFFFFH]	00A68000H
(UN) Unit Number on this Device [0-0FFH]	0001H
(UIN) Unit Info Name [1-17 Chars]	uinfo_215w5
(UDT) Update Timeout [0-0FFFFH]	0064H
(NB) Number of Buffers [nonrandom = 0/rand = 1-0FFFFH]	0004H
(FUP) Fixed Update [True/False]	True
(MB) Max Buffers [0-0FFH]	00FFH

Intel iSBC® 215/iSBX™ 218 Device-Unit Information

(NAM) Device-Unit Name [1-13 Chars]	iw1
(PFD) Physical File Driver Required [Yes/No]	Yes
(NFD) Named File Driver Required [Yes/No]	Yes
(SDD) Single or Double Density Disks [Single/Double]	Single
(SDS) Single or Double Sided Disks [Single/Double]	Single
(EFI) 8 or 5 Inch Disks [8/5]	8
(GRA) Granularity [0-0FFFFH]	0400H
(DSZ) Device Size [0-0FFFFFFFH]	01E2D000H
(UN) Unit Number on this Device [0-0FFH]	0001H
(UIN) Unit Info Name [1-17 Chars]	uinfo_215w
(UDT) Update Timeout [0-0FFFFH]	0064H
(NB) Number of Buffers [nonrandom = 0/rand = 1-0FFFFH]	0004H
(FUP) Fixed Update [True/False]	True
(MB) Max Buffers [0-0FFH]	00FFH

Intel iSBC® 215/iSBX™ 218 Device-Unit Information

(NAM) Device-Unit Name [1-13 Chars]	wmfdx1
-------------------------------------	--------

(PFD) Physical File Driver Required [Yes/No]	Yes
(NFD) Named File Driver Required [Yes/No]	Yes
(SDD) Single or Double Density Disks [Single/Double]	Double
(SDS) Single or Double Sided Disks [Single/Double]	Double
(EFI) 8 or 5 Inch Disks [8/5]	5
(GRA) Granularity [0-0FFFFFH]	0200H
(DSZ) Device Size [0-0FFFFFFFH]	0004F800H
(UN) Unit Number on this Device [0-0FFH]	0009H
(UIN) Unit Info Name [1-17 Chars]	uinfo_shugart48
(UDT) Update Timeout [0-0FFFFFH]	0064H
(NB) Number of Buffers [nonrandom = 0/rand = 1-0FFFFFH]	0004H
(FUP) Fixed Update [True/False]	True
(MB) Max Buffers [0-0FFH]	00FFH

Intel iSBC® 215/iSBX™ 218 Device-Unit Information

(NAM) Device-Unit Name [1-13 Chars]	wfdd1
(PFD) Physical File Driver Required [Yes/No]	Yes
(NFD) Named File Driver Required [Yes/No]	Yes
(SDD) Single or Double Density Disks [Single/Double]	Double
(SDS) Single or Double Sided Disks [Single/Double]	Double
(EFI) 8 or 5 Inch Disks [8/5]	8
(GRA) Granularity [0-0FFFFFH]	0100H
(DSZ) Device Size [0-0FFFFFFFH]	000F9700H
(UN) Unit Number on this Device [0-0FFH]	0009H
(UIN) Unit Info Name [1-17 Chars]	uinfo_215fd
(UDT) Update Timeout [0-0FFFFFH]	0064H
(NB) Number of Buffers [nonrandom = 0/rand = 1-0FFFFFH]	0004H
(FUP) Fixed Update [True/False]	True
(MB) Max Buffers [0-0FFH]	00FFH

Intel iSBC® 215/iSBX™ 218 Device-Unit Information

(NAM) Device-Unit Name [1-13 Chars]	wf1
(PFD) Physical File Driver Required [Yes/No]	Yes
(NFD) Named File Driver Required [Yes/No]	Yes
(SDD) Single or Double Density Disks [Single/Double]	Double
(SDS) Single or Double Sided Disks [Single/Double]	Single
(EFI) 8 or 5 Inch Disks [8/5]	8
(GRA) Granularity [0-0FFFFFH]	0100H
(DSZ) Device Size [0-0FFFFFFFH]	0007C500H
(UN) Unit Number on this Device [0-0FFH]	0009H
(UIN) Unit Info Name [1-17 Chars]	uinfo_215f
(UDT) Update Timeout [0-0FFFFFH]	0064H
(NB) Number of Buffers [nonrandom = 0/rand = 1-0FFFFFH]	0004H
(FUP) Fixed Update [True/False]	True
(MB) Max Buffers [0-0FFH]	00FFH

Intel iSBC® 215/iSBX™ 218 Device-Unit Information

(NAM) Device-Unit Name [1-13 Chars]	wf1
(PFD) Physical File Driver Required [Yes/No]	Yes
(NFD) Named File Driver Required [Yes/No]	Yes
(SDD) Single or Double Density Disks [Single/Double]	Single
(SDS) Single or Double Sided Disks [Single/Double]	Single
(EFI) 8 or 5 Inch Disks [8/5]	8
(GRA) Granularity [0-0FFFFFH]	0080H
(DSZ) Device Size [0-0FFFFFFFH]	0003E900H
(UN) Unit Number on this Device [0-0FFH]	0009H
(UIN) Unit Info Name [1-17 Chars]	uinfo_215f

(UDT) Update Timeout [0-0FFFFH]	0064H
(NB) Number of Buffers [nonrandom = 0/rand = 1-0FFFFH]	0004H
(FUP) Fixed Update [True/False]	True
(MB) Max Buffers [0-0FFH]	00FFH

Intel iSBC® 215/iSBX™ 218 Device-Unit Information

(NAM) Device-Unit Name [1-13 Chars]	wmfdy0
(PFD) Physical File Driver Required [Yes/No]	Yes
(NFD) Named File Driver Required [Yes/No]	Yes
(SDD) Single or Double Density Disks [Single/Double]	Double
(SDS) Single or Double Sided Disks [Single/Double]	Double
(EFI) 8 or 5 Inch Disks [8/5]	5
(GRA) Granularity [0-0FFFFH]	0200H
(DSZ) Device Size [0-0FFFFFFFH]	0009F800H
(UN) Unit Number on this Device [0-0FFH]	0008H
(UIN) Unit Info Name [1-17 Chars]	uinfo_shugart96
(UDT) Update Timeout [0-0FFFFH]	0064H
(NB) Number of Buffers [nonrandom = 0/rand = 1-0FFFFH]	0004H
(FUP) Fixed Update [True/False]	True
(MB) Max Buffers [0-0FFH]	00FFH

Intel iSBC® 215/iSBX™ 218 Device-Unit Information

(NAM) Device-Unit Name [1-13 Chars]	wmfdy1
(PFD) Physical File Driver Required [Yes/No]	Yes
(NFD) Named File Driver Required [Yes/No]	Yes
(SDD) Single or Double Density Disks [Single/Double]	Double
(SDS) Single or Double Sided Disks [Single/Double]	Double
(EFI) 8 or 5 Inch Disks [8/5]	5
(GRA) Granularity [0-0FFFFH]	0200H
(DSZ) Device Size [0-0FFFFFFFH]	0009F800H
(UN) Unit Number on this Device [0-0FFH]	0009H
(UIN) Unit Info Name [1-17 Chars]	uinfo_shugart96
(UDT) Update Timeout [0-0FFFFH]	0064H
(NB) Number of Buffers [nonrandom = 0/rand = 1-0FFFFH]	0004H
(FUP) Fixed Update [True/False]	True
(MB) Max Buffers [0-0FFH]	00FFH

Intel iSBC® 215/iSBX™ 218 Device-Unit Information

(NAM) Device-Unit Name [1-13 Chars]	pw0
(PFD) Physical File Driver Required [Yes/No]	Yes
(NFD) Named File Driver Required [Yes/No]	Yes
(SDD) Single or Double Density Disks [Single/Double]	Single
(SDS) Single or Double Sided Disks [Single/Double]	Single
(EFI) 8 or 5 Inch Disks [8/5]	8
(GRA) Granularity [0-0FFFFH]	0400H
(DSZ) Device Size [0-0FFFFFFFH]	0102C000H
(UN) Unit Number on this Device [0-0FFH]	0000H
(UIN) Unit Info Name [1-17 Chars]	uinfo_215pt
(UDT) Update Timeout [0-0FFFFH]	0064H
(NB) Number of Buffers [nonrandom = 0/rand = 1-0FFFFH]	0008H
(FUP) Fixed Update [True/False]	True
(MB) Max Buffers [0-0FFH]	00FFH

Intel iSBC® 215/iSBX™ 218 Device-Unit Information

(NAM) Device-Unit Name [1-13 Chars]	w0
(PFD) Physical File Driver Required [Yes/No]	Yes
(NFD) Named File Driver Required [Yes/No]	Yes
(SDD) Single or Double Density Disks [Single/Double]	Single

(SDS) Single or Double Sided Disks [Single/Double]	Single
(EFI) 8 or 5 Inch Disks [8/5]	8
(GRA) Granularity [0-0FFFFH]	0400H
(DSZ) Device Size [0-0FFFFFFFH]	00000400H
(UN) Unit Number on this Device [0-0FFH]	0000H
(UIN) Unit Info Name [1-17 Chars]	uinfo_215gen
(UDT) Update Timeout [0-0FFFFH]	0064H
(NB) Number of Buffers [nonrandom = 0/rand = 1-0FFFFH]	0008H
(FUP) Fixed Update [True/False]	True
(MB) Max Buffers [0-0FFH]	00FFH

Intel Line Printer Driver

(IL) Interrupt Level [Encoded Level]	0048H
(ITP) Interrupt Task Priority [0-0FFH]	0082H
(POA) 8255A Port A address [0-0FFFFH]	00C8H
(POB) 8255A Port B Address [0-0FFFFH]	00CAH
(POC) 8255A Port C Address [0-0FFFFH]	00CCH
(CON) 8255A Control Port Address [0-0FFFFH]	00CEH
(TAB) Printer Expanded Tabs [Yes/No]	Yes

Intel Line Printer Driver Device-Unit Information

(NAM) Device-Unit Name [1-13 Chars]	lp
(UN) Unit Number on this Device [0-0FFH]	0000H
(UIN) Unit Info Name [1-17 Chars]	NOT REQUIRED
(MB) Max Buffers [0-0FFH]	0000H

Nucleus

(ASC) All Sys Calls [Yes/No]	Req
(PV) Parameter Validation [Yes/No]	Yes
(ROD) Root Object Directory Size [0-0FF0h]	0020H
(MTS) Minimum Transfer Size [0-0FFFFH]	0040H
(DEH) Default Exception Handler [Yes/No/Deb/Use]	Yes
(NEH) Name of Ex Handler Object Module [1-32chs]	
(EM) Exception Mode [Never/Program/Environ/All]	Never
(NR) Nucleus in ROM [Yes/No]	No

User Jobs

(ODS) Object Directory Size [0-0FF0H]	0010H
(PMI) Pool Minimum [20H - 0FFFFH]	0100H
(PMA) Pool Maximum [20H - 0FFFFH]	0100H
(MOB) Maximum Objects [1-0FFFFH]	0010H
(MTK) Maximum Tasks [1-0FFFFH]	0010H
(MPR) Maximum Priority [0-0FFH]	0000H
(AEH) Address of Exception Handler [CS:IP]	0000H:0000H
(EM) Exception Mode [Never/Prog/Environ/All]	Never
(PV) Parameter Validation [Yes/No]	Yes
(TP) Task Priority [0-0FFH]	0082H
(TSA) Task Start Address [CS:IP]	23A0H:0000H
(DSB) Data Segment Base [0-0FFFFH]	0000H
(SSA) Stack Segment Address [SS:SP]	0000H:0000H
(SS) Stack Size [0-0FFFFH]	0100H
(NPX) Numeric Processor Extension Used [Yes/No]	No

Includes and Libraries

Path Name [1-45 Characters]

(UDF) UDI Includes and Libs

/rmx86/udi/

(HIF) Human Interface Includes and Libs

/rmx86/hi/

(EIF) Extended I/O system Includes and Libs

/rmx86/eios/

(ALF) Application Loader Includes and Libs

/rmx86/loader/

(BIF) Basic I/O System Includes and Libs

/rmx86/ios/

(THF) Terminal Handler and Debugger Includes and Libs

/rmx86/th/

(NUF) Nucleus and Root Job Includes and Libs

/rmx86/nucleus/

(ILF) Interface Libraries

/rmx86/lib/

(CAF) Crash Analyzer Includes and Libs

/rmx86/crash/

(DTF) Development Tools Path Names

:lang:

Generate File Names

File Name [1-55 Characters]

(ROF) ROM Code File Name

rmx86.rom

(RAF) RAM Code File Name

rmx86

Intel Related Publications

iRMX 86 Release 5 Operator's Manual (172764-001)

iRMX 86 Configuration Guide (9803126-05)

System 86/330A Overview Manual (144680-001)

iRMX 86 Documentation Addendum for Release 5 (146032-001)

iRMX 86 Basic I/O System Reference Manual (9803123-05)

iRMX 86 Loader Reference Manual (143318-002)



**APPLICATION
NOTE**

AP-184

August 1984

**Writing Device Drivers
For XENIX* 86 and 286 —
Task or Trivia?**

**MOHANDAS NAIR
APPLICATIONS MARKETING
INTEGRATED SYSTEMS OPERATION**

WRITING DEVICE DRIVERS FOR XENIX 86 AND 286 — TASK OR TRIVIA?

CONTENTS

1.0 INTRODUCTION
2.0 DEFINITIONS
3.0 COMPONENTS OF THE XENIX* I/O ENVIRONMENT
3.1 The process' view of the kernel
3.2 The kernel's view of the process
3.3 The kernel's view of the driver
3.4 The drivers view of the kernel
3.5 The driver's view of the device
3.6 Putting the components together
4.0 TYPES OF DEVICES
4.1 Block devices
4.2 Character devices
4.3 Combination devices
5.0 DEVICES VIEWED FROM THE USER INTERFACE
6.0 THE ENVIRONMENT
6.1 The u-area
6.2 Task-time execution vs. interrupt-time execution
6.3 I/O path through the system	..

7.0 THE ANATOMY OF THE I/O SYSTEM
7.1 The block interface
7.1.1 The system buffers
7.1.2 Driver support routines
7.1.3 Block device driver routines
7.1.4 Review
7.1.5 Steps taken to satisfy requests
7.1.6 iSBC® 254 Bubble Memory board walkthrough
7.1.7 Final look at block drivers
7.1.8 The raw (character interface) to a block device
7.2 The character interface
7.2.1 Clists
7.2.2 Terminal I/O
7.2.3 Useful routines
7.2.4 tty.help — the line discipline routines
7.2.5 Terminal I/O device driver routines
7.2.6 Examples of character I/O (terminal) drivers
7.2.6.1 iSBX™ 270 Walkthrough
7.2.6.2 Low level routines
7.2.6.3 Required routines
7.2.6.4 A final note

CONCLUSIONS

APPENDICES

REFERENCES

ACKNOWLEDGEMENTS

* XENIX is a trademark of MICROSOFT Corp.

1.0 INTRODUCTION

The world of device configuration and device drivers has since time been an area where only hacks could tread. Being the lowest-level of software interfacing to the device, drivers were always examined by self-motivated experts. Also, drivers were hard to come-by and even harder to comprehend.

Intel Corporation's "open systems" concept, coupled with the XENIX* Operating System and our family of microprocessor systems, creates an attractive environment for building and adding new devices and drivers. However, the folklore involved with the XENIX Operating System and its internal functions are exemplified in the lack of device driver details. This paper clears the fog around device drivers and device driver writing. It is written for the general operating system user bringing him/her details of the anatomy of the I/O system, driver interfaces and driver support routines. The details that follow pertain to the Release 1 XENIX 286/86 Operating System which is a superset of the Unix* V7 operating system. This application note discusses writing device drivers for the XENIX Operating System as well as describes the operating environment around device drivers. Note that the reader may not need the discussion on the environment when writing device drivers. However, when the reader begins to debug them, he/she will find these discussions worthwhile.

Most driver writers would agree that few start writing drivers from scratch. Many use existing driver code as templates. This paper includes actual coded and pseudo-coded examples coupled with descriptions which grant the reader (and soon to be writer) a bouncing-board introduction to writing device drivers.

2.0 DEFINITIONS

A device driver is that body of software that allows an operating system to communicate with a device. This body of software is the lowest software level of abstraction in the I/O system. Figure 1a shows these levels identifying device drivers as a set of machine-independent routines (commands) of the operating system which talk to devices.

In the XENIX Operating System, a device driver is a collection of procedures placed in a file that is configured into the system. No source code is needed for this configuration as the operating system, once configured, will talk with these routines in the driver.

Before attempting to write a XENIX device driver, one must:

- 1) Understand the device i.e. know how to talk to the device, initialize it etc.

- 2) Understand how the XENIX Operating interfaces to the driver i.e. what is covered in this application note.
- 3) Begin writing the routines needed for a driver i.e. mimic an existing driver.

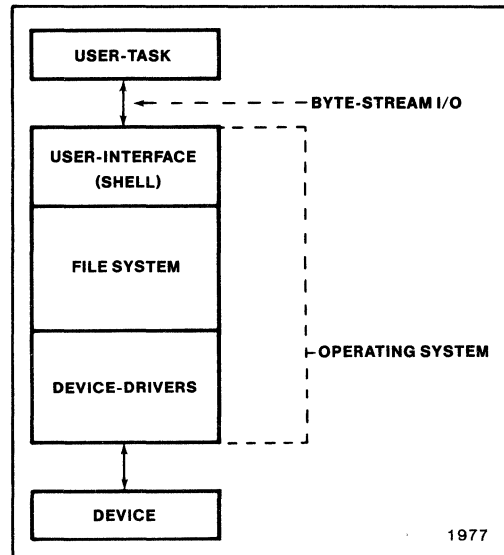


Figure 1a. Driver Model

3.0 COMPONENTS OF THE XENIX I/O ENVIRONMENT

This application note breaks device drivers and the XENIX Operating system into the following components:

- 1) the process
- 2) the kernel
- 3) the I/O buffers
- 4) the driver(s)
- 5) the device(s)

These components are shown in figure 1b. These components communicate with each other in unique ways.

3.1 The Process' View of the Kernel

Processes communicate with the kernel through system calls i.e. open, close, read. These system calls can be found in the XENIX Operating System Documentation (173258001).

3.2 The Kernel's View of the Process

Section 6.0 (THE ENVIRONMENT) defines a process, process synchronization, user-areas and introduces the kernel's view of the process.

*UNIX is a Trademark of Bell Labs

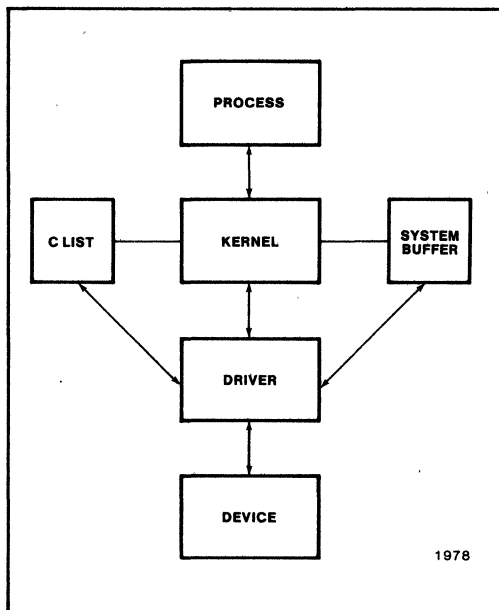


Figure 1b. Components of the I/O Environment

3.3 The Kernel's View of the Driver

Section 4.0 (Types of Devices) serves as an introduction to how the kernel views the driver. Details of this view are found in Section 7.0 (The Anatomy of the I/O System) where the system I/O buffers are described as used by the kernel.

3.4 The Drivers View of the Kernel

Section 7.0 (The Anatomy of the I/O System) explains how the driver communicates with the kernel, by introducing the assist routines (Section 7.1.2, 7.2.3, 7.2.4) available to drivers. Furthermore, the driver must understand the buffer scheme when talking with the kernel. Sections 7.1.1 and 7.2.1 (The I/O buffers) detail what the driver manipulates.

3.5 The Driver's View of the Device

The device driver is a collection of routines that act on the device and the device responds to the driver through interrupts. The driver talks with the device when an action is requested or when the device interrupts the driver on completion of the action. The driver talks with the device through routines which are discussed in Sections 7.1.3 (Block device driver routines), 7.2.5 (Terminal I/O device driver routines) and Appendix D (Interrupt Mapping).

3.6 Putting the Components Together

Section 6.3 (I/O Path through the system) will give

an introduction to how the above-mentioned components interact. To understand this discussion, the following concepts must be covered:

- 1) Types of Devices (Section 4.0)
- 2) How Devices are viewed from the user interface (Section 5.0)
- 3) The kernel's view of the process (The Environment Section 6.0, 6.1 and 6.2)

4.0 TYPES OF DEVICES

The XENIX Operating System supports two kinds of devices - Block and Character. Input/Output to/from these devices are consequently known as block and character I/O.

4.1 Block Devices

A block device is a sectored device that is accessed randomly. A file system resides on it as well. A good example of a block device is a winchester disk or a floppy disk. I/O with a block device is executed through a set of kernel I/O buffers(cache) which intervenes transfers of data (in fixed sized blocks) between user memory and the respective device. Block I/O involves a considerable amount of kernel activity due to these buffering characteristics.

Also:

- 1) The size of Block I/O transfer requests from kernel to device are a multiple of the system's blocksize (BSIZE). BSIZE is 1024 bytes in XENIX 286 Operating System.
- 2) Transfers are seldom done directly to the user task's memory. The transfers are staged through a buffer pool of BSIZE buffers. Also, the XENIX kernel manages these buffers to perform blocking/deblocking and caching. I/O transfers to/from the user task's memory are satisfied from the buffers.

4.2 Character Devices

A character device is unstructured. A file-system cannot reside on a character device. Examples are terminals and printers. In character I/O, data transfer requests occur in 'n' bytes between sections of memory and the device. Hence, "character" is synonymous to "byte." One must realize that there is minimal operating system involvement in data transfer as it is a private transaction between a user task and the device driver (see figure 2).

4.3 Combination Devices

Some devices can be accessed and treated as a block or character device. For example, the disk interface can be accessed either as a block or character I/O

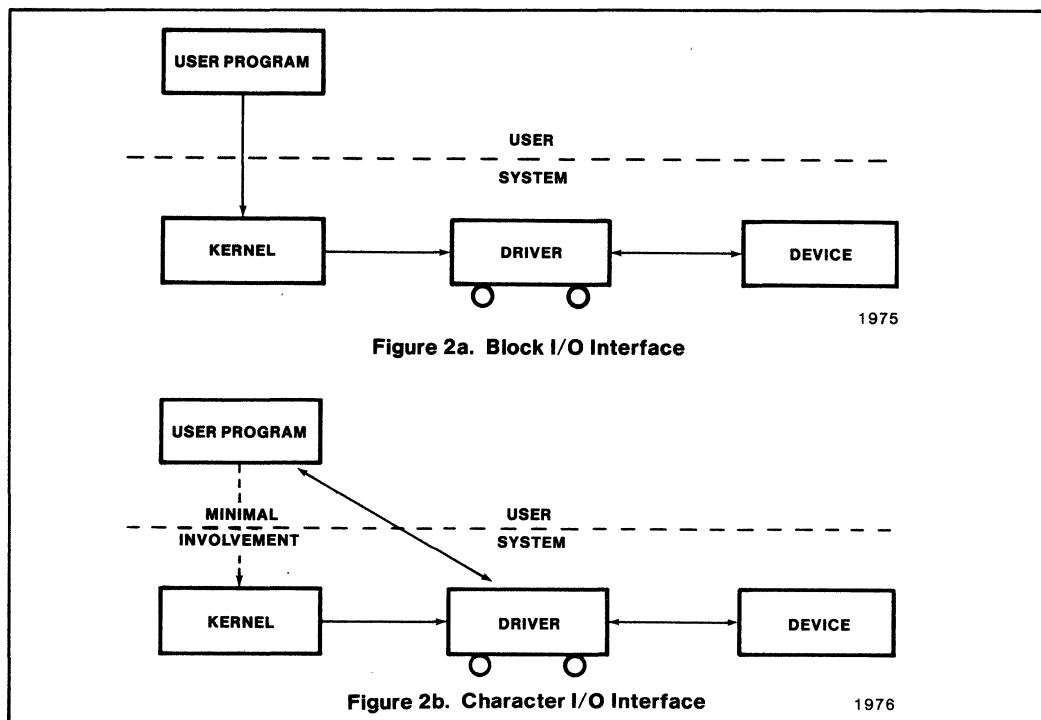


Figure 2a and 2b. Block and Character Interfaces

device. The character device permits direct i/o transfers between user memory and device. This mode of transfer is called RAW I/O. Raw I/O is very useful when direct disk-to-disk copy is necessary. The kernel buffers and file system are bypassed in this operation. Routines such as **dump**, **dd**, **fsck** are examples of such raw operations.

5.0 DEVICES VIEWED FROM THE USER INTERFACE

How are drivers in the XENIX Operating System identified? In the XENIX user interface, a directory called **/dev** exists for the purpose of holding all relevant device driver interfaces. XENIX is a file-oriented operating system and treats all devices as files. As files are accessible by the file-system, so are devices.

An **ls -l** (directory listing) of the **/dev** directory is shown in figure 3.

Since devices are accessible as files, data can be sent to them with:

```
echo "talk to me" > /dev/ttya0
```

Devices can be opened, updated and closed via the filesystem. **/dev/ttya0** is some-user's terminal. Files that identify devices are called device special files as

they provide the hook to the drivers from the file-system. Enforcing device independence where all devices are files in XENIX permits tremendous flexibility and uniformity in the XENIX Operating System. Device special files are identified to the kernel as a 16-bit integer value. This 16-bit value is composed of two other values, which are the device's major and minor numbers. The high-order 8-bits form the major number and the low-order form the minor number. XENIX provides two macros — **major(dev)/minor(dev)** to decode these values from the device file. This **<major,minor>** number pair is used, whenever the device is referenced, to identify the relevant device driver. XENIX is internally very table(array) oriented. These numbers are indices to an array of possible drivers.

Figure 3 shows the **<major,minor>** number-pair for each device. Also note that the leading "c" and "b" characters first on each line denote the character and block files respectively. To see the connection between major number/minor number and drivers, take a look at the **c.c** file (appendix A). **c.c** is created in the configuration process. The following data-structures in **c.c** are created to maintain the relationship between drivers and the major/minor numbers

(identifying the device special file). The data structures are:

- 1) **dinitswl |** is a vector of device-initialization procedures. The procedures mentioned in this vector are called during system initialization to initialize devices.
- 2) **bdevswl |** is the table of block-device interfaces. The index to a driver in this table is the major device number of the Block interface for the device.
- 3) **cdevswl |** is the table of character-device interfaces. The index to the driver in this table is the major device number of the character interface to the device. Note that major numbers do not overlap for block/character devices unless the same device is represented as a block and a character device.
- 4) **vecintswl |** is the table of interrupt procedures. This table contains one entry per supported interrupt level of the cascaded 8259A Programmable Interrupt Controller. The index represents the interrupt level.

```
total 11
crw --w--w- 1 root 4, 1 Feb 1 13:13 console
brw ----- 1 root 0, 9 Jan 23 17:47 df0
brw ----- 1 root 0, 13 Jan 20 09:47 dxfo
brw ----- 1 root 0, 8 Jan 20 09:46 fo
crw ----- 1 sysinfo 2, 1 Jan 20 09:45 kmem
crw --w--w- 1 root 3, 7 Feb 1 14:24 Ip
crw ----- 1 sysinfo 2, 0 Jan 20 09:45 mem
crw -rw-rw- 1 root 2, 2 Feb 1 19:30 null
crw ----- 1 root 0, 9 Feb 1 07:32 rdf0
crw ----- 1 root 0, 13 Jan 20 09:47 rdxfo
brw ----- 2 sysinfo 0, 1 Jan 30 16:40 root
crw ----- 2 sysinfo 0, 1 Jan 20 09:45 rroot
crw ----- 2 sysinfo 0, 3 Jan 30 16:42 rusr
crw ----- 2 sysinfo 0, 1 Jan 20 09:45 rw0a
crw ----- 2 root 0, 2 Jan 20 09:45 rw0b
crw ----- 2 sysinfo 0, 3 Jan 30 16:42 rw0c
crw ----- 1 root 0, 0 Jan 20 09:45 rw0t0
crw ----- 1 root 0, 12 Jan 20 09:47 rxf0
crw -rw-rw- 1 root 6, 0 Jan 20 09:45 tty
crw -w--w- 1 root 3, 0 Feb 1 17:35 ttya0
crw --w--w- 1 nair 3, 1 Feb 1 19:36 ttya1
crw -rw-rw- 1 root 3, 2 Feb 1 08:26 ttya2
crw --w--w- 1 root 3, 3 Feb 1 07:53 ttya3
crw -rw-rw- 1 root 4, 0 Jan 20 09:45 ttyf0
brw ----- 2 sysinfo 0, 3 Jan 24 16:51 usr
brw ----- 2 sysinfo 0, 1 Jan 30 16:40 w0a
brw ----- 2 sysinfo 0, 2 Jan 20 09:45 w0b
brw ----- 2 sysinfo 0, 3 Jan 24 16:51 w0c
brw ----- 1 root 0, 0 Jan 20 09:45 w0t0
```

Figure 3. The ls Listing

Only an overview of configuration details will be covered in this write-up. Knowledge of the configuration process is not needed to write a device driver. Figure

4 illustrates the file and directory structure maintaining drivers and the configuration files/shell scripts. Basically, there are three interesting files:

- 1) **xenixconf**: edit this file to describe the configuration to be built (see Appendix B)
- 2) **master**: contains a master copy of the configuration information (Appendix C)
- 3) **c.c**: generated from the above two using a program "config."

config
i.e master + xenixconf =====> c.c

To configure:

- 1) edit master
- 2) edit xenixconf
- 3) in **/sys/conf** run **MAKEXENIX**

The rest is automatic. Appendix B and C give examples of **xenixconf** and **master**. There will be no further discussion of the configuration details which do not contribute to learning how to write device drivers in XENIX. This overview was meant to describe how **c.c** is created. However, before attempting to write drivers, one must have a minimal understanding of the operating environment surrounding device drivers. This is the topic of discussion that follows.

6.0 THE ENVIRONMENT

The environment of any driver is, of course, the operating system, the device and the user-programs (processes) that communicate with the driver. This section concentrates on the kernel's view of the process. A task or process to the operating system is defined as:

- 1) the existence of an element/structure in the **proc table**. The **proc table** contains details of all processes in the system.
- 2) the existence of a **per-process data area (u-area)** representing it in the kernel. Any process image contains this special area that is copied into the kernel data space when it is active. This area identifies the process and hold parameters of the process.

A **proc table** entry and its corresponding **u-structure**, defines the state of a process at any instance during its birth(fork), lifetime and death(exit). The **proc table** entry is that part of a process that must always remain in memory for process communication and restart capability. The **u-structure** is that part of the process which can be swapped out onto disk, along with the per-process data segment, at times when the process is not in a runnable state.

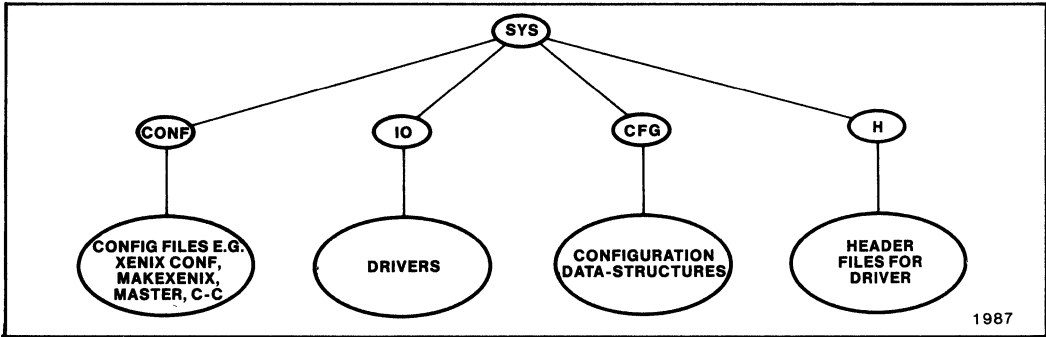


Figure 4. The Configuration Files/Directories

6.1 The u-area

The XENIX operating system does not differentiate a kernel process from a user process. Processes can run in either kernel mode, using system services and privileges (access to i/o drivers, `u_area`) or in user mode, where user-program code is executing. The operating system manages the relevant user-programs using a per-process data area, called the **u-area**. The **u-area** contains pertinent information such as system-stack information, preserved registers and I/O parameters used for data transfer. Throughout the discussion on device drivers, there will be mention of information in the **u-area**. To recap, the **u-area** is that space held in the kernel to maintain information about processes that run on the CPU. Every process has a **u-area** that is made up of a detailed structure, called the **u-structure**. This structure is multiplexed in the kernel i.e. is swapped in and out with the process and contains considerable information about the process such as:

- system call arguments
- process sizes
- registers saved
- error information
- I/O information

Input/output parameters held here are:

- `u.u_error` - error information, 0 means no error
- `u.u_base` - starting user transfer address
- `u.u_count` - bytes to transfer
- `u.u_segflg` - flag telling if transfer is to/from user data space (0) or system memory (1)
- `u.u_offset` - offset in file for I/O

The driver will receive other information too, such as, the target device, the size of the job and the buffer address in the task's memory. For block devices, only `u.u_error` is updated. For character drivers, all other parameters are used. The **u-area** should not be accessed by the running driver at all times. This is because events occur in any operating system that are either:

- 1) Synchronous, as in normal code executing in user space

- 2) Asynchronous, as when an interrupt occurs and the interrupt service routine is called for the device.

Synchronous activities happen as the CPU permits users to run their code. When their code makes requests to the system resources, they are subject to be swapped out of memory. Once the device performs these requests, it interrupts the processor asynchronously. The Operating System then calls on certain routines, called interrupt service, routines, to perform actions that follow the device's completion of its requested job.

The **u-area**, being part of the process, may have been swapped out after resource requests were made. Hence, the asynchronous portions of code called on an interrupt cannot access this area. The terms "task-time execution" and "interrupt-time execution" are used to differentiate times it can and cannot be accessed. Obviously, the device driver must contain routines that are called due to asynchronous and synchronous events.

6.2 Task-time Execution vs. Interrupt-time Execution

As mentioned, the XENIX Operating System does not concern itself with whether tasks are running in user or kernel mode. Hence, there may be several tasks contending for system resources that are non-I/O related. At task-time, tasks are executing user or system code. Their **u-area** may be used whenever the process executes system code i.e. makes a system call and the kernel uses this area for stack and parameter storage. This **u-area** is resident when the process is running and, therefore, can be addressed by the driver for data transfer. The user space is addressable at this time and information about the running process can be placed in this area by the driver.

Contrary to this, during interrupt-time execution, the device has interrupted the CPU. At this time, the running task may not be the task that requested

I/O to/from the device. Usually, the task-time portion of the driver has exited after the I/O request and that process may have been swapped out. Hence, the **u-area** and the user data space cannot safely be accessed or used at interrupt time by the interrupt service routine. However, the interrupt service routine gets relevant information (about what to do) from the task-time portion of the driver via static variables. Obviously, from the nature of task-management, interrupt routines are limited in behavior and they cannot make assumptions about the state of the system or the presence of tasks/data in the system. Figure 5a illustrates the distinction between the task-time portion and the interrupt-time portion of any driver that has to evidently cater and respect the realities of task vs. interrupt-time execution.

Task and interrupt-time routines are separate but they may call common data-manipulating routines.

One such routine is called by both sides and manipulates common buffers used (will be discussed later). There is a mechanism used for mutually excluding an update attempt of one routine from another. This mutual exclusion technique is required as the asynchronous portion of the driver may be removing an element from a list, for example, while the synchronous portion of the same driver may be placing an element onto the list. The technique basically deals with the interrupt structure and priorities of the system. For further details on the interrupt mapping, see Interrupt Mapping (Appendix D).

The operating environment, and the nature of tasks have been discussed only briefly as the device driver is the main focus of this application note. To bring together the basic components of the roadmap (Section 3.0) i.e. how the process, the kernel and the driver communicate, a brief description of the basic I/O path through the system follows.

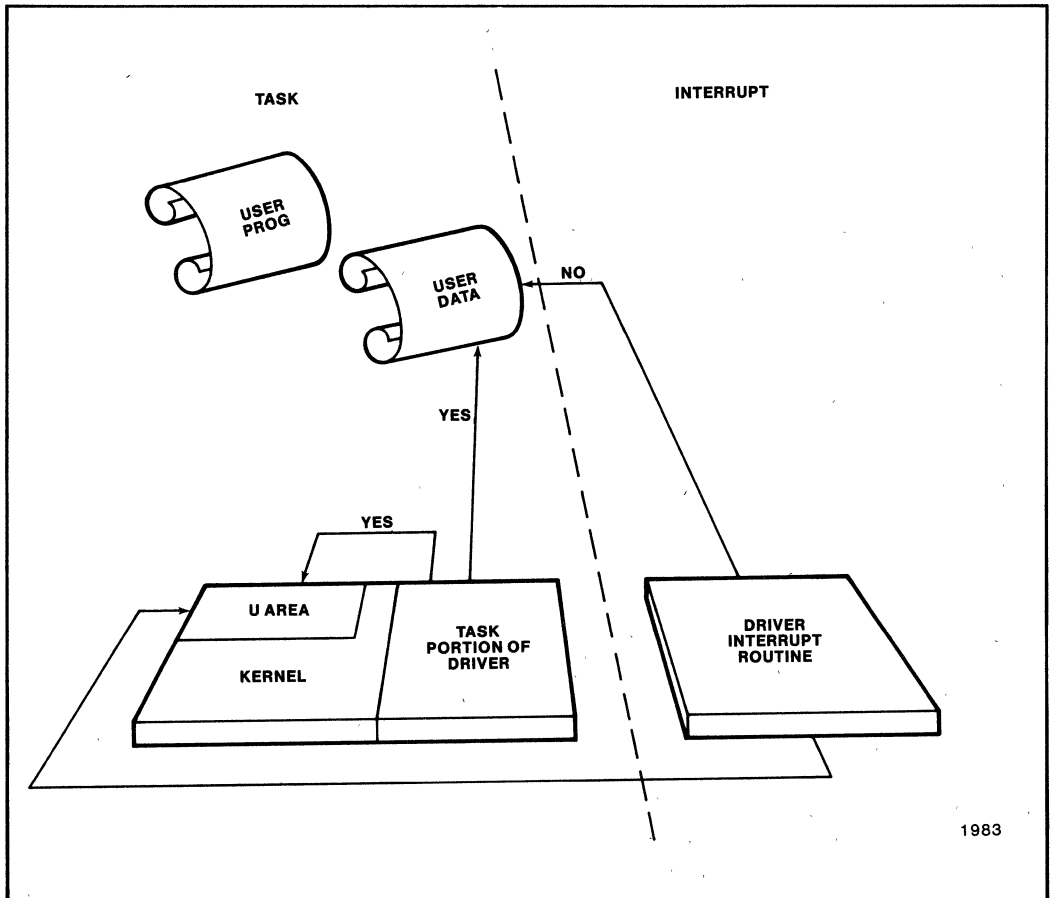


Figure 5a. Task vs. Interrupt Time Execution

6.3 I/O Path Through the System

Thus far, the driver interfaces have yet to be discussed. Knowing the environment a driver exists in is sufficient for understanding the I/O path through the kernel when I/O system calls are made. This will explain how the components of the roadmap (Section 3.0) interact. To illustrate the way all these device tables and system calls relate, the following calling sequences are presented. Details have been avoided to simplify the concepts described.

```
/*
 * user program makes system call
 */
```

```
fd = open("/dev/ttyb1",1)
```

```
/*
 * the kernel reacts with
 */
```

- 1) The kernel entry routine gets the system-call trap, determines it's an "open" call and calls "open" in the kernel.
- 2) Open calls a procedure to parse the path-name (nami) and turn it into an inode. An inode is the structure that represents a file in the file-system.
- 3) Open notices that the inode represents a character special-file and not a normal disk file. Thus, it accessed the `cdevsw[]` table using the major number in the inode, and calls the device-drivers open procedure.
- 4) The driver's open procedure does whatever it needs to do i.e. opens the device. It sets `u.u_error` if an error occurs and returns.
- 5) If the driver did not declare an error, open allocates a file-descriptor in the user-process, and returns it.
- 6) If the driver returned an error, open simply passes it back to the user program.

```
/*
 * user program does
 */
```

```
read(fd, buf, 128);
```

```
/*
 * kernel does
 */
```

- 1) The kernel gets the system-call trap, determines it is a read call and calls read in the kernel.

- 2) Read uses the "fd" argument and determines which inode it represents.
- 3) The inode indicates it is a character special-file. Thus, read uses the major device number stored in the inode to access the `cdevsw[]` table and calls the drivers read routine.
- 4) The drivers read routine does whatever it needs to do transferring data to the user's buffer. If any errors occur, it sets `u.u_error`. In any case, the driver eventually returns after having transferred the data.
- 5) The read routine returns to the kernel, which returns to the user program.

The I/O path will be elaborated on with details on driver functions with respect to block I/O. For the time being, the above sequence should be used as a template.

7.0 THE ANATOMY OF THE I/O SYSTEM

This discussion covers how the kernel and the driver communicate (See Section 3.0 Roadmap) and can be best broken down into two stages. They are:

- 1) the block interface
- 2) the character interface

The following discussions go into much detail on how the buffer schemes for block and character I/O devices work. These details are not needed for device driver writing. However, knowledge of the kernel buffers for block I/O devices and the character lists for character I/O devices will be useful when debugging begins. This knowledge will help the device driver writer to understand the inner workings of the driver he/she is writing.

7.1 The Block Interface

Much has been discussed about what block I/O is. Block I/O transfers require the kernel's intervention when they occur. All block transfers require the use of I/O buffers. These buffers are used as a temporary storage area for caching and blocking/deblocking. Usually, data transfer occurs between user space and devices via these system buffers (see figure 5b).

Each buffer is `BSIZE` bytes long and has a buffer header corresponding to it. `BSIZE` and other system level constants are defined in `param.h` (Appendix E) The header file `buf.h` defines this buffer header structure (Appendix F). `Buf.h` also describes how these kernel buffers are structured. Although it is not necessary to understand all fields in the buffer header, some fields must be noted:

```
b_dev-   device number
b_blkno- block for the transfer
b_bcount- bytes to transfer
```

b_cylin- cylinder number
 b_addr- address to be transferred from/to
 b_flags- nature of transfer i.e. B_READ

7.1.1 THE SYSTEM BUFFERS

Consider a list, called the freelist, that is initialized to be a circular doubly-linked list of buffer-headers. Each header in the list has a pointer to its own BSIZE buffer. Figure 6a shows this list with forward and backward pointers **av_forw** and **av_back**. Consider another list called the device list. This list hangs off each device driver. The buffers forming these device lists are device request that are waiting to be serviced. At start state these lists for each device are empty and their headers point to themselves with **b_forw** and **b_back**. (figure 6b). Each device has its own queue structure that exists to schedule I/O requests. This queue is structured

exactly like the freelist, i.e. is doubly linked and circular. The head of this queue called the static buffer header, is a buffer-header just like all the others. However, some irrelevant fields that it holds (used by other headers in the queue) are redeclared (aliased). These redeclarations are found in **buf.h**. Here, **av_forw**, **av_back**, **b_forw**, **b_back** used on the same buffer headers but form two concurrent lists. Remember, the freelist is the master list. **b_flags** determines, for the enquirer, if the buffer is **BUSY**, **WAITING** and the like. Details of these flags are found in **buf.h** also. (for further details about flags see ref 1).

The configuration file **c.c** (appendix A) holds the data structure **bdevsw[]** which is the table of device driver routines indexed by their major number. The last element in this structure for each device is **ixx-tab** (see naming convention — Appendix G) where

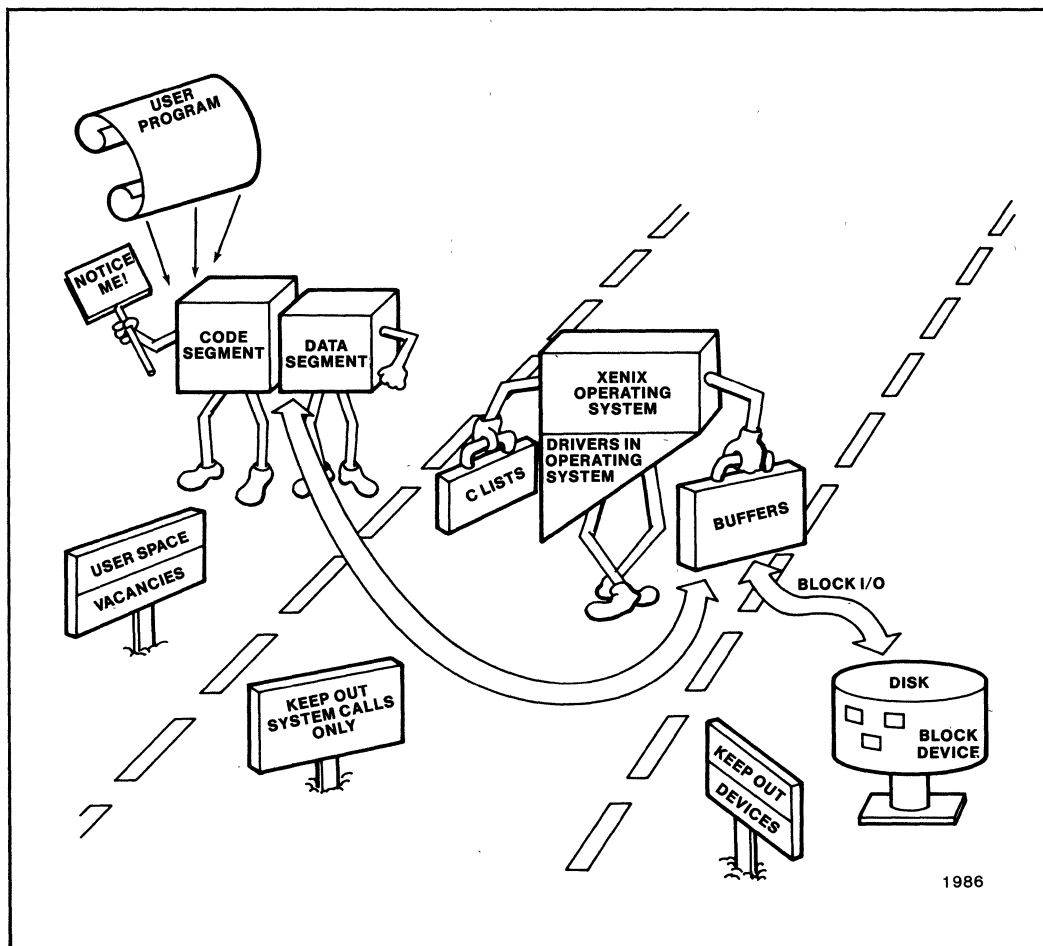


Figure 5b. The System Buffers

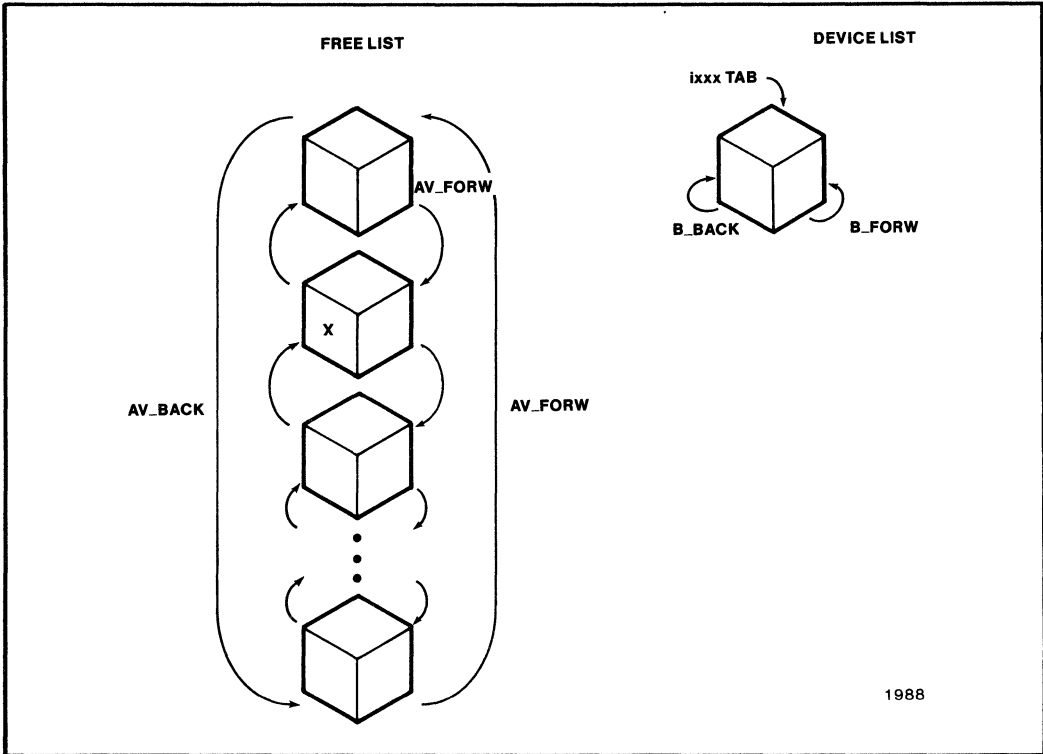


Figure 6a. Start State

xxx is the device identification number e.g. 215, 544. **xxxtab** is the pointer to the system buffers for that device i.e. **xxxtab** is the static buffer header address for the queue of device request. Remember that at start state, these queues are empty.

When a user process requests a write (as a first request) a buffer header from the freelist is removed and placed on the device list. The **av**-pointers of the freelist are unused and **b**-pointers are now used in the device list. The **av**-pointers are re-declared to be **b_actf** and **b_actl** respectively. These links are used to form a new list using the same buffer headers that are in the device queue. Think of two lists i.e. the device list (with items plucked from the freelist) using all four pointers, thus being members of two lists superimposed (see figure 6b). But why two lists superimposed? Well, at first there were two lists — the freelist and the device specific list (of course, each device has one device specific list but for this discussion let it suffice to have only one device).

Upon a request, the kernel takes a buffer header from the freelist and places it on the device queue. The kernel also places details of the write request into the buffer header fields. As the kernel manages

these buffers, the address of this buffer header is given to the device driver. The driver calls a routine **disksort()** which takes the buffer header and orders it in the device specific queue using not the **b**-pointers but the unused **av**-pointers. Note that the buffer header is already in the device queue with the **b_back/b_forw** pointers active. **Disksort()** orders these requests into cylinder order on insert using the **av_forw** (**b_actf**) and **av_back** (**b_actl**) pointers forming a new list of optimally ordered requests (see figure 7a).

There are three lists formed here:

- 1) the device specific list hanging off the device driver (`cdevsw == > xxxtab`)
- 2) the active list that uses the same elements in the device list but uses the **av**-pointers ordered by **disksort()**
- 3) the freelist of buffer headers using the **av**-pointers before redeclaration.

Thus, the write request is then ordered into the active list for the device.

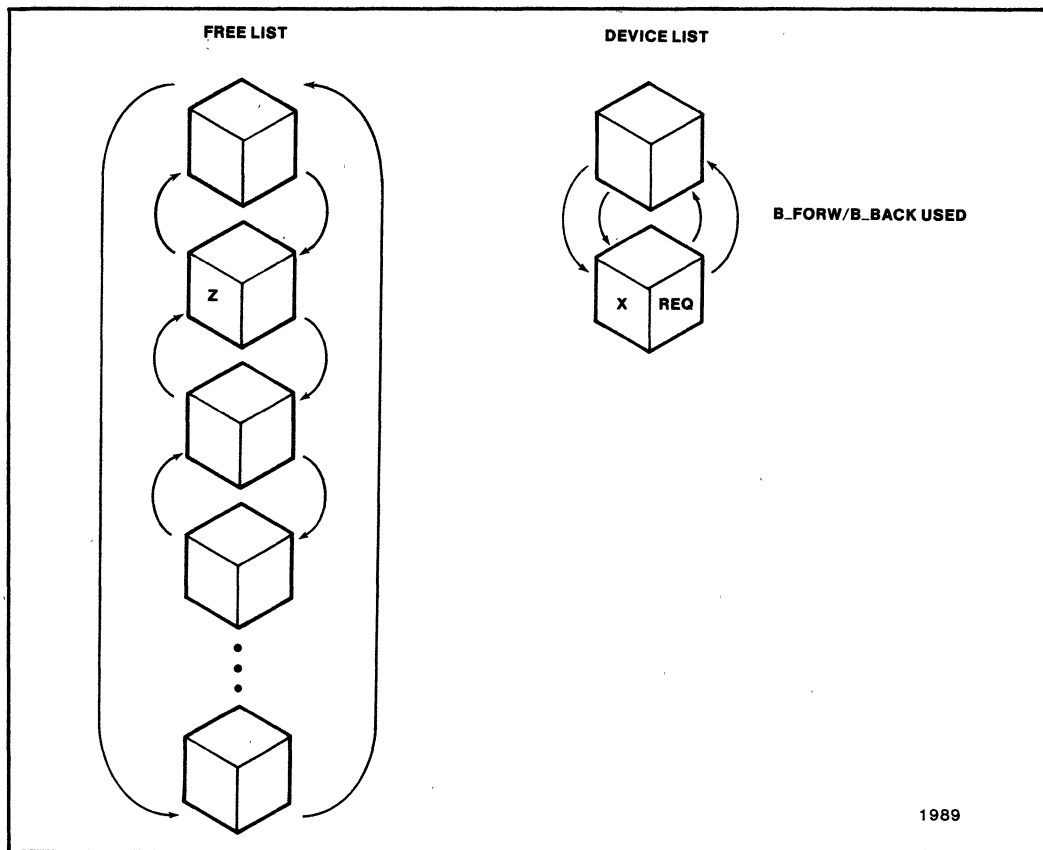


Figure 6b. Buffer placed in device list by kernel

The task-time portion of the driver has been running all this time and this task portion returns after it has made the request. The user process sleeps on the event that the device will complete the requested transaction. In other words, the task sleeps on the buffer-header address. The `sleep()` instruction permits a context switch within the operating system. The operating system can then schedule other tasks for CPU attention.

The device, on completion of the requested write will interrupt the CPU. The interrupt, through the XENIX interrupt scheme, will invoke the respective interrupt service routine which is in the interrupt-time portion of the driver. This is done asynchronously.

This interrupt service routine will awaken all processes associated with the event using `iodone()` (see section 7.1.2). Note that for write, the task-time portion was responsible for the transfer of data from the

user space to the buffers while the interrupt service routine is invoked when the data in buffers are written to disk. The relevant process, on awakening is re-scheduled by the scheduler to run as soon as possible. The interrupt service routine then checks to see if there are other pending requests on the device specific queue. If not empty, it instigates the next transaction from the next buffer on the device queues (by following the `av_pointers` now declared `b_actf/b_actl`).

When interrupt routines are alive and running, mutual exclusion (mutex) is ensured by raising the priority level of the running task in the CPU and on completion, lowering the priority level. Details of this technique is found in Interrupt Mapping (Appendix D). This technique locks out interrupts of an equal and higher level than itself for a short period when shared data structures are manipulated. The interrupt service routine may be looking for the next request on the device queue, when the task-portion

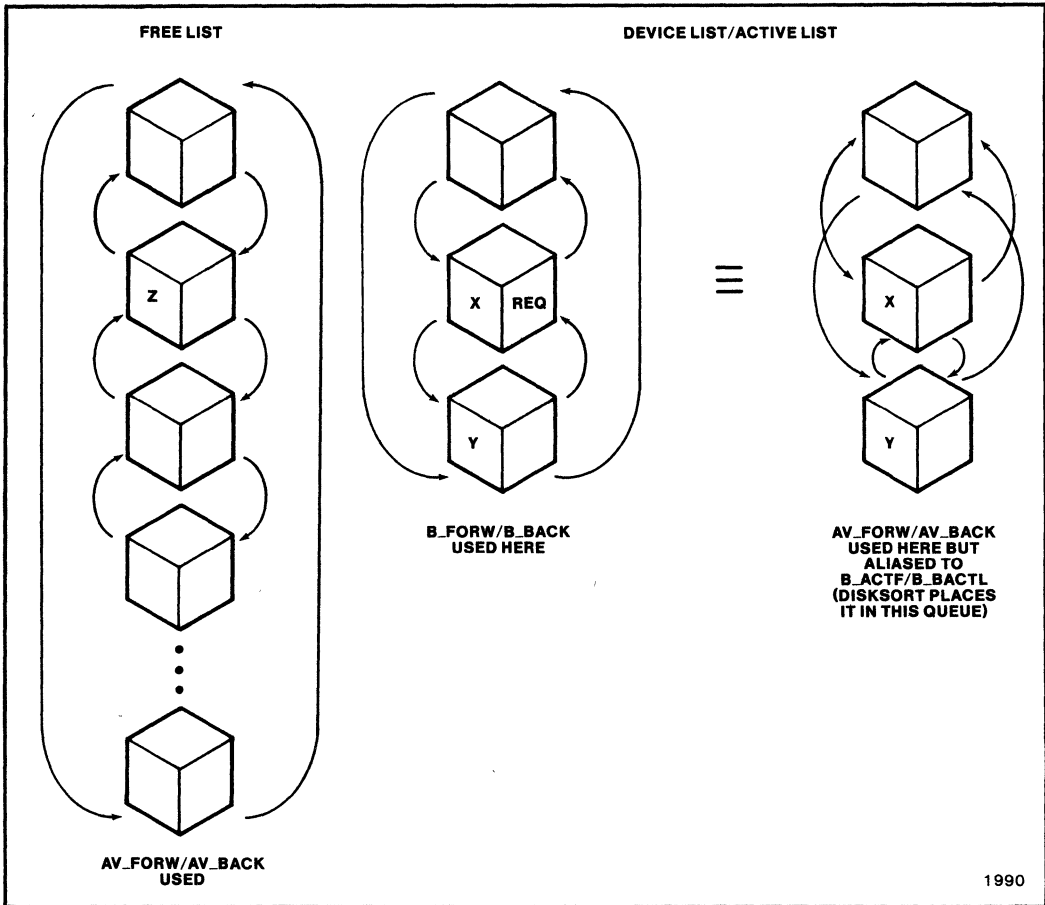


Figure 7a. Another Request Placed in Device List and also Placed in Active List by Driver

of the driver may be calling `disksort()` that orders the device list or when the kernel is placing the buffer header with a request onto the same list.

On completion, the buffer header is marked as IO complete and this header is released from the device specific queue and placed at the end of the freelist using the `av_pointers`. Note that the link to the active list is broken (reusing the `av_pointers`). Remember also, that `b_forw/b_back` pointers are still maintaining membership with the device specific list even when the buffer is now a member of the freelist pool. Here is where the buffer is placed in cache.

As shown in figure 7b, the buffer request X is placed on the freelist using the `av_pointers` but is still a leading member of the device list. On the next request,

the kernel will search the device queue (not the active list) from the beginning. As you can see in figure 7b, the list is diverted into another list of available but recently used buffers by following `b_forw/b_back` pointers. Here caching occurs as most recent transactions can be checked for repetition. Note that the X buffer will bubble up the freelist until it will be re-used for other transactions. Until then it is cached.

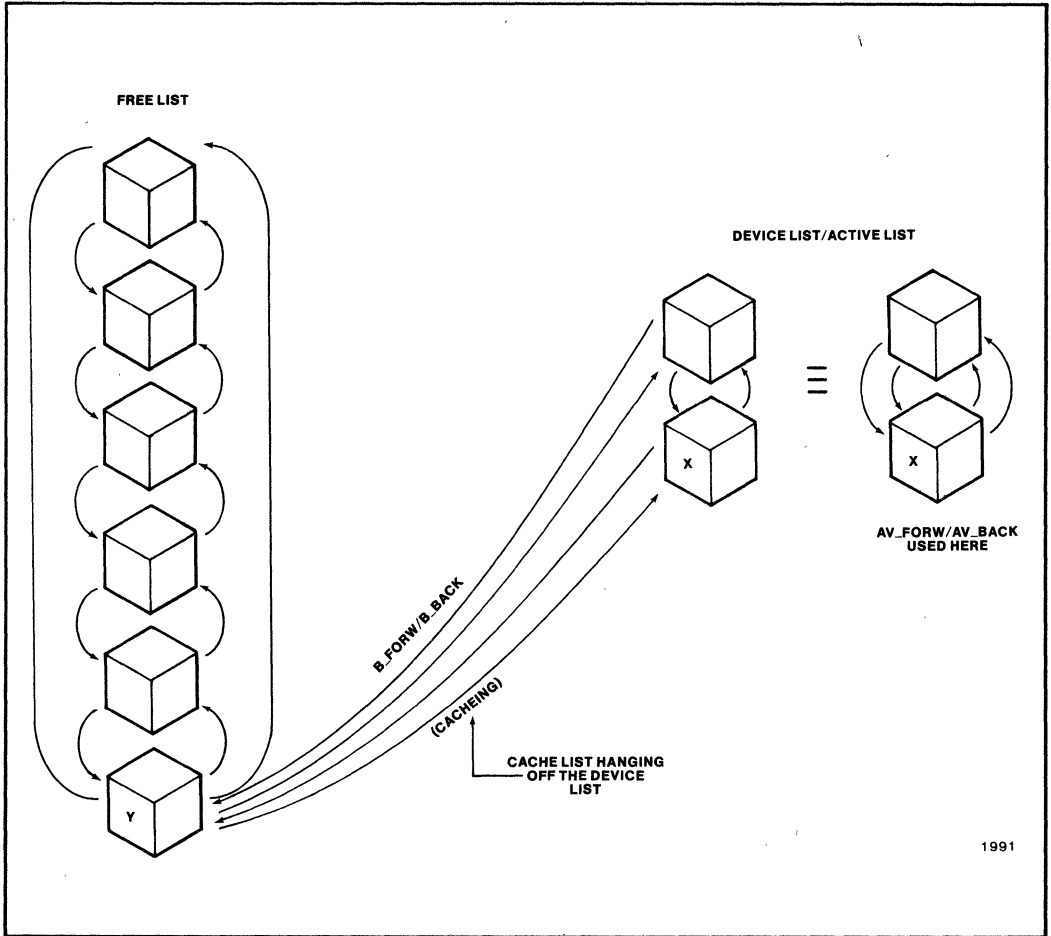


Figure 7b. When Request Granted the Buffer is Returned to Free List but is Still a Member of the Device List and is cached. No Longer in Active List.

For a review of the block drivers duties, in an I/O request, consider a read request and the following point-by-point discussion:

```

user calls read
READ
{
    . maps file fd to inode
    . calls readi
readi (read inode)
    . determines block to be read (bmap()
      called)
    . searches buffer list i.e follows av_pointers
      matching block # to last read/write
    . if cached, copy into user space and return
    . if not cached, flush first buffer from freelist

```

```

    . fill out buffer header
    . with device #
    . block #
    . call task-time portion of device driver
    . on return ,
    . sleep on buffer header waiting for
      wakeup
    . copy into user space

```

```

} The device driver does
{
    issues or queues request to the device
    interrupt handler wakes processes when on I/O
    complete
    when awakened return to readi
}

```

NOTE: the above is only a specific example and a brief one. For example, details of the support routines checking whether it is block/character read was not mentioned.

So far, elements of the I/O system anatomy discussed are invisible to the device driver writer. He/She need not know all of the intricacies of buffer management but there is a need to fully comprehend the routines to be written, the system calls available and the operating environment of device drivers. In accordance with this methodology, block drivers have, in their grasp, many powerful and consistent system calls available from the kernel. They can be called "driver support routines" because some of them are available to character drivers also.

7.1.2 DRIVER SUPPORT ROUTINES

The following list is an informal collection of possible support routines used in block I/O drivers and by the kernel. The kernel deals with the declarations for the arguments and on many situations, places values into these arguments. This is because the kernel is allowed access to most of these arguments and knows their values. Some of these calls are also used in character I/O transfers and will be referenced in the character interface description:

physio(strat, bp, dev, rw)

where **strat** is the address of the strategy routine (a driver procedure) which is the routine that performs read, writes and starts up the device. This routine will be discussed in section 7.1.3. It takes, as an argument, a pointer to a buffer-header (**bp**) which holds detailed information about the transfer.

where **dev** is the relevant device that character I/O is to occur to (the <major, minor>) pair.

where **rw** is a flag indicating the nature of the transfer (B_READ, B_WRITE in param.h)

Physio is used by block devices which can be treated as character devices. In this case, transfer between user space and device space is done directly (direct I/O) with no intervening buffers. Remember that for this to be successful, physical I/O must occur when the instigating user process is in memory and active (and not swapped out). **physio** is a routine called by a driver for physical I/O on a device. Among other functions, **physio** checks the validity of the transfer request. The buffer header pointer that is passed to **physio** does not hold a buffer address but the address of the physical location in memory or the device, depending on the direction of the transfer. Physical I/O is a contiguous transfer feature that is used in **tar**, **fsck** and **dd**, among other utilities.

disksort(&xxxtab, bp);

```
struct buff xxxtab /* static buffer header */
struct buf *bp /* new buff header to be inserted */
```

disksort() is the assist routine ingredient to the buffering/cacheing protocol as it manages the active request queue. It takes, as arguments, the address of the pointer to the static buffer header for the specific device. The active request queue holds all requests for the device. **bp** is the pointer to the new buffer header that holds another request on the device. **disksort()** inserts this request in the queue of requests in cylinder/block order to minimize disk accesses.

iodone(bp)

```
struct buf *bp /* header of completed request */
```

is a clean-up routine that informs the process that the request made is complete. **iodone()** is called by an interrupt service routine and issues a **wakeup()** on the relevant event i.e. the buffer pointer **bp**. The routine pulls the request off the device specific queue and places it onto the free list.

sleep(bp, pri), wakeup(bp) and iowait(bp)

```
struct buf *bp
```

Process/task synchronization is a required feature in the multi-user/multi-tasking XENIX Operating System. Processes have to be informed when to wait for the system's shared resources. The XENIX kernel provides two routines, **sleep()**/**wakeup()**, for this purpose. **Sleep()** takes, as argument, a key or event that the calling process waits on. This key or event is nothing more than a bit-pattern. The key or event in this case is conveniently the buffer-header pointer that the task-time portion of a driver is using for the transfer request. To re-iterate, the task-time portion of the driver, when making an I/O request, may have to wait after the request is made until the actual I/O is completed. The waiting is begun by a system routine called **iowait()** which is called by the kernel and **physio()** which is called by a driver in direct physical I/O (for magnetic tape drivers when driver calls **strat()** and waits for I/O to complete).

PRI, the second argument, is the priority at which the process is to sleep. The sleep priority is higher than what a user process can acquire. When the **wakeup()** occurs, the process continues at the sleep priority thus giving it a higher probability of being scheduled earlier.

Priorities range from 0 to 127. Priorities are not bound by rules but the priority PZERO is used to differentiate two main situations that may occur. If a priority < PZERO is set for the sleep, no signal can wake up the process. Hence, the process will be awakened with an `iodone()` in the future. With a priority > PZERO, signals will awaken the process even before `iodone()`. Also, smaller numerical priorities mean higher priority levels. The safer technique is therefore to place 'sleep' in a loop that tests if the buffer is available for continuation. Hence, if I/O is complete and the buffer freed, the process is awakened legally. Otherwise, continue to sleep.

Also, when the device returns an interrupt `iodone()` is called which calls `wakeup()` that sets the event (buffer header) and induces life to all processes waiting on that event — not just the "first" one. Processes must therefore ensure that they are awake for the correct reason. One way to do this is for the task-time portion to check a predetermined static memory location for instructions left by the interrupt-time portion of the driver on completion.

`timeout(func, arg, time)`

```
int (*func)(); /* function called as argument */
int arg;
int time;
```

Arranges for `func` to be called with argument `arg` in `time` clock-ticks. `timeout()` is a facility that runs a procedure after `n` clockticks. The procedure is called at clock interrupt time and, hence, conforms to the interrupt-time rules. Used for character I/O also.

`iomove(addr, count, flag)`

Used for large data transfers. `addr` gives you information on where in kernel the transfer is to occur. `count` signifies the size of the transfer in bytes. `Flag` tells us if it is a B_READ/B_WRITE. The other transfer address is found in the processes' `u_struct` as `u.u_base`.

7.1.3 BLOCK DEVICE DRIVER ROUTINES

Briefly, a block device driver is composed of one or more of the following routines:

`init()` is a routine called very early during system initialization (at boot time) to initialize the device. It is called with no parameters and returns no values. Interrupts are disabled at this time and the existence of the device is verified. It prints appropriate messages stating that the device is/is not found and remembers if the device is alive (sets a flag). This routine is called once.

`open()` is a routine that opens the device. Prepare it for activity and is called on every open of the device.

```
Its parameters are
dev_t dev;
int flag;
```

`open(dev, flag)` is the calling sequence where `dev` is <major, minor> device number and `flag` is either B_READ or B_WRITE. The program validates the device number and sets-up initial parameters. Any errors detected is recorded in the `u.u_error`. The presence of the device is verified before the open occurs.

`close(dev)`

is called on the final close of the device. The close routine flushes pending transfers in device specific queue and sets flags that remember that the device is closed.

`strat(bp)`

```
struct buf *bp; /* pointer to buffer header */
```

Called by the kernel in response to the user program instigating a read/write request. `strat` is "strategy." Inserts a request on the queue of device requests. The kernel provides a buffer header to the routine and it validates the header to ensure that it has all the necessary information (e.g B_READ). The driver routine calls `start()` and `disksort()`

`intr(level)`

```
int level;
```

The interrupt routine is called by the kernel when the device interrupts. This routine is called when the device is moving from an active state to an idle state. If the device is active on entry to the interrupt service routine, the interrupt service routine was awakened by a spurious interrupt. If not, the device state is changed to idle. In this state, the previous request was satisfied and an `iodone` should be called. The momentum is continued to keep the device busy by calling the `start()` routine if other requests are pending on the request queue (device specific queue).

`start()`

This routine functions to move the device from an idle state to an active state i.e. it talks with device. It is called when the device is idle or a request is queued. It interprets the information on the buffer header at the beginning of the queue of device requests and sends commands to the controller.

These routines form a file called `ixxx.c` where `xxx` is the numerical representation of the device (see Naming Conventions - Appendix G). This file should reside in `/sys/io` directory. Conventionally, `cxxx.c` an adjoining file is also created to identify any data structure relevant for the main program. `cxxx.c` resides in the `/sys/cfg` directory. Finally, constants and `#defines` are found in a header file, generally "included" in `cxxx.c`, called `ixxx.h`. This file is created in `/sys/h`. Hence, a driver for the iSBC 254 Bubble Memory board should be composed of:

- i254.c main driver routines
- i254.h the header file
- c254.c the configuration data structures

With this brief description of the driver routines, a casual discussion of how these routines interact with each other and the kernel is a natural follow-up. Some reiteration of previous details is necessary to give an overall consistent discussion.

7.1.4 REVIEW

User requests to be performed on a device (usually on a file living on the device) are converted by the kernel to simple requests for I/O which are passed to the driver. The kernel does any blocking/deblocking and caching to minimize device accesses. `Strat()` and `Intr()` are the main routines required of a block device driver. The request which is passed to the `strat()` routine is passed in the form of a pointer to a buffer header. This header contains all the information necessary to perform the operation - `B_READ`, `B_WRITE`, device address to use e.g. which track and sector, and the address of the kernel buffer from which the data should be taken or into which the data should be placed. The buffer header points to `B_SIZE`'d buffers and the request will always be for `B_SIZE` operations.

Be sure to keep in perspective the level of software being discussed - the driver itself sees only requests for transfers to or from a physical block of the device - entities like file-structure, disk space allocation, or blocking/deblocking of small or large requests are all performed by higher level kernel software. All the device driver needs to do is examine a request, determine whether it is a read or write and perform the operation between the indicated memory address and the indicated block device.

7.1.5 STEPS TAKEN TO SATISFY REQUESTS

This discussion centers around the `strat()` and `intr()` routines. All requests are passed to the driver by the kernel by calling the `strat()` routine, with a single parameter - a pointer to a buffer header. As before, the header specifies the type of operation that is to be performed, the memory and device addresses to be used, and a field for recording the result of the operation after it has completed. The `strat()` routine places the incoming request on the linked list of active requests to be performed. If the device is currently busy performing a previously queued request, the `strat()` routine has finished its job and returns. If the device is idle (i.e. the request is the only one on the active list), the `strat()` routine must initialize the operation for the request. This typically involves loading parameters into a peripheral controller and initiating a command. At this point, the `strat()` routine has completed and returns.

After a command is started, it is typically a long-time (by cpu standards) until the request is completed and an interrupt occurs. The interrupt routine must field this interrupt and determine the reason for it. If it is the expected "operation-complete" interrupt, the interrupt routine should perform any operation needed to complete the transfer, then call the `iodone()` routine with a single parameter - the pointer to the buffer header of the request just completed. The `iodone()` performs some clean-up, notably waking up the process which was waiting for the I/O to complete. At this point, the interrupt routine may determine if other requests are waiting in the active request queue for the device, and if so, initiate the next one by calling the `start()` routine. Once done, the interrupt routine returns with its job done. The interrupt routine is a trigger that keeps firing-up new requests as they are discovered on the queue. Once the list is exhausted, the `intr()` routine returns without starting another request (none there to start) and the seed is lost and the sequence stops. The `strat()` routine must start another request to "prime the pump" and start the momentum again.

With this understanding of driver routines, a pseudocode example of the iSBC 254 Bubble Memory board driver will complete the discussion of block I/O device drivers. As mentioned, this section of block I/O is not the main thrust of the application note as emphasis has been placed on the character interface. This discussion will culminate in a pseudocode walkthrough.

7.1.6 ISBC® 254 BUBBLE MEMORY BOARD WALKTHROUGH

```

1 /*
2 * SBC 254 Bubble Memory board device driver. (Pseudo-code)
3 *
4 * - implements block and raw interfaces for an SBC 254 -1, -2, or -4.
5 * - always accesses all bubbles in parallel, meaning that there are
6 * always 2048 pages on the board, and the page size can be 64, 128,
7 * or 256 bytes (see c254.c)
8 * - will handle only one 254
9 * - uses DMA mode for bubble accesses
10 * - I/O base address and number of bubbles configurable in c254.c
11 *
12 *
13 */
14
15 #include "../h/param.h"
16 #include "../h/system.h"
17 #include "../h/buf.h"
18 #include "../h/conf.h"
19 #include "../h/dir.h"
20 #include "../h/user.h"
21 #include "../h/i254.h"
22
23
24 extern struct i254cfg i254cfg; /* see c254.c
25 * for values, i254.h for definition
26 */
27 struct buf i254tab; /* static buffer header */
28 struct buf i254rbuf; /* static buffer header for
29 rawinterface */
30 short i254alive, i254isopen; /* device existence, open flags */
31
32 /*
33 * i254init - called early in the system initialization - probes for
34 * 254 by resetting it and watching for appropriate reaction
35 */
36 i254init()
37 {
38
39 /*
40 * this is the first routine of the driver that will be called,
41 * it's a good time to clear the i254isopen flag
42 */
43
44 i254isopen = 0; /* 254 is closed */
45
46 /*
47 * more stuff to init the board and check status
48 */
49 }
50
51 /*
52 * i254open - checks for correct minor number(0), and existence of the
53 * board, and either allows or disallows the open
54 */
55
56 i254open(device, flag)
57 dev_t device; /* device number */
58 int flag; /* what kind of open (for reading, writing, etc.)

```

```

59     we'll ignore this */
60 {
61     if ((minor(dev) == 0) && (i254alive)) {
62         i254isopen = 1;      /* mark 254 as open */
63         return;
64     }
65     else{
66         u.u_error = ENXIO;
67         return;
68     }
69 }
70
71 /*
72 *i254strat - queues the I/O request and starts it if the device is idle
73 */
74 i254strat(bp)
75 struct    buf    *bp;
76
77 {
78     int    x, startpage, numpages, ppb;
79
80     /*
81     * first thing to do is check device is open; otherwise, allow
82     * no I/O
83     */
84
85     if (~i254isopen) {
86         bp->b_flags |= B_ERROR;
87         bp->b_error = ENXIO;
88         iodone(bp);      /* mark it done */
89         return;
90     }
91
92 /*
93 * convert the block number to a page number, and the number of
94 * blocks to number of pages, and the starting block to the starting
95 * page - these could be sped up with some appropriate shifts instead
96 * of * and / */
97 */
98
99     ppb = BSIZE / i254cfg.c_page_size; /* pages per block */
100     numpages = bp->b_bcount * ppb;      * number of pages */
101     startpage = bp->b_blkno * ppb;      /* 1st page of transfer */
102
103 /*
104 * Now check the requested operation for validity in terms of the
105 * the number of pages on the device.
106 * Here's the thinking:
107 * if request is READ on 1st block after the last block -> EOF
108 * "" "" WRITE "" "" "" "" "" -> error
109 * "" "" READ or WRITE 2 or more pages past the end -> error
110 * if request starts on valid page, but runs off end -> EOF
111 */
112
113     if (startpage > BUBPAGES) {
114         /* 2 or more after last page, so error */
115         bp->b_flags |= B_ERROR;
116         bp->b_flags = ENXIO;
117         iodone(bp);
118         return;

```

```

119     }
120     if (startpage == BUBPAGES) {
121         /* 1st block after last one */
122         if (bp->b_flags & B_READ)
123             bp->b_resid = bp->b_bcount; /* read, so just EOF */
124         else {
125             /* write, so error */
126             bp->b_flags |= B_ERROR;
127             bp->b_error = ENXIO;
128         }
129         iodone(bp);
130         return;
131     }
132     if ((startpage + numpages) > (BUBPAGES -1)) {
133         /* starting ok, but running off end */
134         bp->b_resid = bp->b_bcount;
135         iodone(bp);
136         return;
137     }
138
139     /* if we're here, request looks OK, so queue it and
140      * (if necessary) start it
141      */
142
143     bp->b_cylin = startpage; /* use page number as sort field */
144
145     /*
146      * Since we're about to play with the queue, and this can be
147      * accessed at any time via the interrupt handler, we need to
148      * shut down this interrupt level to provide mutex
149      */
150
151     x = SPL();
152     disksort(&i254tab, bp);
153     if (i254tab.b_active == IO_IDLE) i254start(bp); /* start if idle */
154     splx(x); /* reenale this interrupt level */
155 }
156 i254start(bp)
157 struct buf *bp;
158
159 /* this routine is the most device specific of all others.
160  The routine is called by both strat() and intr() at task-time
161  and interrupt time. It starts up the device if it is idle and
162  keeps the momentum going with other requests.
163  */
164
165 /*
166  *i254intr - interrupt handler-checks status of operation just completed
167  * and starts new operation if one is queued
168  */
169 i254intr(level)
170 int level;
171 {
172     short stat;
173
174     /*
175      * point to first buffer header in the active queue, making sure
176      * that it's actually pointing to a buffer header
177      */
178

```

```
179     if ((bp = i254tab.b_actf) == NULL) {
180         printf("No active buffer header, i254intr, level %d0,
181             level);
182         return;
183     }
184
185     /*
186     * clear the interrupt source and disable DMA
187     */
188
189     outb(i254cfg.c_base_port + BMCCMD, CLRINT);
190     outb(i254cfg.c_base_port + DMAMODE, DMADIS);
191
192     /*
193     * Now look at the status of the BMC to determine if this is the
194     * successful end of an operation. Report any errors encountered
195     */
196
197     stat = inb(i254cfg.c_base_port + STATUSPORT);
198     if (stat & BMCBUSY) {
199         printf("BMC still busy, i254intr, status = %d0, stat);
200         return;
201     }
202     if (stat & (BMCOPFAIL | BMCTIMERR | BMCUNCERR)) {
203 #ifdef VERBOSE
204     printf("Error, i254intr, status = %d0, stat);
205 #endif
206     /* call deerror here? */
207     bp->b_flags |= B_ERROR;
208     bp->b_error = EIO;
209 }
210     else if (stat & BMCCORERR)
211         printf("Cor error, i254intr, status = %d0, stat);
212
213     /*
214     * At this point we have determined that a legitimate bubble interrupt
215     * has occurred, and if there has been an error, it's recorded. Now
216     * we need to mark the operation as complete and start the next request
217     * in the queue (if there's one there).
218     */
219
220     i254tab.b_actf = bp->av_forw;
221     iodone(bp);
222     if ((bp = i254tab.b_actf) == NULL) {
223         i254tab.b_active = IO_IDLE;
224         return;
225     }
226
227     /*
228     * At this point, bp is pointing to the next request in the
229     * queue, so start it.
230     */
231
232     i254start(bp);
233 }
234
235 /*
236 * i254close - clears i254isopen flag
237 */
238
```

```

239  i254close(device)
240  dev_t device;
241  {
242      i254isopen = 0;
243  }
244
245  /*
246  * i254read - RAW interface read routine - calls physio
247  */
248
249  i254read(device)
250  dev_t device;
251  {
252      physio(i254strat, &i254rbuf, device, B_READ);
253  }
254
255  i254write(device)
256  dev_t device;
257  {
258      physio(i254strat, &i254rbuf, device, B_WRITE);
259  }

```

The **i254init()** routine (line 36) reports on the devices it expects to find and the ones it actually finds. The normal procedure is to give the device some command (usually a reset or initialization), then do a busy wait loop, waiting for a "sign-of-life" from the device. of course, the busy wait loop should have some other form of termination, so that a non-responsive device does not cause the system to hang at the point. Usually, a simple counter to limit the time spent in the wait loop is used. The **i254init()** routine (line 36) writes a message to the console, such as "SBC 254, port 0x040 found" or "SBC 254, port 0x040 NOT found." Also, it is usual to let the **i254init()** routine set a flag which indicates the presence or absence of the device. The **i254open()** routine (line 56) then checks this flag and will return an error if an open is attempted on a non-existent device.

The **i254open()** routine (line 56) is called upon each open on a device and provides an opportunity for doing any device set-up that is required. This could be the routine that spins-up a Winchester disk, or turns on the motor of a mini-floppy. As before, the **i254open()** routine should check for device existence flag, set by the **i254init()** routine, and return an error if necessary.

The **i254strat()** routine (line 72) is the routine which actually performs the bulk of the work in any block device driver. The **strat()** routine queues the new request onto the active queue (of the device specific queue). This queue is a linked list of buffer headers which contains data transfer parameters and are awaiting service. At the head of this list is a static buffer header that is called **ixxstab**, where **xxx** is the device handler prefix that is used on all the procedures composing the device driver (see Naming

Conventions - Appendix G) In this case "xxx" is "254." This header contains pointers which make forward and backward links to other buffer headers in the list (remember **b_forw/b_back**). Besides the list of pending device requests, the buffer headers can also be linked onto a free-list, which contains all the buffer headers which are not set-up for an operation using **av_forw/av_back**. A third list contains the buffer headers that are currently available or have been used(cache list). This list is the cache mechanism which the kernel uses by looking down to see if perhaps it can re-use the data from another request and avoid physically accessing the device again. It is managed with the **av_forw/av_back** pointers re-declared. Details of these buffers and how they are managed are found in Block I/O Interface (Section 7.1).

The actual manipulation of the device list and the free list is handled completely by the kernel with its higher level routines and does not need to be the concern of the device driver. The active list (device specific list with **av_pointers**) is managed by the device driver, but usually can be handled by a couple of provided subroutines, so that the driver does not have to explicitly deal with the pointers which structure the list. **Disksort()** is the routine normally used to place a buffer header onto the active list. Remember the active list is a reordered version of the device list. Besides the actual mechanics of updating the list, **disksort()** also provides a sorting facility which orders the active requests by cylinder number, to minimize seek time between requests. Since the **disksort()** routine is general in nature, and does not know of the specific physical characteristics of particular devices, its algorithm for sorting is to simply order requests based on the cylinder field of the

buffer header. By filling in this field with an appropriate number `i254strat()` can effectively implement sorting routines which correspond to the device being handled. For example, on a bubble memory board, there is no such thing as a cylinder. By ordering requests based on the page numbers, seek time can be minimized between successive bubble accesses. So, the `i254strat()` routine may just drop the page number into the cylinder field, and let `disksort()` do the rest. If a more complex algorithm is desired, the `i254strat()` routine can implement it and handle the insertion onto the active list itself, by-passing use of `disksort()` completely. When a transfer is complete, the buffer header is removed from the active list (`av_forw/av_back` removed) by the routine `iodone()`. This is typically called by the `i254intr()` routine (line 169).

After `i254strat()` has queued a request onto the active list, it must check and see if the device is currently in the process of performing an operation. If so, the `i254intr()` routine will start pending requests, and `i254strat()`'s work is done. If the device is not busy, `i254strat()` must start the operation to satisfy the request. Since the responsibility of starting an operation rests with both `i254strat()` and `i254intr()` routines, depending on the circumstances, the code used to start a request is usually placed into a procedure call `i254start()` (line 156), which is then called by `i254strat()/i254intr()`.

The `i254start()` routine is typically the most device specific routine in the set composing the device driver. It examines the next request on the active lists, sets-up the operation accordingly and starts it. Thus `i254start()` is the routine which does output to I/O ports and is generally cognizant of the details of performing a function with the device. Once `i254start()` is called to initiate a transfer, `i254strat()` is complete and returns.

The `i254intr()` routine (line 88) is called by the kernel when an interrupt from the device occurs. Details of manipulating the interrupt controller have already been handled by the kernel, and the only thing `i254intr()` needs to be concerned about is handling the device itself. The interrupt typically occurs to indicate that the transfer is complete. Once `i254intr()` has determined that this is the case, it should do anything necessary to finish up the operation as far as the device is concerned, set the status field in the buffer header to indicate successful completion of the operation, and then call `iodone()` (line 221) to finish up as far as the kernel is concerned. `iodone()` performs several functions, among them, awakening the process which was waiting for this operation to complete, removing the buffer header from the active list and placing the buffer header onto the free list. If the interrupt occurred as a result of the operation terminating unsuccessfully, the status field (`u.u_error`) should be set and `iodone()` should be called. Once done, `i254intr()` must then check and see if there are other requests waiting in

the active queue. If so, the `i254start()` routine should be called to start the next request.

Obviously, `i254strat()/i254intr()` manipulate common data structures. Because the `i254strat()` routine can be interrupted at any time by the occurrence of an interrupt (and the subsequent execution of the interrupt handler), care must be taken such that the interrupt routine does not "clobber" something that `i254strat()` was in the process of doing (e.g. like adding a request to the queue, checking to see if the device is active etc.). The only mechanism available to achieve this mutual exclusion (mutex) is to shut down the interrupt level used by the device while `i254strat()` is in the critical region(s) of code. The `slpN` routines are used to do this (see Interrupt Mapping for details - Appendix D).

The `i254close()` routine (line 239) is called only once by the kernel. This routine should also clean-up (e.g. turn off a minifloppy motor) for whatever is appropriate for the device.

Finally, the `i254start()` (line 156) routine is called by both the interrupt routine and the strategy routine i.e. at task-time and interrupt-time. As described before, the routine checks to see if more requests are on the device queue and outputs the appropriate commands to the ports.

7.1.7 FINAL LOOK AT BLOCK DRIVERS

Note that the above discussion on block I/O devices and interfaces is as detailed a discussion as possible given the fact that the next step is to look at existing code and work from it. Device drivers are seldom written from scratch. They are usually based on existing examples. However, there will be problems if every detail about the device is not understood. Knowing the device and its intricacies is of utmost importance to developing a device driver for it.

Kernel overhead is high for block I/O devices i.e. the kernel works hard for these drivers. But not all of driver ideology can be viewed through such structured notions as buffers, blocks and the like. Drivers have an unstructured side to them. This side is the character I/O interface.

7.1.8 THE RAW (CHARACTER INTERFACE) TO A BLOCK DEVICE

A block I/O device like a disk can have a character I/O interface. Note that a new set of routines, namely `ixxxread()`, `ixxxwrite()` and `ixxxioctl()` etc. have to be created for a block device. These routines implement the character interface for a block device driver. This character interface permits a block device, like a disk, to have direct I/O or byte I/O transfer capability. The data structure `cdevswl` |

in `c.c` shows that the disk has both character and block interfaces. Notice that certain locations unused in `cdevsw[]` are either titled `nulldev/nodev`. `Nodev` is a macro that implies a position in the array that is invalid i.e. printers cannot read. `Nulldev` is nothing more than a no-op, that implies it is legal but not implemented.

Moving away from these details, the respective character I/O reads and writes merely call `physio()` (see Device Driver Support Routines Section 7.1.2). With this call,

- 1) the user process is locked in memory
- 2) `strat()` is called with the buffer header already updated with the transfer information. However, the address of the transfer is not a buffer header address but a physical location.
- 3) `iowait()` is called which will wait for I/O completion.

The character I/O interface is often called the raw interface and is used in general utilities like `dd` and `tar`. This interface is fast, unstructured and does not go

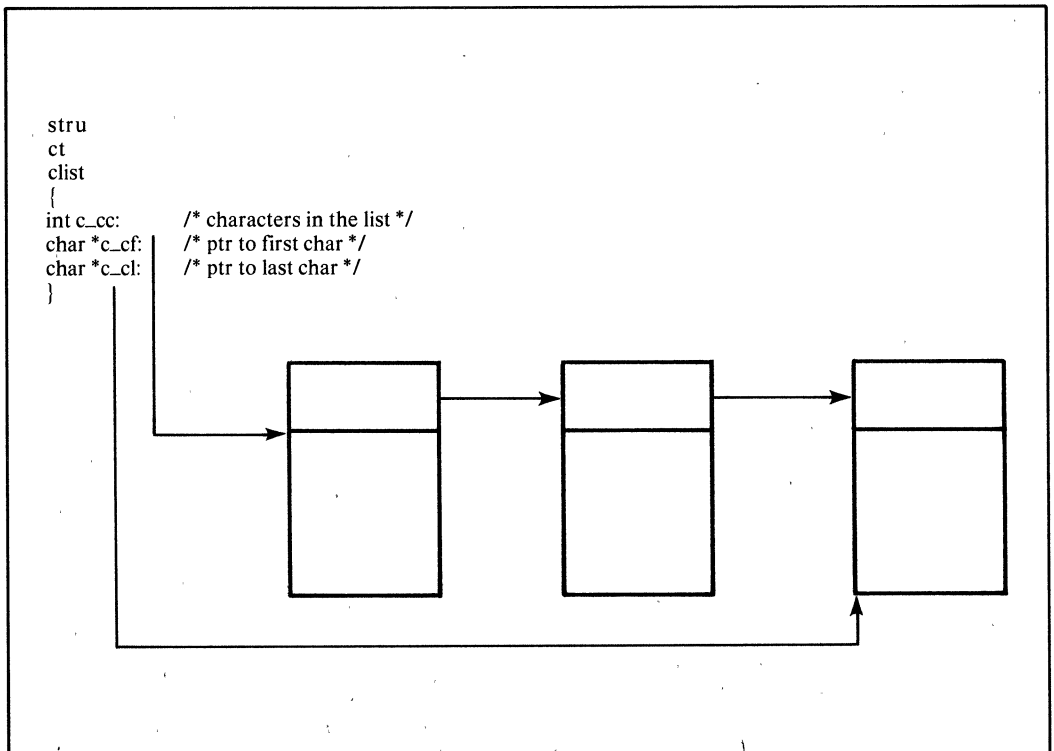
through the kernel buffer scheme. Further details can only surface by investigating the character I/O interface.

7.2 The Character Interface

Character I/O is synonymous to “byte” I/O or direct I/O. Terminals and printers fall under this category of relatively unstructured I/O mediums between device and user-space. In general, to access character I/O devices, user processes must be locked in memory and hence, for overall system throughput, these devices must be fast. For relatively slower devices (guess who? ==> terminals), a data buffering mechanism is employed. These queues are character-based and, hence, are used only for small datatransfers.

7.2.1 CLISTS

Each driver that wishes to use these buffers declares a static buffer header. These buffers are called clists and are linked-lists of buffers. The buffer header is declared in `tty.h` (Appendix H). The structure of the header is:



These clist-headers point to character-holding links that contain four-word blocks of characters (pointer and six characters). Each driver program declares its own clists and this structure accumulates clist-structure-elements (as data transfer prevails) from a "freelist" of buffers. The buffer mechanism is simple compared to the block interface as only a few routines manipulate the clist structure.

Two routines, `getc()` and `putc()` manipulate the clist.

- 1) `getc()` removes a character from the list and moves `c_cf` forward if not in a block boundary i.e. it is not in the end of a six character boundary. If it is, the pointer `c_cf` is set to the beginning of the next block and the block that the last character was read from is placed in the freelist (see figure 8a and b).
- 2) Consequently, `putc()` places a character in the list and obtains a new block when necessary (see Useful Routines).

If precautions are not taken, the freelist may be exhausted by one process. This is alleviated by defining two marks - a low and high -water mark in `tty.h`.

These marks are maintained for each clist by routines that manage them. When a process requests more than its high-water mark, it puts itself to sleep until there are more free to use i.e. when the clist is flushed and it hits its low-water mark). This is an output feature (cannot suspend keyboard input!!). This will alleviate the problem of any process dominating the freelist. Low speed character devices are assisted by the clist structure declared by the relevant driver. Each driver that deals with byte I/O must declare clist structures required for each operation i.e. an input clist and an output clist.

7.2.2 TERMINAL I/O

Each terminal line is associated with line characteristics called the tty structure. Details of these are found in the manual section of `tty(4)` in the XENIX Operating System Documentation (173258-001). The file `tty.h` describes the tty structure for each line.

There exists a set of routines that manipulate this tty structure. These routines are found in `tty.c` (not attached) and are termed "line discipline routines." In `/sys/conf/c.c` (Appendix A) these routines are outlined under the data structure `lineswll`. These

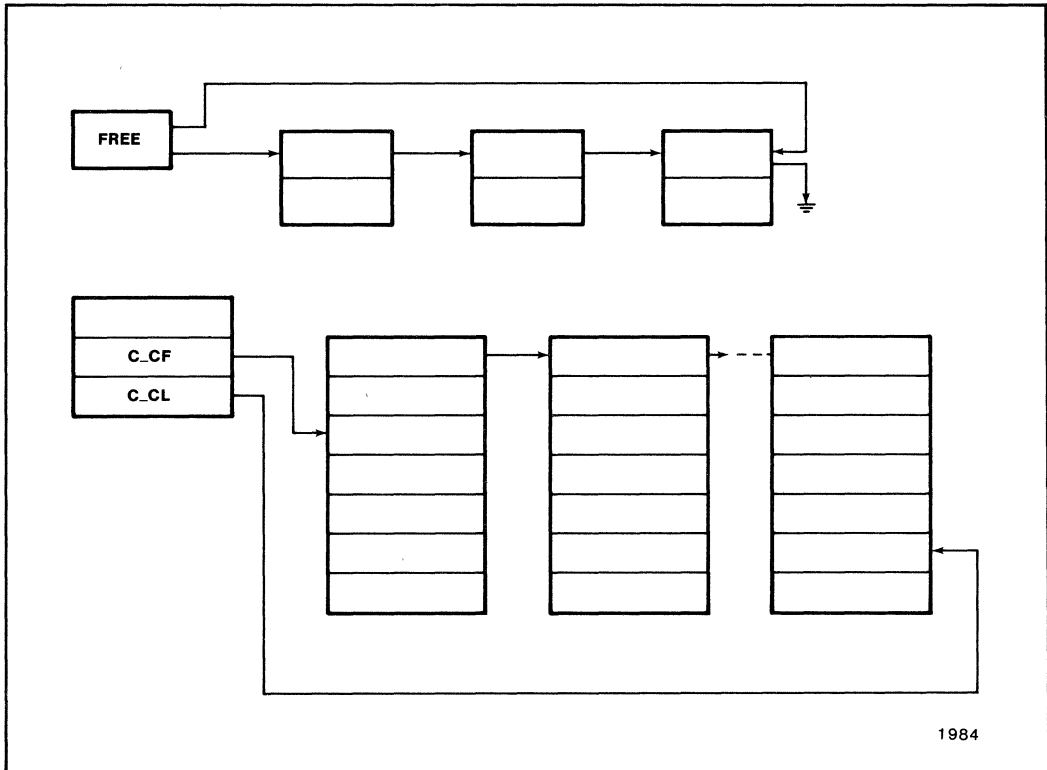
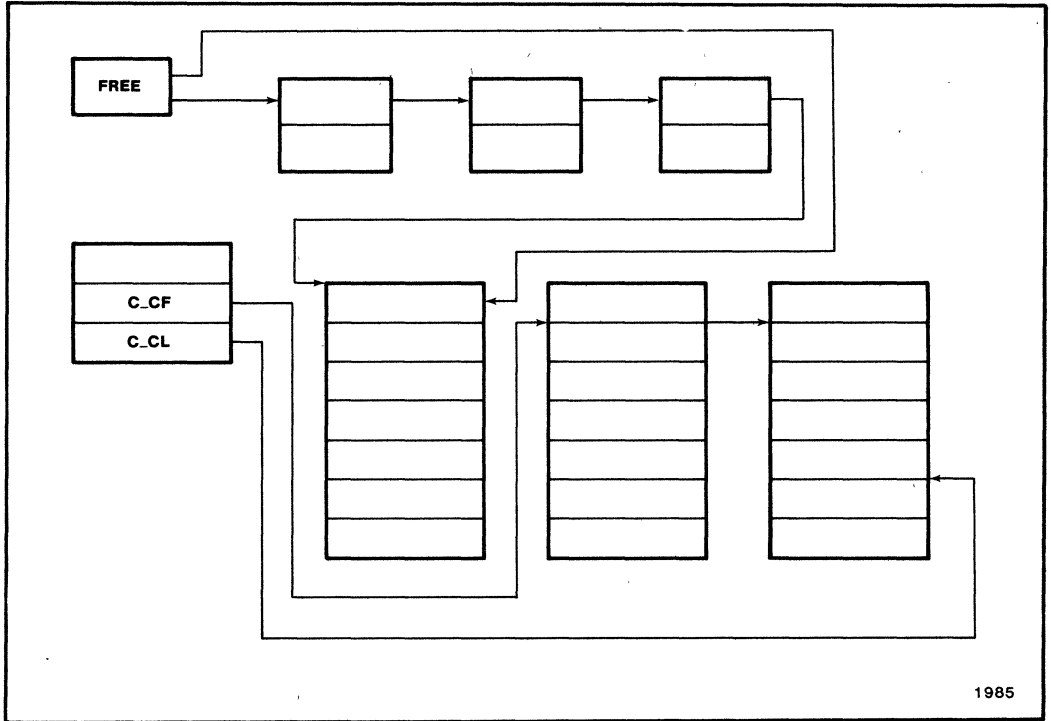


Figure 8a. clists — free list and clist for a device notice they point not to first/last character.



1985

Figure 8b. When getc() takes last char on first block, the clist structure is returned to FREE LIST.

routines are preceded by “tty” or “tt” (Naming Conventions). A summary of these routines is found in `tty.help` (section 7.2.4).

Coming back to the tty structure (Appendix H), three queues are identified for use by each terminal interface. These structures are clists and are:

- . the output queue
- . the input queues
 - . the canonical queue (cooked)
 - . the raw queue

Three static buffer headers (clist headers) are established in the tty structure for each line. The output queue facilitates output to the terminal using the high/low-water marks as gauge. The raw queue is used as the first input vehicle where all characters input are placed. It is from here that the terminal echo occurs. Notice that echo response is generally quick and is irrespective of whether the requesting process is in or out of memory. The canonical queue is that queue maintained for each line that respects all characters especially those that are dependent on surrounding characters i.e. backspace, delete, character expansion etc. This queue is also called the cooked queue to symbolize a ~ raw (not raw) queue

to further establish that XENIX is not lacking in humor! Two queues live to serve the goal of quick and consistent character treatment.

The `tty.c` (line discipline) routines are invisible to the driver-writer. Once these routines are called, they do most of the work. The queue manipulation and I/O functions are purely interface functions that are not in the driver code.

As mentioned before (Section 6.2), the function of any driver can be partitioned into task vs. interrupt-time execution. For character device drivers, the task-time portion deals with moving data to/from the user space and the clist queues. This movement (transfer) uses data derived from the users `u_structure` e.g. `u.u_count` is decremented on data transfer (no device driver involvement) that is performed by the `tty.c` routines.

The interrupt-time portion of the driver manages the transfer of data between the device and the clist queues (namely, the raw and output queues).

7.2.3 USEFUL ROUTINES

The following are useful kernel routines used by the line discipline routines. Further routines are found in Device Driver Support Routines (section 7.1.2).

```
int getc(queue)
struct clist *queue;
```

Returns a character from the clist queue or -1 if the queue is empty. The queues are either the raw, canonical or output queues.

```
int putc(c,queue)
struct clist *queue;
```

Puts a character "c" on the queue. Returns "0" if the character is placed and "-1" if unable to place in queue. The "-1" returns if gone beyond the high-water mark of the respective clist.

7.2.4 TTY.HELP – THE LINE DISCIPLINE ROUTINES

XENIX currently supports one line discipline routines i.e. how to interpret characters on I/O on the line. This line discipline is made up of several routines. They are:

ttyopen(dev, tp)

```
dev_t dev;
struct tty *tp;
```

This routine is called by the device driver's open routine. It is given an address of the line's device number and tty structure. Relevant fields in the tty structure are updated and the raw, canonical and output queues are initialized.

ttyclose(tp)

```
struct tty *tp;
```

All character queues with respect to the respective tty structure are flushed. Relevant fields in the structure are set to "closed."

ttread(tp)

```
struct tty *tp;
```

Handles a read request i.e a system call. Details on the input target addresses are found in the `u`-structure i.e `u.u_base`, `u.u_count` etc. This routine obtains data from the canonical queue and waits, if necessary, for more input. It also calls `canon()`, a routine that transfers characters from the raw input list to the canonical list after processing these lines. `Canon()` basically waits until a full line has been

typed when in cooked mode whereas, in raw, it transfers data immediately.

`ttread()` also waits on `ttinput()` to function in the interrupt time portion of the device driver.

ttwrite(tp)

```
struct tty *tp
```

Handles a write request. `ttwrite()` outputs `u.u_count` characters into the output queue (outq in `tty.h`) guarding the high/low-water marks. Calls `ttoutput()` which places character in output queue adding delays, expanding tabs etc. Calls `ttstart()` to begin transmitting the character.

ttinput(c, tp)

```
struct tty *tp;
char c;
```

Places a character on the raw queue and echoes it if required. This is how input characters are given to the read request. `ttinput()` is run at interrupt-time to add "c" to the raw queue identified by "tp." The echo is done by a call to `ttstart()` to begin character transmittal and a call to `ttoutput()` that basically transports characters from the raw queue to the output queue and prepares them for output.

ttstart(tp)

```
struct tty *tp;
```

is called to cause the next byte to be output if the device is idle. It is called by the task-time portion of the driver as well as the interrupt-time portion. `ttstart()` calls the "xxxstart()" routine in the driver.

ttiocomm(cmd, tp, addr, dev)

```
int cmd;
struct tty *tp;
caddr_t addr;
dev_t dev;
```

Handles common I/O control functions like line-editing, setting line characteristics except baud rates. Consider this call a transfer of input/output control and line characteristic functions to the relevant data structure that holds that information. This routine is called from the `xxxioctl()` routine of the driver.

As mentioned in the discussion on driver interfaces, the file `c.c` holds all kernel interface data-structures to the main driver routines. The data structure `cdevsw[]` is the link to the character I/O drivers. This structure maps the main special devices like `/dev/ttya0` to the actual driver routines.

7.2.5 TERMINAL I/O DEVICE DRIVER ROUTINES

The following routines are typical of a character device driver. The `tty.c` routines are used as most terminal I/O device drivers rely on them. Other drivers for line printers (output only) do not use the `tty.c` routine as less processing of output is necessary and is simpler. The terminal I/O driver-routines are:

`ixxxinit()`

This routine initializes the device. It checks to see if the device is alive and sets a flag to remember this. It then prints messages which tells the user interface that it is/is not alive.

`ixxxopen(dev, flag)`

```
dev_t dev;
int flag;
```

`dev` is `<major,min>` device number. `Flag` is `B_READ`, `B_WRITE`. `Flag` may be ignored for `tty` drivers. Called every time the device is opened. Checks for validity of open, fills out `tty` structure for the device line. The fields it fills are:

```
t_addr - set the device's I/O address
t_oproc - set to the address of the device's
          output start routine
t_lproc - set to address of start routine for output
t_state - device's state
```

Calls `ttyopen`. Calling sequence:

```
ixxxopen() =====> ttyopen
```

`ixxxclose(dev, flag)`

```
dev_t dev;
```

`dev` and `flag` are described for `ixxxopen()`. The routine is called when the last file attached to the device is closed. Calls `ttyclose` and performs clean-up e.g. flushes pending clists. Calling sequence:

```
xenix =====> ixxxclose() =====> ttyclose
```

`ixxxread(dev)`

```
dev_t dev;
```

This routine implements the read system call for the line and calls `tthread()` passing it the `tty` structure for the line. Calling sequence:

```
xenix =====> ixxxread() =====>
tthread
```

`ixxxwrite(dev)`

Like the `read()`. Calling sequence:

```
xenix =====> ixxxwrite() =====> ttwrite
```

`ixxxioctl(dev, cmd, addr, flag)`

```
dev_t dev;
int cmd;
caddr_t addr;
int flag;
```

Implements the `ioctl` system call for this line. I/O control is used for special functions such as rewinding tapes and the like. In terminal drivers, the `ioctl` routine is used to get/set various characteristics of the line. A common `tty.c` routine used is `ttioctcomm()`. Calling sequence:

```
xenix =====> ixxxioctl() =====>
ttioctcomm
```

`ixxxstart(tp)`

```
struct tty *tp; /* ptr to the tty structure */
```

the `start()` routine is called by the common `tty` support routines to start output on the line. The address of this routine is set in the `open()` routine and this address is kept in the line's `tty`-structure. This address is picked up by the `tty.c` routine to initiate action on the device. Typically, if the device is idle, a character is grabbed from the output queue and sent to the device. If the device is busy or no characters are available, the procedure is exited. Calling sequence:

at task time:

```
xenix =====> ixxxwrite() =====> ttwrite()
=====> ttstart() =====> ixxxstart()
```

at intr time:

```
device =====> ixxxintr() =====> ttyinput()
=====> ttstart() =====> ixxxstart()
```

on input,

```
device =====> ixxxintr() =====> ttstart()
=====> ixxxstart()
```

on output

`ixxxintr(level)`

```
int type;
```

This is the interrupt procedure. Level may be ignored. An input interrupt service routine will call `ttinput()` while an output interrupt will call `ttstart()` to begin output of the next character. The `ttstart()` routine will then call the driver's `start()` routine. As in block device, the interrupt service routine is called when the device is returning from the busy state to the idle state. `ttstart()` is called to bring the device back to the busy state if further output is necessary. Calling sequence:

```
HW/input intr ==> xenix ==> ixxxintr()
==> ttyinput()
```

```
HW/output intr ==> xenix ==> ixxxintr()
==> ttstart()
```

Note: These routines are simpler to write due to the `tty.c` routines. Terminal I/O is a special case of char-

acter I/O. Other character I/O devices require the same routines i.e. `ixxxopen()` etc. but cannot rely on the functionality provided by the line discipline routines.

The following diagram (figure 9) illustrates how I/O occurs in terminal I/O. Before any further detail is tackled, a brief diagram of the calling sequences for the main driver routines is shown also. `ixxxstart()` (figure 10) is called from both the `ixxxintr()` / `ixxxwrite()` routines.

7.2.6 EXAMPLES OF CHARACTER I/O (TERMINAL) DRIVERS

There is no real substitute for actual code. However, in XENIX drivers, code can be a challenge to read and understand. To alleviate long hours, a walk-through of a roughly written driver follows. Another example is found in Appendix I.

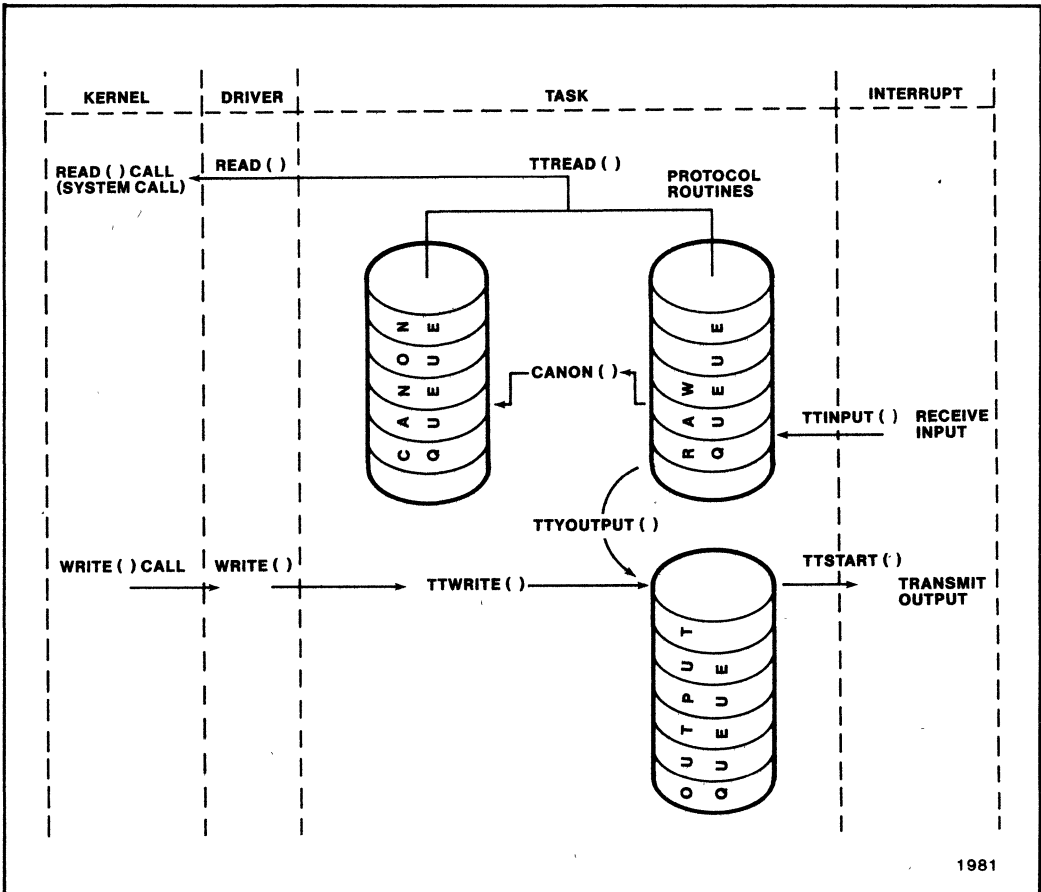


Figure 9. Architecture of Lists and Hold the Work.

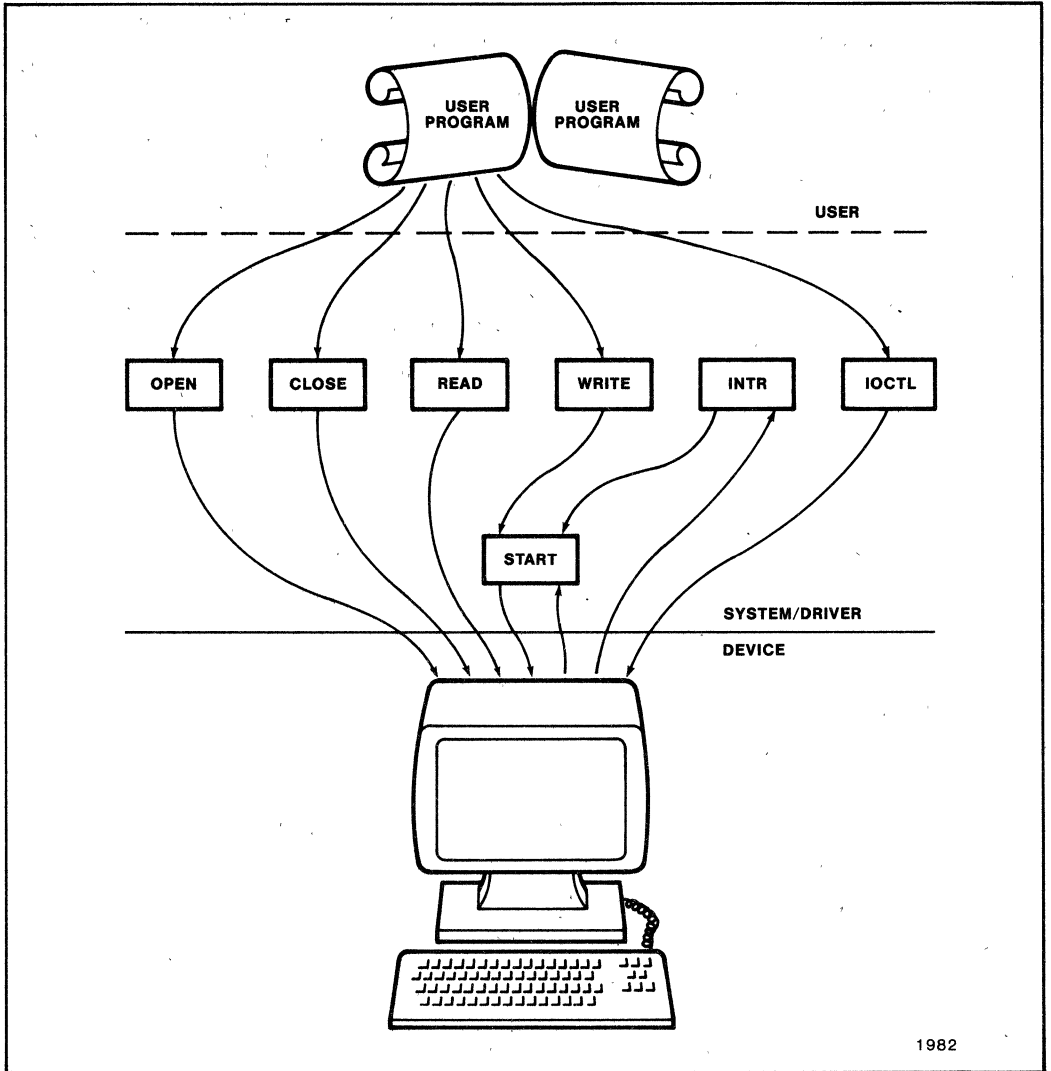


Figure 10. Terminal I/O Driver Routines.

7.2.6.1 iSBX™ 270 Walkthrough

```

1 /*
2 * i270.c - iSBX270 device driver
3 *
4 * - implements terminal device driver for the iSBX270 character
5 *   graphics video display controller
6 *
7 * See also: c270.c - i270 configuration
8 *   i270.h - i270 include files
9 *
10 * Notes on this driver:
11 *   (1) The driver supports the keyboard interface and display
12 *       in scroll mode or page mode
13 *   (2) All manual references in the comments are to the iSBX270
14 *       Video Display Terminal Controller Board Hardware Reference
15 *       Manual, order number 143444-001.
16 */
17
18 #include "../h/param.h"
19 #include "../h/user.h"
20 #include "../h/tty.h"
21 #include "../h/i270.h"
22
23 extern struct    i270cfg    i270cfg;    /* configuration structure */
24
25 short i270_alive;    /* board alive flag */
26 struct tty i270tty; /* tty structure */
27 int c_state;    /* state variable used for escape
28                 sequences */
29 */
30 * i270init()
31 *   - tests for presence of the iSBX270, and reports its presence
32 *     or absence
33 *   - initializes 270 for the configured modes of operation
34 *   - this routine is called very early in the system initialization
35 */
36
37 i270init()
38 {
39     short mode;
40     int    rststat;
41
42 /*
43 * The sequence below sends a reset command to the 270. The
44 * algorithm is based on the flowchart, page 3-8 of the manual.
45 * We perform the additional task of determining if the board is
46 * present or not.
47 */
48 c_state = 0;
49 rststat = rst270(); /* reset 270 */
50 if (rststat == RSTERR) {
51
52     /*
53      * Board not found.
54      */
55
56     printf("iSBX 270 board port %x NOT found.0, i270cfg.c_data);
57     i270_alive = I270DEAD;
58     return;

```

```
59 }
60 else {
61     printf("iSBX 270 board port %x found.0, i270cfg.c_data);
62     i270_alive = I270LIVES;
63 }
64
65 /*
66 * Board lives, so set it up in the configuration specified.
67 * NIMASK is anded with everything to clear any bits which
68 * aren't implemented.
69 * Once we reach this point, we'll assume that the board is
70 * alive to some extent, so we'll just concern ourselves with
71 * getting through the initialization; but we can't afford to
72 * get hung up if the firmware is acting funny. Our approach
73 * will be to protect ourselves against infinite loops, but not
74 * check for error conditions or worry about reporting them.
75 */
76
77 mode = (i270cfg.c_keybrd|i270cfg.c_lpen|i270cfg.c_dma |
78         i270cfg.c_mode|IBEINT|(i270cfg.c_cursor & CURMSK))
79         & NIMASK;
80 mode270(mode);
81 }
82
83 /*
84 * rst270() - resets the 270 board
85 */
86
87 rst270()
88 {
89     unsigned i;
90
91     /*
92     * Clear out any garbage in input and output buffers.
93     * i is a safety valve in case the board is not here and
94     * we read a 1 in the IBF bit - we'll take care of the
95     * presence or absence of the board later - for now we just
96     * need to get out of this loop.
97     */
98
99     i = 0;
100     while ((in_270(i270cfg.c_stat) & I270IBF) && (i++ < 30000)) {
101         if (in_270(i270cfg.c_stat) & I270OBF)
102             in_270(i270cfg.c_data); /* dummy data */
103     }
104     out_270(i270cfg.c_stat, I270RST); /* reset it */
105
106     /*
107     * We'll look for some sign of life from the 270. If
108     * it's not there, we'll either blow right through the
109     * loop, or get stuck in it forever. We'll limit
110     * forever to 30000 iterations, and check for a count of
111     * 0 or > 30000 upon terminating the loop - either one is
112     * then interpreted as a sign that the board isn't here.
113     */
114
115     i = 0;
116     while ((in_270(i270cfg.c_stat) &
117             (I270OBF|I270IBF|I270BUS)) && (i++ < 30000)) {
118         if (in_270(i270cfg.c_stat) & I270OBF)
```



```
119         in_270(i270cfg.c_data);    /* dummy input */
120     }
121     if ((i == 0) | (i > 30000)) return RSTERR;
122     else return RSTOK;
123 }
124
125 /*
126 * mode270(mode) - sets VDTC mode for 270 board to specified mode
127 */
128
129 mode270(mode)
130 short mode;
131 {
132     unsigned i;
133
134 /*
135 * First, go into null busy wait until we can stick another
136 * command into the input buffer - once again, i is used as
137 * an escape in case the firmware on the 270 is acting goofy.
138 */
139 i = 0;
140 while ((in_270(i270cfg.c_stat) & I270IBF) && (i++ < 30000));
141
142 /*
143 * Now we can issue a new command - we do a set VDTC mode.
144 */
145
146 out_270(i270cfg.c_stat, I270SM);
147
148 /*
149 * Wait until the 270 can accept the parameter.
150 */
151
152 i = 0;
153 while (((in_270(i270cfg.c_stat) &
154         (I270OBF | I270IBF | I270BUS)) != I270BUS) && (i++ < 30000)) {
155     if (in_270(i270cfg.c_stat) & I270OBF)
156         in_270(i270cfg.c_data);    /* dummy input if OBF set */
157 }
158 out_270(i270cfg.c_data, mode);
159 }
160
161 /*
162 * i270open(dev) - opens dev
163 */
164
165 i270open(dev)
166 dev_t dev;
167 {
168     int unit, i270start();
169     struct tty *tp;
170
171 /*
172 * First check to make sure that board is alive - if not,
173 * mark error and return.
174 */
175
176 if (i270_alive == I270DEAD) {
177     u.u_error = ENXIO; /* no such device or address */
178     return;
```

```
179 }
180
181 /*
182  * tp will point to the tty structure
183  */
184
185 tp = &i270tty;
186
187 /*
188  * Check for 270 already being exclusively opened
189  */
190
191 if ((tp->t_state & XCLUDE) && u.u_uid) {
192     u.u_error = EBUSY;
193     return;
194 }
195
196 /*
197  * Set up fields in tty structure.
198  */
199
200 tp->t_addr = (caddr_t)i270cfg.c_data; /* io base address */
201 tp->t_oproc = i270start; /* routine to start output */
202
203 /*
204  * If this is first open...
205  */
206
207 if ((tp->t_state & ISOPEN) == 0) {
208     ttychars(tp); /* sets special characters */
209     tp->t_ispeed = tp->t_ospeed = 9600; /* baud rate meaningless */
210     tp->t_flags = ODDP|EVENP|ECHO|CRMOD; /* should add no
211                                         tab expansion here */
212 }
213 tp->t_state |= CARR_ON;
214 ttyopen(dev, tp);
215 }
216
217 /*
218  * i270close(dev, flag)
219  */
220
221 i270close(dev)
222 dev_t dev;
223 {
224     struct    tty    *tp;
225
226     tp = &i270tty;
227     ttyclose(tp);
228     tp->t_addr = (caddr_t) 0; /* forget base address */
229 }
230
231 i270read(dev)
232 dev_t dev;
233 {
234     struct    tty *tp;
235
236     tp = &i270tty;
237     tthread(tp);
238 }
```

```
239
240 i270write(dev)
241 dev_t dev;
242 {
243     struct tty *tp;
244
245     tp = &i270tty;
246     ttwrite(tp);
247 }
248
249 i270iintr(level)
250 int level;
251 {
252     struct tty *tp;
253     short status, chr;
254
255     tp = &i270tty;
256     status = inb(i270cfg.c_stat);
257     chr = inb(i270cfg.c_data);
258     if (status & I270KDR)
259         ttyinput(chr, tp); /* only if a valid keyboard hit */
260 }
261
262 i270ointr(level)
263 int level;
264 {
265     struct tty *tp;
266
267     tp = &i270tty;
268     if (tp->t_state & BUSY) {
269         tp->t_state &= ~BUSY;
270         ttstart(tp);
271         if ((tp->t_state & ASLEEP) &&
272             (tp->t_outq.c_cc <= TLOWAT))
273             wakeup((caddr_t)&tp->t_outq);
274     }
275 }
276 int ttrstrt();
277 extern char partab[];
278
279 i270start(tp)
280 struct tty *tp;
281 {
282     int c,s;
283     short mode;
284
285     s = spl5();
286     if (tp->t_state & (TIMEOUT|BUSY)) {
287         splx(s);
288         return;
289     }
290     if ((c=getc(&tp->t_outq)) >= 0) {
291         tp->t_state|= BUSY;
292         splx(s);
293         switch (c_state) {
294             case 0:
295                 if (c == 0x1b) {
296 #ifdef DEBUG
297                     printf("O %x0, c);
298 #endif
```

```

299         outb(i270cfg.c_data, c);
300         c_state = 1;
301     }
302     else if (c == 0x11) {
303         outb(i270cfg.c_data, c);
304         c_state = 8; /* graphics leadin */
305     }
306     else if ((tp->t_flags & RAW) | (c <= 0x7f))
307         outb(i270cfg.c_data, c);
308     else {
309         tp->t_state |= TIMEOUT;
310         tp->t_state &= ~BUSY;
311         timeout((ttrstrt, (caddr_t)tp, (c & 0x7f));
312         return;
313     }
314     break;
315     case 1:
316         switch (c) {
317             case '=':
318                 /* cursor address sequence */
319 #ifdef DEBUG
320                 printf("O %x0, c);
321 #endif
322                 outb(i270cfg.c_data, 0);
323                 c_state = 2;
324                 break;
325             case 'G':
326                 /* visual attribute sequence */
327 #ifdef DEBUG
328                 printf("O 00000);
329 #endif
330                 outb(i270cfg.c_data, 0);
331                 c_state = 3;
332                 break;
333             case 'M':
334                 /* change mode sequence */
335                 outb(i270cfg.c_data, 0);
336                 c_state = 6;
337                 break;
338             default:
339                 /* regular escape sequence */
340 #ifdef DEBUG
341                 printf("O %x0, c);
342 #endif
343                 outb(i270cfg.c_data, c);
344                 c_state = 0;
345                 break;
346         }
347     break;
348     case 2:
349 #ifdef DEBUG
350         printf("O %x0, c);
351 #endif
352         outb(i270cfg.c_stat, I270SCP);
353         c_state = 4;
354         break;
355     case 3:
356 #ifdef DEBUG
357         printf("O %x0, c0x80);
358 #endif

```

```

359         outb(i270cfg.c_data, ((c & 0x3f) | 0x80));
360         c_state = 0;
361         break;
362     case 4:
363         outb(i270cfg.c_data, (c - 0x30));
364         c_state = 5;
365         break;
366     case 5:
367         outb(i270cfg.c_data, (c - 0x30));
368         c_state = 0;
369         break;
370     case 6:
371         outb(i270cfg.c_stat, I270SM);
372         c_state = 7;
373         break;
374     case 7:
375         mode = (i270cfg.c_keybrd | i270cfg.c_lpen |
376                i270cfg.c_dma | i270cfg.c_mode |
377                IBEINT | (i270cfg.c_cursor & CURMSK))
378                & NIMASK;
379         mode = (mode & 0x3f) | ((c - 0x30) << 6);
380         outb(i270cfg.c_data, mode);
381         c_state = 0;
382         break;
383     case 8:
384     /*
385         * just output character and go back to state 0 -
386         * nothing special, just didn't want to flip into a
387         * special mode as a result of this graphics character
388         */
389         outb(i270cfg.c_data, c);
390         c_state = 0;
391         break;
392     }
393 }
394 else splx(s);
395 }
396
397 i270ioctl(dev, cmd, addr, flag)
398 caddr_t    addr;
399 {
400     struct    tty    *tp;
401     tp = &i270tty;
402     if (ttioctm(cmd, tp, addr, dev)) {
403
404     /*
405         * No ioctl functions supported
406         */
407
408     }
409     else u.u_error = ENOTTY;
410 }
411
412 in_270(port)
413 unsigned    port;
414 {
415     unsigned    i;
416
417     for (i=0; i < DELAY270; i++);
418     return (short) inb(port);

```

```

419}
420 out_270(port, val)
421 unsigned   port;
422 short  val;
423 {
424 unsigned   i;
425
426 for (i=0; i < DELAY270; i+ +);
427 outb(port, val);
428 }

```

7.2.6.2 Low Level Routines

The following routines are not required but were built for clarity and in accordance with structured methodology. The routines are:

```

in_270-line 412
out_270-line 420
rst270-line 87
mode270-line 129

```

Details of the calling sequences for these routines and others are found in figure 11. Consider the procedures `in_270` (412) and `out_270` (420). These routines are just the in and out routines of the kernel with a constant delay in front, and are used only at boot time. This implies that DELAY270 times the execution time of a for loop is lost for every byte transfer (at boot time) to/from the 270 controller, and that the constant DELAY270 must be fixed every time the kernel is ported to a new cpu. These routines would make sense if the 270 required requests to be a certain time interval apart, and the author of the driver could not guarantee that the commands to the 270 would be far enough apart any other way. It would also need to be checked that DELAY270 iterations took less than 100 microseconds, to insure interrupt latency was not adversely affected when these routines were called at a high cpu priority during interrupt service.

`rst270` (87) resets the 270 controller. It begins by reading the status register of the controller. If the input buffer is full (I270IBF), the 270 cannot accept another command (even the reset), so if the data buffer is full, a data byte is read to make room in the 270 hardware's buffer. Once the input buffer of the 270 is no longer full, it is sent a reset command. The variable 'i' is used to insure that if the 270 is not present, this loop will execute at most 30,000 times. Once the reset command is sent, it should be acted upon by the controller. We enter another loop, similar to the first, which is designed to check the existence of the 270 controller. The controller exists if it is now unable to receive input (270IBF), ready to send to the cpu (I270OBF), or busy doing something (perhaps, but not necessarily, the reset). If it is none of the above, the loop will execute zero times. Note that it is possible that the 270 is not there and that

garbage is being read from where the status register would have been. There is code here that assumes that if after 30,000 tries, the controller does not return to an idle state, garbage is being read. Note again that the use of a constant number of iterations to represent an interval of time is bad practice since it implies assumptions about the execution speed of the cpu, which will vary.

`Mode270` (129) is a routine similar to `rst270`, but designed to set the initial mode of the controller after the controller is reset. First the driver waits until the controller is ready to receive the command (or 30,000 tries). Note that this means the controller must become ready within 30,000 executions of an `in_270` to accept the mode change command. This is why there is a delay in `in_270`, but this is still bad practice, since it is cpu dependent code. The 'set mode' command is sent, and a wait is done until the controller can accept the new mode, then the new mode byte is sent to the controller. The code to do this wait ensures that 'while the controller is busy and has input or output flush any input from it'. It should be noted that this is acceptable because this routine is called only by `i270init` at boot time. If `mode270` were called during normal system operation, this input would be meaningful and therefore would need to be processed. It is not clear that this loop will succeed in flushing all input generated while the 270 was in the previous mode, raising the possibility that garbage typed before a reboot will remain in the 270's buffers.

7.2.6.3 Required Routines

`i270init` (37) is called at boot time to be sure the 270 controller is ready to use, and is the only reference to the routines described above. First, `rst270` is called to see if a 270 exists (49, 50). If not, a diagnostic is printed, and the fact there is no 270 controller is remembered (56, 57). If the 270 controller exists, it is set for a standard mode modulo NIMASK, which insures the controller is not placed in an unimplemented mode (77-80). This is also the place where interrupts from the 270 are enabled (78).

`i270open` (165) is called for every open of the 270 device. In line 176 ensures that if the 270 controller was not found at boot time, all opens will fail with a

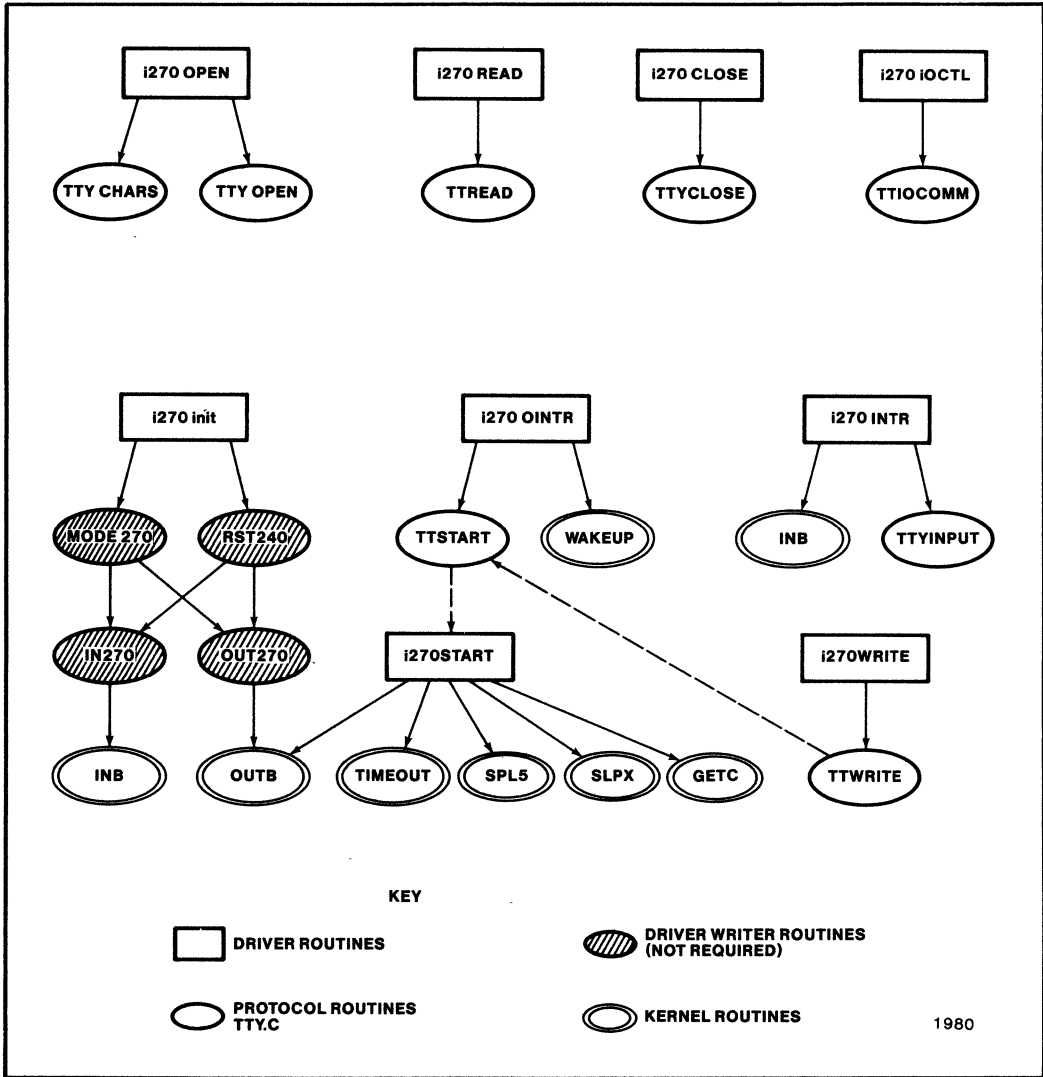


Figure 11. iSBX™ 270 Video Terminal Controller Board Driver Routine Dependencies Calling Sequences.

‘nonexistent device’ (ENXIO). Line 185 sets up a pointer to the status record for the device. If the device has an active exclusive open, line 191 arranges an open failure with a busy return. Line 201 sets a pointer for use by the protocol routines to start output. Line 207 arranges that the 270 looks to the kernel like a 9600 baud terminal with characteristics suited to the device. Line 213 sets a flag to note the device is open for later. Note that ISOPEN is set by ttyopen.

i270close (221) is called on the last close of the device. It just calls the protocol routine `ttclose`, then removes the I/O address of the 270 from the `tty` structure.

i270read (231) and **i270write** (240) respond to reads and writes by simply calling the protocol routines ‘`ttread`’ and ‘`ttwrite`’, respectively.

i270iintr (249) responds to an input interrupt by simply getting the character, and calling the protocol routine 'ttyinput' with it if it came from the keyboard.

i270ointr (262) responds to an output interrupt by checking if there was output in progress. If so, line 269 clears the BUSY flag (in case there are no more characters to output). The protocol routine 'ttstart' will send the next character and set BUSY if there is more output to do. If a process is sleeping because the output queue was at the high water mark, and the queue is now below the low water mark, all such processes are awakened (271-273).

i270start (279) is called after output interrupts and by the protocol routines to initiate output to the 270. This routine sets a cpu priority high enough to prevent reentrancy problems caused by the fact it can be called by interrupts (285). The high cpu priority is used to insure the BUSY bit does not change, so that it may be used as a lock to prevent more than one flow of control from getting into the switch statement of line 293. Once this lock is obtained, the cpu priority is lowered. The interrupt priority should remain raised for no more than the 100 microsecond. If the driver is doing output, this routine will return (line 286). It is necessary that all paths out of this routine lower the cpu priority (note lines 287, 292, and 394) within the time constraint.

Line 290 uses the 'getc' routine to get the next character from the output clist. The way interrupt driven output to tty devices is performed is by maintaining the output portion of the terminal in one of two states (idle or BUSY). In the idle state the device is ready to receive a character for output, and in the BUSY state a character has been sent to the device and the device has not yet signaled (via an output interrupt) that it is finished initiating the output. Output interrupts do not necessarily signal that the character has been output, only that the output has been started and the output device is ready to accept the next output character. A call to **i270start**, then should force the terminal into the BUSY state except in the case that the output clist is empty (lines 290 and 394). At the next output interrupt the terminal goes from BUSY to idle (line 269), and protocol routine ttstart calls the appropriate start routine (set by line 201 to be i270start) to attempt to put the device back into the BUSY state.

A major reason for the bulk of this routine is that it maintains a finite state machine which controls the disposition of characters sent to the 270. This machine implements certain escape sequences which do device control operations. Some of these device control operations cannot be done by writing to the data port, and so require special code. There is also code implementing an escape for graphic attribute bytes

which prevents them from being mistaken for device control escape sequence leadins. A state transition diagram, shown in figure 12, details the control character sequences and other input sequences expected by the device. The code for the default treatment for most characters is in state 0 at lines 306 and 307. Characters with the high bit set (non-ASCII characters) are used to implement a timed delay at lines 308 through 313. If this state is not cleared, the driver will hang. The code at lines 295 through 301 implement a transition to state one in the event an escape character is written to the 270. The graphic attribute escape is in state 0 at lines 302 through 305 and in state 8 at lines 383 through 392.

An escape is followed by a character which determines which escape sequence is involved. At state 1 lines 317 through 324 we see a transition to state 2 if a cursor motion sequence (<ESC> =) is found, after sending an ASCII NUL to the 270 controller to abort the escape sequence. State 2 sends a set cursor position (I270SCP) command to the status port and sends us to state 4 (lines 352 and 353). State 4 then sends a data byte minus an ASCII '0' (the X axis position), and sends us to state 5 (lines 363 and 364). State 5 then sends another data byte minus an ASCII '0' (the Y axis position), and sends us back to state 0 (lines 367 and 368). State 1 lines 325 through 332 abort the escape sequence and move to state 3 to implement a 'set attributes' (<ESC> G) command. State 3 sends the character after masking it to insure a valid attribute byte (line 359), the returns to state 0. State 1 lines 333 through 337 abort the escape sequence and move to state 6 to initiate a 'set mode' command. State 6 sends a 'set mode' to the controller's status port and moves to state 7 (lines 371 and 372). State 7 shifts the output byte's low bit into the 'page mode' bit, sends the new mode to the data port and returns to state 0 (lines 375 through 381). Note that an <ESC> M sequence followed by a character other than an ASCII '0' or '1' is not guaranteed to result in a valid mode. The remaining state 1 code (lines 338 through 347) implement the default action, which is to send the escape sequence on to the device and return to state zero. Note that this works because no 270 hardware escape sequence is over two characters (the escape and the command). Longer escape sequences would probably need a protection similar the protection by state 8 of attribute bytes, to prevent them from being mistaken for lead-in characters. If all escape sequence support were implemented by the device via the data port, as is the case for terminals on a serial line, none of this state machine logic would be appropriate.

i270ioctl (379) is called to respond to every ioctl issued to the 270. It responds by using the protocol routine 'tтиocomm' to process all terminal ioctls, and rejecting all others since the board does not support baud rates.

APPENDIX A: THE c.c FILE CREATED BY MASTER AND XENIXCONF	
APPENDIX B: XENIXCONF	
APPENDIX C: THE CONFIGURATION FILE MASTER	
APPENDIX D: INTERRUPT MAPPING	
APPENDIX E: param.h FILE THAT LISTS THE SYSTEM CONSTANTS	
APPENDIX F: THE buf.h FILE DESCRIBING THE BUFFER-HEADER STRUCTURE	
APPENDIX G: NAMING CONVENTIONS	
APPENDIX H: THE tty.h FILE DESCRIBING THE TTY STRUCTURE	
APPENDIX I: THE c254.c AND i254.h FILES	
APPENDIX J: THE iSBC® 534	

APPENDIX A:

THE C.C FILE CREATED BY MASTER AND XENIXCONF

```

/*
 * Configuration information
 */

#define NBUF 29
#define NINODE 120
#define NFILE 120
#define NMOUNT 8
#define SMAPSIZ (NPROC/2)
#define NCALL 25
#define NPROC 100
#define NTEXT 40
#define NCLIST 150
#define NFLOCKS 100
#define MAXUPRC 15
#define TIMEZONE (8*60)
#define NCOREL 1
#define DSTFLAG 1
#define GENBOOT 0
#define CMASK 0
#define MTOP 512

#include ".../h/param.h"
#include ".../h/buf.h"
#include ".../h/tty.h"
#include ".../h/conf.h"
#include ".../h/proc.h"
#include ".../h/text.h"
#include ".../h/dir.h"
#include ".../h/a.out.h"
#include ".../h/user.h"
#include ".../h/file.h"
#include ".../h/inode.h"
#include ".../h/acct.h"
#include ".../h/mmu.h"
#include ".../h/map.h"
#include ".../h/callo.h"
#include ".../h/mount.h"
#include ".../h/var.h"
#include ".../h/elist.h"

extern nodev(), nulldev(), novect();

int clock();
int dbgintr();
int 1544intr();
int 1215intr();
int 174intr();
int lpintr();

int (*vecintsw[])() =
{
    clock,
    dbgintr,
    novect,
    1544intr,
    novect,
    1215intr,
    174intr,

```



```

extern lpclose(), lpwrite(), lpopen(), lpinit(), lpclose(), lpwrite();
extern mmread(), mmwrite();
extern syopen(), syread(), sywrite(), syioctl();

struct bdevsw bdevsw[]=
{
/* 0*/ 1215open, 1215close, 1215strategy, &1215tab,
};

struct cdevsw cdevsw[]=
{
/* 0*/ 1215open, 1215close, 1215read, 1215write, 1215ioctl, nulldev, 0,
/* 1*/ syopen, nulldev, syread, sywrite, syioctl, nulldev, 0,
/* 2*/ nulldev, nulldev, mmread, mmwrite, nodev, nulldev, 0,
/* 3*/ 1544open, 1544close, 1544read, 1544write, 1544ioctl, nulldev, 0,
/* 4*/ 174open, 174close, 174read, 174write, 174ioctl, nulldev, 0,
/* 5*/ lpclose, lpwrite, nodev, nulldev, 0,
};

int nblkdev= 1;
int nchrdev= 6;

dev_t rootdev= makedev(0,1);
dev_t pipedev= makedev(0,1);
dev_t swapdev= makedev(0,2);
daddr_t swplo= 1;
int nswap= 1180;

int (*dinitsw[])()=
{
1215init,
1544init,
174init,
lpinit,
(int (*)())0
};

int ttyopen(), ttyclose(), ttread(), ttyinput(), ttstart();
char *ttwrite();

struct linesw linesw[]=
{
/*0*/ ttyopen, ttyclose, ttread, ttwrite, nodev, ttyinput, nulldev,
nulldev, ttstart, nulldev,
0
};

int nldisp = 1;

int Timezone=TIMEZONE;
int Dstflag=DSTFLAG;
int Genboot=GENBOOT;
int Cmask=CMASK;

struct buf buf[NBUF];
char buffers[NBUF][BSIZE+BSLOP];
struct file file[NFILE];
struct inode inode[NINODE];
struct locklist locklist[NFLOCKS];
struct proc proc[NPROC];
struct text text[NTEXT];
struct map swapmap[SMAPSIZ];
struct callo callout[NCALL];
struct cblock cfree[NCLIST];
struct mount mount[NMOUNT];

struct var v=
{
NBUF,

```

```
NCALL,  
NINODE,  
  (char *)(&inode[NINODE]),  
NFILE,  
  (char *)(&file[NFILE]),  
NMount,  
  (char *)(&mount[NMount]),  
NPROC,  
  (char *)(&proc[NPROC]),  
NTEXT,  
  (char *)(&text[NTEXT]),  
NCLIST,  
MAXUPRC,  
NFLOCKS  
};
```

```
short mm_free = 0;  
short mm_nfree = 0;  
short mem_top = MTOP;
```

APPENDIX B:

XENIXCONF

The following file establishes the devices to be selected and configured into the system.

```
*
*   Devices
*
1215    1
*1534   1
1544    1
174     1
*i270   1
lp      1
*sm     1
debug   1
*fd     1
*i287   1
root    1215 1
pipe    1215 1
swap    1215 2 1 1180
*
*   Local parameters
*
timezone (8*60)
daylight 1
cmask    0
*
*   Tunable Parameters
*
*   Dont change them unless you're sure you know what you're doing!
*
buffers 29
procs   100
mounts  8
inodes  120
files   120
clists  150
locks   100
maxproc 15
mem_top 512
```

APPENDIX C:

The configuration file-Master

```

*
* The following devices are those that can be specified in the system
* description file. The name specified must agree with the name shown
*name vsiz msk typ hndlr na bmaj cmaj # na vec1 vec2 vec3 vec4
* 1 2 3 4 5 6 7 8 9 10 11 12 13 14
1215 2 0137 014 1215 0 0 0 2 -1 0005 0 0 0 0a
fd 0 0137 014 fd 0 6 6 1 -1 0 0 0 0 0a
1534 2 0137 004 1534 0 0 1 1 -1 0002 0 0 0 0a
1544 2 0137 004 1544 0 0 3 1 -1 0003 0 0 0 0a
174 2 0137 004 174 0 0 4 1 -1 0006 0 0 0 0a
1270 2 0133 004 1270 0 0 7 1 -1 0106 0 0 0 0a
sm 0 036 010 sm 0 1 0 1 -1 0 0 0 0 0a
lp 2 0132 004 lp 0 0 5 1 -1 0107 0 0 0 0a
debug 2 0 0 dbg 0 0 0 0 1 -1 0001 0 0 0 0a
slave7 2 0 0 sl 0 0 0 0 1 -1 0007 0 0 0 0a
1287 2 0 300 1287 0 0 0 1 -1 0008 0 0 0 0a
*
* The following devices must not be specified in the system description
* file. They are here to supply information to the config program.
*
memory 0 06 0324 mm 0 -1 2 1 0 0 0 0 0
tty 0 027 0324 sy 0 -1 1 1 0 0 0 0 0
$$$
*
* The following is the line discipline table
*
tty ttyopen ttyclose tthead ttwrite nodelv ttyinput nulldev nulldev ttstart nulldev
$$$$$
*
* The following entries form the alias table.
*
1215 disk
1534 serial
sm sim
$$$
*
* The following entries form the tunable parameter table.
*
buffers NBUF 50
inodes NINODE 100
files NFILE 100
mounts NMOUNT 8
swapmap SMAPSIZ (NPROC/2)
calls NCALL 25
procs NPROC 60
texts NTEXT 40
clists NCLIST 150
locks NFLOCKS 200
maxproc MAXUPRC 15

timezone TIMEZONE (8*60)
pages NCOREL 1
daylight DSTFLAG 1
genboot GENBOOT 0
cmask CMASK 0
mem_top MTOP 512

```


APPENDIX D:

Interrupt Mapping

Interrupts are not vectored directly to the interrupt routine procedure of a driver. Rather, the interrupt is vectored inot part of the Xenix kernel. The kernel code takes care of playing with the 8259A PIC, setting up an appropriate interrupt mask, switching to the kernel map and stack for the process, saving and restoring registers and handline scheduling semantics. The outcome to this in that the interrupt routine can be written in C. Xenix handles the other details

The interrupt model is one of multiple levels of priorities. An interrupt is unique in priority and can be served only if it higher(smaller in numerical form) than the current interrupt level

The `splN()` command is used to lock out other interrupts which are lower in priority (i.e. <N). `Spl()` (set priority level) is a privileged operation and not one any process can use. All interrupt driven routines need a method to interlock data access. Data items such as buffer pools and private data. The calls `splbuf()/splcli()` are these features that permit routines to interlock their allocation and de-allocation of buffers. `Spl` only raises the current CPU interrupt level, it never changes the priority level to a lower level with as with an `splx()`. The system buffers/clists can also be mutually excluded during access by `splcli()` and `splbuf()` which is an `splN()` with N high enough to lock out device interrupts that affect them. In mutual exclusion, all relevant interrupt levels are locked out from access by.

short s

```

.
.
s = spl6() /* returns 16 bit value, not-decipherable */
.
.
splx() /* must accompany each spl call to return pri level */
/* note that spl7 is most restrictive and spl0 is the least */

```

APPENDIX E:

Param.h file that lists the system constants

```

#define MAXMEM 0x180      /* max core per process - 750K */
#define SSIZE 4096      /* initial stack size (bytes) */
#define SINCR 1024      /* increment of stack (bytes) */
#define NOFILE 20       /* max open files per process */
#define CANBSIZ 256     /* max size of typewriter line */
#define HZ 20           /* Ticks/second of the clock */
#define DHZ 20          /* Ticks/second of the clock */
#define MSGBUFS 128     /* Characters saved from error messages */
#define NCARGS 5120     /* # characters in exec arglist */
#define USTK_SIZE 4096 /* default size of user stack */
#define MAXTTYS 16      /* Max # open ttys */
#define NIOSTAT 50      /* max number of bufs to keep stats for */

/*
 * priorities
 * probably should not be
 * altered too much
 */

#define PSWP 0
#define PINOD 10
#define PRIBIO 20
#define PZERO 25
#define NZERO 20
#define PPIPE 26
#define PWAIT 30
#define PSLEP 40
#define PUSER 50

/*
 * signals
 * dont change
 */

#define NSIG 17
/*
 * No more than 16 signals (1-16) because they are
 * stored in bits in a word.
 */
#define SIGHUP 1 /* hangup */
#define SIGINT 2 /* interrupt (rubout) */
#define SIGQUIT 3 /* quit (FS) */
#define SIGINS 4 /* illegal instruction */
#define SIGTRC 5 /* trace or breakpoint */
#define SIGIOT 6 /* iot */
#define SIGEMT 7 /* emt */
#define SIGFPT 8 /* floating exception */
#define SIGKIL 9 /* kill, uncatchable termination */
#define SIGBUS 10 /* bus error */
#define SIGSEGV 11 /* segmentation violation */
#define SIGSYS 12 /* bad system call */
#define SIGPIPE 13 /* end of pipe */
#define SIGCLK 14 /* alarm clock */
#define SIGTRM 15 /* Catchable termination */
#define SIGFNC 16 /* function key */

/*
 * fundamental constants of the implementation--
 * cannot be changed easily
 */

```

```

#define NBPW    sizeof(int)    /* number of bytes in an integer */
#define BSIZE  1024           /* size of secondary block (bytes) */
/* BSLOP can be 0 unless you have a TIU/Spider */
#define BSLOP   4             /* In case some device needs bigger buffers */
#define NINDIR (BSIZE/sizeof(daddr_t))
#define BMASK   01777        /* BSIZE-1 */
#define BSHIFT  10           /* LOG2(BSIZE) */
#define NMASK   0377        /* NINDIR-1 */
#define NSHIFT   8           /* LOG2(NINDIR) */
#define INOPB   (BSIZE/sizeof(struct dinode)) /* # inodes per block */
#define LINOPB   4           /* LOG2(INOPB) */
#define NULL    0
#define DCMASK  0           /* default mask for file creation */
#define NODEV   (dev_t)(-1)
#define ROOTINO ((ino_t)2) /* i number of all roots */
#define SUPERB  ((daddr_t)1) /* block number of the super block */
#define DIRSIZ  14          /* max characters per directory */
#define NICINOD 100         /* number of superblock inodes */
#define NICFREE 100        /* number of superblock free blocks */
/* #define INFSIZE 138      /* size of per-proc info for users */
#define CBSIZE  6           /* number of chars in a clist block */
#define CROUND  07         /* clist rounding: sizeof(int) * + CBSIZE - 1*/

/*
 * MMU parameters.
 */

#define MMPGSZ  2048        /* bytes/page in the MMU */
#define LMMPGSZ 11         /* log2(MMPGSZ) */
#define NPAGEPS 32         /* There are 32 pages in a segment */
#define NSEG    0          /* max seg / user (see user.h) */
#define MMFRAGMENTS 256   /* maximum number of free segments */

/*
 * Some macros for units conversion
 */

/* pages to disk blocks */
#define ptod(x) ((x)*(MMPGSZ/BSIZE))
/* bytes to disk blocks */
#define btod(x) ((x)+(BSIZE-1))>>BSHIFT

/* inumber to disk address */
#define itod(x) (daddr_t)(((unsigned)(x)+(INOPB+INOPB-1))>>LINOPB)

/* inumber to disk offset */
#define itoo(x) (int)(((x)+(INOPB+INOPB-1))&(INOPB-1))

/* bytes to pages */
#define ptob(x) ((x)<<LMMPGSZ)

/* bytes to pages */
#define btob(x) (((unsigned)(x)+(MMPGSZ-1))>>LMMPGSZ)

/* bytes to page number */
#define btobn(x) (((unsigned)(x))>>LMMPGSZ)

/* page to address */
#define ptoa(x) ( ((long)(x) << LMMPGSZ) )

/* address (long (32 bit)) to page number (int)*/
#define atopn(x) ((int)(((long)(x))>>LMMPGSZ))

/* address (long (32 bit)) to page count (int)*/
#define atop(x)  ((int)(((long)(x)+(MMPGSZ-1))>>LMMPGSZ))

/* address (long (32 bit)) to offset (int) get bits LMMPGSZ-1 - 0 */
#define atoo(x) ((int)((x)&(MMPGSZ-1))

/* long address to short address (get low 16 bits of long address */
#define atos(x) ((int)( (x) & 0x0000FFFF))

```

```
/* long address to short address (get low 16 bits of long address */
#define atoh(x) ((int)(x) >> 16)

/* page number to long */
#define ptol(x) ((long)((int)(x)<<LMPGSZ)

/* major part of a device */
#define major(x) (int)(((unsigned)(x)>>8))

/* minor part of a device */
#define minor(x) (int)((x)&0377)

/* make a device number */
#define makedev(x,y) (dev_t)((x)<<8 | (y))

/* extract low word of long */
#define LOWWORD(x) ((int)x)

/* extract high word of long */
#define HIGHWORD(x) ((int)((long)x >> 16))

/* 8086 base from an absolute physical address */
#define base86(x) ((short)(x)>>4)

typedef struct { int r[1]; } * physadr;
typedef struct { unsigned short off;
                unsigned short seg; } segadr;
typedef long daddr_t;
typedef char * caddr_t;
typedef unsigned short Ino_t;
typedef long time_t;
typedef int label_t[5]; /* return, sp, si, di, bp */
typedef int dev_t;
typedef long off_t;

/*
 * Machine-dependent bits and macros
 */
#define SPLOMASK 0x00 /* 0xC0 ==> SM on On-Board USART */
#define USERMODE(ps) ((ps)&03 == 03)
#define CLKONLY(ps) ((ps)&0x8000) /* IO10 --- PLB */
```

APPENDIX F:

The buf.h file describing the buffer-header structure

```

/*
 * Each buffer in the pool is usually doubly linked into 2 lists:
 * the device with which it is currently associated (always)
 * and also on a list of blocks available for allocation
 * for other use (usually).
 * The latter list is kept in last-used order, and the two
 * lists are doubly linked to make it easy to remove
 * a buffer from one list when it was found by
 * looking through the other.
 * A buffer is on the available list, and is liable
 * to be reassigned to another disk block, if and only
 * if it is not marked BUSY. When a buffer is busy, the
 * available-list pointers can be used for other purposes.
 * Most drivers use the forward ptr as a link in their I/O
 * active queue.
 * A buffer header contains all the information required
 * to perform I/O.
 * Most of the routines which manipulate these things
 * are in bio.c.
 */
struct buf
{
    int b_flags; /* see defines below */
    struct buf *b_forw; /* headed by d_tab of conf c */
    struct buf *b_back; /* " */
    struct buf *av_forw; /* position on free list, */
    struct buf *av_back; /* if not BUSY*/
    dev_t b_dev; /* major+minor device name */
    unsigned b_bcount; /* transfer count */
    union { /* always points to buffer area */
        caddr_t b_addr; /* as low order core address */
        int *b_words; /* as words for clearing */
        struct_filsys *b_filsys; /* as superblocks */
        struct_dinode *b_dino; /* as ilist */
        daddr_t *b_daddr; /* as indirect block */
    } b_un;
    daddr_t b_blkno; /* block # on device */
    char b_xmem; /* high order core address */
    char b_error; /* returned after I/O */
    unsigned int b_resid; /* bytes not transferred after error */
    unsigned int b_cylin; /* cylinder number for disk i/o queue */
};

extern struct buf buf[]; /* The buffer pool itself */
extern struct buf bfreelist; /* head of available list */

/*
 * These flags are kept in b_flags.
 */
#define B_WRITE 0 /* non-read pseudo-flag */
#define B_READ 01 /* read when I/O occurs */
#define B_DONE 02 /* transaction finished */
#define B_ERROR 04 /* transaction aborted */
#define B_BUSY 010 /* not on av forw/back list */
#define B_PHYS 020 /* Physical IO potentially using UNIBUS map */
#define B_MAP 040 /* This block has the UNIBUS map allocated */
#define B_WANTED 0100 /* issue wakeup when BUSY goes off */
#define B_AGE 0200 /* delayed write for correct aging */
#define B_ASYNC 0400 /* don't wait for I/O completion */
#define B_DELWRI 01000 /* don't write till block leaves available list */

```

```
#define B_TAPE 02000 /* this is magtape (no bdwrite, raw i/o at any loc) */
#define B_PBUSY 04000
#define B_PACK 010000
#define B_PURGE 020000 /* bpurge() in progress--invalidate buf when releas

/*
 * special redeclarations for
 * the head of the queue per
 * device driver.
 */
#define b_actf av_forw
#define b_actl av_back
#define b_active b_bcount
#define b_errcnt b_resid

/*
 * collect io statistics
 */
#define DISKMON 1

#ifdef DISKMON
struct {
    int nbuf;
    long nread;
    long nreada;
    long ncache;
    long nwrite;
    long bufcount[NIOSTAT];
    long nswabp;
} io_info;
#endif
```

APPENDIX G:

Naming Conventions

The convention followed is:
ixxxppp

where

xxx is the number of the device (i.e. 534,524)
ppp is the procedure name i.e. init, open

Thus, the ISBX 270 Video Terminal Controller Board driver has the interface procedures:

```
1270init()
1270open()
1270start() etc.
```

This naming convention allows the kernel procedures to understand unique driver interfaces. Usually, data structures also follow this convention to identify variable names and symbols.

APPENDIX H:

The `tty.h` file describing the `tty` structure

```

/*
 * A clist structure is the head
 * of a linked list queue of characters
 * The characters are stored in 4-word
 * blocks containing a link and several characters.
 * The routines getc and putc
 * manipulate these structures.
 */
struct clist
{
    int c_cc; /* character count */
    char *c_cf; /* pointer to first char */
    char *c_cl; /* pointer to last char */
};

/*
 * A tty structure is needed for
 * each UNIX character device that
 * is used for normal terminal IO.
 * The routines in tty.c handle the
 * common code associated with
 * these structures.
 * The definition and device dependent
 * code is in each driver. (kl.c dc.c dh.c)
 */
struct tc {
    char t_intrc; /* interrupt */
    char t_quitc; /* quit */
    char t_startc; /* start output */
    char t_stopc; /* stop output */
    char t_eofc; /* end-of-file */
    char t_brkrc; /* input delimiter (like nl) */
};

struct tty
{
    struct clist t_rawq; /* input chars right off device */
    struct clist t_canq; /* input chars after erase and kill */
    struct clist t_outq; /* output list to device */
    int (* t_oproc)(); /* routine to start output */
    int (* t_iproc)(); /* routine to start input */
    struct chan *t_chan; /* destination channel */
    caddr_t t_linep; /* aux line discipline pointer */
    caddr_t t_addr; /* device address */
    dev_t t_dev; /* device number */
    short t_flags; /* mode, settable by ioctl call */
    short t_state; /* internal state not visible externally */
    short t_2state; /* continuation of state, driver specific */
    short t_pgrp; /* process group name */
    char t_delc; /* number of delimiters in raw q */
    char t_line; /* line discipline */
    char t_col; /* printing column of device */
    char t_erase; /* erase character */
    char t_kill; /* kill character */
    char t_char; /* character temporary */
    char t_ispeed; /* input speed */
    char t_ospeed; /* output speed */
    union {
        struct tc t_tc;
        struct clist t_ctlq;
    } t_un;
};

```



```

#define tun tp->t un
/*
 * structure of arg for ioctl/
 */
struct ttiocb {
    char ioc_ispeed;      d
    char ioc_ospeed;     e
    char ioc_erase;      f
    char ioc_kill;       a
    int ioc_flags;       u
};                          l
                                t

#define TTIPRI 28
#define TTOPRI 29
#define CERASE ' '
#define CEOT 004
#define CKILL '@'
#define CQUIT 034
#define CINTR 0177
#define CSTOP 023
#define CSTART 021
#define CBRK 0377

/* limits */
#define TTHIWAT 100
#define TTLOWAT 70
#define TTYHOG 256

/* modes */
#define TANDEM 01
#define CBREAK 02
#define LCASE 04
#define ECHO 010
#define CRMOD 020
#define RAW 040
#define ODDP 0100
#define EVENP 0200
#define NLDELAY 001400
#define TBDELAY 006000
#define XTABS 006000
#define CRDELAY 030000
#define VTDELAY 040000

/* Hardware bits */
#define DONE 0200
#define IENABLE 0100

/* Internal state bits */
#define TIMEOUT 01 /* Delay timeout in progress */
#define WOPEN 02 /* Waiting for open to complete */
#define ISOPEN 04 /* Device is open */
#define FLUSH 010 /* outq has been flushed during DMA */
#define CARR_ON 020 /* Software copy of carrier-present */
#define BUSY_040 /* Output in progress */
#define ASLEEP 0100 /* Wakeup when output done */
#define XCLUDE 0200 /* exclusive-use flag against open */
#define TTSTOP 0400 /* Output stopped by cti-s */
#define HUPCLS 01000 /* Hang up upon last close */
#define TBLOCK 02000 /* tandem queue blocked */
#define DKCMD 04000 /* datakit command channel */
#define DKMPX 010000 /* datakit user-multiplexed mode */
#define DKCALL 020000 /* datakit dial mode */
#define DKLINGR 040000 /* datakit lingering close mode */
#define CNTLQ 0100000 /* interpret t_un as clist */

/* Driver specific state bits */
#define INBUSY 01 /* Input in progress */
#define INSTOP 02 /* Stop input interrupts */

```

```
/*
 * tty ioctl commands
 */
#define TIOCGETD (('t'<<8)|0)
#define TIOCSETD (('t'<<8)|1)
#define TIOCHPCL (('t'<<8)|2)
#define TIOCMODG (('t'<<8)|3)
#define TIOCMODS (('t'<<8)|4)
#define TIOCGETP (('t'<<8)|8)
#define TIOCSETP (('t'<<8)|9)
#define TIOCSETN (('t'<<8)|10)
#define TIOCEXCL (('t'<<8)|13)
#define TIOCNXCL (('t'<<8)|14)
#define TIOCFLUSH (('t'<<8)|16)
#define TIOCSETC (('t'<<8)|17)
#define TIOCGETC (('t'<<8)|18)
#define TIOCGETS (('t'<<8)|19)
#define DIOCLSTN (('d'<<8)|1)
#define DIOCNTL (('d'<<8)|2)
#define DIOCMPX (('d'<<8)|3)

#define DIOCNPX (('d'<<8)|4)
#define DIOCSCALL (('d'<<8)|5)
#define DIOCRCALL (('d'<<8)|6)
#define DIOCPGRP (('d'<<8)|7)
#define DIOCGETP (('d'<<8)|8)
#define DIOCSETP (('d'<<8)|9)
#define DIOCLOSE (('d'<<8)|10)
#define DIOCTIME (('d'<<8)|11)
#define DIOCRESET (('d'<<8)|12)
#define FIOCLEX (('f'<<8)|1)
#define FIONCLEX (('f'<<8)|2)
#define FIORDCHK (('f'<<8)|3)
#define MXLSTN (('x'<<8)|1)
#define MXNBLK (('x'<<8)|2)

/*
 * tty ioctl commands (extension)
 */
#define MLCRESET (('m'<<8)|0)
#define MLCBOOT (('m'<<8)|1)
#define MLCREAD (('m'<<8)|2)
#define MLCWRITE (('m'<<8)|3)
```

APPENDIX I:

The c254.c and 1254.h files

```
/*
 * include file for 254 driver ... this is 1254 h
 *
 * mask constants for BMC status:
 */

#define  BMCBUSY   0x80
#define  BMC

/*
 * configuration structure for 254
 */
struct  1254cfg {unsigned c_base_port;
        unsigned c_page_size;
};

/* this is c254.c
 */

#include "../h/1254.h"
struct  1254cfg 1254cfg={0x40, /* I/O base port address */
        256} /* bubble page size
        - 64 for 1 bubble,
        128 for 2 bubbles,
        256 for 4 bubbles */
```

APPENDIX J:

The 1SBC 534

```
/*
 * INTEL CORPORATION PROPRIETARY INFORMATION. THIS LISTING IS
 * SUPPLIED UNDER THE TERMS OF A LICENSE AGREEMENT WITH INTEL
 * CORPORATION AND MAY NOT BE COPIED NOR DISCLOSED EXCEPT IN
 * ACCORDANCE WITH THE TERMS OF THAT AGREEMENT.
 */

/*
 * isbc534 device driver.
 *
 * This is the set of procedures that make up the isbc534 device driver.
 * The procedures provided include 1534open, 1534close, 1534intr, 1534start,
 * 1534ioctl which are the interfaces between xenix and the hardware.
 * The subroutines used are 1534init, 1534param which are used to program the
 * hardware. The isbc534 hardware consists of 4 usarts, 2 pic's, 2 pit's and
 * a ppi.
 *
 * Multiple isbc534 minor number structure:
 * bits 0-4:
 * Minor #:      Board:
 * 0-3   usarts  1st Board lowest intr level
 * 4-7   usarts  2nd Board intr level
 *
 * 20-23 usarts 6th Board last intr level(7)
 * bits 5-6 reserved for future use.
 *
 * NOTES: The base address of the board MUST be non-zero!!!
 *        The isbc86/12 board must have the fail safe timer installed.(default)
 *
 * The isbc534 REQUIRES a HARDWARE MODIFICATION for MODEM SUPPORT
 * The isbc534 requires a default jumper removed from
 * pin 105-106
 * and add a jumper from
 * pin 105-104
 * This modification cascades timer bdg4 to bdg5 to allow a
 * 2 second timer used in detecting carrier from a modem.
 * The carrier loss signal is generated via a separate interrupt.
 * The above modification is ONLY NEEDED to FOR MODEM SUPPORT but should
 * be done for consistency.
 *
 * Debug switches are: DEBUG for isbc534 support.
 * 1534debug: output control
 *          0 == no output except spurious intrs
 *          1 == special currently same as 0
 *          2 == little but useful output
 *          3 == all output
 *
 * Written by Jim Chorn
 * on 12/29/81
 *
 * History: modified 1/15/82 for multiple board support.
 *          modified 1/29/82 for console support.
 *          modified 3/29/82 for addition of modem support
 *          mods affect 1534open,1534close,1534intr.
 *          modified 4/22/82 moved console support out to support isbx351
 *          modified 6/22/82 added OR tie'ng of 534's on the same interrupt
 *          level.
 *          Changed the modem support bit to 0xC0 meaning configure
 *          the line for detection of aquisition AND loss of carrier detect
 *          signal. Bit 0x40 means detection of aquisition and bit 0x80
 *          means detection of loss of carrier detect signal.
 *          The detection of aquisition of carrier without detection of
```

```

*      loss of carrier is meaningless and is not mentioned in the
*      manual entry.
*
*/

#include ".../h/i534.h"      /* hardware structure and local commands */
#include ".../h/param.h"
#include ".../h/system.h"    /* system */
#include ".../h/conf.h"      /* system configuration */
#include ".../h/dir.h"       /* system directory structures */
#include ".../h/a.out.h"     /* needed for user.h */
#include ".../h/user.h"      /* user structures (system) */
#include ".../h/tty.h"       /* device structures (system) */
#include ".../h/usart.h"     /* baud rates */
#include ".../h/intr.h"     /* some pic commands from system */

#ifdef DEBUG
int i534debug = 1;          /* debug output control */
#endif

int i534wakep;              /* wakeup variable for modems */
struct tty i534tty[N534*4]; /* 4 USARTs per 534 */
struct i534cfg i534cfg[N534]; /* board software addresses von conf*/
int i534base[8];           /* board number -> board base addr */
int i534alive[N534];       /* does it live ?? */

/*
* This procedure verifies that a i534 board is presently
* configured by putting the board into test mode and
* then checking if the board actually is in test mode.
* This test mode check is a one bit test. If the board configured is not
* present an array variable for each board called i534alive is set to false.
*
* TITLE: i534probe
*
* CALL: i534probe();
*
* INTERFACES: i534open, i534intr (thru the variable i534cfg[])
*
* CALLS: none
*
* History:
*
*/

i534probe()
{
    register board;
    register struct i534cfg *cf;
    struct db534 *DBbase; /* set up the i/o boards base address */
    int alive;

    for (board=0; board<N534; board++){
        cf = &i534cfg[board];
        if(cf->c base != 0) {
            alive = 1; /* assume it lives */
            DBbase = cf->c base;
            outb(&DBbase->stestmd, 1); /* select test mode */
            if(((inb(&DBbase->stestmd) & 1) == 0) /* is test mode selected? */
                alive = 0; /* trash base addr for intr() */
            outb(&DBbase->stestmd, 0xff);
            if(((inb(&DBbase->stestmd) & 1) == 0)
                alive = 0;
            outb(&DBbase->stestmd, 0); /* deselect test mode */
            printf('i534 Based %x board %d %s.\n',
                cf->c base, board,
                alive ? "found" : "NOT found" );
            i534base[board] = cf->c base; /* associate board & tty struc*/
            i534alive[board] = alive;
        }
    }
}

```

```

/*
 * This procedure initializes the isbc534 when the call to dinit is
 * made. This procedure is done ONCE ONLY in the following sequence:
 * initialize the isbc534 structures to point at the board,
 * reset the board,
 * initialize the usarts with a special hardware sequence,
 * initialize the ppi port for input,
 * initialize and mask the on-board pic's.
 * After this has been accomplished there is no reason to reinitialize the
 * isbc534 except when hardware failure occurs.
 * NOTE: The baud rate clocks are not programmed here; this
 * is done on the first device open in the call to 1534param; see 1534open.
 *
 * TITLE: 1534init
 * CALL: 1534init();
 * INTERFACES: dinit
 *
 * CALLS: delay
 *
 * History: 1/11/82 Shortened the delay time from 100 to 10 to speed things
 *          up a bit.
 *          1/15/82 Added probing for boards.
 */

1534init()
{
    struct db534      *DBbase;          /* set up the i/o boards base address */
    struct cb534      *CBbase;
    register int i, board;

#ifdef DEBUG
    if(1534debug>=2)
        printf("1534 init, ");
#endif
    1534probe();
    for(board=0; board<N534; board++) {
        if(1534alive[board] == 0)
            continue;          /* Board not there! */
        CBbase = DBbase + 1534cfg[board].c_base;
        outb(&DBbase->reset, 0);
        outb(&DBbase->seldata, 0);
        for (i=0; i<4; i++){          /* init each usart */
            151uinit(&DBbase->USART[i].cntrl);
        }
        outb(&DBbase->PIC[0].csr, PICICW1);
        outb(&DBbase->PIC[0].msr, PICICW2);
        outb(&DBbase->PIC[1].csr, PICICW1);
        outb(&DBbase->PIC[1].msr, PICICW2);
        outb(&DBbase->PIC[0].msr, MASKINT);
        outb(&DBbase->PIC[1].msr, MASKINT);
        outb(&CBbase->selcntr, 1);
        153tprog(&CBbase->PIT[1].timer[1],          /* timer bdg4*/
                &CBbase->PIT[1].pcr,              /* pcr */
                (RATEMD0|0x40),                    /* mode */
                U534SPEED);                          /* 2 sec */

        153tprog(&CBbase->PIT[1].timer[2],          /* timer bdg5*/
                &CBbase->PIT[1].pcr,              /* pcr */
                (RATEMD0|0x80),                    /* mode */
                U534SPEED );                          /* 2 sec */
        outb(&CBbase->seldata, 1);
    }
}

/*
 * This procedure sets up a usart timer for a load operation.

```

```
* The code depends on having the ttystructure filled out before a call is made
* to 1534param. This is the sequence of events;
* check for valid speed
* program timer (using 153tprog)
* This procedure will program bdg0 to bdg4 as a baud rate generator.
*
*
*
```

```
* TITLE: 1534param
* CALL: 1534param(dev);
* INTERFACES: 1534init,
* CALLS: 153tprog
* History: 1/20/82 : removed bdg4, bdg5 programming options.
*           These timers aren't used.
*           1/29/82 : added console programming
*           4/7/82 : added 153tprog to handle pit programming
*           4/22/82 : removed console programming
*
*/
```

```
#define MAXBAUDS 15 /* maximum indexes into 1534baud[] */
int 1534baud[] = {
    US_B0, US_B50, US_B75, US_B110, 0,
    US_B150, US_B200, US_B300, US_B600, US_B1200,
    0, US_B2400, US_B4800, US_B9600, 0,
    0
};
int 1534speed[N534*4]; /* track record */

1534param(dev)
dev_t dev;
{
    struct cb534 *CBbase; /* set up the i/o boards base address */
    register struct tty *tp;
    int unit, s, speed, mode, pit;

    unit = minor(dev) & MINORMSK;
    tp = &1534tty[unit];
    CBbase = tp->t_addr & 0xf0;
    s = (int)tp->t_ospeed;
    if(s==0) { /* hangup signal via stty */
        outb(tp->t_addr+1, SHANGUP);
        return;
    }
    if(s == 1534speed[unit])
        return; /* already that fast */
    else
        1534speed[unit] = s;
    unit %= 4; /* which usart? */
    speed = 1534baud[s];
    if ((s > MAXBAUDS) || ((s != 0) && (speed == 0))) {
        u.u.error = EINVAL; /* invalid baud rate */
        return;
    }
    if (unit == 3) {
        pit = &CBbase->PIT[1].timer[0];
        mode = RATEMDO;
    } else {
        pit = &CBbase->PIT[0].timer[unit];
        mode = RATEMDO | (unit << 6);
    }
    s = SPL(),
    outb(&CBbase->selcntr, 1);
    153tprog(pit, (pit|0x03), mode, speed);
    outb(&CBbase->seldata, 1);
    splx(s);
}
```

```

/*
 * This procedure opens one of the 4 lines on the isbc534 board for
 * exclusive use by a user. The file structure is initialized
 * and control is passed to ttyread which does the actual open.
 * Not supported is the fifth device which is the parallel port.
 *
 *
 * TITLE: 1534open
 * CALL: 1534open(dev, flag);
 *
 * INTERFACES: xenix
 *
 * CALLS: 1534init, ttyopen
 *
 * History: 1/15/82: Modified code for multiple i534's to: index a
 * configuration table to get the board base address.
 */
int 1534start();

1534open(dev)
dev_t dev;
{
    struct db534 *DBbase; /* set up the i/o boards base address */
    register struct tty *tp;
    register int unit;
    int modem; /* modem bit in minor dev numb */

    unit = minor(dev) & MINORMSK;
    if (unit >= (N534*4)) { /* not enough tp's */
        u.u error = ENXIO;
        return;
    }
    tp = &i534tty[unit];
    if (i534alive[unit/4] == 0) { /* Board not there! */
        u.u error = ENXIO;
        return;
    }
    DBbase = (struct db534 *)i534cfg[unit/4].c_base;
    unit %= 4;
    tp->t_addr = (caddr_t)&DBbase->USART[unit].data;
    modem = minor(dev) & MODEMSK;
    tp->t_oproc = 1534start;
    if ((tp->t_state & ISOPEN) == 0) {
        ttychars(tp);
        tp->t_ispeed = tp->t_ospeed = ISPEED, /* channel speed */
        tp->t_flags = ODDP | EVENP | ECHO | CRMOD;
        i534param(dev); /* load baud clock */
        outb((tp->t_addr + 1), SANSWER); /* turn usart (dtr) on */
        if (modem) {
            if (modem & MODEMWAIT) /* mask detect leaving aqua */
                while((inb((tp->t_addr + 1)) & DTRON) == 0)
                    sleep((caddr_t)&i534wakeupt, TTIPRI);
            outb(&DBbase->PIC[1].msr, ((inb(&DBbase->PIC[1].msr) &
                (~0x10 << unit)) & TIMERGO));
            /* unmask carrier/detect */
        }
        outb(&DBbase->PIC[0].msr, ((inb(&DBbase->PIC[0].msr) & (~ (3 << unit * 2))));
        /* unmask txrdy, rxrdy */
    }
    if (tp->t_state & XCLUDE && u.u_uid != 0) {
        u.u error = EBUSY;
        return;
    }
    tp->t_state |= CARR ON;
    (*linesw[tp->t_line].l_open)(dev, tp);
}

```



```

/*
 * This procedure performs the close operation on one of the devices of the
 * isbc534. A close masks the device on board; reinstalls the flags that
 * state the device is closed; calls ttyclose the do the operation.
 * Not implemented yet is device 4 which is the parallel port; it is
 * unknown device at this minute.
 *
 * TITLE: 1534close
 *
 * CALL: 1534close(dev, flag);
 *
 * INTERFACES: xenix
 *
 * CALLS: ttyclose
 *
 * History:
 *
 */

```

```

1534close(dev)
dev_t dev;

```

```

{
    struct db534 *DBbase; /* set up the i/o boards base address */
    register struct tty *tp;
    register unit;
    register mask;
    int s,

    unit = minor(dev) & MINORMSK;
    tp = &i534tty[unit],
    DBbase = (tp->t_addr & 0xf0);
    if (unit < N534*4) {
        if (tp->t_state & HUPCLS) {
            tp->t_state &= ~CARR ON;
            outb((tp->t_addr + 1, SHANGUP); /* dtr off */
        }
        (*linesw[tp->t_line].l_close)(tp);
        ttyclose(tp);
        unit%=4;
        s = SPL();
        mask = inb(&DBbase->PIC[0].msr) | (3 << (unit * 2));
        outb(&DBbase->PIC[0].msr, mask); /* RxRDY, TxRDY off */
        splx(s);
    }
    tp->t_addr = (caddr_t) 0.
}

```

```

/*
 * This procedure interfaces the read request with the system read operation
 * to obtain a byte from the usart. The usart's character is read after an
 * interrupt so this procedure calls the system to wait for the interrupt
 * procedure to pass the character on to the input character queue.
 *
 * TITLE: 1534read
 *
 * CALL: 1534read(dev)
 *
 * INTERFACES: xenix
 *
 * CALLS: ttread
 *
 * History:
 *
 */

```

```

1534read(dev)
dev_t    dev;

{
    register struct tty *tp;
    register int unit;

    unit=minor(dev) & MINORMSK;
    tp = #1534tty[unit];
    (*linesw[tp->t_line].l_read)(tp);
}

/*
 * This procedure is the compliment of the i534read routine. A call is
 * made to ttwrite which watches the output queue for characters and
 * gets the characters in the queue out to the device
 *
 * TITLE: 1534write
 *
 * CALL: 1534write(dev),
 *
 * INTERFACES: xenix
 *
 * CALLS: ttwrite
 *
 * History:
 *
 */

1534write(dev)
dev_t    dev;

{
    register struct tty *tp;
    register int unit;

    unit=minor(dev) & MINORMSK;
    tp = #1534tty[unit];
    (*linesw[tp->t_line].l_write)(tp);
}

/*
 * This procedure is called by xenix with interrupts off (spl5) when the
 * isbc534 interrupts. The interrupt process polles the 8259's on the isbc534
 * to find out which device ; (if the device is a usart receiving it gets the
 * character) then sends the character to ttyinput or restarts output by
 * calling ttstart depending on which interrupt was set off. Ttstart calls
 * 1534start to make sure that no more characters need to be transmitted and
 * to let every body know a character has been transmitted. The carrier detect,
 * ring indicator, present next digit and pit interrupt signals are not
 * implemented yet. The present next digit signal comes from the external
 * source on line 4.
 *
 * NOTE : all carrier detect signals both interrupt and latch on the 8255 ppi.
 * Refer to the H/W manual for possible uses of these signals
 * (ie ACU | printer applications).
 * The rxrdy/txrdy lines from the older usarts (8251A/s2657 & older) cause
 * glitches on the pic interrupt lines. This is a problem with the USART.
 * If possible replace usart with a newer version.
 *
 *
 * TITLE: 1534intr
 *
 * CALL: 1534intr(level);
 *
 * INTERFACES: xenix
 *
 * CALLS: ttyinput, ttstart

```

```

*
* History: 1/13/82: Condensed the usart Rxrdy/txrdy intr switch to
* run more efficiently using an if.. ; Added the
* unset of busy flag which gets set in i534start.
* 1/15/82: changed variable type to level which was incorrect.
* added multiple isbc534 support.
*/
int wakeup(),
i534intr(level)
int level;
{
    struct db534 *DBbase; /* set up the i/o boards base address */
    register struct tty *tp;
    register char c;
    int status,mask; /* mask & status to/from PIC */
    int gotone,board;

    do {
        gotone=0;
        for(board=0;board<N534;board++) {
            if(i534alive[board]) {
                DBbase = i534base[board];
                outb(&DBbase->PIC[0].csr, GETINT);
                status = inb(&DBbase->PIC[0].csr);
                if ((status & GOODINT) == GOODINT) { /* check bit 8 for an int */
                    gotone++;
                    outb(&DBbase->PIC[0].csr, PIC_EOI);
                    status &= 0x07; /* mask off garbage bits */
                    tp = &i534tty[board*4] + (status >> 1);
                    if ((status & 0x01) == 0){ /* Rxrdy intr */
                        c = inb(tp->t_addr);
                        (*linesw[tp->t_line].l_rint)(c, tp);
                    }else{ /* Txrdy intr */
                        tp->t_state &= ~BUSY; /* the character is out */
                        (*linesw[tp->t_line].l_start)(tp); /* do the next one */
                        if((tp->t_state & ASLEEP) && (tp->t_outq.c_cc <= TLOWAT)) {
                            tp->t_state &= ~ASLEEP;
                            wakeup((caddr_t)&tp->t_outq);
                        }
                    }
                }
            }
        }
        outb(&DBbase->PIC[1].csr, GETINT);
        status = inb(&DBbase->PIC[1].csr);
        if ((status & GOODINT) == GOODINT) { /* check bit 8 for an int */
            gotone++;
            outb(&DBbase->PIC[1].csr, PIC_EOI);
            status &= 0x07; /* mask off garbage bits */
            if (status >= 4)
                tp = &i534tty[board*4] + (status -4);
            switch(status) {
                case 0 : /* pit 1 cntr 4 */
                    break;
                case 1 : /* pit 1 cntr 5 */
                    wakeup((caddr_t)&i534wakeup);
                    break;
                case 2 : /* ring ind all */
                    break;
                case 3 : /* present next */
                    break;
                case 4 : /* port 0 detect*/
                case 5 : /* port 1 detect*/
                case 6 : /* port 2 detect*/
                case 7 : /* port 3 detect*/
                    if((tp->t_state & (CARR_ON|ISOPEN))
                       == (CARR_ON|ISOPEN)) {
                        signal(tp->t_pgrp, SIGHUP);
                        tp->t_state &= ~CARR_ON;
                        /* flick dtr off to cause

```

```

        * hardware hang up on
        * modem
        */
        mask =   inb(&DBbase->PIC[1].msr)
                | (1<<status);
        outb(&DBbase->PIC[1].msr, mask);
        /* carrier detect off */
        outb((tp->t_addr + 1), SHANGUP);
    }
    break;
}
}

#ifdef DEBUG
/* else printf('1534: Spurious Int level %d0, level); */
#endif

/* no interrupt from this device
 * a call should be made to handle
 * some form of accounting as this
 * interrupt is probably caused by
 * an out of date usart 8251A/s2857
 * or older. (glitches occasionally
 * the rxrdy/txrdy lines)
 */
}
} while(gotone);

* This procedure starts output on a usart if needed. 1534start gets a
* character from the character queue, outputs the character to the usart,
* and sets the BUSY flag. The busy flag gets unset when the character
* has been transmitted by 1534intr().
*
* TITLE: 1534start
*
* CALL: 1534start(tp)
*
* INTERFACES: ttystart
*
* CALLS: none
*
* History: 1/13/82: Removed the hardware probing for txrdy and added
* a set of the busy flag which gets unset on txrdy
* interrupt.
*
*/
int ttrstrt();
char partab[];

1534start(tp)
register struct tty *tp;
{
    register c;
    register s;

#ifdef DEBUG
    if(1534debug>=3)
        printf('1534start: called on unit at %x0, tp->t_addr);
#endif
    s = spl5();
    if (tp->t_state&(TIMEOUT|BUSY)) {
        splx(s);
        return;
    }
    splx(s);
    if ((c=getc(&tp->t_outq)) >= 0) {
        if (tp->t_flags & RAW) {
            outb(tp->t_addr, c);
        }else{
            if (c<=0x7f) {
                outb(tp->t_addr, c | (partab[c]&0200));
            }else{

```

```

        tp->t_state |= TIMEOUT;
        timeout(ttrstrt, (caddr_t)tp, (c&0x7f));
        return;          /* i'm timed out & !BUSY */
    }
}
tp->t_state |= BUSY;
}

/*
 * This procedure handles the ioctl system calls for such things as baud rate,
 * changes and various hardware control changes from the initial set up.
 * Currently only baud rate changes are supported.
 *
 * TITLE: 1534ioctl
 *
 * CALL: 1534ioctl(dev, cmd, addr, flag)
 *
 * INTERFACES: ioctl
 *
 * CALLS: 1534param, ttioccomm
 *
 * History:
 *
 */

1534ioctl(dev, cmd, addr, flag)
caddr_t addr;
{
    register struct tty *tp;
    register int unit;

    unit = minor(dev) & MINORMSK;
    tp = #1534tty[unit];
    if (ttioccomm(cmd, tp, addr, dev)) {
        if (cmd==TIOCSETP || cmd ==TIOCSETN) /*if baud change do it*/
            1534param(dev);
    }else
        u.u_error = ENOTTY;
}

```

REFERENCES

- 1) Ritchie, Dennis M., *The Unix I/O System*, undated.
- 2) Scheulen, Bob, *Microsoft Device Driver Guide*, unpublished '82.
- 3) Letwin, Gordon, *Interrupt Structure*, unpublished (MICROSOFT) '82.
- 4) Short, Antony, *The XENIX I/O System*, unpublished (MICROSOFT) '82.
- 5) Beck, Bob, *The Anatomy of XENIX Device Drivers*, unpublished.
- 6) Byrant, McNamara, Vaish, *Writing Device Drivers*, UNIFORM '84.

ACKNOWLEDGEMENTS

- 1) Jim Emmons, for hours of shared discussion on device drivers and for his iSBC 254 Bubble Memory Board Pseudo-Code.
- 2) Dilip Ratnam, Phil Barret, Jean McNamara Rick Byrant and other members of the XENIX team for sharing their ideas.
- 3) Vince Slyngstad, for his iSBX 270 Video Terminal Controller walkthrough.



**APPLICATION
NOTE**

AP-221

October 1984

**An Introduction to
Task Management in the
iRMX™ 86 Operating System**

**CATHERINE J. LUNDBERG
APPLICATIONS ENGINEERING**

An Introduction to Task Management in the iRMX™ 86 Operating System

Contents

Introduction	
iRMX™ 86 Operating System Nucleus Architecture	
Memory Management	
Task Scheduling	
Single Task Application Example	
Inittask	
Onetask	
Creating Tasks	
Mailboxes	
Using the System Debugger (SDB) ..	
Exception Handling	
Multiple Task Application Example	
Main\$task	
Supervisor\$task	
Widget\$task	
IO\$task	
Mailboxes	
Deadlock	
Initialization	
Debugging	
Configuration	
Configuring the iRMX™ 86 Operating System	
Linking and Locating the Application	
Conclusion	
Appendix A: Inittask Code	
Appendix B: Onetask Code	
Appendix C: Tasks Code	
Appendix D: Submit file	
Appendix E: iRMX 86 Operating System Definition File	
Appendix F: Related Publications	

INTRODUCTION

The purpose of this application note is to help users understand the nucleus of the iRMX™ 86 Operating System, and how to use the nucleus. The other layers of the Operating System are not discussed. It is assumed that the reader has a basic understanding of the iRMX 86 Operating System, which can be gained by reading the product documentation. This application note does not discuss all areas of the nucleus with equal depth, so readers wishing to understand areas other than task scheduling should refer to the iRMX 86 Nucleus Reference Manual for more information. Related areas are covered in enough detail to provide the necessary background for understanding how to use tasks in the iRMX 86 Operating System.

This application note focuses on the nucleus and its task scheduling and resource management functions. The nucleus of the iRMX 86 Operating System must handle two major functions: task scheduling, which also involves interrupt handling; and resource management, in particular memory management. The iRMX 86 Operating System has other layers, which increase the functions provided by the Operating System and which rely on the nucleus as their base.

The iRMX 86 Operating System is a real time operating system which can have multiple jobs and tasks. It is a preemptive, priority based operating system. Since only one task can be executing on the central processor at any time, the task scheduler is the heart of the operating system.

The application note has two examples. The first example creates a single task to show what is involved in creating a task. The second example shows multitasking, creating three tasks. It demonstrates how the relative priorities of the tasks affect the way the application behaves, and is used as the basis for discussing deadlock between tasks.

iRMX™ 86 OPERATING SYSTEM NUCLEUS ARCHITECTURE

The nucleus of the iRMX 86 Operating System is essentially a resource manager. There are many resources which an operating system must handle. These can be divided into three areas for the iRMX 86 Operating System: processor time, objects and memory. The main concern in this application note is control of processor time, but the reader should remember that in controlling processor time, the nucleus also manages each of the other two areas.

Processor time is the key resource the iRMX 86 Operating System manages. The Operating System should use the processor as efficiently as possible.

When a task requires the processor, that task is placed in the ready state. The processor always executes the highest priority ready task first. If more than one task of that priority is ready, the processor is allocated to the task that has been ready the longest. Once a task gains control of the processor, that task retains control until preempted by a higher priority task or interrupt, or the task gives up control.

Objects are the building blocks of the iRMX 86 Operating System. There are several types of objects: tasks, jobs, segments, mailboxes, semaphores, regions, extension objects, and composite objects. Tasks are the primary focus of this application note. However, the use of segments and mailboxes will also be discussed since they are used to communicate between the tasks in the example programs.

Memory usage is usually a critical factor in an application. It is desirable to use as little memory as possible, while still allowing the application to run efficiently. Insufficient memory can slow down the execution of a task or cause an error when executing the application.

Some other functions of the iRMX 86 Operating System that are implemented at the nucleus level are exception handling, interrupt management, and hardware manipulation of devices such as the programmable interrupt controller, the system clock, and the numeric data processor. These functions are not specifically discussed in this application note.

Memory Management

In the iRMX 86 Operating System, memory is partitioned into pools. That portion of the nucleus that allocates memory is called the free space manager. The root job, which is the first job in the Operating System and the ancestor of all other jobs, has a memory pool consisting of all available memory when the Operating System initializes. As jobs are initialized, they are allocated memory from the root job's pool for their own use.

Jobs which are created during system initialization, such as those that make up the iRMX 86 Operating System layers, are allocated memory based on configuration parameters. There are two parameters used in requesting memory. The first is the minimum memory pool size. This parameter specifies the minimum amount of memory that the job requires in order to run. The second parameter is maximum memory pool size. This parameter can either have the same value as the minimum pool size, or it can have some larger value. If it has a larger value, the job will be allowed to borrow memory dynamically from its parent or some more

distant ancestor. If the maximum memory pool is set to a smaller value than the minimum memory pool, an error will result. If the minimum and maximum memory pool sizes are the same, the job will be allocated that amount of memory and will not be allowed to borrow memory.

If there is sufficient memory to fulfill the request when a job is created, it is given the minimum amount of memory requested. If there is not enough memory, the job is not created and an error (E\$MEM) will be returned to the creator. If the job is created and later needs more memory, the request will be honored up to the maximum memory pool size by borrowing. The memory is borrowed from the job's ancestors. A job with the same minimum and maximum memory pool size cannot borrow memory from its parent.

Usually, only one job created by the root job should be allowed to borrow memory. This prevents more than one job borrowing memory from the same memory pool. Multiple jobs borrowing from the same memory pool can cause deadlock between the jobs, so borrowing should be used with great caution. In an iRMX 86 Operating System with all the layers, the Human Interface is usually chosen to be the layer allowed to borrow memory.

Memory is allocated by the free space manager on a first come, first served basis. The job that is created first will receive the memory it requested if there is sufficient memory available to satisfy the minimum memory pool request. Then the next job created will request the memory it needs. If a job is not able to get as much memory as it needs, the operating system will return an E\$MEM error to the creating task and the job will fail to be created. The free space manager will continue to try to allocate memory to the next job that is created.

The free space manager for Release 6 of the iRMX 86 Operating System keeps a sequential doubly linked list of the available segments of memory within each job pool. Each block of memory has a header which contains two links: one forward, and one backward. A pointer called the rover always points to the next entry of the linked list's unallocated memory.

When a memory request is made, the next memory entry in the linked list is checked to see if it is large enough. The first segment found which is large enough is allocated and removed from the free space manager's list. The rover points to the remainder of the segment just allocated. Memory is always allocated in contiguous segments, including allocating minimum memory pools. The rover keeps the lower portion of memory from becoming more fragmented than the upper portions. Using the rover and a first

fit algorithm means that the average number of segments that must be checked is also decreased. (See Knuth: The Art of Computer Programming: Vol. 1 pp 435-453. In particular, refer to exercise 6 on p. 452 and its answer on p. 597.) When memory is returned to a pool, it is merged with existing segments when possible.

The Application Loader allocates memory to jobs at run time under control of the iRMX 86 Operating System. The OMF (Object Module Format) for the iAPX 86 processor has two areas in which minimum and maximum memory pools are specified. First are the program's minimum and maximum static memory requirements. These static memory requirements correspond to the code and data size of the job. The second area in which memory pools are specified are a minimum and maximum dynamic memory pool which are specified in the LINK86 process.

When the Application Loader allocates memory for a program, the Application Loader calls the Nucleus to create a memory pool as large as possible within the specified bounds from the user's memory pool. In Releases 5 and 6 of the iRMX 86 Operating System, with each failed attempt to allocate memory, the application loader decrements the size of the memory pool requested by 3% of the difference between the current size attempted and the minimum size. The Application Loader then tries again to allocate the memory. This approach is used so that the application is given the largest amount of memory possible in as few tries as possible, and so that the loading time is decreased.

Task Scheduling

Tasks are the active objects in an iRMX 86 Operating System, and they do all the work. They run inside of jobs, which provide the environment the tasks need, such as the memory pools. There are five possible execution states for an iRMX 86 task. These states are running, ready, asleep, suspended, and asleep-suspended. Nucleus system calls can change the state of a task. External events can also affect the state of the task. Figure 1 shows the state transition diagram for tasks.

Tasks can have different priorities. A numerically lower priority is a logically higher priority task. A task which has a logically higher priority will execute first if it is in the ready state. Tasks will be put on the ready list in priority order, and within a priority, the task which has been ready the longest will execute first.

Normal tasks are assigned priorities between 80H and 0FFH so that they can be serviced with minimum delay. Interrupts are usually at a higher priority than normal tasks, and will always interrupt

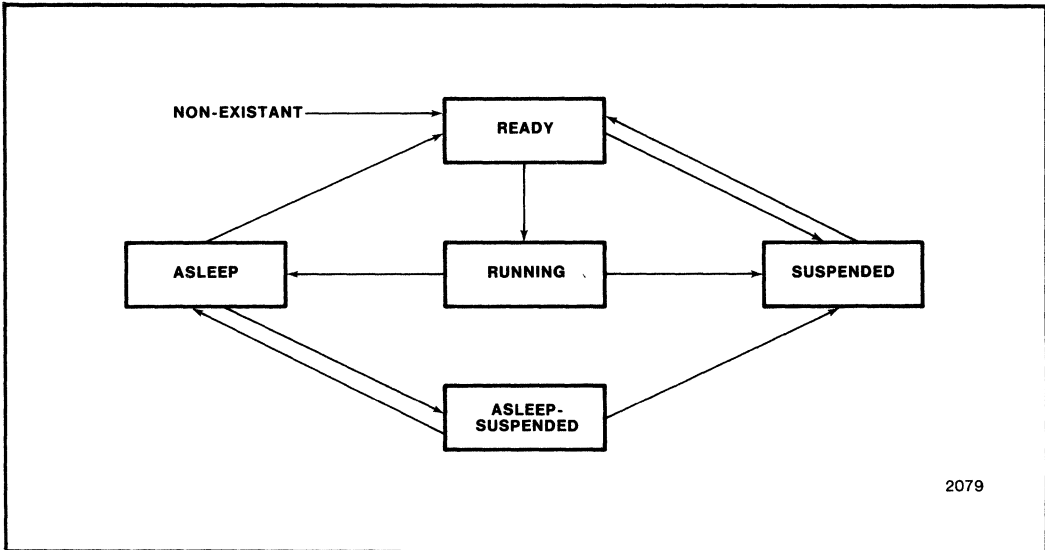


Figure 1. Task State Transition Diagram

the processor when they occur. The interrupt handler may be able to handle the interrupt directly, or it may invoke an interrupt task to handle the interrupt. The interrupt handler will retain control of the processor until the handler exits or a higher priority event occurs.

When mailboxes are used, queues of either tasks and objects can form at the mailbox. Task queues can form at semaphores. Task queues can be priority ordered or FIFO (First In First Out) ordered. This order is specified when the mailbox is created. A FIFO queue on a mailbox can cause a task with a lower priority to execute before a task with higher priority. If both tasks are waiting on the mailbox before continuing execution and the lower priority task is first on a FIFO queue, the lower priority task will execute first. However, when the higher priority task receives the object for which it was waiting, the task now becomes the ready task with the highest priority and can take control of the processor.

SINGLE TASK APPLICATION EXAMPLE

This section explains how to create a task, how to use mailboxes, and how to use the System Debugger. It also covers exception handling, as well as how to configure the iRMX 86 Operating System, and how to link and locate the application job.

The single task example shows how to create a single task that writes to the terminal. The structure of the operating system used for this application example is shown in Figure 2. The code has an initial module,

called Inittask (Appendix A), which is used to provide a stable entry point for the application code. The entry point of Inittask is the start address for the task. In the User Job screen of the ICU, this value must be supplied for the task start address parameter. Inittask calls Onetask (Appendix B) which does all the work of the application. The submit file which links and locates the user job is given in Appendix D.

INITTASK

The initial module, shown in Appendix A, is very simple. It illustrates how to set up a stable entry point for a user job. There is no data in this module, and there are only two calls. Inittask is never changed, and it is linked first, so its entry point is stable no matter what changes are made to the rest of the application code. This approach allows the user job's entry point to be set up only once in the configuration of the operating system, and removes the need to generate a new operating system whenever the application code is changed.

The .MP2 file generated by the LOC86 utility shows that the module has the entry point Inittask at 1500:0002H. This address is used as the task start address in the definition file, in the User Job screen of the ICU. Inittask calls RQ\$END\$INIT\$TASK, which is a requirement for any job created by the root job in the iRMX 86 Operating System. Calling RQ\$END\$INIT\$TASK allows the root task to resume execution and create another first level job. Once RQ\$END\$INIT\$TASK has been called,

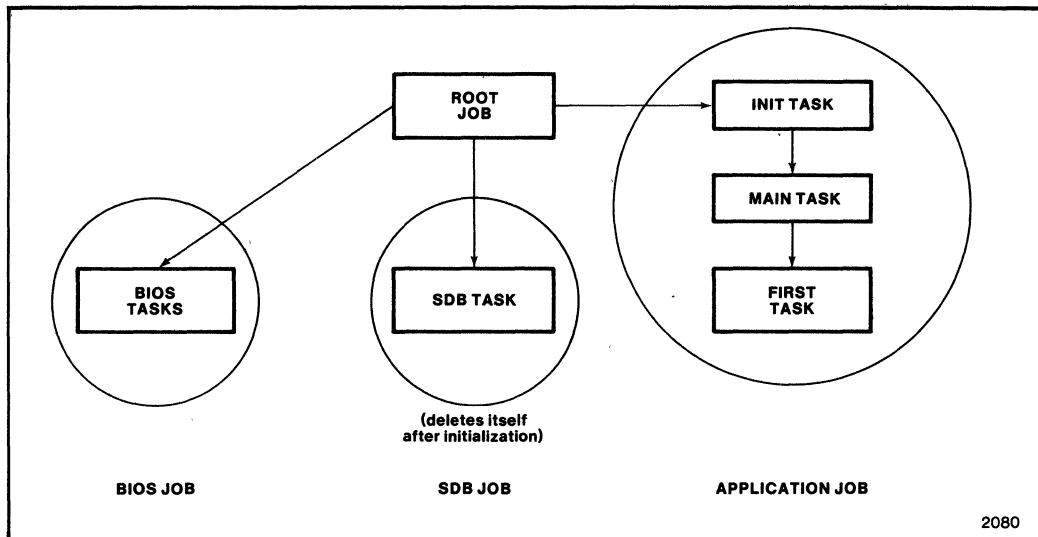


Figure 2. Single Task System

Inittask calls an external procedure called Main\$task, which is the actual code that creates the example task. The same initial module is used with both application examples.

ONETASK

In Appendix B, the code for Onetask is shown. There are two parts to Onetask. First is the main module, called Main\$task. Second is the task which is created by Main\$task, called First\$task. Note that there are really two tasks, only one of which is created specifically in the example code.

Main\$task creates a mailbox and catalogs it in the user job's directory under the name DONEMBX. It uses this mailbox to synchronize Main\$task and the task which is created. It creates First\$task and then waits at the mailbox to receive a message from First\$task to indicate that the task has finished. Main\$task then deletes the task and deletes the mailbox.

Main\$task deletes segments received from the mailbox, and then deletes the mailbox. In the code shown in Appendix B, there is a loop around creating and deleting the task, with the PL/M 86 call CAUSE\$INTERRUPT (3) at each end of the loop. This code as used in this example was for debugging purposes, but could have also allowed stopwatch timing of the routine. Note also that using CAUSE\$INTERRUPT (3) calls in your code will result in all other tasks in the system halting, including those of other users.

At the point where Main\$task calls RQ\$RECEIVES\$MESSAGE to wait at the mailbox, First\$task begins to run. Main\$task has a higher priority than First\$task, so First\$task cannot run until Main\$task either suspends itself or goes to sleep. In this case, Main\$task is waiting on the mailbox, which puts it in the sleeping state, and Main\$task cannot continue until an object is received from the mailbox. This situation gives First\$task a chance to run, since it is now the highest priority ready task.

First\$task creates some mailboxes and segments, and does a lookup to find the mailbox DONEMBX which it must use to communicate with Main\$task. First\$task then physically attaches the terminal, opens a file connection, and writes a buffer to the terminal. Then it closes the connection, deletes the file connection, and detaches the device. It cleans up by deleting the segments and mailboxes it created, and signals Main\$task that it is done by sending a message to the mailbox DONEMBX.

Main\$task receives control of the processor after First\$task sends a message to DONEMBX to indicate completion. Main\$task then deletes the segments and mailboxes which are left, and then deletes the application job. All memory allocated to the application job will then be returned to the root job's memory pool.

If there was an error while running this application, the task would end up looping in one of the 'error' routines. If the application completed successfully and exited, the nucleus idle task for Release 6 of the iRMX 86 Operating System would begin executing.

The Basic I/O System (BIOS) was used in this application to provide immediately visible results. The section which involves using the BIOS is the most complex part of the example code. Many applications will have no need of the BIOS.

These applications were done in the LARGE model of compilation to provide simpler examples. The COMPACT model can be used if the application's code and data are less than 64K each. COMPACT code can usually execute faster because calls will be within the same segment, so won't require changing as many registers to execute the call.

Creating Tasks

To illustrate all the areas that are involved in creating a task, let's go through each of the parameters of the RQ\$CREATE\$TASK system call. The call looks like this.

```
task$token = RQ$CREATE$TASK
              (priority,
               start$address$pointer,
               data$segment,
               stack$pointer,
               stack$size,
               task$flags,
               exception$pointer);
```

The parameters in the RQ\$CREATE\$TASK call are explained below.

Priority: Task scheduling involves setting relative priorities of tasks. Unless a task is involved in processing interrupts, its priority should be between 129 and 255. When a task having a priority in the range 0 to 128 is running, certain external interrupt levels are disabled, depending on the priority. The task for this application used a priority of 202. The initial task itself was given a priority of 82H, or 130, at configuration time.

Start\$address\$pointer: The start address pointer is used to point to the beginning of the task which is being created. In the PL/M 86 LARGE and COMPACT models, the pointer points to the label of the procedure containing the task. The task was a procedure within the same main module for these examples. If the task had been compiled separately, it would have to be defined as an external procedure within the main module which created the task. The actual locations are resolved when the application is linked and located.

Data\$segment: In the PL/M 86 LARGE model, the data segment is set equal to zero when creating the task. Setting the data segment to zero allows the task to set up its own data segment. In other

models of PL/M 86, the user task must explicitly set up its own data segment or the value of the data segment must be obtained from the locate map and used in the call. Refer to the iRMX 86 Configuration Guide, which is part of the *iRMX 86™ Installation and Configuration Guide for Release 6* for more information on how to set up the data segment of a task.

Stack\$pointer: The stack pointer is also set to zero to allow the iRMX 86 Operating System to automatically allocate a stack of size stack\$size. While the task is running, the SS register will show which stack segment is being used for the application task.

Stack\$size: The stack size will need to vary with stack requirements of the task. If the task is reentrant, or makes calls to subroutines with many parameters, or if the task makes iRMX 86 Operating System calls, the amount of stack must be larger than if the task only keeps local variables on the stack.

There are two ways to determine the stack size needed. The first method involves arithmetically determining the stack size needed, based on three things: the number of bytes required for interrupts, the number of bytes required for system calls, and the amount of stack required by the task's code segment. This method is explained in *iRMX 86 Programming Techniques*, which is part of the *iRMX™ 86 Programmer's Reference Manual for Release 6, Part II*. The other method involves choosing a relatively large stack size and reducing it through empirical methods. To use the empirical method, display the stack with a debugger. If there are "C7"s on the stack when the application has completed, that part of the stack hasn't been used. You can also watch the stack pointer, kept in the SP register, to see how low it goes. It will grow toward zero from the value given as the stack pointer when the task is created.

While testing the example application code, the stack size was set to 2000 (7D0H) which was much too large. A stack size of 300 was sufficient for this task.

Task\$flags: Task flags are used in the iRMX 86 Operating System Releases 5 and 6 to tell the nucleus whether the task contains floating point instructions. This task did not, so task\$flags was set to 0. Setting bit 0 of task\$flags to 1 to indicate the use of floating point instructions will result in memory being reserved for the NPX registers. The other bits of task\$flags are reserved. The iAPX 8087 or iAPX 80287 must

be included in the system if floating point instructions are used.

Exception\$pointer: This pointer gives the location of the word where the status of this call will be returned. The result is checked after the call to make sure that an E\$OK was returned. For convenience while debugging, the condition code can also be found in the CX register. The nucleus manual (Chapter 7 for iRMX 86 Release 6) contains a table of exceptional conditions that can be returned and their numeric codes.

Mailboxes

There are two object queues associated with every mailbox. One is a fast queue, which has a fixed length determined when the mailbox is created. The other object queue is an overflow queue, and memory must be allocated for that queue each time it is used. This example used one mailbox to communicate between the original task and the task it created. The fast queue has a length of 4, which is the default value, indicated by the 0 as the first parameter of the create\$mailbox call. If this mailbox frequently had large numbers of objects on its queue, it might have been useful to use a larger fast queue. This approach, of course, means that more memory would be allocated to the mailbox when it is created. In this application, the minimum fast queue length was used.

The DONEMBX mailbox is being used as if it was a semaphore. It is used only to tell the parent task that the child task has completed. First\$task also creates some mailboxes which are used to send information. For instance, when a physical connection is made to the terminal, the mailbox is used to receive a token for the physical connection. Similarly, when a file connection is made, the mailbox receives a file connection token.

Different protocols can be set up to handle objects which are received at a mailbox. In this application, after an object is received from a mailbox, that object is deleted after it has been used in an appropriate manner (such as to extract the file connection token). A protocol can also be set up so that objects are reused. A response can be sent to the task that sent the object. Only Operating System created objects can be sent to a mailbox. Objects should be deleted when they are no longer needed, because they take up memory. Creating extra objects and neglecting to delete them can eventually cause a task to use up all its available memory. The objects which are sent to mailboxes in this example are segments. Examples of how to check for the presence of objects are shown in the following section on using the System Debugger.

Using the System Debugger (SDB)

The System Debugger is a job which is added to the iRMX 86 Operating System at configuration time. The only configurable parameter in the SDB is its interrupt level, which in this example should be the default value of 018H, master interrupt level one. The SDB knows most of the data structures of the iRMX 86 Operating System, and can be used to determine what is happening within the Operating System while an application job is running.

The first thing that must be done when using the SDB is to activate it. If you are writing an application which can be run from the Human Interface, the DEBUG cusp can be used. The example application wasn't run from the Human Interface. Instead, it is built in as a user job. The SDB can be invoked by pressing the front panel interrupt button, or by inserting a 'CAUSE\$INTERRUPT (3)' call into the source code. While bootloading an application, the 'CAUSE\$INTERRUPT (3)' call is a more useful tool, since pressing the interrupt button cannot stop the application at a specific point. The monitor won't get control immediately as it would if the job were loaded from a development system. The 'CAUSE\$INTERRUPT (3)' call can be taken out of the application when the code is debugged. Any use of the SDB should be done only in a single-user system, since the SDB will stop all jobs.

CAUSE\$INTERRUPT (3) is used in three places in the single task application code. The first occurrence is as soon as the module Main\$task is entered, but before any of its code has executed. The second place is before the loop to create and delete the task. The third place is after the loop is completed. During the debugging phase of developing the code, there were also CAUSE\$INTERRUPT (3) calls within the task, to help determine what was happening. Break-points can also be used once the code has been stopped so that the monitor has control. A break-point is set by the monitor command 'g, address'. The code will execute until it gets to that address, and then it will break to the monitor.

The job tree is found by using the SDB command 'vj' for view job. An example follows for the application task.

```
.vj
BFDD
    B68A
    BE9E
```

BFDD is the token for the root job. By knowing the memory pools of each of the layers, and looking at the memory pools of each of the other job tokens, a user can determine that B68A is the user job, (the application code), and BE9E is the BIOS. The 'vt' command will show the current state of each token.

```
.vt BE9E
Object type = 1      Job      (NOTE: This token is for the BIOS.)

Current tasks      0003      Max tasks      FFFF      Max priority    00
Current objects    000A      Max objects    FFFF      Parameter obj   BFB8
Directory size     0000      Entries used   0000      Job flags       0000
Except handler     0EE4:01C0  Except mode    00        Parent job      BFDD
Pool min           0800      Pool max       0800      Initial size    0800
Pool size          0800      Allocated     0077      Largest seg     0764
```

```
.vt B68A
Oject type = 1      Job      (NOTE: This token is for the application job.)

Current tasks      0001      Max tasks      0010      Max priority    00
Current objects    0001      Max objects    0020      Parameter obj   BFB8
Directory size     0010      Entries used   0001      Job flags       0001
Except handler     0EE4:01C0  Except mode    00        Parent job      BFDD
Pool min           0500      Pool max       0500      Initial size    0500
Pool size          0500      Allocated     01FD      Largest seg     0303
```

Notice from these tables that the application job may have been created with too much memory. It is only using 01FD of memory (Allocated), and has a minimum and a maximum pool size of 0500H (Pool min, Pool max). A pool size of 0250H would probably have been sufficient. If this command was executed early in the application, all the objects might not be created yet. So the job might require more memory than is currently being used at some point in its existence.

The command 'vo job-token' shows all the objects that have been created in a job. The token for each object contained by that job will be shown, listed after a designation for which type of object it is. You can check for the presence of leftover segments at the end of a task's execution with this command. Before executing the application code, the following list shows what the command and its results look like.

```
.vo B68A
Child Jobs:
Tasks:      B491
Mailboxes:
Semaphores:
Regions:
Segments:
Extensions:
Composites:
```

By using the SDB command 'vk', the tasks that are ready or sleeping can be seen. After running the user job to completion, the result of the 'vk' command looks like this:

```
.vk
Ready Tasks:
Sleeping Tasks: BE64 BE2E BE06 ICCF
```

If the code hadn't completed executing, results like this might indicate deadlock. The tasks would have to be examined to see how long they were asleep. If they are all asleep forever, nothing further will happen without an external event.

Exception Handling

In this example, the default system exception handler is used, and the exception mode is set to 'never'. This setting means that the application code either wishes to handle exceptions in-line or through a call to an exception handler. The system exception handler will not be invoked for errors. If the handler were invoked, it would simply delete the task containing the call which caused the exception.

This application handles exceptions in line. After each system call, the task checks the status word for E\$OK. If an exception is detected, the task will jump to ERROR and loop there forever. This technique is to help identify where an error occurred, and is useful for debugging the application. The CX register contains the status returned from the call, so it is possible to find out which error occurred by using the monitor command 'x' to display the registers. The problem then becomes the following: to find out where the error occurred. This procedure usually involves stepping through the code, or setting several breakpoints (using the monitor commands to break at given points).

It is useful to insert 'CAUSE\$INTERRUPT (3)' calls at points when the task is likely to transfer control (such as after sending or receiving messages). When the application is running properly, remove the 'CAUSE\$INTERRUPT (3)' calls to allow the code to execute unattended. Breakpoints can also be used to monitor the code.

If you write your own exception handler, you have to decide upon which conditions it will be invoked. You must compile, link and locate the exception handling code, and determine the starting address of the exception handler. This value must be configured into the operating system as the User Job's exception handler address. Each task can also have its own exception handler by using the call `RQSETEXCEPTION$HANDLER`.

An exception handler is the preferred method of handling exceptions, but exception handlers are beyond the scope of this application note. An exception handler would eliminate the need for GOTOs in the code. GOTOs are considered bad programming practice in most structured languages.

MULTIPLE TASK APPLICATION EXAMPLE

For the second example, the same initial code, `Inittask`, was used. The main module is called `Main$task` as it was for the single task example. The same configuration of the iRMX 86 Operating System is used, since the start address is the same for both applications. The code for this example, called `Tasks`, is shown in Appendix C. The main module for this example creates three tasks, and lets them do the work. A diagram of the system is shown in Figure 3.

The three tasks are set up according to their function. This very simple example of a machine control system makes the classic product, widgets. One task is the supervisor and controls the other two tasks. It sends messages to the other tasks to tell them what to write or how many widgets to make. The second task, called `IO$task`, outputs messages which it has received from `Supervisor$task` to the terminal. The third task, called `Widget$task`, makes widgets. In this example, `Widget$task` is essentially a no-op task, but it could easily be replaced with code that implemented a real application.

MAIN\$TASK

The code in the second example has an initial module which creates three tasks, and then waits at a mailbox for them to complete execution. The initial module then deletes the three tasks and the mailboxes it has created. When everything is cleaned up, it deletes the application job. The initial task not only creates the mailbox it needs for signaling when the tasks are done, but it also creates the other four mailboxes that are used by the three tasks to communicate with each other. The mailboxes are cataloged in the user job's directory, and each task must look up the mailboxes it needs to use.

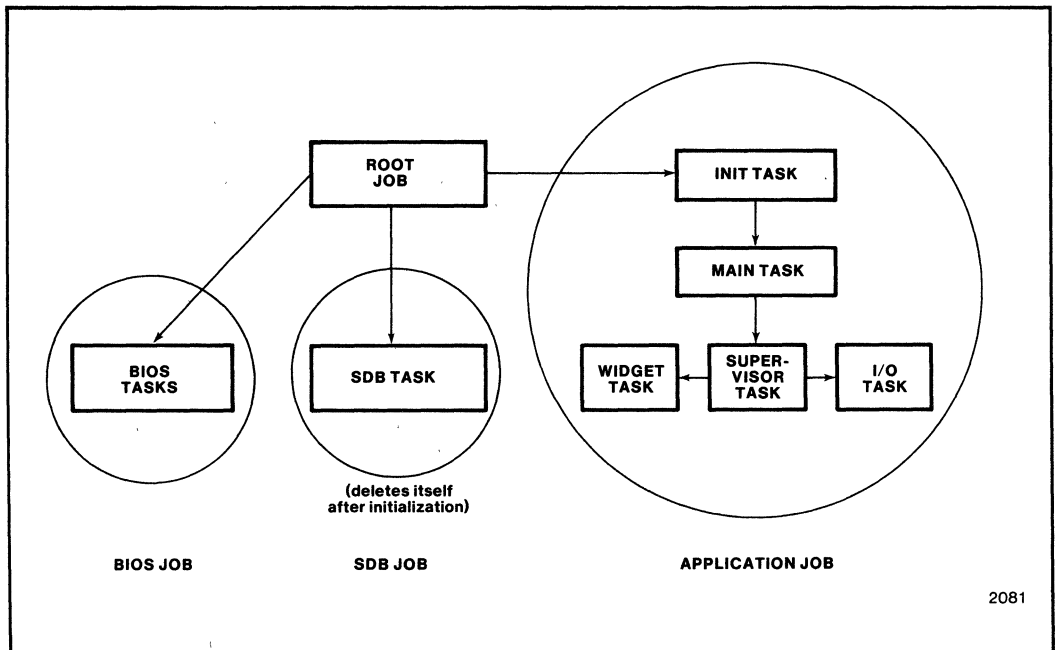


Figure 3. Multiple Task System

Pseudocode for Main\$task

```

create mailboxes
catalog mailboxes in job's directory
create Supervisor$task
create Widget$task
create IO$task
receive message from Supervisor$task (done)
delete tasks
delete mailboxes
delete myself

```

SUPERVISOR\$TASK

Supervisor\$task is created with the highest priority of the three tasks. It happens to be created first, but because it has the highest priority it would execute first regardless of when it was created. Supervisor\$task controls the other two tasks. The other two tasks are created with priorities lower than both Supervisor\$task and the creating task. If one of them had been created with a priority higher than the creating task and had been the first one created, it would have begun executing as soon as it was created, preventing the creator task from creating Supervisor\$task.

Pseudocode for Supervisor\$task

```

lookup mailboxes in job's directory
do 1 to 10
    send message to Widget$task (make widget)
    send message to IO$task (making widget, message)
    receive message from Widget$task (done)
    receive message from IO$task (done)
end
send message to Widget$task (cleanup)
send message to IO$task (cleanup)
receive message from Widget$task (done)
receive message from IO$task (done)
cleanup by deleting segments
send message to Main$task (done)

```

The supervisor allows Widget\$task and IO\$task to work as independently as possible. It sends messages to both of them, and then waits for both to reply before it repeats the loop. This approach allows the tasks to execute when they can get the processor, completely independently of each other. We can also look at the ready list at various points in the code and see which task is executing. Tasks will not always execute in the same order with this method. Because each task is required to wait for a message indicating that it may run, IO\$task cannot inform the console that a widget is being made any sooner than Widget\$task begins to make the widget. If the sends and receives had instead been paired (sending then receiving from the same task) Supervisor\$task could have guaranteed which task would be executing at any given point in the code.

Supervisor\$task is sending information in the mailboxes to each task. In the single task example, a simple segment with no information content was sent between the creating task and the created task to signal that the created task was done executing. A semaphore could have been used just as well in that example. In this multitasking example a semaphore would not work. Information is being passed to each of the tasks to tell them whether or not to continue executing, or that they should clean up their environments. In addition, Supervisor\$task is passing to IO\$task a message that will be printed out. To pass the message, a structure is used. The structure contains values, rather than tokens for objects. The values are moved into the structure with the MOVBL/M call prior to sending the structure's token through the mailbox. Structures which contain tokens should be avoided, especially when using mailboxes which communicate between jobs. On more advanced processors than the iAPX 86, operating systems may be implemented in which jobs have disjoint address spaces. In that case, a token may have different values in different jobs. The iRMX 86 Operating System will expect this convention to be followed but will not enforce it. Future operating systems may enforce the convention. This example is completely contained within one job, so it isn't quite as restricted.

WIDGET\$TASK

Widget\$task is extremely simple for this example. This portion in a real application would probably be the most complex, since it would involve the machine interfaces for a control process. It could also include any mathematical calculations which need to be done.

Pseudocode for Widget\$task

```

lookup mailboxes
receive message from Supervisor$task (make
    widget)
do while make widget is true
    send message to Supervisor$task (done)
    receive message from Supervisor$task (make
        widget)
end
cleanup the environment
send message to Supervisor$task (done)

```

IO\$TASK

This task is very similar to First\$task in the first example. The main difference between the two is the way the information for the messages is given to the tasks. In Onetask, the message was defined within the task. In IO\$task, the information for the message is passed to IO\$task via a mailbox from Supervisor\$task. This task illustrates how to do simple I/O with just the BIOS.

Pseudocode for IO\$task

```

look up mailboxes
receive message from supervisor (making
    widget, message)
create user
physically attach terminal
get device connection
create file
get file connection
open file
receive message from Supervisor$task (making
    widget, message)
do while making widget is true
    write message (Making widget)
    delete segments
    send message to Supervisor$task (done)
    receive message from Supervisor$task (making
        widget, message)
end
close file
delete file connection
detach device
delete user
cleanup environment by deleting segments
send message to Supervisor$task (done)

```

Mailboxes

In this application, mailboxes are used for several purposes. Their most obvious purpose is to send information between tasks. Less obvious but more important is the role they play in allowing mutual exclusion and synchronization between tasks.

The simplest messages in the application return an empty segment to the mailbox to indicate that the task has completed some portion of work, and the receiving task can continue. This empty segment is the kind of message that Widget\$task and IO\$task send to Supervisor\$task, and the kind of message that Supervisor\$task sends to Main\$task when it has completed execution. This exchange could be accomplished just as well by using a semaphore rather than a mailbox.

The more complex messages contain some information. In this example, the message contains information indicating that the task should continue executing a loop, or that it is time to clean up the environment and exit. The message that was sent between Supervisor\$task and IO\$task also contained the message that Supervisor\$task wanted IO\$task to print out. This example illustrates the kind of information that can be passed between tasks using mailboxes.

Mailboxes are also used in this application to implement mutual exclusion and synchronization between the tasks. One alternative implementation which is

not shown here keeps the three tasks all at the same priority, and uses mailboxes to allow the tasks to execute in a strictly defined order. The implementation shown in this example is quite general purpose, and doesn't use as many mailboxes as the alternative implementation would. This implementation also allows the tasks to have more freedom in when they can run, and uses the processor more efficiently if one of the tasks is blocked while doing I/O.

Deadlock

As more tasks are used, and as more mailboxes are used to communicate between the tasks, the possibility of deadlock increases. Deadlock usually is caused by faulty design, and may appear when debugging of the code begins. Evidence of possible deadlock occurs when all the tasks are sleeping, and no tasks are ready when you use the SDB command 'vk' to check what's going on. This situation can be caused by sending a message to the wrong mailbox, or by not creating segments to send within a loop that is sending messages. An example of deadlock can be obtained by changing the code in Supervisor\$-task in some minor ways.

Executes correctly:

```

Do i = 1 to 10
    create segments
    send message to Widget$task (make widget)
    send message to IO$task (making widget, message)
    receive message from Widget$task (done)
    delete segment
    receive message from IO$task (done)
    delete segment
end

```

Causes deadlock:

```

create segments
Do i = 1 to 10
    send message to Widget$task (make widget)
    delete segment
    send message to IO$task (making widget, message)
    delete segment
    receive message from Widget$task (done)
    receive message from IO$task (done)
end

```

The second code example results in deadlock because the object which is sent to the mailbox is created outside the loop. Once it is sent, there is no longer an object to send, and the receiving task can't continue unless it has a timeout specified because it never receives another object. With a different synchronization scheme, if the receiving task hadn't deleted the message, that object could have been sent again.

Key places to watch for deadlock in the code are where some communication occurs between the tasks. Other situations that can cause deadlock are tasks needing the same resources, such as a unit from a semaphore, or a region. Insufficient memory will cause an E\$MEM error rather than deadlock.

Initialization

Supervisor\$task controls the other two tasks. This control is necessary since the tasks cannot execute more than a few lines of code without receiving a message from the supervisor. The tasks execute a few lines of code to lookup the mailbox which they will use to communicate with the other tasks. Then they can make the RQ\$RECEIVE\$MESSAGE call. The task with the highest priority which was created first, and as a result, has been ready the longest, will execute first. Since Supervisor\$task was created with a higher priority than the other two tasks, it will execute first. By the time it gives up control of the processor, it has already sent messages to both of the other tasks. The task that has been ready the longest at this point will execute first. In this example, the first executed task happens to be Widget\$task, since it was created before IO\$task was.

Debugging

The same techniques are used to debug a multiple task application as were discussed in the single task example. Look at the .MP2 file before beginning debugging, and find the entry points to each task. The .MP2 file is produced as a result of the LOC86 step in building the application. Use 'CAUSE\$INTERRUPT (3)' calls at the beginning of each task, and keep track of which task is executing at a given time. One technique that was used in this application to make it easier to debug was to send all errors to an error routine within each task. The error routine was different in each task (outputting A1H for the first task, A2H for the second task, etc.). It was immediately obvious by looking at the disassembled code containing the call which task caused the exception.

As a second debug alternative, the iRMX 86 Dynamic Debugger is also useful in a multiple task application. Rather than halting the entire system like the SDB does, the Dynamic Debugger allows users to examine vital system objects while the system is running. The Dynamic Debugger must be configured into the Operating System with its own terminal handler and its own terminal if the BIOS is used.

CONFIGURATION

There are two steps involved in configuration: configuring the operating system; and compiling, linking

and locating the application code. The same definition file was used for both applications, and the same submit file was used to link and locate the applications.

Configuration of the iRMX™ 86 Operating System

For both examples in this application note, the same operating system configuration was used. The layers used are the Nucleus (for scheduling and intertask communication) and the BIOS (so that I/O could be done). The SDB, for debugging the code, was also configured into the operating system as a user job. The only device driver required is the terminal driver. The listing of the operating system definition file is shown in Appendix E.

User Jobs

This application code is configured into the operating system as a user job. The Interactive Configuration Utility (ICU) requires information to be given about the user job, and sets up a %JOB macro for the job. However, the ICU does not set aside memory for the user job, and it does not link and locate the job as it does for the layers of the operating system.

An application that uses only nucleus and BIOS calls is a user job. If the application uses the EIOS or the Human Interface, it is an I/O job and must be configured as a child job of the EIOS. Applications can also be run from the Human Interface level, as jobs under the Human Interface. In a real-time application, treating the application as a user job or user jobs is usually most appropriate. During development, however, the application could be run under the Human Interface. This technique would eliminate rebooting after each code change or trial run to test the application.

The following parameters appear in the User Jobs screen in the ICU. Each parameter is defined and explained in the context of the example application.

JOB NAME (NAM)

The first question in the User Jobs screen for the iRMX 86 Release 6 ICU is Job Name. This question is optional, and is just used for the user to keep track of which user job is being configured in the screen. It is not used by the ICU.

OBJECT DIRECTORY SIZE (ODS)

Object directory size refers to how many objects can be cataloged in the job's directory. In the first example, the only object that is being cataloged is

the mailbox DONEMBX which is used to let the main task know that First\$task has completed. In the second example, five mailboxes are cataloged, so the object directory size doesn't have to be very big for this application. The default value for this parameter is 10H, which is large enough for this application. If you have a large application and many objects are cataloged, this number would have to be increased. For small applications, a small value can be used to conserve memory. Memory is allocated for the object directory of a job when the operating system is initialized.

POOL MINIMUM, POOL MAXIMUM (PMI, PMA)

Pool minimum and pool maximum are closely related. They specify the amount of memory that the job requests from its parent. For this example, a pool minimum and pool maximum size of 500H was chosen. The minimum and maximum should be set to the same value in the user job to avoid memory fragmentation. The Human Interface is usually the only exception to this rule.

MAXIMUM OBJECTS (MOB)

This parameter defines how many objects can be created by the application. For this application, the maximum objects was set to 20H, but only 10 objects existed at any given time. In a larger application, of course, 20H could easily become insufficient. To determine how many objects are being used at a given point in time, use the SDB command 'vo user-job-token'.

MAXIMUM TASKS (MTK)

The next parameter specifies the maximum number of tasks which can be created by tasks in this user job. This application used the default value of 10H, but could have used 4H, since only three tasks were created (one task was created by the root job to be the user job task). The definition file was being used by both the single task example and the multiple task example, so the default value is appropriate.

MAXIMUM PRIORITY (MPR)

The maximum priority parameter refers to the maximum priority allowed of any task in this job which is created. This parameter was set to 0H where 0H indicates that the priority of the root job is the maximum allowable priority of the tasks. The root job's priority in this operating system is 00H, which doesn't limit the maximum allowable priority.

ADDRESS OF EXCEPTION HANDLER (AEH)

This application is using the system exception handler, so the address of exception handler used is

0000H:0000H. If the user job had its own exception handler, the correct address of that exception handler would have to be found by first linking and locating the application code and the exception handler, and then looking in the .MP2 file for the address of the exception handler.

EXCEPTION MODE (EM)

The exception mode is set to 'never' for this user job, indicating that the exception handler won't be invoked for any kind of error condition. Instead, exceptions will be handled in-line in the example code.

PARAMETER VALIDATION (PV)

Parameter validation is used by the nucleus to determine if the parameters passed in system calls are valid. This question should be answered yes until the application code is debugged. If the BIOS or other upper layers are being used in the operating system, parameter validation should be enabled even when the application code is debugged, or the Operating System will not work. If parameter validation is turned off, the nucleus calls will execute faster, so setting parameter validation to 'no' can improve performance.

TASK PRIORITY (TP)

Task priority sets up the static priority of the initial task which is created for the user job. For this application, the priority of the task is set to 82H, or 130.

TASK START ADDRESS (TSA)

The Task Start Address is the start address of the job's initialization task. This address is determined from the .MP2 file after locating the application code. For both example applications this address was 1500H:0002H.

DATA SEGMENT BASE (DSB)

The data segment base is set to 0000H, which allows the task to set up the data segment base for the initial task of the user job. Since the LARGE model of compilation was used, the parameter can be set to zero.

STACK SEGMENT ADDRESS (SSA)

The stack segment address is also set to 0000H:0000H to allow the nucleus to allocate a stack segment to the task and take care of initializing the SS and SP registers. This setting permits dynamic stack allocation and deallocation.

STACK SIZE (SS)

The stack size for the task is set to 300, which is the amount that is considered necessary to make any nucleus system calls. Since this application was very small, it didn't need a very large stack. If a job used a lot of subroutines and nested procedures with many parameters, or if the job was recursive, the amount of stack needed could increase.

NUMERIC PROCESSOR EXTENSION USED (NPX)

The 8087 Numeric coprocessor was not used in this application task. If any floating point functionality is needed within a task, this parameter should be set to yes in the configuration process.

Ram

The last parameter which must be considered when creating the definition file for the application is the amount of RAM required and where it is located. This is a parameter in the memory screen of the ICU. Remember that the ICU does not locate the user job for you, so memory must be specifically set aside to be used by the user job. In this application, the RAM that was used was from 0104H to 1500H, and from 1800H to F7FFH. The user job was allowed to use RAM from 1500H to 1800H (these numbers are specified in paragraphs of 16 bytes each). The operating system itself was put into the memory from 104H to just under 1500H. This location can be determined by looking at the .MP2 files for each of the layers of the operating system after completing the configuration.

Linking and Locating the Application

The submit file that was used for this application is shown in Appendix D. Note first of all that instead of using the name of the application code program, a %0 was used. This convention allows you to invoke the submit file with a parameter which is the name of the application code program. The same submit file was used for both examples. Note that Inittask is linked with the INITCODE option. This is necessary with LINK86 v 2.0 and LOC86. The INITCODE option is not necessary with other versions of LINK86.

While the submit file is running, some warnings will be generated. The following errors are normal and should be ignored.

WARNING 12: UNRESOLVED SYMBOLS

WARNING 26: DECREASING SIZE OF
SEGMENT
SEGMENT: STACK

WARNING 66: START ADDRESS NOT SPECIFIED IN OUTPUT MODULE

CONCLUSION

This application note is an introduction to the basic functions of the iRMX 86 Operating System Nucleus. Task scheduling and memory management functions were covered in detail. Two applications were discussed, using some pieces of code in common. The functions involved in developing and testing real-time code were explained while using the application code for examples and reference. Configuration of the application operating system was also covered in detail.

The examples shown in this application note illustrate how to develop a real-time application. The first example shows how to use the nucleus and BIOS to do simple I/O at the lowest, most optimizable level. The second example builds on the concepts developed in the first example, expanding the application to a more realistic process control situation.

Both examples shown earlier are fairly simple. They illustrate what has to be done to create and use a task. They are good examples for a user who has written a limited amount of PL/M 86 code using the nucleus system calls. The multiple task example would be a good foundation for a process control application. Each of the tasks in the second example shows a major function of real time code, demonstrating control, I/O, and supervisory functions.

The iRMX 86 Operating System is ideally suited to multi-tasking, real-time applications. The ability to use the same system for both development and as the target system is a great benefit to the development engineer and the company which is developing real-time applications. The modularity provided by the iRMX 86 Operating System and the PL/M 86 language make it easier to develop code for one application, then modify it for another application. The ultimate benefits to users are reduced development time, added cost savings, and shorter time-to-market for new products.

APPENDIX A:
APPENDIX B:
APPENDIX C:
APPENDIX D:
APPENDIX E:
APPENDIX F:

APPENDIX A

PL/M-86 COMPILER single task creation; for ap note

iRMX 86 PL/M-86 V2.1 COMPILATION OF MODULE INITTASK
 OBJECT MODULE PLACED IN INITTASK.OBJ
 COMPILER INVOKED BY: :LANG:p1m86 INITTASK.P86

```
$large rom debug
$title('single task creation; for ap note')
```

```

/*****
 * This is an example to be used for an ap note on task *
 * scheduling. *
 * Cathy Lundberg 03/22/84 *
 *****/
```

```
1    inittask: do;
      $INCLUDE (/RMX86/INC/NEINIT.EXT)
      = $SAVE NOLIST
```

```
4    1    main$task: PROCEDURE EXTERNAL;
5    2    END main$task;
```

```

/*****
 * This separate module is used to keep the user job's *
 * start address constant while changing the code. *
 * This module has no data or constants in it, all it *
 * does is call the main routine, main$task, after *
 * calling rq$end$init$task. *
 *****/
```

```
6    1    begin: PROCEDURE PUBLIC;
```

```
7    2    CALL rq$end$init$task;
8    2    CALL main$task;
9    2    END begin;
```

```
10   1    END inittask;
```

MODULE INFORMATION:

```

CODE AREA SIZE       = 0018H     24D
CONSTANT AREA SIZE  = 0000H     0D
VARIABLE AREA SIZE  = 0000H     0D
MAXIMUM STACK SIZE  = 0008H     8D
36 LINES READ
0 PROGRAM WARNINGS
0 PROGRAM ERRORS
```

DICTIONARY SUMMARY:

112KB MEMORY AVAILABLE
3KB MEMORY USED (2%)
0KB DISK SPACE USED

END OF PL/M-86 COMPILATION

APPENDIX B

PL/M-86 COMPILER single task creation; for ap note

iRMX 86 PL/M-86 V2.1 COMPILATION OF MODULE ONETASK
 OBJECT MODULE PLACED IN ONETASK.OBJ
 COMPILER INVOKED BY: :LANG:PLM86 ONETASK.P86

```

1      $large rom debug
      $title('single task creation; for ap note')
      onetask: DO;

      /*****
      * This is an example to be used for an ap note on task *
      * scheduling.                                           *
      * Cathy Lundberg 03/22/84                               *
      *****/

/* The include for LTKSEL.LIT must be done before any other includes,
 * because anything that uses TOKENS must have the data type defined
 * for a token. */

      $INCLUDE (/RMX86/INC/LTKSEL.LIT)
      = $SAVE NOLIST
      $INCLUDE (/RMX86/INC/NEXCEP.LIT)
      = $save nolist
      $INCLUDE (/RMX86/INC/NUC.EXT)
      = $SAVE NOLIST
      $INCLUDE (/RMX86/INC/BIOS.EXT)
      = $SAVE NOLIST

/*****
 * main$task is the procedure that is called by inittask to create the *
 * task 'first$task'. It creates a mailbox, creates the task, and then *
 * waits at the mailbox, allowing first$task to execute. When first$task *
 * finishes and sends a message to the mailbox, control is returned to *
 * main$task, and it deletes first$task and the mailbox.         *
 *****/

280  1      main$task: PROCEDURE REENTRANT PUBLIC;

281  2      DECLARE job                TOKEN,
           data$seg                    TOKEN,
           user$token                  TOKEN,
           taska                       TOKEN,
           done$writing$mbx            TOKEN,
           resp                         TOKEN;

282  2      DECLARE task$flags          WORD,
           status                      WORD,
           i                           WORD;

```

```

283 2   DECLARE seg$pointer      POINTER,
        start$address        POINTER,
        taska$ptr            POINTER,
        stack$pointer        POINTER;
284 2   DECLARE stack$size$300 LITERALLY '300';
285 2   DECLARE priority$level$202 LITERALLY '202';
286 2   DECLARE iors$token      SELECTOR;
287 2   DECLARE done$obj       TOKEN AT(@iors$token);
288 2   DECLARE iors            BASED iors$token STRUCTURE
        (status              WORD,
         unit$status         WORD,
         actual              WORD);

289 2       data$seg = 0; /* task sets up own data segment */
290 2       stack$pointer = 0; /* automatic stack allocation */
291 2       task$flags = 0; /* no floating point */
292 2       job = 0; /* catalog object in containing job */

293 2   CAUSE$INTERRUPT (3);
        /* create a mailbox to use for letting this task know
        * that the task it created is done writing. */
294 2       done$writing$mbx = rq$create$mailbox (0, @status);
295 2       IF status <> E$OK THEN GOTO error;
297 2       CALL rq$catalog$object (0, done$writing$mbx, @(7,'DONEMBX'),
        @status);
298 2       IF status <> E$OK THEN GOTO error;
300 2       CAUSE$INTERRUPT (3);
301 2       DO I = 1 TO 1000;
        /* Now create taska */
302 3       taska = rq$create$task (priority$level$202, @first$task,
        data$seg, stack$pointer,
        stack$size$300, task$flags, @status);
303 3       IF status <> E$OK THEN GOTO error;

305 3       done$obj = rq$receive$message (done$writing$mbx, Offfffh,
0, @status);
306 3       IF status <> E$OK THEN GOTO error;

308 3       CALL rq$delete$segment (done$obj, @status);
309 3       IF status <> E$OK THEN GOTO error;
311 3       CALL rq$delete$task (taska, @status);
312 3       IF status <> E$OK THEN GOTO error;
314 3       END; /* of DO WHILE loop */
315 2       CAUSE$INTERRUPT (3);
316 2       CALL rq$delete$mailbox (done$writing$mbx, @status);
317 2       IF status <> E$OK THEN GOTO error;
319 2       GOTO ok;

320 2   error: /* output to usart to determine which */
        DO WHILE 1; /* task had the error. For debugging.*/
321 3       OUTPUT(9CH) = OAAH;
322 3       END;

```

```

323  2      ok:
          CALL rq$delete$job (0, @status); /* delete myself */

324  2      END main$task;

/*****
* FIRST$TASK is the task which is created by main$task. It creates
* the necessary mailboxes and segments, and then attaches the terminal
* physically. It then creates a file so that it has a file connection.
* It opens the file connection, and writes the contents of a buffer to
* the terminal. Then it closes the file connection, deletes the
* device connection, and detaches the device. Last, it looks up the
* mailbox created in main$task and sends a message to the mailbox.
* This allows control to return to main$task.
*****/

325  1      first$task: PROCEDURE REENTRANT PUBLIC;

326  2      DECLARE  job          TOKEN,
                  mbx$token    TOKEN,
                  seg$token    TOKEN,
                  user$token   TOKEN,
                  file$connection TOKEN,
                  device$connection TOKEN,
                  done$token   TOKEN,
                  done$writing$mbx TOKEN;

327  2      DECLARE  status      WORD;
328  2      DECLARE  hard        BYTE;
329  2      DECLARE  iors$token  TOKEN;
330  2      DECLARE  dev$conn$object TOKEN AT (@iors$token),
                  file$conn$object TOKEN AT (@iors$token),
                  object        TOKEN AT (@iors$token);

331  2      DECLARE  iors        BASED  iors$token    STRUCTURE
                  (status      WORD,
                   unit$status WORD,
                   actual      WORD);

332  2      DECLARE  buffer      BASED          seg$token (1) BYTE;

333  2      DECLARE  user$object  STRUCTURE
                  (length      WORD,
                   count       WORD,
                   id (1)      WORD);

334  2      DECLARE message(*)   BYTE    DATA ('SINGLE TASK TEST');

335  2          user$object.length = 1;
336  2          user$object.count  = 1;
337  2          user$object.id(0)  = OFFFH;

338  2          job = 0;          /* catalog object in containing job */
339  2          hard = OffH;     /* request a hard detach of the device */

```

```
340 2      user$token = rq$create$user (@user$object, @status);
341 2      IF status <> E$OK THEN GOTO error;
343 2      mbx$token = rq$create$mailbox (0, @status);
344 2      IF status <> E$OK THEN GOTO error;
346 2      seg$token = rq$create$segment ( 48, @status);
347 2      IF status <> E$OK THEN GOTO error;

349 2      CALL rq$a$physical$attach$device ( @(2,'TO'), 1, mbx$token,
                                         @status);
350 2      IF status <> E$OK THEN GOTO error;
352 2      dev$conn$object = rq$receive$message (mbx$token, OFFFFH, 0,
                                         @status);
353 2      IF status <> E$OK THEN GOTO error;
355 2      device$connection = dev$conn$object;

356 2      CALL rq$a$create$file (user$token, device$connection,
                              0, 0, 0, 0, 0, mbx$token, @status);
357 2      IF status <> E$OK THEN GOTO error;
359 2      file$conn$object = rq$receive$message (mbx$token, OFFFFH, 0,
                                         @status);
360 2      IF status <> E$OK THEN GOTO error;
362 2      file$connection = file$conn$object;

363 2      CALL rq$a$open (file$connection, 2, 0, mbx$token, @status);
364 2      IF status <> E$OK THEN GOTO error;
366 2      object = rq$receive$message (mbx$token, OFFFFH, 0, @status);
367 2      IF status <> E$OK THEN GOTO error;
369 2      CALL rq$delete$segment (object, @status);
370 2      IF status <> E$OK THEN GOTO error;

372 2      CALL movb( @message, @buffer, SIZE(message));
373 2      CALL rq$a$write (file$connection, @buffer, size(message),
                       mbx$token, @status);
374 2      IF status <> E$OK THEN GOTO error;
376 2      object = rq$receive$message (mbx$token, OFFFFH, 0, @status);
377 2      IF status <> E$OK THEN GOTO error;
379 2      CALL rq$delete$segment (object, @status);
380 2      IF status <> E$OK THEN GOTO error;

382 2      CALL rq$a$close (file$connection, mbx$token, @status);
383 2      IF status <> E$OK THEN GOTO error;
385 2      object = rq$receive$message (mbx$token, OFFFFH, 0, @status);
386 2      IF status <> E$OK THEN GOTO error;
388 2      CALL rq$delete$segment (object, @status);
389 2      IF status <> E$OK THEN GOTO error;

391 2      CALL rq$a$delete$connection (file$connection, mbx$token,
                                    @status);
392 2      IF status <> E$OK THEN GOTO error;
394 2      object = rq$receive$message (mbx$token, OFFFFH, 0, @status);
395 2      IF status <> E$OK THEN GOTO error;
397 2      CALL rq$delete$segment (object, @status);
398 2      IF status <> E$OK THEN GOTO error;
```

```

400 2      CALL rq$a$physical$detach$device (device$connection, hard,
                                         mbx$token, @status);
401 2      IF status <> E$OK THEN GOTO error;
403 2      object = rq$receive$message (mbx$token, OFFFFH, 0, @status);
404 2      IF status <> E$OK THEN GOTO error;
406 2      CALL rq$delete$segment (object, @status);
407 2      IF status <> E$OK THEN GOTO error;

409 2      CALL rq$delete$mailbox (mbx$token, @status);
410 2      IF status <> E$OK THEN GOTO error;

412 2      CALL rq$delete$user (user$token, @status);
413 2      IF status <> E$OK THEN GOTO error;
415 2      CALL rq$delete$segment (seg$token, @status);
416 2      IF status <> E$OK THEN GOTO error;
418 2      done$token = rq$create$segment ( 16, @status);
419 2      IF status <> E$OK THEN GOTO error;
421 2      done$writing$mbx = rq$lookup$object (0, @(7,'DONEMBX'), 500,
                                         @status);

422 2      IF status <> E$OK THEN GOTO error;
424 2      CALL rq$send$message (done$writing$mbx, done$token, 0,
                              @status);

425 2      IF status <> E$OK THEN GOTO error;
427 2      CALL rq$delete$segment (done$token, @status); /* this code */
428 2      IF status <> E$OK THEN GOTO error;           /* shouldn't be*/
430 2      CALL rq$delete$mailbox (done$writing$mbx, @status);
431 2      IF status <> E$OK THEN GOTO error;           /* executed */
433 2      GOTO ok;

434 2      error:                                     /* output to usart for debugging */
          DO WHILE 1;
435 3          OUTPUT(9CH) = 0AAH;
436 3      END;

437 2      ok:
          DO;
438 3          CALL rq$suspend$task (0, @status ); /* suspend myself */
439 3      END;
440 2      END first$task;
441 1      END onetask;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 0554H   1364D
CONSTANT AREA SIZE = 0023H    35D
VARIABLE AREA SIZE = 0000H     0D
MAXIMUM STACK SIZE = 0040H    64D
1520 LINES READ
0 PROGRAM WARNINGS
0 PROGRAM ERRORS

```

DICTIONARY SUMMARY:

84KB MEMORY AVAILABLE
20KB MEMORY USED (23%)
0KB DISK SPACE USED

END OF PL/M-86 COMPILATION

APPENDIX C

PL/M-86 COMPILER three task creation; for ap note

iRMX 86 PL/M-86 V2.1 COMPILATION OF MODULE TASKS
 OBJECT MODULE PLACED IN TASKS.OBJ
 COMPILER INVOKED BY: :LANG:PLM86 TASKS.P86

```

1      $large rom debug
      $title('three task creation; for ap note')
      tasks: DO;

/*****
 * This is an example to be used for an ap note on task *
 * scheduling.                                           *
 * Cathy Lundberg 04/23/84                               *
 *****/

/* The include for LTKSEL.LIT must be done before any other
 * includes, because anything that uses TOKENS must have the
 * data type defined for a token. */

      $INCLUDE (/RMX86/INC/LTKSEL.LIT)
=     $SAVE NOLIST
      $INCLUDE (/RMX86/INC/NEXCEP.LIT)
=     $save nolist
      $INCLUDE (/RMX86/INC/NUC.EXT)
=     $$SAVE NOLIST
      $INCLUDE (/RMX86/INC/BIOS.EXT)
=     $$SAVE NOLIST

/*****
 * MAIN$TASK is the procedure that is called by inittask to create the *
 * three tasks. IO$task outputs to the screen. Widget$task makes *
 * widgets. Supervisor$task controls what IO$task and widget$task are *
 * doing. When widget$task has made X widgets, supervisor$task sends *
 * a message to IO$task saying that X widgets have been made, and the *
 * message is printed by IO$task. Then it sends a message to *
 * main$task to tell it that the tasks are done executing. Main$task *
 * has been waiting at a mailbox for that welcome news, and when it *
 * receives the message, it deletes all three tasks. *
 *****/

280  1      main$task: PROCEDURE REENTRANT PUBLIC;

281  2      DECLARE data$seg          TOKEN,
           IO$task$token          TOKEN,
           widget$task$token      TOKEN,
           supervisor$task$token  TOKEN,

```

```

main$super$mbx      TOKEN,
write$msg$mbx       TOKEN,
start$write$mbx     TOKEN,
done$write$mbx      TOKEN,
start$widget$mbx    TOKEN,
done$widget$mbx     TOKEN;
282  2  DECLARE task$flags  WORD,
        status        WORD,
        i             WORD;
283  2  DECLARE stack$pointer  POINTER;
284  2  DECLARE stack$size$300  LITERALLY '300';
285  2  DECLARE priority$level$202  LITERALLY '202';
286  2  DECLARE priority$level$190  LITERALLY '190';
287  2  DECLARE iors$token          SELECTOR;
288  2  DECLARE done$obj          TOKEN  AT(@iors$token);
289  2  DECLARE iors              BASED  iors$token  STRUCTURE
        (status              WORD,
         unit$status         WORD,
         actual              WORD);

290  2  data$seg      = 0;          /* task sets up own data segment */
291  2  stack$pointer = 0;          /* automatic stack allocation */
292  2  task$flags   = 0;          /* no floating point */

293  2  CAUSE$INTERRUPT (3);
        /* create a mailbox to use for letting this job know
        * that the tasks are done executing. */
294  2  main$super$mbx = rq$create$mailbox (0, @status);
295  2  IF status <> E$OK THEN GOTO error;
297  2  start$write$mbx = rq$create$mailbox (0, @status);
298  2  IF status <> E$OK THEN GOTO error;
300  2  done$write$mbx = rq$create$mailbox (0, @status);
301  2  IF status <> E$OK THEN GOTO error;
303  2  start$widget$mbx = rq$create$mailbox (0, @status);
304  2  IF status <> E$OK THEN GOTO error;
306  2  done$widget$mbx = rq$create$mailbox (0, @status);
307  2  IF status <> E$OK THEN GOTO error;
309  2  CALL rq$catalog$object (0, main$super$mbx, @(9,'MAINSUPER'),
        @status);
310  2  IF status <> E$OK THEN GOTO error;
312  2  CALL rq$catalog$object (0, start$write$mbx,
        @(10,'STARTWRITE'), @status);
313  2  IF status <> E$OK THEN GOTO error;
315  2  CALL rq$catalog$object (0, done$write$mbx, @(9,'DNEWRITE'),
        @status);
316  2  IF status <> E$OK THEN GOTO error;

318  2  CALL rq$catalog$object (0, start$widget$mbx,
        @(11,'STARTWIDGET'), @status);
319  2  IF status <> E$OK THEN GOTO error;
321  2  CALL rq$catalog$object (0, done$widget$mbx,
        @(10,'DNEWIDGET'), @status);
322  2  IF status <> E$OK THEN GOTO error;
324  2  CAUSE$INTERRUPT (3);

```



```

/* Now create the tasks. */
325 2      supervisor$task$token = rq$create$task (priority$level$190,
          @supervisor$task, data$seg, stack$pointer,
          stack$size$300, task$flags, @status);
326 2      IF status <> E$OK THEN GOTO error;
328 2      widget$task$token = rq$create$task (priority$level$202,
          @widget$task, data$seg, stack$pointer,
          stack$size$300, task$flags, @status);
329 2      IF status <> E$OK THEN GOTO error;
331 2      IO$task$token = rq$create$task (priority$level$202, @IO$task,
          data$seg, stack$pointer,
          stack$size$300, task$flags, @status);
332 2      IF status <> E$OK THEN GOTO error;
334 2      done$obj = rq$receive$message (main$super$mbx, OFFFHH, 0,
          @status);
335 2      IF status <> E$OK THEN GOTO error;
337 2      CAUSE$INTERRUPT (3);

338 2      CALL rq$delete$segment (done$obj, @status);
339 2      IF status <> E$OK THEN GOTO error;
341 2      CALL rq$delete$task (IO$task$token, @status);
342 2      IF status <> E$OK THEN GOTO error;
344 2      CALL rq$delete$task (widget$task$token, @status);
345 2      IF status <> E$OK THEN GOTO error;
347 2      CALL rq$delete$task (supervisor$task$token, @status);
348 2      IF status <> E$OK THEN GOTO error;
350 2      CAUSE$INTERRUPT (3);
351 2      CALL rq$delete$mailbox (main$super$mbx, @status);
352 2      IF status <> E$OK THEN GOTO error;
354 2      CALL rq$delete$mailbox (start$write$mbx, @status);
355 2      IF status <> E$OK THEN GOTO error;
357 2      CALL rq$delete$mailbox (done$write$mbx, @status);
358 2      IF status <> E$OK THEN GOTO error;
360 2      CALL rq$delete$mailbox (start$widget$mbx, @status);
361 2      IF status <> E$OK THEN GOTO error;
363 2      CALL rq$delete$mailbox (done$widget$mbx, @status);
364 2      IF status <> E$OK THEN GOTO error;
366 2      GOTO ok;
367 2      error:          /* output to usart for debugging */
          DO WHILE 1;
368 3          OUTPUT(9CH) = OAAH;
369 3      END;

370 2      ok:

          CALL rq$delete$job (0, @status); /* myself */

371 2      END main$task;

```

```

/*****
* This is the task which is controlling the other two tasks. It has *
* higher priority than them now, but could have the same priority *
* since it uses the mailboxes to synchronize the tasks. *
*****/

```

```

372 1      supervisor$task: PROCEDURE REENTRANT PUBLIC;

373 2      DECLARE done$token          TOKEN,
           main$super$mbx          TOKEN,
           start$write$mbx        TOKEN,
           done$write$mbx         TOKEN,
           start$widget$mbx       TOKEN,
           done$widget$mbx        TOKEN,
           cleanup$token          TOKEN,
           make$widget$token      TOKEN;

374 2      DECLARE status             WORD,
           i                       WORD;

375 2      DECLARE cleanup$ptr       BASED cleanup$token (1) BYTE,
           make$widget$ptr         BASED make$widget$token (1) BYTE;

376 2      DECLARE cleanup$d         BYTE DATA(0),
           make$widget$d           BYTE DATA(1),
           message$d(*)            BYTE DATA('Making widget'),
           making$widget$d        BYTE DATA(1);

377 2      DECLARE send$to$io$token  TOKEN;
378 2      DECLARE send$to$io        BASED send$to$io$token STRUCTURE(
           making$widget (1) BYTE,
           message      (13) BYTE);

379 2      CAUSE$INTERRUPT (3);
380 2      main$super$mbx = rq$lookup$object (0, @(9,'MAINSUPER'),
           OFFFFH, @status);

381 2      IF status <> E$OK THEN GOTO error;
383 2      start$write$mbx = rq$lookup$object (0, @(10,'STARTWRITE'),
           OFFFFH, @status);

384 2      IF status <> E$OK THEN GOTO error;

386 2      done$write$mbx = rq$lookup$object (0, @(9,'DNEWRITE'),
           OFFFFH, @status);

387 2      IF status <> E$OK THEN GOTO error;
389 2      start$widget$mbx = rq$lookup$object (0, @(11,'STARTWIDGET'),
           OFFFFH, @status);

390 2      IF status <> E$OK THEN GOTO error;
392 2      done$widget$mbx = rq$lookup$object (0, @(10,'DNEWIDGET'),
           OFFFFH, @status);

393 2      IF status <> E$OK THEN GOTO error;

395 2      DO i = 1 TO 10;
396 3      CAUSE$INTERRUPT (3);
           /* Send message to widget task, mailbox STARTWIDGET */
397 3      make$widget$token = rq$create$segment (SIZE(make$widget$d),
           @status);

398 3      IF status <> E$OK THEN GOTO error;
400 3      CALL MOVB (@make$widget$d, @make$widget$ptr,
           SIZE(make$widget$d));
401 3      CALL rq$send$message (start$widget$mbx, make$widget$token,
           0, @status);
402 3      IF status <> E$OK THEN GOTO error;
404 3      CAUSE$INTERRUPT (3);

```

```

/* Send message to IO task, mailbox STARTWRITE */
405 3  send$to$io$token = rq$create$segment ( 32, @status);
406 3  IF status <> E$OK THEN GOTO error;
408 3  CALL MOVB (@message$d, @send$to$io.message,
        SIZE(message$d));
409 3  CALL MOVB (@making$widget$d, @send$to$io.making$widget,
        SIZE(making$widget$d));
410 3  CALL rq$send$message (start$write$mbx, send$to$io$token, 0,
        @status);
411 3  IF status <> E$OK THEN GOTO error;
413 3  CAUSE$INTERRUPT (3);

/* Receive message from widget task, mailbox DONEWIDGET */
414 3  done$token = rq$receive$message (done$widget$mbx, OFFFFH,
        0, @status);
415 3  IF status <> E$OK THEN GOTO error;
417 3  CAUSE$INTERRUPT (3);
418 3  CALL rq$delete$segment (done$token, @status);
419 3  IF status <> E$OK THEN GOTO error;

/* Receive message from IO task, mailbox DONEWRITE */
421 3  done$token = rq$receive$message (done$write$mbx, OFFFFH, 0,
        @status);
422 3  IF status <> E$OK THEN GOTO error;
424 3  CAUSE$INTERRUPT (3);
425 3  CALL rq$delete$segment (done$token, @status);
426 3  IF status <> E$OK THEN GOTO error;
/* receiving messages from the tasks, they are done and we *
* are ready to tell them to clean up. */
428 3  END;

/* Send message to widget task, mailbox STARTWIDGET */
429 2  cleanup$token = rq$create$segment ( SIZE(cleanup$d), @status);
430 2  IF status <> E$OK THEN GOTO error;
432 2  CALL MOVB (@cleanup$d, @cleanup$ptr, SIZE(cleanup$d));
433 2  CALL rq$send$message (start$widget$mbx, cleanup$token, 0,
        @status);
434 2  IF status <> E$OK THEN GOTO error;
436 2  CAUSE$INTERRUPT (3);

/* Send message to IO task, mailbox STARTWRITE */
437 2  send$to$io$token = rq$create$segment ( 32, @status);
438 2  IF status <> E$OK THEN GOTO error;
440 2  CALL MOVB (@message$d, @send$to$io.message, SIZE(message$d));
441 2  CALL MOVB (@cleanup$d, @send$to$io.making$widget,
        SIZE(cleanup$d));
442 2  CALL rq$send$message (start$write$mbx, send$to$io$token, 0,
        @status);
443 2  IF status <> E$OK THEN GOTO error;

/* Receive message from widget task, mailbox DONEWIDGET */
445 2  done$token = rq$receive$message (done$widget$mbx, OFFFFH, 0,
        @status);

```

```

446 2      IF status <> E$OK THEN GOTO error;
448 2      CAUSE$INTERRUPT (3);
449 2      CALL rq$delete$segment (done$token, @status);
450 2      IF status <> E$OK THEN GOTO error;

      /* Receive message from IO task, mailbox DONEWRITE */
452 2      done$token = rq$receive$message (done$write$mbx, OFFFFH, 0,
      @status);
453 2      IF status <> E$OK THEN GOTO error;
455 2      CAUSE$INTERRUPT (3);
456 2      CALL rq$delete$segment (done$token, @status);
457 2      IF status <> E$OK THEN GOTO error;

459 2      CAUSE$INTERRUPT (3);

      /* Send message to main task, mailbox MAINSUPER */
460 2      done$token = rq$create$segment ( 16, @status);
461 2      IF status <> E$OK THEN GOTO error;
463 2      CALL rq$send$message (main$super$mbx, done$token, 0, @status);
464 2      IF status <> E$OK THEN GOTO error;

466 2      CALL rq$suspend$task (0, @status); /* suspend myself */
467 2      IF status <> E$OK THEN GOTO error;
469 2      GOTO ok;

470 2      error:          /* output to usart for debugging */
      DO WHILE 1;
471 3      OUTPUT(9CH) = 0A3H;
472 3      END;

473 2      ok:
      CALL rq$suspend$task (0, @status) /* myself */
      END supervisor$task;

/*****
* IO$task is a task which is created by main$task. It creates
* the necessary mailboxes and segments, and then attaches the terminal
* physically. It then creates a file so that it has a file connection.
* It opens the file connection, and writes the contents of a buffer to
* the terminal. Then it closes the file connection, deletes the
* device connection, and detaches the device.
*****/
474 1      IO$task: PROCEDURE REENTRANT PUBLIC;

475 2      DECLARE  mbx$token      TOKEN,
      seg$token      TOKEN,
      user$token    TOKEN,
      file$connection  TOKEN,
      device$connection  TOKEN,
      cleanup$token  TOKEN,
      done$token    TOKEN,
      make$widget$token  TOKEN,
      continue$token  TOKEN,
      done$write$mbx  TOKEN,
      start$write$mbx  TOKEN;

```

```

476 2    DECLARE status          WORD;
477 2    DECLARE hard             BYTE;
478 2    DECLARE iors$token      TOKEN;
479 2    DECLARE cleanup$ptr    BASED cleanup$token (1) BYTE;
480 2    DECLARE send$to$io$token TOKEN;
481 2    DECLARE send$to$io      BASED send$to$io$token STRUCTURE(
                                making$widget (1) BYTE,
                                message      (13) BYTE);
482 2    DECLARE dev$conn$object TOKEN AT (@iors$token),
                                file$conn$object TOKEN AT (@iors$token),
                                msg$received$obj TOKEN AT (@iors$token),
                                object          TOKEN AT (@iors$token);
483 2    DECLARE iors          BASED iors$token STRUCTURE
                                (status        WORD,
                                unit$status   WORD,
                                actual        WORD);
484 2    DECLARE buffer        BASED seg$token (9) BYTE;
485 2    DECLARE user$object   STRUCTURE
                                (length      WORD,
                                count        WORD,
                                id (1)      WORD);

486 2    user$object.length = 1;
487 2    user$object.count = 1;
488 2    user$object.id(0) = OFFFFH;

489 2    hard = 0ffh; /* request a hard detach of the device */
490 2    CAUSE$INTERRUPT (3);
491 2    /* Set up the mailboxes for this task to use */
491 2    start$write$mbx = rq$lookup$object (0, @(10,'STARTWRITE'),
                                OFFFFH, @status);
492 2    IF status <> E$OK THEN GOTO error;
494 2    done$write$mbx = rq$lookup$object (0, @(9,'DONEWRITE'),
                                OFFFFH, @status);
495 2    IF status <> E$OK THEN GOTO error;

/* Receive message from supervisor task, mailbox STARTWRITE
*/
497 2    send$to$io$token = rq$receive$message (start$write$mbx,
                                OFFFFH, 0, @status);
498 2    IF status <> E$OK THEN GOTO error;
500 2    CAUSE$INTERRUPT (3);

501 2    user$token = rq$create$user (@user$object, @status);
502 2    IF status <> E$OK THEN GOTO error;
504 2    mbx$token = rq$create$mailbox (0, @status);
505 2    IF status <> E$OK THEN GOTO error;
507 2    seg$token = rq$create$segment ( 48, @status);
508 2    IF status <> E$OK THEN GOTO error;

510 2    CALL rq$a$physical$attach$device ( @(2,'TO'), 1, mbx$token,
                                @status);

```

```

511 2      IF status <> E$OK THEN GOTO error;
513 2      dev$conn$object = rq$receive$message (mbx$token, OFFFFH, 0,
                                         @status);
514 2      IF status <> E$OK THEN GOTO error;
516 2      device$connection = dev$conn$object;

517 2      CALL rq$a$create$file (user$token, device$connection,
                               0, 0, 0, 0, 0, mbx$token, @status);
518 2      IF status <> E$OK THEN GOTO error;
520 2      file$conn$object = rq$receive$message (mbx$token, OFFFFH, 0,
                                         @status);

521 2      IF status <> E$OK THEN GOTO error;
523 2      file$connection = file$conn$object;

524 2      CALL rq$a$open (file$connection, 2, 0, mbx$token, @status);
525 2      IF status <> E$OK THEN GOTO error;
527 2      object = rq$receive$message (mbx$token, OFFFFH, 0, @status);
528 2      IF status <> E$OK THEN GOTO error;
530 2      CALL rq$delete$segment (object, @status);
531 2      IF status <> E$OK THEN GOTO error;

```

```

/*****
* This is the part of the code that will be repeated. It receives *
* the message from second task and writes it to the screen, then *
* returns control to the second task.                               */

```

```

533 2      DO WHILE send$to$io.making$widget(0) = 1;
534 3      CAUSE$INTERRUPT (3);
535 3      CALL rq$a$write (file$connection, @send$to$io.message,
                        size(send$to$io.message),
                        mbx$token, @status);
536 3      IF status <> E$OK THEN GOTO error;
538 3      object = rq$receive$message (mbx$token, OFFFFH, 0,
                                         @status);
539 3      IF status <> E$OK THEN GOTO error;
541 3      CALL rq$delete$segment (object, @status);
542 3      IF status <> E$OK THEN GOTO error;
544 3      CALL rq$delete$segment (send$to$io$token, @status);
545 3      IF status <> E$OK THEN GOTO error;

/* Send message to supervisor task, mailbox DONEYRITE */
547 3      done$token = rq$create$segment ( 16, @status);
548 3      IF status <> E$OK THEN GOTO error;
550 3      CALL rq$send$message (done$write$mbx, done$token, 0,
                              @status);
551 3      IF status <> E$OK THEN GOTO error;
553 3      CAUSE$INTERRUPT(3);

/* Receive message from supervisor task; mailbox
* STARTWRITE */
554 3      send$to$io$token = rq$receive$message (start$write$mbx,
                                                OFFFFH, 0, @status);
555 3      IF status <> E$OK THEN GOTO error;

```

```

557 3      CAUSE$INTERRUPT (3);
558 3      END;

559 2      CALL rq$delete$segment (send$to$io$token, @status);
560 2      IF status <> E$OK THEN GOTO error;
562 2      CALL rq$a$close (file$connection, mbx$token, @status);
563 2      IF status <> E$OK THEN GOTO error;
565 2      object = rq$receive$message (mbx$token, OFFFFH, 0, @status);
566 2      IF status <> E$OK THEN GOTO error;
568 2      CALL rq$delete$segment (object, @status);
569 2      IF status <> E$OK THEN GOTO error;
571 2      CALL rq$a$delete$connection (file$connection, mbx$token,
                                     @status);
572 2      IF status <> E$OK THEN GOTO error;
574 2      object = rq$receive$message (mbx$token, OFFFFH, 0, @status);
575 2      IF status <> E$OK THEN GOTO error;
577 2      CALL rq$delete$segment (object, @status);
578 2      IF status <> E$OK THEN GOTO error;

580 2      CALL rq$a$physical$detach$device (device$connection, hard,
                                           mbx$token, @status);
581 2      IF status <> E$OK THEN GOTO error;
583 2      object = rq$receive$message (mbx$token, OFFFFH, 0, @status);
584 2      IF status <> E$OK THEN GOTO error;
586 2      CALL rq$delete$segment (object, @status);
587 2      IF status <> E$OK THEN GOTO error;

589 2      CALL rq$delete$mailbox (mbx$token, @status);
590 2      IF status <> E$OK THEN GOTO error;

592 2      CALL rq$delete$user (user$token, @status);
593 2      IF status <> E$OK THEN GOTO error;
595 2      CALL rq$delete$segment (seg$token, @status);
596 2      IF status <> E$OK THEN GOTO error;

/* Send message to supervisor task, mailbox DONEYRITE */
598 2      done$token = rq$create$segment ( 16, @status);
599 2      IF status <> E$OK THEN GOTO error;
601 2      CALL rq$send$message (done$write$mbx, done$token, 0, @status);
602 2      IF status <> E$OK THEN GOTO error;
604 2      CAUSE$INTERRUPT(3);
605 2      GOTO ok;

606 2      error:
        DO WHILE 1;
607 3          OUTPUT(9CH) = 0A1H;
608 3      END;

609 2      ok:
        CALL rq$suspend$task (0, @status);

610 2      END IO$task;

611 1      widget$task: PROCEDURE REENTRANT PUBLIC;

```

```

612 2      DECLARE done$token          TOKEN,
           continue$token          TOKEN,
           cleanup$token           TOKEN,
           start$widget$mbx       TOKEN,
           done$widget$mbx        TOKEN,
           make$widget$token      TOKEN;
613 2      DECLARE status              WORD;
614 2      DECLARE cleanup$ptr         BASED  cleanup$token (1) BYTE,
           continue$ptr            BASED  continue$token (1) BYTE,
           make$widget$ptr        BASED  make$widget$token (11) BYTE;

615 2      CAUSE$INTERRUPT (3);
616 2      start$widget$mbx = rq$lookup$object (0, @(11,'STARTWIDGET'),
                                           OFFFFH, @status);

617 2      IF status <> E$OK THEN GOTO error;
619 2      done$widget$mbx = rq$lookup$object (0, @(10,'DONEWIDGET'),
                                           OFFFFH, @status);

620 2      IF status <> E$OK THEN GOTO error;

/* Receive message from supervisor task, mailbox STARTWIDGET */
622 2      make$widget$token = rq$receive$message (start$widget$mbx,
                                           OFFFFH, 0, @status);
623 2      IF status <> E$OK THEN GOTO error;

625 2      DO WHILE make$widget$ptr(0) = 1; /* Repeat endlessly.
           * Let supervisor task take care of controlling. */
626 3      CAUSE$INTERRUPT (3);
627 3      CALL rq$delete$segment (make$widget$token, @status);
628 3      IF status <> E$OK THEN GOTO error;

           /* Send message to supervisor task, mailbox DONEWIDGET */
630 3      done$token = rq$create$segment ( 16, @status);
631 3      IF status <> E$OK THEN GOTO error;
633 3      CALL rq$send$message (done$widget$mbx, done$token, 0,
                               @status);
634 3      IF status <> E$OK THEN GOTO error;
636 3      CAUSE$INTERRUPT(3);

/* Receive message from supervisor task, mailbox STARTWIDGET */
637 3      make$widget$token = rq$receive$message (start$widget$mbx,
                                           OFFFFH, 0, @status);
638 3      IF status <> E$OK THEN GOTO error;

640 3      END; /* do making the widgets */

641 2      CAUSE$INTERRUPT (3);
           /* cleanup the environment now. */
642 2      CALL rq$delete$segment (make$widget$token, @status);
643 2      IF status <> E$OK THEN GOTO error;

           /* Send message to supervisor task, mailbox DONEWIDGET */
645 2      done$token = rq$create$segment ( 16, @status);
646 2      IF status <> E$OK THEN GOTO error;
648 2      CALL rq$send$message (done$widget$mbx, done$token, 0,
                               @status);

```



```
649 2      IF status <> E$OK THEN GOTO error;
651 2      CAUSE$INTERRUPT(3);
652 2      GOTO ok;

653 2      error: /* output to usart for debugging */
        DO WHILE 1;
654 3          OUTPUT(9CH) = 0A2H;
655 3      END;

656 2      ok:
        CALL rq$suspend$task (0, @status);

657 2      END widget$task;

658 1      END tasks;
```

MODULE INFORMATION:

```
CODE AREA SIZE      = 0D1BH   3355D
CONSTANT AREA SIZE  = 00A9H   169D
VARIABLE AREA SIZE  = 0000H    0D
MAXIMUM STACK SIZE  = 0044H   68D
1796 LINES READ
0 PROGRAM WARNINGS
0 PROGRAM ERRORS
```

DICTIONARY SUMMARY:

```
84KB MEMORY AVAILABLE
22KB MEMORY USED   (26%)
0KB DISK SPACE USED
```

END OF PL/M-86 COMPILATION

APPENDIX D

```
PLM86 %0.P86
PLM86 inittask.P86
LINK86 inittask.OBJ to inittask.lnk initcode
LINK86 %0.OBJ to %0.lnk1 initcode
link86 inittask.LNK, &
    %0.LNK1, &
    /lib/ndp87/dcon87.lib, &
    /lib/ndp87/ce187.lib, &
    /lib/ndp87/eh87.lib, &
    /lib/ndp87/8087.lib, &
    /lib/rmx86/epif1.lib, &
    /lib/rmx86/ipif1.lib, &
    /LIB/RMX86/RPIFL.LIB, &
    /LIB/PLM86/PLM86.LIB &
    TO %0.LNK
LOC86 %0.LNK TO %0 &
    SEGSIZE(STACK(0)) ADDRESSES(CLASSES(CODE(015000H),DATA(017000H))) NOINITCODE
lib86
delete /boot/%0(inittask)
add %0 to /boot/%0
e
```

APPENDIX E

ICU86 V2.0 ONETASK.DEF

05/21/84 07:04:32

Hardware

(CPU) Processor used in the system	8086
(OSP) 80130 Operating System Extension [Yes/No]	No
(TP) 8253/8254 Timer Port [0-0FFFFH]	00D0H
(CIL) Clock Interrupt Level [0-7]	0002H
(CN) Timer Counter Number [0,1,2]	0000H
(CI) Clock Interval [0-0FFFFH msec]	000AH
(CF) Clock Frequency [0-0FFFFH khz]	04CDH
(TPS) Timer Port Separation [0-0FFH]	0002H
(NPX) Numeric Processor Extension [Yes/No]	No
(NIL) NPX Interrupt Level [Encoded]	0008H

Interrupts

(MP) 8259A Master Port [0-0FFFFH]	00C0H
(MPS) Master PIC Port Separation [0-0FFH]	0002H
(SIL) Slave Interrupt Levels [0-7/None]	None
(LSS) Level Sensitive Slaves [0-7/None]	None

Memory

Type : RAM = low, high
 Type : ROM = low, high
 Type : RAM = 0104H, 1500H
 Type : RAM = 1800H, F7FFH

Sub-systems

(UDI) Universal Development Interface [Yes/No]	No
(HI) Human Interface [Yes/No]	No
(AL) Application Loader [Yes/No]	No
(EIO) Extended I/O System [Yes/No]	No
(BIO) Basic I/O System [Yes/No]	Yes
(SDB) System Debugger [Yes/No]	Yes
(DDB) Dynamic Debugger [Yes/No]	No
(TH) Terminal Handler [Yes/No]	No
(CA) Crash Analyzer [Yes/No]	No

BIOS

(ASC) All Sys Calls in BIOS [Yes/No]	Yes
(ADP) Attach Device Task Priority [1-0FFH]	0081H
(TF) Timing Facilities Required [Yes/No]	Yes
(TTP) Timer Task Priority [0-0FFH]	0081H
(CON) Connection Job Delete Priority [0-0FFH]	0082H
(ACE) Ability to Create Existing Files [Yes/No]	Yes
(SMI) System Manager ID [Yes/No]	Yes
(CUT) Common Update Timeout [0-0FFFFH]	03E8H

(CST) Control-Sequence Translation [Yes/No]	Yes
(OSC) Terminal OSC Controls [Yes/No]	Yes
(TS) Tape Support for iSBC 215G [Yes/No]	No
(PMI) BIOS Pool Minimum [0-0FFFFH]	0800H
(PMA) BIOS Pool Maximum [0-0FFFFH]	0800H

8251A Driver

(IIL) Input Interrupt Level [Encoded]	0068H
(OIL) Output Interrupt Level [Encoded]	0078H
(UDP) USART Data Port [0-0FFFFH]	00D8H
(USP) USART Status Port [0-0FFFFH]	00DAH
(IRP) 8253 Inrate Port [0-0FFFFH]	00D4H
(ICP) 8253 Input Control Port [0-0FFFFH]	00D6H
(IRC) 8253 Input Counter Number [0-2]	0002H
(IRF) Inrate Frequency [0-0FFFFFFFH]	0012C000H
(ORP) 8253 Outrate Port [0-0FFFFH]	0000H
(OCP) 8253 Output Control Port [0-0FFFFH]	0000H
(ORC) 8253 Output Counter Number [0-2]	0000H
(ORF) Outrate Frequency [0-0FFFFFFFH]	00000000H

8251A Unit Information

(NAM) Unit Info Name [1-17 Chars]	t0info
(LEM) Line Edit Mode [Trans/Normal/Flush]	Normal
(ECH) Echo Mode [Yes/No]	Yes
(IPC) Input Parity Control [Yes/No]	Yes
(OPC) Output Parity Control [Yes/No]	Yes
(OCC) Output Control in Input [Yes/No]	Yes
(OSC) OSC Controls [Both/In/Out/Neither]	Both
(DUP) Duplex Mode [Full/Half]	Full
(TRM) Terminal Type [CRT/Hard Copy]	CRT
(MC) Modem Control [Yes/No]	No
(RPC) Read Parity Checking [See Help/0-3]	0000H
(WPC) Write Parity Checking [See Help/0-4]	0000H
(BR) Baud Rate [0-0FFFFH]	2580H
(SN) Scroll Number [0-0FFFFH]	0017H

251A Device-Unit Information

(NAM) Device-Unit Name [1-13 chars]	TO
(UN) Unit Number on this Device [0-0FFH]	0000H
(UIN) Unit Info Name [1-17 Chars]	t0info
(MB) Max Buffers [0-0FFH]	0000H

System Debugger

(SLV) SDB Interrupt Level [Encoded Level/None]	0018H
--	-------

Nucleus

(ASC) All Sys Calls [Yes/No]	Yes
(PV) Parameter Validation [Yes/No]	Yes
(ROD) Root Object Directory Size [0 - 0FF0h]	0020H

(MTS) Minimum Transfer Size [0-0FFFFH]	0040H
(DEH) Default Exception Handler [Yes/No/Deb/Use]	Yes
(NEH) Name of Ex Handler Object Module [1-32chs]	
(EM) Exception Mode [Never/Program/Environ/All]	Never
(SRR) Start Root job from Reset [Yes/No]	No

User Jobs

(NAM) Job Name [0-14 characters]	onetask
(ODS) Object Directory Size [0-0FF0H]	0010H
(PMI) Pool Minimum [20H - 0FFFFH]	0500H
(PMA) Pool Maximum [20H - 0FFFFH]	0500H
(MOB) Maximum Objects [1 - 0FFFFH]	0020H
(MTK) Maximum Tasks [1 - 0FFFFH]	0010H
(MPR) Maximum Priority [0 - 0FFH]	0000H
(AEH) Address of Exception Handler [CS:IP]	0000H:0000H
(EM) Exception Mode [Never/Prog/Environ/All]	Never
(PV) Parameter Validation [Yes/No]	Yes
(TP) Task Priority [0-0FFH]	0082H
(TSA) Task Start Address [CS:IP]	1500H:0002H
(DSB) Data Segment Base [0-0FFFFH]	0000H
(SSA) Stack Segment Address [SS:SP]	0000H:0000H
(SS) Stack Size [0-0FFFFH]	1F40H
(NPX) Numeric Processor Extension Used [Yes/No]	No

User Modules

Module : 1-55 characters

ROM code

(BIR) Basic I/O System in ROM [Yes/No]	No
(SIR) SDB in ROM [Yes/No]	No
(NIR) Nucleus in ROM [Yes/No]	No
(RIR) Root Job in ROM [Yes/No]	No

Includes and Libraries

Path Name [1-45 Characters]

(UDF) UDI Includes and Libs	/rmx86/udi/
(HIF) Human Interface Includes and Libs	/rmx86/hi/
(EIF) Extended I/O System Includes and Libs	/rmx86/eios/
(ALF) Application Loader Includes and Libs	/rmx86/loader/
(BIF) Basic I/O System Includes and Libs	/rmx86/ios/
(SDF) System Debugger Includes and Libs	/rmx86/sdb/
(THF) Terminal Handler and Dynamic Debugger Includes and Libs	/rmx86/th/
(NUF) Nucleus and Root Job Includes and Libs	/rmx86/nucleus/

(ILF) Interface Libraries /rmx86/lib/
(CAF) Crash Analyzer Includes and Libs /rmx86/crash/
(DTF) Development Tools Path Names :lang:

Generate File Names

File Name [1-55 Characters]

(ROP) ROM Code Prefix none
(RAF) RAM Code File Name /boot/onetask

Appendix F: Related publications

Knuth, The Art of Computer Programming, Vol 1, pp 435-453, and exercise 6, p 452 with answer p 597. (c) 1973, 1978. Addison-Wesley Publishing Co., Redding, MA

iRMX™ 86 Introduction and Operator's Reference Manual For Release 6 (146194-001)

iRMX™ 86 Programmer's Reference Manual, Part I, For Release 6 (146195-001)

iRMX™ 86 Programmer's Reference Manual, Part II, For Release 6 (146196-001)

iRMX™ 86 Installation and Configuration Guide For Release 6 (146197-001)

June 1983

Software That Resides in Silicon

**Ron Stamp
and Jim Person
Intel Corporation**

Software That Resides in Silicon

Ron Slamp and Jim Person, Intel Corporation

Silicon software sounds like a contradiction in terms. The casting of software in silicon implies that the software cannot be changed; yet software does and must change. For example, it must be possible to alter a microprocessor operating system so that the system will support different hardware and software designs, as well as accommodate new hardware components and applications. And if the software has been committed to silicon, then a way must exist to overcome any bugs that are discovered later.

Design Considerations

Silicon software consists of two kinds of code: on-chip code and off-chip code (see Figure 1). In a typical case, some of the off-chip code works closely with the on-chip code, and is developed as part of the silicon software package. This special off-chip (or "support") code might contain initialization, interface, system, and version update codes. For silicon software to tolerate change and be usable in more than one system, the on-chip code must have three qualities: position independence, configuration independence and stepping independence.

Position Independence

Because the most advanced microprocessors address at least 1 megabyte of memory, system software that resides in silicon must work right regardless of its location in memory. Absolute addresses in the read-only, on-chip code or data restricts the configuration of the system. Because the on-chip code recognizes only offsets, absolute addresses are unacceptable. On-chip code cannot presume to know the location of any code or data, it can only presume to know the structure of the data which it accesses. It cannot know, except relatively, where in memory it (or any other code) resides. If the on-chip code is to be position independent, then any absolute addresses needed by the on-chip code must be obtained via the processor's registers.

Position independence is not a new concept; in fact, it is rather an obvious requirement for silicon software. Compilers and relocatable assemblers allow linking and locating, thus making it easier to produce position-independent code. But most of these tools can also produce code that is not position independent. Silicon software developers need to be aware of the position-independence requirement throughout the design, implementation and test phases for their products.

Configuration Independence

The second requirement for silicon-resident software is that the on-chip code must not depend on the underlying hardware and software configuration of the system. Instead, the on-chip code must have indirect access to other code or data, and must then check the run-time data to deduce the system configuration.

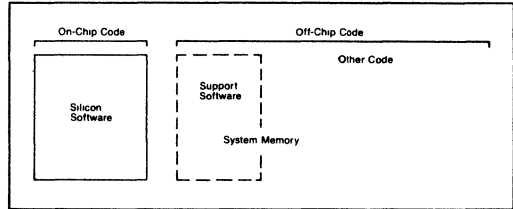


FIGURE 1. Silicon software is divided into on-chip code and off-chip code. The off-chip code either directly supports the on-chip code or contains other applications code.

Because of the read-only nature of silicon software, constants can cause problems when they are located within the on-chip code. Values representing a hardware device must not reside on-chip if that device can be located anywhere in the system, or when values support several devices having similar functions but different programming interfaces. Indirect access is necessary for all values that vary depending on the configuration of the system.

Stepping Independence

Stepping independence is an expansion of configuration independence, and is perhaps the most elusive of the requirements to be met by software intended for residence in silicon. A "step" is an updated version of the on-chip code. The on-chip code and the off-chip code must remain compatible, regardless of changes in either of them. Stepping independence exists when all versions of the on-chip code work with all versions of the off-chip code.

If stepping independence is taken into consideration when the silicon software is developed, then provisions can be made for the subsequent additions of options without changing the on-chip code. Otherwise, the static nature of the on-chip code might make it impossible to add options. Although configuration independence can be designed into software from the start, stepping independence can be achieved only if a system's existing silicon software does not include features that prevent it.

One type of data that is likely to change between steps is the value representing the size of a data area. If the software is to be stepping independent, it cannot know the sizes of the data areas accessed by on-chip code prior to run time. (No problems arise if on-chip and off-chip code agree on the size of the data area.)

But what happens if the on-chip code is not from the same version of the product as the off-chip code, and if the size of the data area has changed between versions? If the size of the data area is defined by a constant in the on-chip code, then that area might be smaller than the off-chip code expects it to be. This misunderstanding can lead to disaster as the off-chip code reads and writes beyond the data area.

This problem is solved when the on-chip code ascertains the size of the data area from off-chip data. Thus, the size of the data areas for the system becomes a configuration option.

Getting the Bugs Out of Silicon Software

Every large program contains bugs. Designers usually remove bugs by modifying the program to correct the problem, and then discarding the old program. However, a program in silicon cannot be modified without stepping the component. And even so, it is undesirable to discard the outdated component.

Software designed for silicon should include a facility for fixing bugs in on-chip code. One way to fix an on-chip bug is to prevent access to the routine containing the bug. A correct version of the routine is provided off-chip, and program execution is forced to branch to the off-chip version whenever the routine is invoked. Modular programming practices during development help reduce the cost of such off-chip duplication.

This on-chip bug-fix works well over time. Each component step has an associated collection of bug-fix modules. The collection is updated for each new version of the product, as component steps fix known bugs. During system configuration, the user specifies which component step is being used; the fixes for that step are included automatically in the off-chip code. Because of this facility, one step looks just like another to the user.

Intel's OSF: A Software Component

The Operating System Firmware (OSF) component consists of several hardware modules (see Figure 2). These modules provide two functions that are essential to operating systems: interrupts and timers. The OSF modules include a Control Store (16K bytes of fast ROM) to contain the silicon software, three programmable interval timers, an eight-input programmable interrupt controller, a bus interface, control logic, a data buffer, and address latch logic.

The 80130: The iRMX™ 86 Kernel in Silicon

Intel's first software-on-silicon product is the 80130. It provides a functional subset of the iRMX™ 86 Nucleus, which is the heart of the iRMX 86 operating system (OS). The iRMX 86 OS is a real-time, multi-tasking, multiprogramming operating system intended for 16-bit microprocessor designs. The iRMX 86 family of standard software modules includes a nucleus, a stand-alone terminal handler, a stand-alone debugger, an asynchronous I/O system, a synchronous I/O system, a loader, a human interface, and options required for real-time applications. The nucleus manages the creation and dynamic deletion of all system architectural features (tasks, program environments, memory segments, data-communication managers, etc.). It also schedules tasks, based on priority, interrupt management, memory management, validation of parameters, management of exceptional conditions, and co-processor support.

How the 80130 Satisfies the Silicon Software Criteria

The iRMX 86 Nucleus provides both the on-chip and off-chip codes needed to implement the operating system. The on-chip code resides in the 16K-byte ROM space of the 80130. It is the main portion of the Nucleus code, and includes the kernel of the

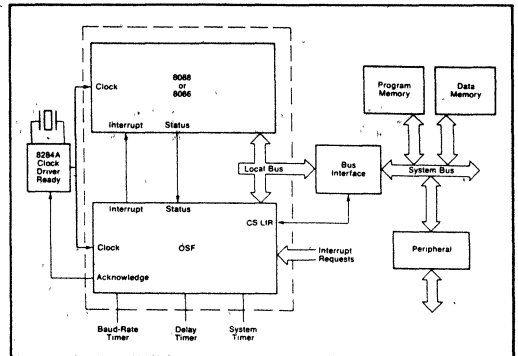


FIGURE 2. The OSF component works with systems that use the IAEX 86, 88, 186, or 188 microprocessor. Close coupling of the CPU and the OSF allows maximum zero-wait-state performance of the OSF software.

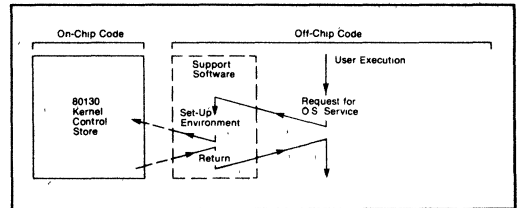


FIGURE 3. The position-independent interface supplies data location and run-time values, and starts on-chip execution of the software.

operating system and the primitives, which are present in the basic 80130 configuration. The off-chip code is stored in external RAM or ROM. It consists of initialization code, and code that either cannot be position independent or cannot be known before a given system is configured.

Position independence is guaranteed if entry to the on-chip code is possible only through an interface in the off-chip code that sets up the necessary registers. The off-chip position-independence interface (see Figure 3) provides an absolute data location and begins on-chip execution by the silicon-resident code. All run-time values can be determined based on the data location. On-chip execution gives the processor a location in the on-chip code from which other on-chip locations can be calculated.

It was relatively easy to make the 80130 configuration independent, because (like most operating-system kernels) it contains only general-purpose functions. The off-chip code contains all the drivers for particular peripheral chips. The Interactive Configuration Utility integrates the drivers with the 80130.

The interface between the off-chip and on-chip codes remains stable across component steps. The stepping-independence interface (see Figure 4) resides on the chip, and is a map of the on-chip code. This interface gives the off-chip code indirect access to all on-chip "publics" (e.g., externally accessible routines, modules, and labels). It is also a chart that routes execution to the proper on-chip location. The off-chip code uses an index of this chart to specify which public should

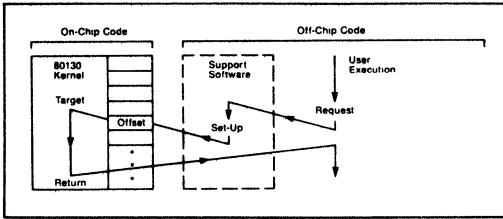


FIGURE 4. All on-chip accesses are routed through the on-chip stepping-independence interface, which provides compatibility between on-chip and off-chip code. Because the interface structure stays constant, the external reference also stays constant, while the on-chip OFFSET changes to point to the new location of the on-chip code.

be accessed. The index of a given routine remains the same across component steps, even though the actual address (offset into the component) of the public has changed. For different versions of the on-chip and off-chip codes to work correctly, all access from outside the component must be routed through the stepping-independence interface.

The 80150: CP/M-86* in Silicon

Intel's decision to implement CP/M-86 operating system in silicon (the 80150) raised a different design problem. With the 80130, Intel only had to deal with Intel-designed software. Code design, implementation, extensions, corrections, support, and the subsequent effect on the end user were all under Intel's control. The selection of an independent software system such as CP/M-86 (a product of Digital Research, Inc.) introduced new factors into the implementation.

The CP/M-86 Architecture

The CP/M-86 operating system consists of three modules. The Console Command Processor (CCP) handles command line processing, and executes built-in utilities. The Basic Disk Operating System (BDOS) performs logical disk I/O, including disk reading and writing, directory management, and sector allocation. The Basic Input/Output System (BIOS), which contains the configuration-dependent code and data, also provides I/O for specific peripheral chips.

CP/M-86 is a single-user, single-tasking operating system written in position-dependent code. The 80150 contains the entire CP/M-86 operating system; for many configurations, it requires no off-chip code. Intel's goal was to use the configuration-independent CCP and BDOS elements as a base, and add to them a BIOS that supported a variety of peripheral components but was still configuration independent.

The 80150 BIOS supports the following two functional configuration options:

1. A *preconfigured-mode system*, for which the system designer needs to do no operating-system code development or extension.
2. A *configurable-mode system*, for which the designer makes a selection from among the Intel drivers supplied, and makes changes as required to meet hardware needs.

The 80150 BIOS includes drivers for the following chips:

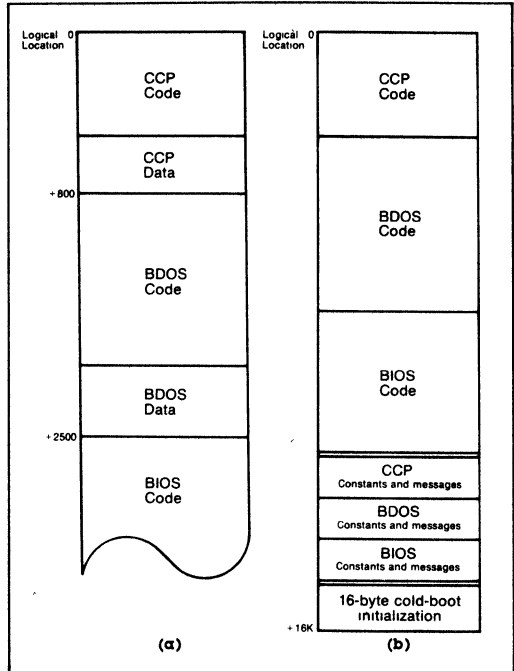


FIGURE 5. (a) The standard disk-based CP/M-86 module is one long structure containing both code and data. (b) Intel reorganized the basic CP/M-86 architecture to fit the operating system into the 80150 OS firmware component.

- 8251A Universal Asynchronous Receiver/Transmitter (UART)
- 8274 Multi-Protocol Serial Controller (MPSC)
- 8255A Programmable Parallel Interface (PPI)
- 8275 Floppy-Disk Controller
- 8237 Direct Memory Access (DMA) Controller

If the 80150 is used as a co-processor with the iAPX 186 or the 188, then the on-chip peripherals of these processors (DMA, timers, interrupt controller, chip-select logic) are also used.

Configuration independence is achieved via the Configuration Block (CB), with which whole BIOS drivers, data structures, and built-in utilities can be selected independently by the system integrator.

CP/M-86 Transformations

Intel and Digital Research together addressed the issues of position dependence and intermixed code, data, buffers, and stacks. The CCP and BDOS were reorganized to consolidate code and to use the 80150's ROM space efficiently.

CP/M-86 was originally developed using an 8080 model structure. The use of this structure implied that the code and data groups would overlap, as they do in the classical 8080-based CP/M design. Each module contained set-aside buffer areas, and included separate data stacks. Therefore, all variable areas

*CP/M-86 is a trademark of Digital Research, Inc.

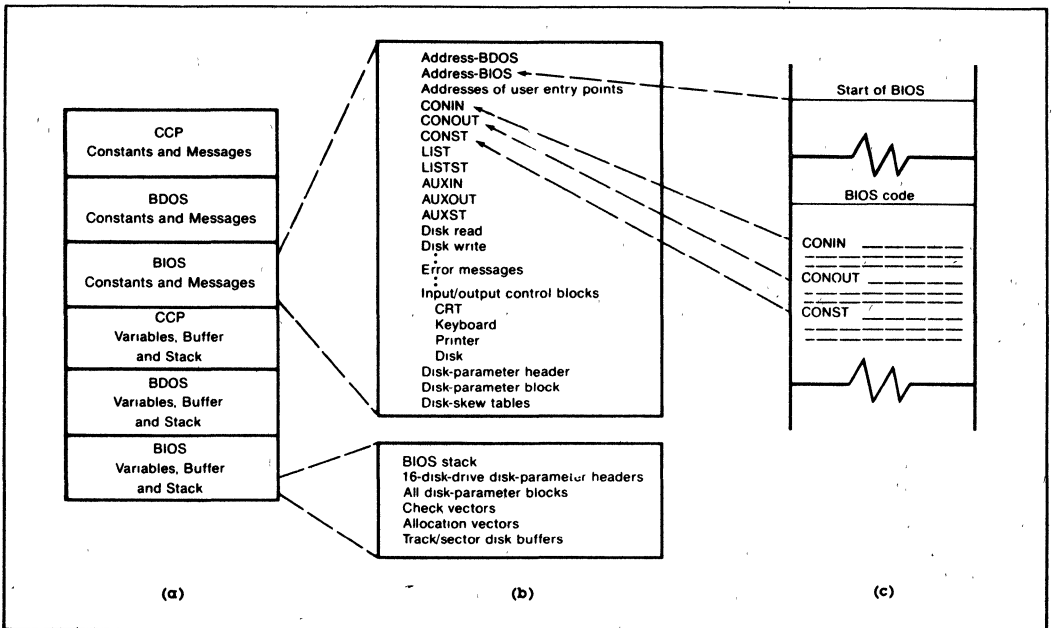


FIGURE 6. The Configuration Block (CB) reconfigures the 80150 for specific hardware systems. a) The CB constants read down from the 80150, and variables used at run-time. b) The BIOS portion of the CB contains configuration-dependent data. c) These addresses provide access to the 80150 on-chip code, to alter execution paths for different configurations and steppings.

and stack areas had to be removed from code that would reside in ROM.

Figure 5(a) shows the general structure of the original CCP and BDOS. Although a natural separation between code and data is clear, Digital Research did not distinguish between constants, literal messages, and pure scratch storage.

Intel's first step in the transformation of CP/M-86 was to group all variables within each module, including buffers and stacks. We then placed this data grouping at the end of the constants and literal messages for each of the CCP and BDOS modules.

The new structure (Figure 5(b)) includes all code, constants, and internal messages, as well as a 16-byte initial-program-load (IPL) boot resident in the 16K-byte OSF ROM. We removed all variables from the body of CP/M-86, and put them in an external RAM-based structure.

Second, the implementation of CP/M via the Intel 8086 "small model" (separate code and data segments) rather than via the 8080 model (intermixed code and data), meant that the necessary additional variable data space would be available at 80150 execution time. The segmented architecture of the iAPX 86 family made this implementation easy, because separate CPU registers were available for data and code addresses. As part of the BIOS initialization, we moved the constant data structures for the CCP, BDOS, and BIOS to the base of a RAM-resident Configuration Block (CB). An additional amount of RAM equivalent to the total variable space was also allocated and preset to zero. This 8086 "small-model" transformation not only made it easy to separate code and data, but also

made the code more efficient and eliminated approximately 2100 bytes.

We achieved configuration and stepping independence via the off-chip RAM-based Configuration Block. Figure 6(a) shows the overall structure of the CB as constructed during BIOS initialization. During initialization, the 80150 BIOS copies the CCP, BDOS, and BIOS constant and literal structures into the Configuration Block, and appends additional space for variable and scratch-pad storage. Even the location of the CB is alterable, based on the address stored in locations 0:3FE-3FF.

Figure 6(b) shows expanded portions of the CB. The data area contains pointers that can be changed to select custom off-chip code instead of the standard on-chip code. The entire BIOS can be replaced. (The BIOS code insert in Figure 6(c) and the various code labels are reflected back to the CB.) Complete I/O control block structures are provided for each CP/M logical device, including CRT, keyboard, list, auxiliary, and disk. The control block includes port addresses, protocol support, and other default data needed to detect and control the status of each peripheral. Figure 6(b) also expands the systems tables and buffers created for disk support.

The addresses in Figure 6(b) indicate how stepping independence is achieved. Any off-chip routines changed by the user can be selected by altering the address of the CB. If Intel updates an on-chip routine, the address in the CB is updated automatically when the 80150 copies its constant structures into the CB. As explained above, full stepping independence is maintained, because any ROM changes can also be imple-

mented off-chip by having the address in the CB point to an off-chip patch. (The CB contains BDOS entry points (shown in Figure 6(b)) that make this change possible.)

The Configuration-Independent Interface

Use of the predefined configuration requires that the 80150 be installed at the top of the 8086 memory address space (FC00:0). The 16-byte internal hardware boot is activated at all POWER ON and hardware resets, and passes control to the 80150. The 80150 initialization sequence uses this positioning to indicate the default hardware configuration (floppy disk, printer port, serial console, or auxiliary port). Each device has predefined port addresses, interrupt assignments, and protocols. The iAPX 186 or 188 CPU supports programmable chip-selection and the on-chip DMA drives the floppy disk controller.

If the configuration must be altered, or if the BIOS code needs revision, the 80150 can be installed on any 16K code boundary except at the very top or bottom of memory. A PROM that contains off-chip code and data for a user's particular configuration is also installed at the top of memory.

The 80150 initializes the default system hardware tables, then calls an EPROM to complete or revise the existing data in the off-chip CB RAM area. At this point, the CB contains the addresses that select either on-chip or off-chip code. When the configuration is complete, control is returned to the 80150. The 80150 completes the CP/M initialization, displaying the familiar CP/M "A" sign-on.

Conclusion

Converting software to silicon is not new. But redesigning software to consist of on-chip ROM code and configurable RAM data is somewhat more innovative. One silicon-related specter that haunts software designers is the fear of "committing code before its time." But software designers can *never* expect to produce bug-free code the first time. And system designers cannot always predict the capabilities or the implementation requirements of peripheral devices that have yet to be built. Nevertheless, software designers who use the general silicon-implementation strategies of position independence and configuration independence, and who provide for stepping independence, can create standard silicon hardware without fear of component obsolescence. □

About the Authors

Ron Slamp received the A.S. degree in software technology from Portland Community College, and gained much of his skill in electronics at Clark Community College in Vancouver, Washington. He has worked in Intel's OEM Module Operation in Hawthorne, Oregon since 1978 and is currently the project leader for component software.

Jim Person received the B.S. degree in mathematics in 1962 from the University of Arizona. He was the engineering project manager at Intel for the 80150 "CP/M-on-a-chip."

June 1983

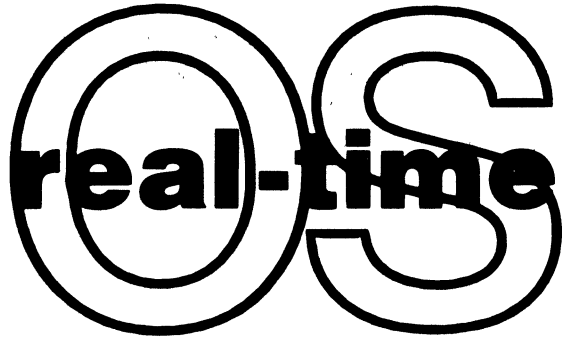
Putting Real-Time Operating Systems to Work

Stephen Evanczuk
Electronics Magazine

SPECIAL REPORT

Punching in for real-time jobs in industry, R&D, and offices, operating systems use special software structures to squeeze better-than-ever performance out of 16-bit microprocessors

by Stephen Evanczuk, *Software Editor*



□ A special class of operating systems is hard at work in the 16-bit microsystem world. For controlling environmental processes, acquiring data at high speed, or even handling transactions at a commercial bank, these operating systems contain mechanisms that enable them to respond rapidly to external events and that differentiate them from the more familiar general-purpose operating systems.

In fact, all the operating systems for 16-bit microprocessors respond in a reasonable period of time. But the general-purpose, or developmental, operating systems like CP/M, Bell Laboratories' Unix, and MS-DOS are intended for standard programming activities like editing, compiling, and file management [*Electronics*, March 24, 1982, p. 113]. As such, they lack certain software structures needed for reliable control of processes producing data at a high speed.

Real-time operating systems tend to fall into two general categories—multipurpose and embedded, reflecting the type of hardware they run on. Multipurpose real-time systems are typically built around full-fledged microcomputer systems with terminal, keyboard, plenty of system memory, and mass storage. Furthermore, in process-control or data-acquisition applications, some special-purpose hardware is usually included in these systems to serve equipment or high-speed data input operations. Besides the familiar applications for research and development, transaction-processing environments are an example of situations needing multipurpose real-time systems.

No doubt the largest class in volume because of their growing use in consumer items, embedded systems are minimal hardware systems, often just one-chip microprocessors that control limited parts of a larger system. Programmers ordinarily employ a special development system to create the software, which is loaded into the target system for use and ideally is never seen again.

To meet the needs of these two classes of applications, real-time operating systems come in three flavors for 16-bit microprocessors. Serving multipurpose real-time systems, one type—discussed in the

first part of this report (see p. 106)—includes all the software development support found in their general-purpose counterparts. Furthermore, many can be stripped of the layers needed in the developmental environment and placed in programmable read-only memory for use in an embedded system.

For those who swear by Unix, the group of Unix-based operating systems discussed in the second part (see p. 111) may mean no need to swear at it in real-time applications. A growing number of vendors are starting to convert this admittedly non-real-time operating system into versions that can be used to handle external processes. Although the industry is cautious, if not downright skeptical, of real-time versions of Unix, the fact that C—the language of Unix—is so highly regarded for use in real-time applications may help swing this group into the forefront.

The potential for distributed-control systems based on embedded microprocessors hinges largely on the availability of high-performance real-time operating systems that can be plugged into the application with the same ease as an integrated circuit. Called silicon software, these operating systems discussed in the last part (see p. 114) have been designed to be stored in read-only memory. Providing a fixed set of system calls, they present programmers with a consistent set of high-level commands to perform the low-level functions usually built from scratch.

Building system-level software from scratch has long been the hallmark of real-time programmers, even a mark of honor. Fortunately, however, the increased acceptance of ready-made operating systems using well-understood algorithms (described in the first part) is helping to replace this software "random logic" with rather more standardized packages.

On still another level, the unique responsiveness and throughput demonstrated by real-time operating systems is a truly user-friendly feature. For this reason, these systems should find their way into less obvious real-time applications, such as transaction processing, word processing, and personal work stations for office automation.



Algorithms star in multipurpose systems

Whatever environment it finds itself in, the function of an operating system is the efficient management of shared resources by a number of users, whether these are human beings accessing a computer through terminals or programs vying for a single central processing unit. In fact, the degree of sophistication of an operating system is reflected by the number and types of physical resources it manages and by the fineness of control it exercises in their management. And operating systems targeted for control of the external environment must wrestle with the most demanding resource of all—time. The degree of care with which such software is designed to manage time is what determines its suitability for the real-time environment.

Schedulers and queues

Two critical aspects of the real-time environment are the random nature of physical events and the simultaneous occurrence of physical processes. Consequently, interrupt handling and multitasking are primary attributes of a real-time operating system. In fact, it might be

argued that the mechanism for handling multitasking—the scheduler—is the heart of the operating system. The rest of the operating system lies atop this kernel and serves the specific demands of the application environment.

In particular, the lists, or queues, and their managers that surround the scheduler are constructed to deal with the different physical resources supported by the operating system. Thus, one queue may contain those tasks (processes, or programs in the course of being run) that are ready to execute on the processor, another queue may be tasks waiting for access to input/output hardware, and another queue may contain tasks waiting for some specified event to occur.

In any multitasking operating system, the scheduler uses the queues as input. Its output, on the other hand, is a single task that has been activated and allowed to execute on the central processing unit. The scheduling algorithm in large part defines the operating system.

In one system, the scheduler may simply select a task on a first-come, first-served basis, allowing it to run until completion or until some specified period of time has elapsed. This type of relatively primitive algorithm was commonly used in mainframe computers running simple batch-oriented operating systems.

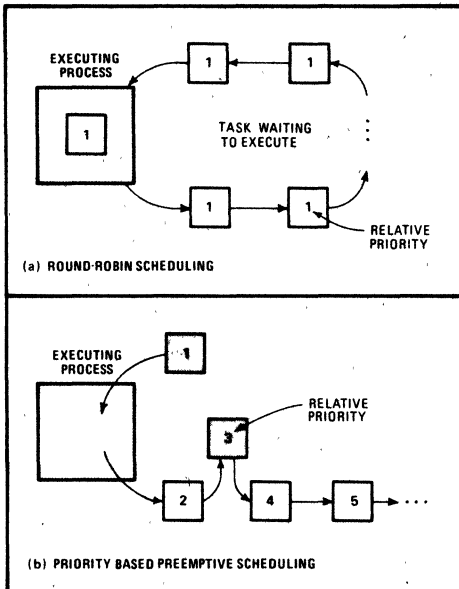
In a slightly more sophisticated operating system that can be used interactively through terminals, the scheduler may select tasks on a round-robin basis and permit each of them to run for a specified period of time (Fig. 1). Once the task exceeds its time slice, it is placed at the end of the queue and forced to wait until all other tasks have had a chance to execute.

Round-robin scheduling with equal time slices is adequate if every task is no more important than any other task. However, if some are considered to possess a higher priority, then a more sophisticated scheduling algorithm must be used—one that recognizes that some tasks are more important, but that no task should be excluded from using the CPU.

One solution is the use of several queues, where the length of the time slice is related to the priority of elements in the queue. In this case, the scheduler would allow all tasks in each queue of a different priority to execute on the CPU, but lower-priority tasks would be given less time.

A further refinement permits higher-priority tasks to suspend a running task. This technique, called preemptive scheduling, is an important feature for real-time environments, in which the delayed execution of a high-priority task could have disastrous results, rather than simply disappointing the user.

In scheduling algorithms, tasks may exist in a number of logical states, depending on their readiness to run. In the Versatile Real-Time Executive (VRTX) from Hunter



1. Priorities. In round-robin scheduling (a), tasks (or processes) take equal turns executing, while a higher-priority task will supersede a lower-priority one in priority-based preemptive scheduling (b). Most schedulers employ some combination of these techniques.

& Ready Inc., Palo Alto, Calif., for example, tasks are driven through four possible states by external events, by other tasks and system utilities, or by their own system calls (Fig. 2). For example, an executing task may delete itself—in which case it enters a dormant state—or may cause itself to be blocked either explicitly through a call to suspend itself or implicitly through a call to perform some I/O function. On the other hand, once suspended, a task may reschedule itself through a system call, or an external real-time event may bring the task back into the ready queue.

Recognizing the importance of scheduler design, at least one software vendor has made it easier for real-time users to build systems around a prepared kernel. United States Software of Portland, Ore., is offering a basic scheduler that assembles into less than 100 bytes of object code for the target microprocessor [*Electronics*, Nov. 17, 1982, p. 206]. Furthermore, in anticipation of real-time systems targeted for specific application areas, U. S. Software supplies a list of design notes detailing extensions to the basic kernel.

Another use for queues

In addition to having queues serving the scheduler directly, most systems use them as the preferred means of associating a task with a required resource. For example, one capability commonly found in real-time operating systems is the ability to suspend a task for a specified period of time. Typically, the operating system contains a special queue for this function. Each element in the queue is a task in a suspended state. Associated with each task is a counter that contains the number of clock ticks remaining until it should be reactivated.

For example, in iRMX-86 from Intel Corp., Santa Clara, Calif., the counters keep track of the incremental time remaining with respect to the previous element in the queue, rather than the total time remaining before

the task may be reactivated. Thus at each clock tick only the counter in the element at the head of the queue need be decremented, rather than every counter in every queue element. This method takes longer to insert new elements into the queue and so requires slightly higher overhead for insertion than when the total time is maintained by each counter; however, that overhead is more than offset by the time saved by updating only a single counter.

Real-time environments pose a special set of problems for resource allocation. Besides all the more familiar problems of scheduling, a real-time operating system must maintain reliable behavior under extremes of load when it is driven by a high rate of external stimuli. From the system user's point of view, the system must maintain a predictable level of response and throughput.

In an interactive environment, users sitting at terminals measure response as the time the system needs to react to a keystroke. In general, system response is the time that the system needs to detect and collect data from some external stimulus. Throughput, in an interactive environment, is seen as the number of users able to utilize the installation simultaneously. In a more general real-time environment, throughput is the rate at which the system is able to collect, process, and store data.

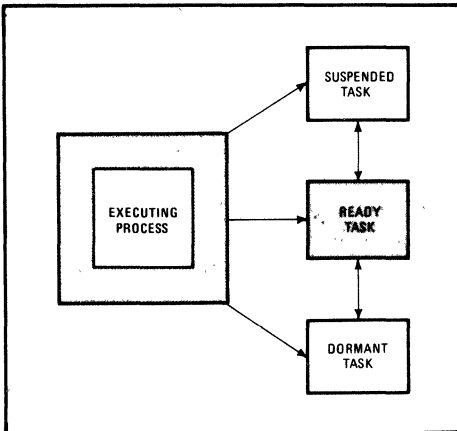
In fact, although response and throughput share some common software elements, operating-system designers will invariably find themselves forced to make choices that will tend to optimize one at the expense of the other. Often, the interrupt-handling requirements of a real-time operating system force this choice.

Interrupt processing is hardware and software integration at its most demanding (see "Handling hardware interrupts," p. 108). To handle interrupts, operating systems often place layers of software between the user and the microprocessor in order to allow different levels of performance and capability.

Intel's RMX-86 is a typical example of distinct levels of software used to perform basic interrupt processing. At the lowest level, an interrupt handler works intimately with the hardware to execute some operation, such as sending a message character by character to a printer. Code for interrupt handlers is kept compact and simple, since system interrupts are disabled during their operation. The higher level, called the interrupt task, works at a priority associated with the particular hardware it services. Interrupt tasks act as interfaces between application tasks, working with specific interrupt handlers to complete execution of operations dealing with external devices. RMX makes this interrupt-handling mechanism available to application programs through a special set of system calls.

Protection and communication

Once the interrupt software has completed its function, tasks that use the data are indistinguishable from any other task in the system as far as the operating system is concerned. Unless special care is taken, conflicts could still arise between two separate tasks that might need to use the same resource, such as the same location in memory. MP/M-86, for example, employs a special queue, called a mutual exclusion queue, that contains a unique message representing the shared resource. In or-



2. Task states. As one task (or process) runs, others may be in various states of readiness. In Hunter & Ready's VRTX, for example, tasks can be ready (able to run immediately), suspended (waiting for a resource), or dormant (deleted by a system call).



der to use the resource, a task must first capture the message, much as a node in a token-passing network must first obtain the token before being at liberty to transmit.

Per Brinch Hansen¹ identified such shared resources as key elements in multi-tasking systems. Sections of code that access critical resources are called critical regions. The simple expedient of ensuring that only one task at a time is allowed in a critical region guarantees that multiple tasks may share the same critical resource without fear that its integrity may be compromised when two of them attempt to access it simultaneously (Fig. 3).

This concept of the mutual exclusion of tasks from critical regions is implemented in a structure called a monitor, in which critical regions are gathered in one section of code and protected from use by more than one task at a time. The MSP operating system from Hemenway Corp. of Boston [*Electronics*, Jan. 27, 1983, p. 119] explicitly supports mutual exclusion through monitors in its internal structure.

Furthermore, user-written routines needing monitor protection are provided with four functions in MSP that are implemented using hardware traps for rapid access: Entermon, Exitmon, Wait, and Signal. Entermon and Exitmon serve as monitor entry and exit points, respectively, performing required housekeeping functions. Entermon disables system interrupts and preserves all registers, while Exitmon reverses these actions. Wait and Signal, on the other hand, work in tandem to control access to a critical resource. Wait queues up tasks needing an unavailable resource. Signal releases them from the queue when the resource becomes available.

Wait and Signal are examples of an intertask communication mechanism, called semaphores, found in most real-time operating systems. As noted, these commands simply queue up and release tasks needing a critical resource. Such a resource may be an I/O device, a memory location, or simply a go-ahead signal that synchronizes a pair of tasks. For example, task A may execute only after task B has completed. In this case, task A would begin with a Wait (flag) command, where the flag is used as an associated variable. Task B, on the other hand, would end with a Signal (flag) command. In this way, task A would be blocked until task B had executed its Signal command at the end of its processing. But exchanging simple go-no-go signals is not sufficient for many multitasking environments.

For longer messages, real-time operating systems offer extensive intertask communication facilities called mailboxes. Mailboxes are essentially semaphores with storage. As such, tasks needing data from another task will wait until the other has loaded the mailbox with the information. Intel's object-oriented RMX-86 transfers any of the defined objects in the system through mailboxes. Hemenway's MSP, on the other hand, provides a buffer of fixed size that may be used without restriction on its contents, as long as the 256-byte buffer is not exceeded. With its Multibus message exchange (IMMX) extension to RMX for

Handling hardware interrupts

Underlying the special software of a real-time system is the assumption that the hardware itself can respond in a coordinated fashion to external events, or interrupts. In fact, microprocessors contain subsystems whose sole function is to deal with interrupts in a way that eases integration of the interrupt-handling software.

All modern computers integrate interrupt-handling hardware and software at a very low level of design. When a user accesses a microprocessor through a terminal, the same hardware interrupt facilities come into play as when, for example, an analog-to-digital converter sends data to the same type of microprocessor. The software response, on the other hand, depends on the type of operating system, but both real-time and general-purpose operating systems must take some action, like read in the data value or the character.

Examining the details of a simple keyboard task illustrates the complex nature of real-time processing. It also serves as a vehicle for introducing some of the basic vocabulary in this field.

A standard software subsystem in a microcomputer system, called the keyboard monitor, is responsible for working with the hardware interrupt system to detect a character, collect it, and effect some action based on the input character. When a key is struck on a terminal, the corresponding byte is converted into a serial stream of bits that are passed from the terminal to a universal asynchronous receiver-transmitter. Once it receives the full character, the UART generates a hardware signal, or interrupt, that notifies the processor. Since interrupt management is a common activity, processors contain special hardware to respond to this signal.

Although the details may vary from one particular microprocessor to the next, the result is the same for all. When its interrupt-request line is asserted, the processor ceases its current processing and places values from its internal registers into system memory. Typically, the processor status and instruction-address registers are saved in the system stack, a last-in, first-out buffer located in some portion of system memory. As the figure shows, the processor responds to the original interrupt-request signal by issuing a signal of its own, called an interrupt acknowledge.

The peripheral hardware that originated the interrupt detects the interrupt-acknowledge signal on the system bus and responds by returning the memory addresses of both the interrupt-handling subroutine and the new processor status. Typically, the new processor status will provide for disabling any further interrupts. This latter action is a simple precaution, preventing a single external stimulus from causing a continuous series of interrupts that will eventually result in an overflow of the system stack.

Such an interrupt mechanism, called a vectored interrupt, allows the speediest identification and reaction to an interrupt. (An alternative interrupt mechanism used by earlier processors, called a device-polling interrupt, simply forced the processor to switch to a defined address in memory containing software that polled each peripheral device until the device that generated the interrupt was discovered.) At

this point in the interrupt-handling task, all the activity was exclusively in hardware, but nevertheless resulted in extensive processor activity and bus traffic due to multiple accesses of system memory and the involved peripheral-device controller.

Consequently, it is not surprising that the time for hardware to set the processor to handle the interrupt—the hardware-interrupt latency—should be several processor cycle times in length. In general, hardware-interrupt latency is not a fixed number, but will lie within some range, since the processor will need a variable length of time to complete its current instruction and to initiate the interrupt-acknowledge signal. For example, if a processor is involved in a lengthy floating-point operation, several microseconds could elapse before the interrupt is acknowledged.

Once the processor has reached the interrupt-handling subroutine, the contents of only a minimal set of its internal registers have been preserved. However, before the real work of the subroutine may commence, the contents of other registers and variables shared by independent sections of the operating system must be preserved. The time needed to perform this action is called the context-switching time. Only after the software context is switched is the system ready to begin handling the special requirements of the device that originated the interrupt. The period of time between the occurrence of the external event and this state is the total interrupt-response latency.

In real-time operating systems, interrupt-response latency is usually a specified value—around 100 microseconds in very high-performance systems based on 16-bit microprocessors. Designers often bypass the constraints imposed by response latency by including special-purpose hardware to boost system response to external events.

Throughout all this time, system interrupts are still disabled. However, now that the context switch has taken place, the keyboard handler is free to transfer the character from the UART. Deciding where to put the character is important in terms of system throughput and overall efficiency. When it is put in some specified location in system memory, system interrupts must remain disabled; otherwise, if the handler attempted to service a subsequent interrupt, the new character would overwrite the character already in the location, but not yet fully processed.

In general, there are two methods for handling this problem. In the first method, the character is simply placed on the system stack and referenced through the relevant pointer. In an alternative method, the character is placed in a block of memory that has been reserved just for the handler and is called a context block. In this case, the character is referred to by using a specified offset from the base of the context block. Each time the keyboard handler is called in response to an interrupt, one of

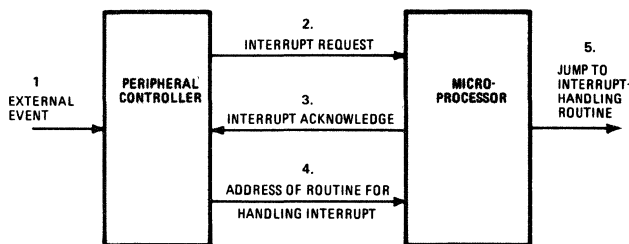
these context blocks is reserved from available system memory. Setting up a context block and switching the processor to it in a context switch accounts for a significant fraction of the time that is needed to respond to an interrupt.

Software code, such as the UART handler in this example, that does not contain any memory locations for variables is called reentrant because the processor may asynchronously enter it, be called away by an interrupt (even one that results in another call to the same piece of code), and return without loss of data or context. If the code is not already resident in system memory, another routine causes a copy of the code to be read from storage into memory. With reentrant code, only a single copy of the program or task need be resident at any time. Each context block, or logical copy of the task, is called an instance of the task.

Multiple instances of a task help explain some of the confusion associated with performance figures reported as a result of benchmarks. In examining benchmark figures, it should be clear just what the values are that are being reported. Total interrupt latency generally includes hardware interrupt latency, the time to create an instance of a task (plus the time to call in the task into memory if not already resident), the context-switch time, and an additional period needed to execute a variable amount of code that causes the data to be read from the peripheral registers. Creating a new task means either calling in a new task and creating a context block for an instance of it or just creating a new instance of a task already existing in memory.

Once the handler in the UART example reads in the character from the receiver buffer, it will reenables interrupts. The time between entry to the interrupt routine, when interrupts were disabled, until the time when interrupts are reenabled is an important factor in determining the effective latency of system response.

This dead time must be minimized, or the system will remain deaf to external stimuli for unacceptably long periods of time. In fact, the length of time that system interrupts are disabled is one of the criteria for determining the usefulness of an operating system for real-time applications. The longest period during which interrupts are disabled is a direct measure of the responsiveness of the system. Because of the weight of disabled interrupts on total system performance, modern microprocessors use a number of hardware-interrupt levels, or priorities, that disable interrupts at or below the priority level of the device originating the interrupt.





multiprocessor-based systems, Intel replaces the concept of a mailbox with that of a software port connecting different tasks, whether they exist on the same or different physical processor.

Unlike memory-intensive software development systems, real-time environments find less need to support a virtual address space. In fact, the increased system overhead is less than desirable, because the designer seeks to minimize response latency. A useful feature, however, that can be found in some real-time operating systems is a set of system calls responsible for dynamically allocating and deallocating memory.

For example, in the ZRTS system from Zilog Corp., which comes in different versions for the Cupertino, Calif., firm's segmented Z8001 and nonsegmented Z8002, a set of three system calls provides for dynamic allocation and deallocation, as well as information on the status of memory allocation. The system call for memory allocation allows application programs to specify the attributes of the memory block to be allocated and returns a name referring to the created structure.

Besides similar system calls, Intel's RMX adds some calls suited to its context-based architecture. In RMX, each task lies within the context of a job environment that bounds the scope of tasks within it (Fig. 4). As such, each task is allowed to draw from the memory pool of its job. In case more memory is required than that initially allocated to the job, a pair of system calls provides for querying the system on the size of the job memory pool and for dynamically changing it.

Dynamic memory allocation and deallocation is a relatively advanced concept that exacts some overhead during runtime. However, the alternative—static allocation before runtime based on expected requirements—may be less suitable for applications in which the real-time environment is relatively unpredictable.

In real-time operating systems, disk-file management is treated as just another asynchronous task possessing a particular set of critical resources—mass-storage devices. In real-time environments, file-management utilities have

to meet not only the requirements of general-purpose systems but some additional demands.

In terms of system response, a requirement of real-time operating systems in heavily loaded systems is the ability to conduct asynchronous I/O operations. In such an operation, the calling task simply queues up the I/O request, then immediately returns as if the task were completed in zero time. When the I/O request is fulfilled, the operating system switches the processor to a separate routine whose address is supplied when the original asynchronous request was made. This completion routine then may continue any processing that may be required following the I/O request.

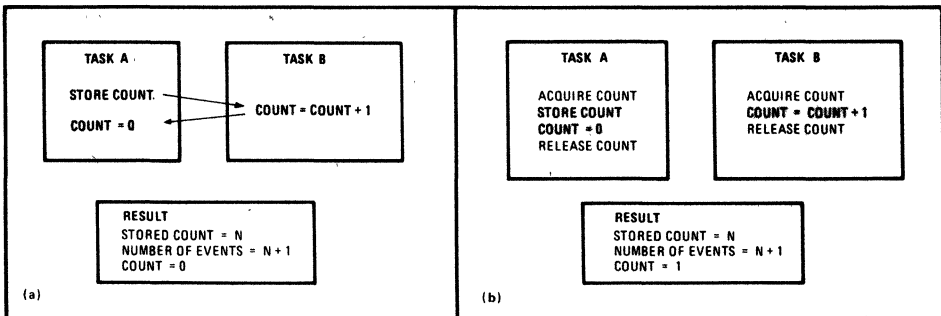
System throughput depends heavily on the efficiency and performance of the I/O subsystem. Peripheral controllers with direct memory access and the ability to move the disk's read-write head without necessarily performing data transfer can significantly reduce the overhead associated with data movement.

Reducing overhead

System software can also contribute to reduced overhead by providing a simple disk organization when high throughput is needed. One of the simplest structures is a file consisting of an unbroken series of disk sectors, such as the contiguous file in Hemenway's MSP or the physical file in Intel's RMX. By ensuring that the next block of data will be written to the next physical sector on a disk, the operating system can reduce the delay caused by head movement on the disk.

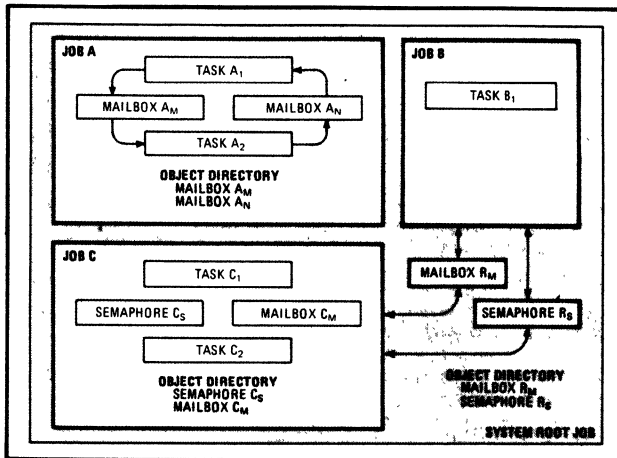
In their use of an I/O interface that is common to all system device drivers, MSP and RMX attack another important aspect of system design, though one not necessarily tied to their utility in real-time applications. In MSP, a basic I/O routine called Iohdr serves for all operations by accessing a special block of information in memory. RMX, on the other hand, uses a number of device-independent system calls to handle communication with peripheral devices.

Next to multiprocessor-based software systems, real-time software systems are the most difficult to debug. Again, the cause is the distinguishing feature of real-time operating systems—precise management of time. Standard debugging tools for single-user general-purpose op-



3. Critical regions. If two asynchronous tasks use a counter, events can be miscounted if task B interrupts task A before the counter is reset (a). Forcing the tasks to acquire a counter before using it (b) ensures synchronization through the critical regions (tinted).

4. Job context. In Intel's RMX, all jobs exist within the context of another job. A directory defines the objects that are known to other objects in the same context. For example, all three jobs may use mailbox R_M since it is in the system's root-job object directory.



erating systems generally disable all system interrupts in various phases of the debugging routines. Since the object of a real-time software system is asynchronous involvement with the task under control, this effect makes standard debugging tools useless.

Ideally, debugging real-time software would use performance-analysis tools and troubleshooting aids built into the operating system itself. Unfortunately, the processing overhead and additional memory requirements imposed by such a technique make this an unpopular notion in the design of an operating system. However, some systems do provide some means for run-time error handling. The exception handlers in RMX, for example, are procedures that are associated with each task when it is created. If a task attempts to use a system call but encounters an error, called an exception, the operating system invokes the associated exception handler to allow some graceful recovery from the error.

Although the technique in VRTX is not true exception handling, Hunter & Ready's silicon-software system does include a mechanism to build run-time debugging software. A special location in the VRTX configuration table (see p. 115) causes a user-defined routine to be called whenever a context switch is performed. By recording information about the task as well as the processor, such

a routine can be used to create a list, called a trace, of the history of task execution.

Because real-time systems often include special-purpose hardware, the accepted technique for debugging user-written routines uses the classical approach of collecting data before and after passing through a suspect region, along with a logic analyzer to monitor timing of traffic through critical regions.

Intel offers some relief to this problem through the iRMX debugger. In particular, the debugger allows the user to work with individual tasks without interfering in the operation of other tasks, as well as to monitor the activity of the system as a whole without disturbing it. The debugger recognizes data structures in the RMX kernel, so the user may examine system objects. In addition, Intel's crash analyzer brings mainframe debugging power to microprocessor-based applications using RMX.

Zilog's ZRTS configuration language offers another level of support to the development of systems targeted to specific hardware complements. By defining the details of the hardware, a system designer can configure ZRTS to particular systems.

Reference
1. Per Borch Hansen, "Operating System Principles," Prentice-Hall, Englewood Cliffs, N. J., 1973, p. 84



Designers tune Unix for real-time use

□ With an eye on the growing momentum of Bell Laboratories' Unix, real-time system designers have endeavored to squeeze this complex operating system into the rigid confines imposed by the demands of real-time environments. Although Unix brought advanced system ca-

pability to mini- and microcomputers, the original intent was to provide a hospitable software-development environment, rather than to include the features considered necessary for real-time uses.

Until now, data-acquisition systems employing unmod-

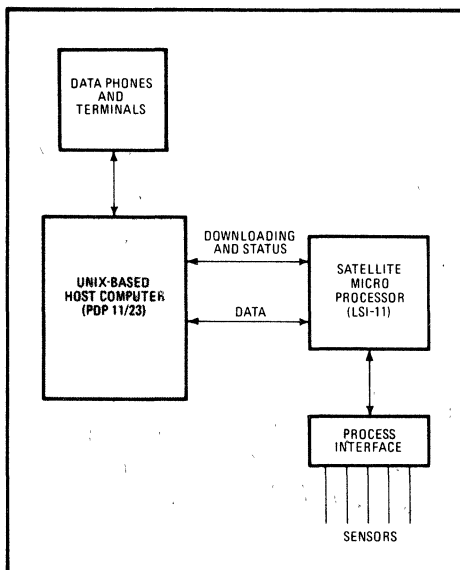


ified Unix typically used dedicated microprocessors to buffer a central computer from constant random activity caused by external events. For example, in the Concepts process-control system

Unix goes real-time

from Bell Laboratories, Murray Hill, N. J., a Unix-based host is linked with auxiliary microprocessors. In each microprocessor, software derived from Unix software handles the low-level details of real-time activity (Fig. 1).
 Appearing in all shapes and sizes, Unix-compatible executives, Unix lookalikes, and new Unix versions are bringing this popular environment into real-time applications. However, unlike their colleagues creating totally new operating systems (see pp. 106-111), designers of these second-generation systems are constrained by the boundaries set by the original. Caught between Unix's complex organization and the high-speed needs of some real-time applications, they have opted for preserving the basic architecture. Still, for intensive data-acquisition applications, vendors like VenturCom, Cambridge, Mass., and Masscomp, Littleton, Mass., add on dedicated hardware like high-speed peripheral controllers to link devices into the main system without losing the generality of the Unix software architecture.

For microprocessor-based dedicated systems, memory-



1. Satellite processing. In Bell Labs' Concepts system, separate microprocessors handle low-level details of process control. Yet another processor—a host computer that runs the Unix operating system—is in charge of coordinating these satellite machines.

resident kernels like the C Executive bring a measure of Unix compatibility to even dedicated systems. Offered by JMI Software Consultants of Roslyn, Pa., the C Executive combines support of an extensive C-language run-time library with many of the features considered important in real-time applications. Although not directly supporting shared data in its multitasking architecture, the executive's intertask-communication facilities include data exchange through a queuing mechanism. As befits a real-time executive, the task-scheduling algorithm allows higher-priority tasks to preempt lower-priority ones. Because it is intended primarily for embedded systems—that is, dedicated microsystems that do not have disks—the C Executive is totally contained in system memory and does not support the extensive Unix file-management subsystem.

Controlling real-time tasks

Full-blown Unix lookalikes, on the other hand, find themselves forced to deal with some of the very internal structures that aided Unix's rise in popularity. For applications like program development where regular scheduling is more important than instant response, scheduling is aided by Unix's manipulation of the priority levels of tasks (or processes, in Unix's preferred terminology). For real-time applications, however, the slight uncertainties this feature introduces could destroy the synchrony of timed events controlled by the system.

Consequently, one enhancement commonly found in the real-time offshoots is the addition of some mechanism to ensure more precise control of real-time tasks. A technique that sits well within Unix's task-oriented (that is, process-oriented) design is the definition of a real-time class of tasks (or processes). This class earns special rights in the operating system, such as a guarantee that each task will not be swapped out of memory, but remain locked in and ready to respond more rapidly to events.

VenturCom's Venix, for example, defines a real-time priority level. The scheduler allows tasks running at this level to maintain control of the processor for as long as necessary. In contrast, Regulus from Alcyon Corp. of San Diego, Calif., speeds response to real-time events through the use of 32 user-defined priority signals.

Better I/O handling

In addition to its scheduling algorithm, Unix's method of handling input/output operations needs improvement to perform well in real-time applications. Aiding total system response, the asynchronous I/O procedure in Venix supplements the conventional synchronous procedure in Unix, in which the requesting task must be suspended until the I/O operation is completed (Fig. 2). By placing asynchronous requests at the head of the I/O request queue, Venix's manager lets real-time tasks issue a write request, for example, and immediately continue processing, assured that the request will be honored next. Concentrating instead on improving what happens when I/O requests have been completed, Masscomp's enhanced version of Bell Labs' Unix System III adds a modified signal called an asynchronous signal trap. Similar to the concept of completion routines in other operat-

Going Forth with alternatives

Few nightmares evoke the feelings of dread experienced by a programmer who must alter code that has been developed by another programmer—worse yet if the code is all assembly language for an embedded system. Fortunately, system developers are seeing the light of day and are specifying one of the commercially available real-time operating systems, so programmers now are dealing with a set of well-defined software calls for system functions. Still, for the true diehard who feels restricted by using someone else's system or the developer trying to eliminate all processing overhead caused by the operating system, alternatives do exist.

For straightforward, yet high-performance, process-control applications, the use of a finite-state machine as the controller is an easily implemented technique. A finite-state machine is simply some device that produces a defined output state based on its input state. For example, a microprocessor may read some input register, access a table in memory using this input as the address, and send out the value contained in the accessed location. In such a system, a value could be created with a single indirect move instruction in a microprocessor using a memory-mapped input/output scheme. Clearly, using a microprocessor this way would allow only a relatively small number of states.

Besides this hardware approach, the software alternatives include the interpreters for high-level languages, such as Forth and concurrent versions of Pascal, that are appearing in the read-only memory of single-chip 8-bit microcomputers. For example, the CDP1804P complementary

MOS single-chip microcomputer from RCA Corp.'s Solid State division, Somerville, N. J. [*Electronics*, Nov. 30, 1982, p. 127], contains a core interpreter for Micro Concurrent Pascal (mCP) from Enertec Inc. of Lansdale, Pa. Based on Per Brinch Hansen's Concurrent Pascal, mCP contains all the constructs necessary for real-time applications, such as shared data, monitors, interrupt handling, and task queuing and switching. RCA also provides a ROM that extends the core interpreter to include full multitasking support. Software for this microsystem is developed using an RCA cross-compiler available on various host machines.

In parallel with the use of modified high-level languages like mCP, Forth interpreters are on the verge of appearing as single-chip microcomputers like the CDP1804 or the RF1/12 from Rockwell International Corp.'s Newport Beach, Calif., Electronic Devices division [*Electronics*, Jan. 13, 1983, p. 41]. After its development in the 1960s for real-time applications, Forth gained a slow acceptance among system developers. But with the inception of Forth standards committees and the spread of interpreters into more systems, this stack-oriented language is rapidly attracting the attention of larger houses.

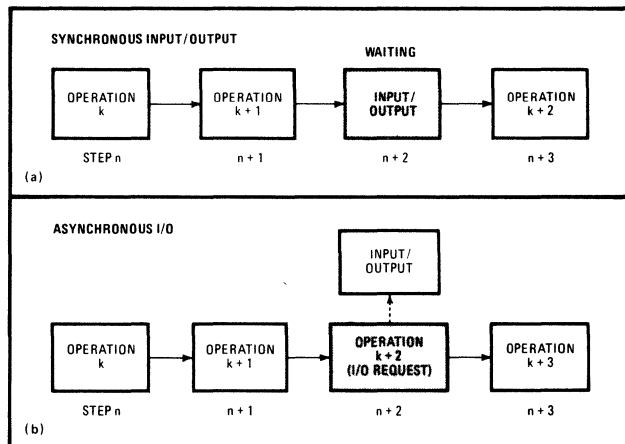
Forth is a threaded language in which basic procedure calls, or words, are used to build up more complex words. Because of the threading, programs tend to be very compact. Once the programmer gets used to reverse-Polish notation, program development is simply a matter of building up the system dictionary with the words needed for the particular application.

ing systems, the AST mechanism allows tasks to perform operations that were contingent on the completion of a separate real-time operation. For example, by issuing an AST when it has completed its work, a read task is able to notify another task that a buffer has been filled. The other task is then free to initiate whatever calculation

may be needed to make use of this new data.

Besides such modifications improving Unix's response to asynchronous events, Masscomp upgraded the system's throughput by adding support for contiguous files to the file-management system. In this way, large amounts of data may be written at a high speed to

2. No blocking. In synchronous I/O, execution of a task blocks, or waits (tinted), until the data transfer is completed (a). Since I/O is handled independently, a task need only request an I/O operation (shaded) and continue on to the next operation



consecutive disk sectors. Since other disk accesses are locked out in this mode, the disk head will be positioned correctly, thereby eliminating unnecessary and time-consuming movements.

In addition to these I/O add-ons, Masscomp boosted intertask communication capability by enlarging the Unix standard intertask communication mechanism, called pipes, to allow tasks to transfer buffers. In an alternative approach, Charles River Data Systems of Natick, Mass., allows tasks in its Unix-like Unos system to share data directly. A number of independently constructed software tasks may use a common set of locations in memory to transfer data between themselves or to perform some sequence of calculations. However, whenever asynchronous tasks share some common re-

source, their use of the resource could result in corrupted data—unless some mechanism coordinates their activities, such as the monitor concept described on page 108. Unos provides a mechanism called event counts to help avoid these conditions.

Event counts are integer values that are a nondecreasing count of the number of times some particular event has occurred. By using an event count associated with some task that produces shared data and another event count for a task that consumes the shared data, programmers may ensure the correct sequencing of asynchronous data-producing and -consuming tasks. Similarly, event counts serve as primitive operations for emulating the synchronization function that is provided by semaphores and the mutual exclusion that is furnished by monitors.



Chips come to aid of embedded systems

□ Storing machine instructions in read-only memory is hardly a new concept in microprocessors. If supporting software totally breaks down, Digital Equipment Corp.'s LSI-11, for example, resorts to a basic keyboard monitor stored in a special ROM that is logically placed in the input/output address space. Using a primitive on-line debugging technique stored in the same ROM as the monitor, a software designer may read and alter memory locations and initiate a bootstrap loading operation from storage—a common provision in computer systems.

From these primitive beginnings, however, ROM-based software has evolved into complete operating systems in memory, engendering the term silicon software. Complementing hardware for distributed-processing architectures, such silicon-software systems signal a migration of application software into dedicated microcomputers previously considered unable to gain full systems capability. For developers of dedicated microcomputers embedded in some larger real-time system, silicon software spells the end of the need to reinvent the wheel to carry out the fundamental functions of a real-time operating system.

Extending the microprocessor

Functionally, silicon operating systems extend the microprocessor's instruction set to include system-level instructions that perform operations on software structures, like queues and tables, rather than on hardware registers. Application-program developers are then presented with a virtual machine—one that is perceived by the programmer as different from the actual host processor. In these virtual operating-system machines, their instruction set includes a well-defined set of system calls as well as the basic machine instructions of the host microprocessor. For example, with systems like VRTX and RMX, the virtual microprocessor has a special set of instructions for handling interrupts (see Table 1).

For system developers, however, the problems in developing reliable silicon software extend beyond resource

protection, timing, and communication problems (see pp. 106-111). In fact, the development problems extend beyond the purely logistical exercise of maintaining a separate ROM-based instruction store and one for variables that need to be placed in system read-write memory. Treading a fine edge between the full function of a general operating system and the fine-tuned performance of special-purpose software, silicon systems need to balance the need for a wide range of system functions with the requirement that they squeeze into a minimal amount of ROM.

Flexibility for expansion

Still, once a system meets a reasonable compromise between capability and size, it should not irrevocably lock the user into accepting its choices. For example, many real-time applications require some custom peripheral-device drivers and system-level functions. Consequently, the program should provide a mechanism for logically incorporating user-written extensions to the operating system, such as the user-defined pointers in the VRTX system from Hunter & Ready, Palo Alto, Calif.

In VRTX, a configuration table (Table 2) in system random-access memory allows specification of a custom routine that is to be executed whenever the system is initialized. For even more delicate control of system operations by custom software, a trio of pointers in the table specifies user-written routines to be accessed whenever a task is created or deleted or whenever a context switch is performed. Hunter & Ready also includes a location in this baseline configuration table for its anticipated file-management extensions to VRTX.

The 80130, an RMX-86 kernel in silicon from Intel Corp., Santa Clara, Calif., generalizes this approach through an index table containing pointers to system routines. If circumstances require the replacement of an existing system routine, the index-table pointer is merely altered to indicate the address of the new routine. In an

TABLE 1: SYSTEM CALLS FOR HANDLING INTERRUPTS

Instruction	Description
Versatile Real Time Executive (VRTX)	
UI POST	deposit message from interrupt handler
UI EXIT	exit from interrupt handler
UI TIMER	timer interrupt
UI RXCHR	receiver ready interrupt
UI TXRDY	transmitter ready interrupt
iRMX-86	
ROSSETSINTERRUPT	assign interrupt handler
ROSRESETSINTERRUPT	deassign interrupt handler
ROSGETSLEVEL	return number of highest priority interrupt level currently being processed
ROSSIGNALSINTERRUPT	signal from interrupt handler that event has occurred
ROSWAITSINTERRUPT	wait for occurrence of event
ROSEXITSINTERRUPT	relinquish control of the system
ROSENABLE	enable hardware to accept interrupts
ROSDISABLE	disable hardware from accepting interrupts

embedded system, this new routine could be placed in ROM along with application software.

Now that programs in ROM have matured into silicon systems, the development of software for embedded systems may now follow a more hospitable development cycle. The particular method used to create embedded systems will, in general, fall into one of two paths represented by the two major camps.

On one hand, kernels in silicon from systems such as RMX-86 or the MSP from Hemenway Corp., Boston, Mass., for the 68000 or Z8000 are self-contained subsets of the full operating system. Consequently, software programmers may use the full development version of the same operating system as that in the eventual target to create the application package. On the other hand, development of application programs around the ZRTS system from Zilog Corp., Cupertino, Calif., or Hunter & Ready's VRTX for the Z8002, iAPX-86 family, or 68000 relies on the use of a separate development system to create software for the target microprocessor, since this software does not have development versions.

Two approaches

The significance of these two approaches as usual depends on the intended application. Hunter & Ready views VRTX as a set of processor-independent building blocks that programmers use to construct application packages for embedded systems. As such, the programmers employ the same development systems that they might use to build application code, but now with the benefit of a sophisticated set of ready-made system-software components.

In playing its part in Intel's systematic drive toward

TABLE 2: VRTX CONFIGURATION TABLE

Table Entry	Entry Description
sys RAM addr	system beginning address
sys RAM size	system memory size
sys stack size	system stack size
user RAM addr	starting address for available memory in initial partition
user RAM size	size of initial partition
user block size	size of memory block for dynamic allocation
user stack size	size of stack for user tasks
user task addr	address of first user task
user task count	maximum number of tasks
sys init addr	address of user supplied initialization routine
sys tcreate addr	address of user supplied routine accessed when a task is created
sys tdelete addr	address of user supplied routine accessed when a task is deleted
sys tswap addr	address of user supplied routine accessed when a context switch occurs
[RESERVED]	address of Hunter & Ready future extensions to VRTX

providing an integrated environment around the iAPX-86 family, the 80130 holds the anchor position in an interlocked set of components. Able to function independently of the upper layers of the operating system, it provides a hardware base for the rest of RMX-86. Serving as a viewport into this system-software base for the central processing unit, Intel's universal run-time and development interfaces offer the mechanism for software portability needed for the next stage in the company's plan to grow into higher-performance microprocessors, such as the 186, 286, and 386.

While interlocking with the software in this way, the 80130 also must play its role in the complementary relationships being established at the hardware level. As such, it includes on-chip hardware support for system-level functions, including timers, interrupt controller, bus control, and bus interface.

Meanwhile, Intel's plan for software-in-silicon becomes evident as it gathers the other pieces of the puzzle, such as the 82730 text-coprocessor chip, the 82586 local-network coprocessor, and the 82720 graphics processor chip. Similar to the 80130 software connection, the 82720 graphics part interlocks with the rest of the system at the software level through its support of another well-defined software interface—the virtual device interface. Yet to come are pieces for voice I/O support, as well as some level of hardware support for data-base access. □

June 1983

Intel's Matchmaking Strategy: Marry iRMX™ Operating System with Hardware

Chappell Brown
Software Editor
Electronic Engineering Times

Intel's Matchmaking Strategy: Marry iRMX™ Operating System With Hardware

Intel's major software product, the iRMX™-86 16-bit operating system, which is now in its fifth release, represented a three-year development investment which most independent software vendors would have found a daunting prospect in 1978 when the project was conceived.

The investment was essential. By the mid-1970s, feedback from OEMs working with Intel's hardware revealed problems with system integration—the marriage of software with hardware. It consequently slowed sales, with the prospect of even greater problems at higher levels of circuit integration. Intel management, looking for ways of coping with the ballooning software requirements of the rapidly accelerating hardware program, began stepping up software development programs in the mid-1970s.

“The RMX program illustrates a number of things one needs to keep in mind with developing a real-time operating system,” explained Bill Lattin, Intel's OEM microcomputer systems manager. “Foundations must be well laid so the system can grow and evolve over time. And there is a need for the system to be open to modification by typical OEM-specific applications.

“Although the RMX program has been around since 1978, it has only recently hit its stride, as processor technology has advanced to use the full range of its features,” Lattin said.

The fast-paced microcomputer market had created a new situation for systems designers in terms of a radical shift in the hardware/software cost ratio. Earlier hardware generations involved various expensive centralized facilities. Not only was software cheap in comparison, but the hardware environment changed slowly, so that it was also feasible to rewrite systems as needed.

But when the price of a computer drops to as low as \$5, the hardware environment becomes volatile and software turns into a major investment. Intel was finding that customers might invest as much as two-thirds of their development costs in software, only to see it eclipsed by evolving VLSI technology.

It became evident that merely supplying components would become increasingly counterproductive. Thus, the Intel “total solution” emerged—a consistent systems approach to hardware sales, which naturally depends heavily on a viable software program.

Object-oriented programming is a method which has worked best in creating a software program blending with the component approach. By hiding data representation within an object with its own object manager, changes in the hardware environment that affect the data can be accommodated without having to change the rest of the software.



A price is paid in terms of program size with this approach, however. And it was difficult at the time to justify this kind of liability with the existing onboard memories of the 8-bit generation.

Bill Stevens, iRMX-86 program manager for release five, explained the difficult decisions that had to be made at the outset of the program. “Every engineering decision involves a trade-off. We wanted to optimize program productivity and we had to have modularity. The consequence of this was large size. It turned out that a minimum configuration was 12 kbytes wide and the full configuration was 128 kbytes. At the time we did not have 64k dynamic RAMs and 64k EPROMs, so we didn't have the technology to realize the systems of initial specifications times. Bruce Schafer has to take credit for making that decision to go ahead anyway, early on. . . it was a gutsy decision, and it turned out to be absolutely right!”

Had Intel known of the difficulty it was about to encounter in producing its 64-kbyte RAM, Schafer may have had second thoughts.

Schafer joined Intel in 1976 and began working on iRMX-80. "It was a nice little system," Schafer said. "A miniature dispatcher had evolved to handle multiple asynchronous events and became a primitive OEM operating system. It was tempting to do an enlarged version of it, mainly because I was already working on it for the 16-bit generation."

Schafer soon found himself centrally involved in the task of heading off the 16-bit software crunch, laying groundwork for a system that could cover a wide range of applications, many of them unknown at the time, and a system which could also evolve with hardware advances.

"When you set out to design a system of that scope, you don't just sit down and start writing code. It's definitely a top-down process," explained Schafer. He discovered early in the project that the purely technical hurdles in writing software were minor compared to orchestrating a team of engineers on such a comprehensive project.

The iRMX-86 system is multi-layered, and the project had to be coordinated across these layers along with the sequence of planning, design and implementation. On top of that, a thorough testing program had to be coordinated with all phases.

"I had a difficult time convincing engineers on the project that documentation of their work was as important as the work itself. Specifications were absolutely crucial to the development phase," said Schafer.

Schafer began with a customer survey to discover the kind of problems OEMs were experiencing with system design. He wrote a production implementation plan, which was critiqued by marketing and engineering personnel. This was approved in June 1978 and formed the basis for engineering specifications. A critiquing process evolved as the organizing principle behind initial product design; engineers on the project would exchange documentation and then meet to evaluate the progress of the system.

The sessions were lively and the problems of coordinating implementation, testing and design along with the pressure of deadlines for the whole program generated quite a bit of excitement.

Development testing turned out to be a particularly thorny problem—the asynchronous interrupts and multiple-

processing aspects of real-time applications required a special test apparatus to simulate a real-world environment.

What they came up with is a nucleus executing directly on the 8086 and 8088 processors as the basic building block of the system. Together with the next layer—a basic I/O system—a minimal operating system can be configured, which has been found useful in many applications.

However, it was necessary to develop an application on the Series-III development system even though the target was going to be RMX. "We quickly realized that users want to be able to do development work on the machine they target on," said Schafer. "This is particularly important for field maintenance . . . you can't drag a Series-III out to an oil derrick." To realize this goal, Intel built higher layers around iRMX so that program development could be done without a Series-III. Higher layers involve extended I/O and human interface facilities. After this, customer-written software can be added in high-level languages.

A major objective has been to provide a stable base for independent software vendors; with its latest release, Intel also announced an ISV program initially involving three major vendors; Microsoft, Digital Research and Mark Williams Inc.

The first release of iRMX-86 came out in April 1980. Since then, the system has been refined and released four more times, with release five appearing last December. An Interactive Configuration Utility appeared for the first time with release five, a further attempt to aid OEMs in putting their systems together. The system designer runs the ICU program on a terminal and is quizzed on his requirements, after which the program generates the unique iRMX software for his application.

"It has been a successful product in its own right, apart from its role in the hardware program, but I doubt that anyone would have wanted to invest in a three-year development process before there was a chance at some return," observed Stevens, who has been most excited by the diverse applications he has seen. "I've really enjoyed the iRMX symposiums. There is always some new system demonstrated. In Tokyo, I just saw an 8086-based scientific system with really first-class graphics put together by Seiko. Another time I saw a blood analyzer based on the system. There are even RMX-based personal computers."

APRIL, 1984

**Industrial PC Systems
Demands Real-Time
Operating System**

Kathryn S. Norris
Intel Corporation

"The tasks
must be able
to coordinate
with one another."

PC ■ INTEGRATION SERIES

Industrial PC systems demands real-time operating system

Personal computers find a wide range of applications in an industrial process control environment. For example, the computers can be used for temperature monitoring and control, production line testing, wear analysis, frequency signature monitoring, analysis in noisy or hostile environments, and vibration analysis.

All of these jobs for the computer have certain characteristics in common. All require that the computer process asynchronously occurring events that are happening in real time. Moreover, handling multiple asynchronously occurring events, some of which can happen concurrently, demands that the computer process more than one operation or task at a time.

For example, in monitoring and controlling the temperature of a burn-in oven used to stress printed circuit boards, the computer must have a task that keeps track of time so that the temperature is read at prescribed intervals. It requires a second task to read the temperature sensor, and a third to control the application or removal of heat.

Furthermore, certain tasks are more time critical than others. For example, an error routine that detects a critical overtemperature condition in the burn-in oven must be given the highest priority and executed before any other routine vying for computer time.

Finally, the tasks must be able to coordinate with one another. They must be able to communicate so that the results of one needed by a second can be passed between the two. Tasks must also be able to exclude one another so that, for example, one can have use of commonly accessed data without interference from a second. Finally, tasks must be able to synchronize with one another to ensure, for example, control of a chemical reaction that requires an ordered sequence of elements be added to produce the desired result.

To meet the requirements of an industrial control environment, a personal computer must have an operating system like the iRMX-86 from Intel Corp. (Hillsboro, Ore.), which can meet the requirements just described. One recently announced industrial personal computer which offers iRMX-86 is the MSC 8807 Industrial PC from Monolithic Systems Corp. (Englewood, Colo.), see ("Industrial PC goes to work").

The operating system contains a nucleus which gives an applications program or task the means to monitor and control external events. Tasks running concurrently are called into execution by interrupts generated by the real-world process being controlled.

A scheduler inside the operating system decides whether an executing task should be interrupted to process data from a device generating an interrupt. In addition, three facilities inside the operating system provide for tasks to interact with one another. Each of the three is optimized for data transfer, synchronization or mutual exclusion.

Handling multiple tasks. The essence of real-time application systems is the ability to process numerous events occurring at random times. Any single program that attempts to process multiple, concurrent, asynchronous events is bound to be complex. It must process each event and remember which have already occurred and the order in which they happened. The complexity obviously grows greater as the system monitors more events.

Multitasking capability in an operating system unwinds this confusion. Rather than writing a single monolithic program to process N events, N programs are written, each of which processes a single event. Each of these N programs forms an iRMX 86 task. Multitasking eliminates the need to monitor the order in which events occur.

The operating system is an interrupt processor. When an interrupt occurs, it schedules a task to process the interrupt. This method of event detection improves the performance of an application system.

There are two ways that computer systems can schedule processing associated with detecting and

Kathryn S. Norris, Software Products Marketing Manager
Intel Corp
5200 N.E. Elam Young Pkwy
Hillsboro, Ore 97123

PC ■ INTEGRATION SERIES

"iRMX 86 uses preemptive, priority-based scheduling to decide which tasks runs at any instance."

controlling events in the real world: polling and interrupt processing. Polling has the major shortcoming of requiring a significant amount of the processor's time to test to see if events have occurred.

The second method of controlling processing is the interrupt. An event occurring generates an interrupt to the computer. Rather than executing the next sequential instruction, the processor begins to execute a task associated specifically with the detected event.

Interrupt processing allows a system to spend all of its time running the tasks that processes events, rather than executing a polling loop to see if events have occurred. Since there is a direct correlation between interrupts and tasks, a system can easily be modified to process different events. All that is needed is to write the tasks to process the new interrupts.

Because interrupt processing allows a system to respond to events by means of modularly coded tasks, system programs are more structured and easier to understand. Modular programs are less costly to develop and maintain, and modules can be developed more quickly than a monolithic program containing the equivalent of several independent modules.

Scheduling with priority. The iRMX 86 operating system uses preemptive, priority-based scheduling to decide which task runs at any instant. This technique ensures that if a more important task becomes ready while a less important task is running, the more important task begins execution immediately.

In multitasking systems, there are two common techniques for deciding which task is to be run at any given moment. One called time slicing, better known as the familiar round-robin approach, involves tasks running in rotation. Each task is allotted a fixed quantity of computer time in which to execute. If it does not complete in that time, it must relinquish the CPU and wait until its turn with the CPU comes up again. The technique is commonly employed in time-sharing systems.

The second technique, priority-based scheduling, uses assigned priorities to decide which task is to be run next. Within priority-based scheduling, there are two approaches. Non-preemptive scheduling

allows a task to run until it relinquishes the processor. Even if while running it causes a higher priority task to become ready for execution, the original task continues to run until it explicitly surrenders the processor.

The second approach to priority-based scheduling is preemptive. Using preemptive scheduling, the system always executes the highest priority task that is ready to run. In other words, if the executing task or an interrupt causes a higher priority task to become ready, the operating system switches the processor to the higher priority task.

Preemptive, priority-based scheduling goes hand-in-hand with interrupt processing. The priorities of tasks can be tied to the relative importance of the events that they process. Thus, the processing of more-important events preempts the processing of less-important ones.

Allowing tasks to interact. The iRMX 86 operating system provides simple techniques for tasks to coordinate with one another. These techniques allow programs in a multitasking system to mutually exclude, synchronize, and communicate with each other. The processing of several events may be related. For instance, the task processing event A may need to know how many times event B has occurred since event A last occurred. This processing requires coordination between programs.

Tasks exchange information for two purposes. One is to pass data from one program to another. Suppose that one task accumulates keystrokes from a terminal until a carriage return is encountered. The keyboard program then passes the entire line of text to another task, which is responsible for decoding commands.

The second reason for passing data is to draw attention to a specific object, a mailbox for example, in the application system. In effect, one task says to another, "I am talking about that object."

The iRMX 86 system facilitates intertask communications by supplying objects called mailboxes along with system calls to manipulate them. The system calls associated with mailboxes are CREATE MAILBOX, DELETE MAILBOX, SEND MESSAGE, and RECEIVE MESSAGE. Tasks use the first two commands to build and eradicate a particular mailbox. They use the remainder to communicate with each other.

"The priorities of tasks can be tied to the relative importance of the events that they process."

PC ■ INTEGRATION SERIES

If Task A wants task B to become aware of a particular object, it uses the SEND MESSAGE system call to send the object to the mailbox. Task B uses the RECEIVE MESSAGE system call to retrieve the object from the mailbox. Why don't tasks just send messages directly between each other rather than through mailboxes? Tasks are asynchronous; they execute in unpredictable order. Mailboxes allow tasks to communicate with each other even though tasks are asynchronous.

If the receiver uses the RECEIVE MESSAGE system call before the message has been sent, the receiver waits at the mailbox until a message arrives. Similarly, if the sender uses the SEND MESSAGE system call before the receiver is ready to receive, the message is held at the mailbox until a task requests a message from the mailbox.

Providing tasks exclusivity. Occasionally, when tasks are running concurrently, the following kind of situation arises. Task A is in the process of reading information from a memory segment. An interrupt occurs and task B, which has a higher priority preempts task A. Task B modifies the contents of the segment that task A was in the midst of reading.

Task B finishes processing its event and surrenders the processor and task A resumes reading the segment. However, task A might have information that is completely invalid. For instance, suppose the application is air traffic control. Task A is responsible for detecting potential collisions and task B is responsible for updating the plane location table with the new X- and Y-coordinates of each aircraft's location. Unless task A can obtain exclusive use of the plane location table, task B can make task A fail to spot a collision.

Here's how it could happen. Task A reads the X-coordinate of the plane's location and is preempted by task B. Task B updates the entry that task A was reading, changing both the X- and Y-coordinates of the plane's location. Task B finishes its function and surrenders the processor. Task A resumes execution and reads the new X- and Y-coordinate of the aircraft's location. As a direct result of task B changing the plane location table while task A was reading it, task A thinks the plane is at old X and new Y coordinates. This misinformation could easily lead to disaster.

This problem can be avoided by mutual exclusion. If task A can prevent task B from modifying the table until after A has finished using it, A can be assured of valid information. Somehow, task A must obtain exclusive use of the table. The iRMX 86 operating system provides a type of object that can be used to provide mutual exclusion in the form of the semaphore.

A semaphore is an integer counter that tasks can manipulate using four system calls: CREATE SEMAPHORE, DELETE SEMAPHORE, SEND UNITS and RECEIVE UNITS. The creation and deletion system calls are used to build and eradicate semaphores. The send and receive system calls can be used to achieve mutual exclusion.

Semaphores can only take on non-negative integer values. Tasks can modify a semaphore's value by using the SEND UNITS or RECEIVE UNITS system calls. When a task sends N units to a semaphore, the value of the counter is increased by N. When a task uses the RECEIVE UNITS system call to request M units from a semaphore, one of two things happens: If the semaphore's counter is greater than or equal to M, the operating system reduces the counter by M and continues to execute the task. Otherwise, the operating system begins running the task having the next highest priority, and the requesting task waits at the semaphore until the counter reaches M or greater.

To use a semaphore to achieve mutual exclusion, the task wanting exclusivity creates a semaphore with an initial value of one. Before any task uses the shared resource, it must receive one unit from the semaphore. Also, as soon as a task finishes using the resource, it must send one unit to the semaphore. This technique ensures that at any given moment, no more than one task can use the resource, and any other tasks that want to use it must await their turn at the semaphore.

Semaphores allow mutual exclusion; they don't enforce it. All tasks (there can be more than two) sharing the resource must receive one unit from the semaphore before using the resource. If one task fails to do this, mutual exclusion is not achieved. Also, each task must send a unit to the semaphore when the resource is no longer used. Failure to do this can permanently lock all tasks out of the resource.

PC ■ INTEGRATION SERIES

“Occasionally a task must know that a certain event has occurred before it starts running.”

Synchronizing tasks. Tasks are asynchronous. Nonetheless, occasionally a task must know that a certain event has occurred before it starts running. For instance, suppose that a particular application system requires that task A cannot run until after task B has been completed.

An application system can achieve synchroniza-

tion also by using semaphores. Before executing either task A or task B, a semaphore is created with an initial value of zero. Task A issues RECEIVE UNITS requesting one unit from the semaphore. Task A is forced to wait at the semaphore until task B sends a unit. This achieves the desired synchronization.

Industrial PC goes to work

One portable computer designed exclusively for the multitasking industrial and scientific environment is the MSC Model 8807 Industrial PC from Monolithic Systems Corp. (Englewood, Colo.), the first of a planned family of system and support products that emphasizes software flexibility by offering two operating systems, four major languages and a variety of utility programs.

Based on the Intel Multibus architecture, the industrial PC integrates a 16-bit 80186 CPU, up to 512 kbytes of parity RAM, 9-in. CRT screen, dual 3½-in. floppy disk drives, parallel and serial I/O ports, and ACSII interface. The total weighs approximately 35 lb and includes a 100-W power supply, cooling fan and built-in carrying handle. One of two operating systems available for the PC is Intel's iRMX-86 operating system.

RMX-86 is a multi-user, multitasking, real-time operating system, which provides such advanced features as hierarchical file structure with variable file granularity. It schedules tasks with true real-time preemptive priorities. It enables dynamic memory allocation among concurrent applications, device independent I/O and intertask communication via mailboxes and semaphores.

The PC is built around the company's MSC 8186 single-board computer, which, in turn, is based on the Intel 80186 microprocessor. The processor board contains 128 kbytes of dual-port, dynamic parity RAM, a dynamic RAM controller, up to 64 kbytes of EPROM, and programmable parallel and serial I/O ports. Twenty-bit

addressing, plus four bits for bank select, enables addressing up to 16 Mbytes of system memory. The serial I/O port is controlled by a programmable communications interface for operation in most synchronous or asynchronous data transmission formats. Parallel I/O is implemented with a basic 24 lines controlled by a programmable peripheral interface.

An on-board programmable interrupt controller allows the system to handle up to eight levels of interrupt priority under software control. The CPU operates at 8 MHz and has the enhanced 80186 instruction set. The processor board contains two ISBX bus connectors for piggyback expansion modules.

Product packaging also reflects the targeted industrial market. The enclosure is metal, rather than plastic, and the top of the unit is hinged for easy access to all internal parts. This permits a user to run an interface directly off the processor board in addition to external port connectors.

The chassis is designed with six board slots, of which three are intended for customer-specified modules. This is a tool for system integrators who specialize in factory automation, test systems, process control, R&D laboratories, and a multitude of other on-line applications where system portability is important.

The product can monitor units under test concurrent with statistical analysis or data processing applications. As a software development tool, the system can do large compiling jobs concurrent with code writing or editing.

Translators and Utilities for Program Development

3

TRANSLATORS AND UTILITIES FOR PROGRAM DEVELOPMENT

Intel offers an extensive selection of program development tools for its microprocessor (8080, 8086, 8088, 80186, 80286) and microcontroller (8048, 8051, 8096 etc.) families. These tools include translators and programming utilities such as linkers, relocators, and library managers. These program development tools are high quality, time tested tools for the professional. Based on a set of well-defined standards, they provide an integrated development environment. The result is an extremely flexible and productive program development environment.

A LANGUAGE FOR EVERY NEED

The iAPX-86 family has the most comprehensive set of translators available for a microprocessor. These include a macro assembler and compilers for PL/M, Pascal, FORTRAN, and C (see Table 1). The macro assembler produces the most optimum code. PL/M is the most popular 8086 language for systems programming and provides the best of both optimal code and high level language capabilities.

The main advantage of 'C' is portability across different target machines. Pascal and FORTRAN are used extensively for applications programming. To allow applications to be portable, Pascal and FORTRAN conform to ISO and ANSI77 standards respectively, with many useful extensions for microprocessor applications.

Intel's microcontroller family (8048, 8051, 8096 etc.) is similarly the best supported in the industry. PL/M-51 was the first high level language ever to be introduced for a microcontroller. The 8096 is similarly supported with PL/M-96. Every microcontroller in the family is supported with an assembler and linkage utilities.

USE A MIXTURE OF LANGUAGES FOR MAXIMUM FLEXIBILITY

Programs are typically decomposed into modules to exploit the many benefits of modular programming. Intel's integrated programming technology allows different modules of the same program to be programmed in a variety of languages. For instance, the most performance-sensitive system modules may be coded in assembler or in PL/M. The application modules, on the other hand, can be written in Pascal to speed up programming. The system and application modules can then be linked into one program using the linker. Hence, the various modules of a program can each be coded in the most suitable programming language.

UTILITIES ENHANCE PROGRAMMING PRODUCTIVITY

A set of utilities is provided to support modular and position independent programming. The linkers combine the constituent modules of a program into one system. A locator is provided to position the code in memory. This allows code to be placed in appropriate ROM and RAM locations. Also, coding can be done in a position-independent way. The librarian provides a structured way of organizing frequently used routines. The routines needed by a particular program can be linked in by the linker. The linker automatically selects only those modules from the library that are needed by the program. For the protected, virtual-memory, and multi-tasking processor iAPX 286, a sophisticated operating system configuration utility BUILD-286, is provided.

FULL RANGE OF DEBUG SUPPORT

The programming tools are integrated with the debugging tools via the well-defined Intel object module format standard. iAPX-86 family programs may be debugged using any of the Intel 8086 debug tools. This includes PSCOPE which provides source level software debug, and the ICE products which provide in-target real-time debug. Microcontroller software is similarly supported by the various emulators and ICE units.

CHOOSE FROM A VARIETY OF HOST CONFIGURATIONS

The programming tools are provided on a variety of development host environments to meet the needs of different project sizes and development budgets (see Table 1). The environments span personal development systems (IPDS), stand alone development systems, network development systems (NDS-II) and even the *VAX/VMS microcomputer. The programming tools work identically, no matter which of the available host configurations is chosen. This allows the user to grow his development environment, as his needs grow, without impacting previous investment in software.

* VAX/VMS is a trademark of Digital Equipment Corporation.

Table 1. Intel Translator/Host Summary

Language	Component Family	Host Code
Macro Assembler + Utilities	2920	1,2
	MCS-85 Family	1
	MCS-48 Family	1
	MCS-51 Family	1
	iACX-96 Family	2
	iAPX-86 Family	1,2,3
	iAPX-286 (Protected Mode)	2,3
PL/M	MCS-85 Family	1
	MCS-51 Family	1
	iACX-96 Family	2
	iAPX-86 Family	1,2,3
	iAPX-286 (Protected Mode)	2,3
PASCAL	MCS-85 Family	1
	iAPX-86 Family	2,3
	iAPX-286 (Protected Mode)	2
FORTRAN	MCS-85 Family	1
	iAPX-86 Family	2
	iAPX-286 (Protected Mode)	2*
"C"	iAPX-86 Family	2,3*
	iAPX-286 (Protected Mode)	2*,3*
Ada	iAPX-86 Family	3*
	iAPX-286 (Protected Mode)	3*

NOTE: * = Planned

HOST CODES

- 1 = 8085 Based Development System
- 2 = iAPX-86 family Based Development System
- 3 = VAX/VMS Minicomputer



PL/M 80 HIGH LEVEL PROGRAMMING LANGUAGE

- Provides Resident Operation on Intellec® Microcomputer Development System and Intellec® Series II Microcomputer Development Systems
- Produces Relocatable and Linkable Object Code
- Sophisticated Code Optimization Reduces Application Memory Requirements
- Speeds Project Completion with Increased Programmer Productivity
- Cuts Software Development and Maintenance Costs
- Improves Product Reliability with Simplified Language and Consequent Error Reduction
- Eases Enhancement as System Capabilities Expand

The PL/M 80 High Level Programming Language Intellec Resident Compiler is an advanced, high level programming language for Intel 8080 and 8085 microprocessors, iSBC-80 OEM computer systems, and Intellec microcomputer development systems. PL/M has been substantially enhanced since its introduction in 1973 and has become one of the most effective and powerful microprocessor systems implementation tools available. It is easy to learn, facilitates rapid program development and debugging, and significantly reduces maintenance costs. PL/M is an algorithmic language in which program statements naturally express the algorithm to be programmed, thus freeing programmers to concentrate on system development rather than assembly language details (such as register allocation, meanings of assembler mnemonics, etc.). The PL/M compiler efficiently converts free-form PL/M programs into equivalent 8080/8085 instructions. Substantially fewer PL/M statements are necessary for a given application than would be using assembly language or machine code. Since PL/M programs are problem oriented and thus more compact, programming in PL/M results in a high degree of productivity during development efforts, resulting in significant cost reduction in software development and maintenance for the user.



FUNCTIONAL DESCRIPTION

The PL/M compiler is an efficient multiphase compiler that accepts source programs, translates them into object code, and produces requested listings. After compilation, the object program may be first linked to other modules, then located to a specific area of memory, and finally executed. The diagram shown in Figure 1 illustrates a program development cycle where the program consists of three modules: PL/M, FORTRAN, and assembly language. A typical PL/M compiler procedure is shown in Table 1.

Features

Major features of the Intel PL/M 80 compiler and programming language include:

Resident Operation — on Intel microcomputer development systems eliminates the need for a large in-house computer or costly timesharing system.

Object Code Generation — of relocatable and linkable object codes permits PL/M program development and debugging in small modules, which may be easily linked with other modules and/or library routines to form a complete application.

Extensive Code Optimization — including compile time arithmetic, constant subscript resolution, and common subexpression elimination, results in generation of short, efficient CPU instruction sequences.

Symbolic Debugging — fully supported in the PL/M compiler and ICE-85 in-circuit emulators.

Compile Time Options — includes general listing format commands, symbol table listing, cross reference listing, and "innerlist" of generated assembly language instructions.

Block Structure — aids in utilization of structured programming techniques.

Access — provided by high level PL/M statements to hardware resources (Interrupt systems, absolute addresses, CPU input/output ports).

Data Definition — enables complex data structures to be defined at a high level.

Re-entrant Procedures — may be specified as a user option.

Benefits

PL/M is designed to be an efficient, cost-effective solution to the special requirements of microcomputer software development as illustrated by the following benefits of PL/M use:

Low Learning Effort — even for the novice programmer, because PL/M is easy to learn.

Earlier Project Completion — on critical projects, because PL/M substantially increases programmer productivity while reducing program development time.

Lower Development Cost — because increased programmer productivity requiring less programming resources for a given function translates into lower software development costs.

Increased Reliability — because of PL/M's use of simple statements in the program algorithm, which are easier to correct and thus substantially reduce the risk of costly errors in systems that have already reached full production status.

Easier Enhancement and Maintenance — because programs written in PL/M are easier to read and easier to understand than assembly language, and thus are easier to enhance and maintain as system capabilities expand and future products are developed.

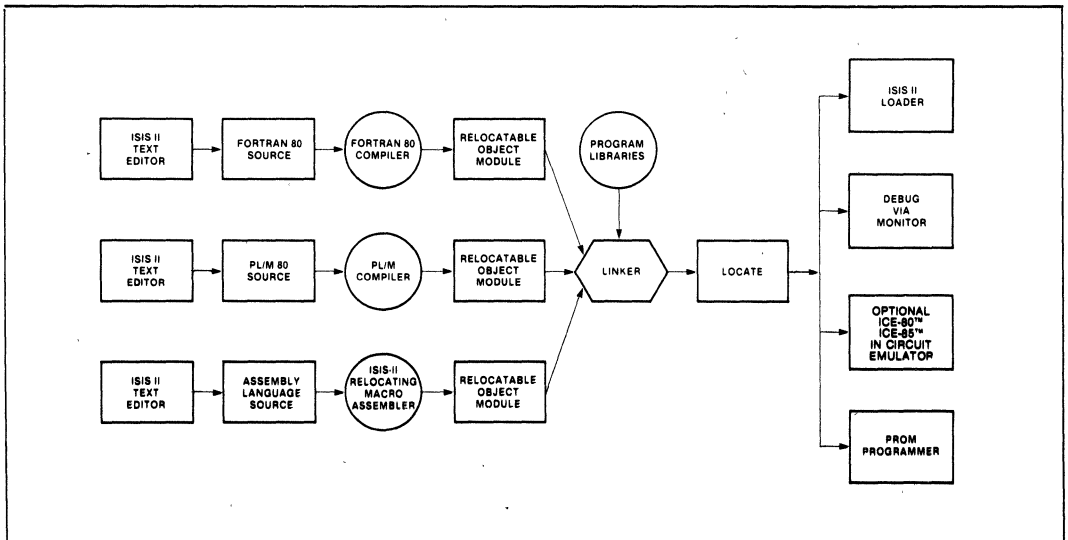


Figure 1. Program Development Cycle Block Diagram

Simpler Project Development — because the Intel microcomputer development system with resident PL/M 80 is all that is needed for developing and debug-

ing software for 8080 and 8085 microcomputers, and the use of expensive (and remote) timesharing or large computers is consequently not required.

Table 1. PL/M-80 Compiler Sample Factorial Generator Procedure

		\$OBJECT(:F1:FACT.OB2)
		\$DEBUG
		\$XREF
		\$TITLE('FACTORIAL GENERATOR — PROCEDURE')
		\$PAGEWIDTH(80)
1		FACT:
		DO;
2	1	DECLARE NUMCH BYTE PUBLIC;
3	1	FACTORIAL. PROCEDURE (NUM,PTR) PUBLIC;
4	2	DECLARE NUM BYTE, PTR ADDRESS;
5	2	DECLARE DIGITS BASED PTR (161) BYTE;
6	2	DECLARE (I,C,M) BYTE;
7	2	NUMCH = 1; DIGITS(1) = 1;
9	2	DO M = 1 TO NUM;
10	3	C = 0;
11	3	DO I = 1 TO NUMCH;
12	4	DIGITS(I) = DIGITS(I)*M + C;
13	4	C = DIGITS(I)/10;
14	4	DIGITS(I) = DIGITS(I) — 10*C;
15	4	END,
16	3	IF C<>0 THEN
17	3	DO;
18	4	NUMCH = NUMCH + 1; DIGITS(NUMCH) = C;
20	4	C = DIGITS(NUMCH)/10;
21	4	DIGITS(NUMCH) = DIGITS(NUMCH) — 10*C;
22	4	END
		END;
24	2	END FACTORIAL;
25	1	END;

SPECIFICATIONS

OPERATING ENVIRONMENT

Intel Microcomputer Development Systems
(Series II, Series III, Series IV)
Intel Personal Development System

DOCUMENTATION

PL/M 80 Programming Manual
ISIS-II PL/M 80 Compiler Operator's Manual

ORDERING INFORMATION

Product Code	Description
MDS PLM	PL/M 80 High Level Language Compiler. Needs Software License

SUPPORT:

Hotline Telephone Support, Software Performance Report (SPR), Software Updates, Technical Reports, and Monthly Technical Newsletters are available

MDS is an ordering code only and is not used as a product or trademark. MDS is a registered trademark of Mohawk Data Sciences Corporation



FORTRAN 80 8080/8085 ANS FORTRAN 77 INTELLEC® RESIDENT COMPILER

- Meets ANS FORTRAN 77 Subset Language Specification plus adds Intel microprocessor extensions
- Supports Intel Floating Point Standard with the FORTRAN 80 software routines, the ISBC-310™ High Speed Mathematics Board, or the ISBC-332™ math multimodule
- Executes on Intellec Microcomputer Development System, Intellec Series II Microcomputer Development System, and Personal Development System
- Supports full symbolic debugging with ICE-80™ and ICE-85™
- Produces relocatable and linkable object code compatible with resident PL/M 80 and 8080/8085 Macro Assembler
- Provides optional run-time library to execute in RMX-80™ environment
- Has well defined I/O interface for configuration with user-supplied drivers

FORTRAN 80 is a computer industry-standard, high-level programming language and compiler that translates FORTRAN statements into relocatable object modules. When the object modules are linked together and located into absolute program modules, they are suitable for execution on Intel 8080/8085 Microprocessors, ISBC-80 OEM Computer Systems, Intellec Microcomputer Development Systems and Personal Development Systems. FORTRAN 80 meets the ANS FORTRAN 77 Language Subset Specification¹. In addition, extensions designed specifically for microprocessor applications are included. The compiler operates on the Intellec Microcomputer Development System and Personal Development System under the ISIS-II Disk Operating Systems and produces efficient relocatable object modules that are compatible for linkage with PL/M 80 and 8080/8085 Macro Assembler modules.

The ANS FORTRAN 77 language specification offers many powerful extensions to the FORTRAN language that are especially well suited to Intel 8080/8085 Microprocessor software development. Because FORTRAN 80 conforms to the ANS FORTRAN 77 standard, the user is assured of compatibility with existing FORTRAN software that meets the standard as well as a guarantee of upward compatibility to other computer systems supporting an ANS FORTRAN 77 Compiler.

¹ANSI X3J3/90



FORTRAN 80 LANGUAGE FEATURES

Major ANS FORTRAN 77 features supported by the Intel FORTRAN 80 Programming Language include:

- Structured Programming is supported with the IF ... THEN ... ELSE IF ... ELSE ... END IF constructs.
- CHARACTER data type permits alphanumeric data to be handled as strings rather than characters stored in array elements.
- Full I/O capabilities include:
 - Sequential and Direct Access files
 - Error handling facilities
 - Formatted, Free-formatted, and Unformatted data representation
 - Internal (in-memory) file units provide capability to format and reformat data in internal memory buffers
 - List Directed Formatting
- Supports arrays of up to seven dimensions.
- Supports logical operators
 - .EQV. — Logical equivalence
 - .NEQV. — Logical nonequivalence

Major extensions to FORTRAN 77 in Intel FORTRAN-80 include:

- Direct 8080/8085 port I/O supported by intrinsic subroutines.
- Binary and Hexadecimal integer constants.
- Well defined interface to FORTRAN-80 I/O statements (READ, OPEN, etc.), allowing easy use of user-supplied I/O drivers.
- User-defined INTEGER storage lengths of 1, 2 or 4 bytes.
- User-defined LOGICAL storage lengths of 1, 2 or 4 bytes.
- REAL STORAGE lengths of 4 bytes.
- Bitwise Boolean operations using logical operators on integer values.
- Hollerith data constants.
- Implicit extension of the length of an integer or logical expression to the length of the left-hand side in an assignment statement.
- A format descriptor to suppress carriage return on a terminal output device at the end of the record.

FORTRAN 80 COMPILER FEATURES

- Supports multiple compilation units in single source file.
- Optional Assembly Language code listing.
- Comprehensive cross-reference, symbol attribute and error listing.
- Compiler controls and directives are compatible with other Intel language translators.
- Optional Reentrancy.
- User-defined default storage lengths.
- Optional FORTRAN 66 Do Loop semantics.
- Source files may be prepared in free format.

- The INCLUDE control permits specified source files to be combined into a compilation unit at compile time.
- Transparent interface for software and hardware floating point support, allowing either to be chosen at time of linking.

FORTRAN 80 BENEFITS

FORTRAN 80 provides a means of developing application software for Intel MCS-80/85 products in a familiar, widely accepted, and computer industry-standardized programming language. FORTRAN 80 will greatly enhance the user's ability to provide cost-effective solutions to software development for Intel microprocessors as illustrated by the following:

- *Completely Complementary to Existing Intel Software Design Tools* — Object modules are linkable with new or existing Assembly Language and PL/M Modules.
- *Incremental Runtime Library Support* — Runtime overhead is limited only to facilities required by the program.
- *Low Learning Effort* — FORTRAN 80, like PL/M, is easy to learn and use. Existing FORTRAN software can be ported to FORTRAN 80, and programs developed in FORTRAN 80 can be run on any other computer with ANS FORTRAN 77.
- *Earlier Project Completion* — Critical projects are completed earlier than otherwise possible because FORTRAN 80 will substantially increase programmer productivity, and is complementary to PL/M Modules by providing comprehensive arithmetic, I/O formatting, and data management support in the language.
- *Lower Development Cost* — Increases in programmer productivity translates into lower software development costs because less programming resources are required for a given function.
- *Increased Reliability* — The nature of high-level languages, including FORTRAN 80, is that they lend themselves to simple statements of the program algorithm. This substantially reduces the risk of costly errors in systems that have already reached production status.
- *Easier Enhancements and Maintenance* — Like PL/M, program modules written in FORTRAN 80 are easier to read and understand than assembly language. This means it is easier to enhance and maintain FORTRAN 80 programs as system capabilities expand and future products are developed.
- *Comprehensive, Yet Simple Project Development* — The Intel Microcomputer Development System and Personal Development System, with the 8080/8085 Macro Assembler, PL/M 80 and FORTRAN 80 are the most comprehensive software design facilities available for the Intel MCS-80/85 Microprocessor family. This reduces development time and cost because expensive (and remote) timesharing or large computers are not required.

SAMPLE FORTRAN-80 SOURCE PROGRAM LISTING

```

*   ** THIS PROGRAM IS AN EXAMPLE OF ISIS-II FORTRAN-80 THAT
*   ** CONVERTS TEMPERATURE BETWEEN CELSIUS AND FARENHEIT

PROGRAM CONVRT

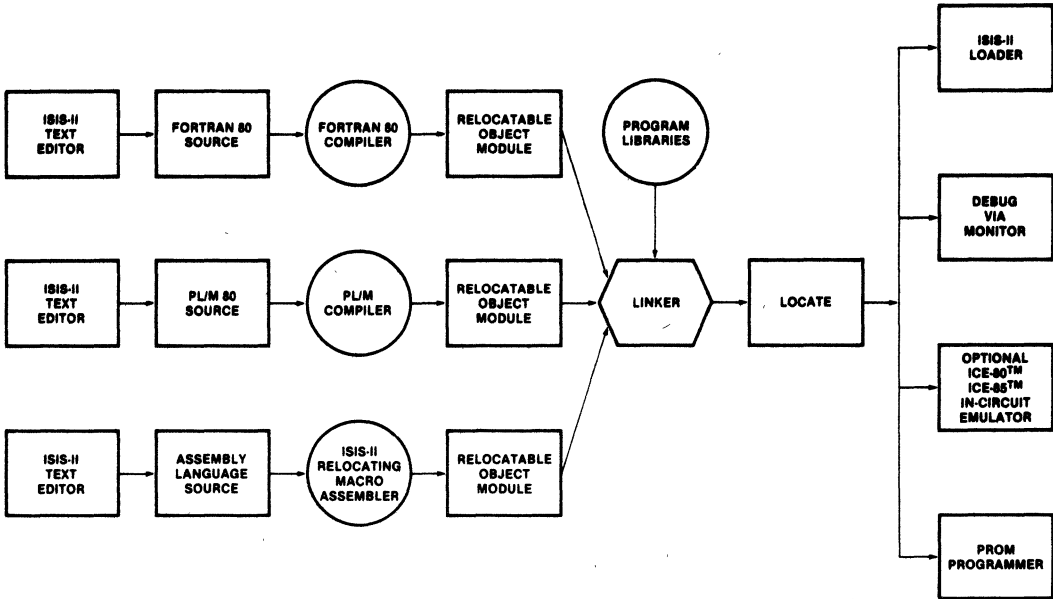
CHARACTER*1 CHOICE, SCALE

PRINT 100
*   ** ENTER CONVERSION SCALE (C OR F)
10  PRINT 200
    READ (5,300) SCALE

    IF (SCALE .EQ. 'C')
+     THEN
        PRINT 400
        ** ENTER THE NUMBER OF DEGREES FARENHEIT
        READ (5,*) DEGF
        DEGC = 5./9.*(DEGF-32)
        ** PRINT THE ANSWER
        WRITE (6,500) DEGF,DEGC
        ** RUN AGAIN?
20  PRINT 600
        READ (5,300) CHOICE
        IF (CHOICE .EQ. 'Y')
+         THEN
            GOTO 10
        ELSE IF (CHOICE .EQ. 'N')
+         THEN
            CALL EXIT
        ELSE
            GOTO 20
        END IF
    ELSE IF (SCALE .EQ. 'F')
+     THEN
        ** CONVERT FROM FARENHEIT TO CELSIUS
        PRINT 700
        READ (5,*) DEGC
        DEGF = 9./5.*DEGC+32.
        ** PRINT THE ANSWER
        WRITE (6,800) DEGC,DEGF
        GOTO 20
    ELSE
+     ** NOT A VALID ENTRY FOR THE SCALE
        WRITE (6,900) SCALE
        GOTO 10
    END IF
100  FORMAT(' TEMPERATURE CONVERSION PROGRAM',//,
+ ' TYPE C FOR FARENHEIT TO CELSIUS OR',/,
+ ' TYPE F FOR CELSIUS TO FARENHEIT',//)
200  FORMAT(/, ' CONVERSION? ', $)
300  FORMAT(A1)
400  FORMAT(/, ' ENTER DEGREES FARENHEIT: ', $)
500  FORMAT(/, F7.2, ' DEGREES FARENHEIT = ', F7.2, ' DEGREES CELSIUS')
600  FORMAT(/, ' AGAIN (Y OR N)? ', $)
700  FORMAT(/, ' ENTER DEGREES CELSIUS: ', $)
800  FORMAT(/, F7.2, ' DEGREES CELSIUS = ', F7.2, ' DEGREES FARENHEIT', /)
900  FORMAT(/, 1H 'A1, ' NOT A VALID CHOICE - TRY AGAIN!', /)
END

```

The FORTRAN 80 Compiler is an efficient, multiphase compiler that accepts source programs, translates them into relocatable object code, and produces requested listings. After compilation, the object program may be linked to other modules, located to a specific area of memory, then executed. The diagram shown below illustrates a program development cycle where the program consists of modules created by FORTRAN 80, PL/M 80 and the 8080/8085 Macro Assembler.



SPECIFICATIONS

OPERATING ENVIRONMENT

Required Hardware:

- 1 Intel Microcomputer Development Systems
—MDS-800 and Series II
or
2. Personal Development System

DOCUMENTATION PACKAGE

- FORTRAN-80 Programming Manual
- ISIS-II FORTRAN-80 Compiler Operator's Manual
- FORTRAN-80 Programming Reference Card

ORDERING INFORMATION

PART NO.	DESCRIPTION
Model MDS-301	FORTRAN 80 Compiler for Inteltec Microcomputer Development Systems

Requires Software License.

SUPPORT

Intel offers several levels of support for this product which are explained in detail in the price list. Please consult the price list for a description of the support options available.

*MDS is an ordering code only and is not used as a product name or trademark. MDS is a registered trademark of Mohawk Data Sciences Corporation.



MICROSOFT*, INC. BASIC-80 INTERPRETER SOFTWARE PACKAGE

- Compatible with other Microsoft BASIC compilers and interpreters
- Sophisticated string handling and structured programming features for applications development
- Direct transfer of BASIC programs to the 8085, 8086 and 8088
- Random and sequential file manipulation where random file record length is user-definable
- Read or write memory location capabilities
- Meets the requirements for the ANSI subset standard for BASIC, and supports many enhancements
- Extensive text editing features built-in
- Automatic line number generation and renumbering
- Supports assembly language subroutine calls
- Trace facilities for easier debugging

BASIC Release 5.0 from Microsoft is an extensive implementation of BASIC. Microsoft BASIC gives users what they want from a BASIC—ease of use plus the features that are comparable to a minicomputer or large mainframe.

BASIC-80 meets the requirements for the ANSI subset standard for BASIC, as set forth in document BSRX3.60-1978. It supports many unique features rarely found in other BASICs.

FEATURES

- Four variable types: Integer (–32768, +32767), String (up to 255 characters), Single-Precision Floating Point (7 digits), Double-Precision Floating Point (16 digits).
- Formatted output using the PRINT USING facility, including asterisk fill, floating dollar sign, scientific notation, trailing sign, and comma insertion.
- Trace facilities (TRON/TROFF) for easier debugging.
- Direct access to I/O ports with the INP and OUT functions.
- Error trapping using the ON ERROR GOTO statement.
- Extensive program editing facilities via EDIT command and EDIT mode subcommands.
- PEEK and POKE statements to read or write any memory location.
- Assembly language subroutine calls (up to 10 per program) are supported.
- Automatic line number generation and renumbering, including reference line numbers.
- IF/THEN/ELSE and nested IF/THEN/ELSE constructs.
- Matrices with up to 255 dimensions.
- Supports variable-length random and sequential disk files with a complete set of file manipulation statements: OPEN, CLOSE, GET, PUT, KILL, NAME, MERGE.
- Boolean operators OR, AND, NOT, XOR, EQV, IMP.



BASIC-80 Commands, Statements, Functions

AUTO	RENUM	NAME
LIST	WIDTH	SAVE
NULL	CONT	EDIT
TROFF	MERGE	NEW
CLEAR	RUN	TRON
LOAD	DELETE	

Program Statements

CALL	RANDOMIZE	RETURN
GOSUB	COMMON	WAIT
END	DEF FN	ON GOSUB
GOTO	ERROR	DIM
STOP	POKE	FOR/NEXT/ STEP
WHILE/ WEND	RESUME SWAP	IF/THEN/ ELSE
CHAIN	DEFDBL	ON ERROR
DEFUSR	DEFSTR	GOTO
LET	DEFSGN	OPTION BASE
REM	DEFINT	

Input/Output Statements and Functions

CLOSE	GET	NAME
KILL	POS	PUT
OUT	FIELD	EOF
RESTORE	LSET/RSET	SPC
READ	PRINT	INKEY\$
TAB	USING	INPUT
DATA	LOC	OPEN
LINE	MKI\$	CVD
INPUT	MKS\$	CVI
PRINT	MKD\$	CVS
WRITE	LLIST	
LPRINT	LPOS	

Arithmetic Functions

ABS	SIN	LOG
INT	CDBL	FIX
SGN	CSNG	COS
ATN	CINT	RND
EXP	SQR	TAN

String Functions

ASC	STR\$	INSTR
LEN	HEX\$	RIGHT\$
STRING\$	OCT\$	MID\$
CHR\$	VAL	SPACE\$
LEFT\$		

Operators

=	*	XOR
^	<=	NOT
<	+	EQV
>	< >	MOD
-	\	IMP
/	>=	OR
		AND

Special Functions

ERL	ERR	VARPTR
USR	FRE	PEEK

SPECIFICATIONS

Operating Environment

The standard disk version of Microsoft BASIC-80 occupies 24K bytes of memory. Microsoft BASIC-80 Interpreter is compatible with Intel's ISIS operating system or CP/M* operating system.

Required Hardware

- Intellex Microcomputer Development System
- IPDS (Personal Development System)
- minimum of 1 diskette drive

Required Software

ISIS Operating System or CP/M Operating System.

Documentation Package

One copy of each manual is supplied with the software package.

Description
BASIC-80 Reference Manual
BASIC Reference Book



ORDERING INFORMATION

Order Code	Description
SD102CPM80F	Microsoft BASIC-80 Interpreter Software Package, CP/M version (Double-Sided, Double Density 5¼" Floppy) iPDS format
SD102ISS80F	Microsoft BASIC-80 Interpreter Software Package, ISIS version (Double-Sided, Double Density 5¼" Floppy) iPDS format

SUPPORT

Intel offers several levels of support for this product, depending on the system configuration in which it is used. Please consult the price list for a detailed description of the support options available.

An Intel Software License required.

*Microsoft is a trademark of Microsoft, Inc.

*CP/M is a registered trademark of Digital Research, Inc.

*MP/M-II is a trademark of Digital Research, Inc.



MICROSOFT*, INC. BASIC-80 COMPILER SOFTWARE PACKAGE

- Produces highly optimized, true machine code
- Compiled programs are fast and compact because of extensive optimizations performed during compilation
- Supports all the commercial language features of the Microsoft BASIC interpreter (except direct mode commands)
- Supports double-precision transcendental functions
- Machine code for application programs may be placed on diskette, ROM, or other Media
- Provides source program security because only compiled code need be distributed to end-users
- Loader format identical to Microsoft's MACRO-80 assembler, COBOL-80 compiler, and FORTRAN-80 compiler: Compiled BASIC programs can be loaded and linked with any of these languages

Microsoft's BASIC-80 compiler is a powerful tool for programming BASIC applications or microprocessor system software. The single-pass compiler produces extremely efficient, optimized 8080 machine code that is in Microsoft-standard, relocatable binary format. Execution speed is typically 3-10 times faster than Microsoft's BASIC-80 interpreter.

FEATURES

Optimized, Compatible Object Code

The BASIC compiler produces object code that is highly optimized for speed and space, relocatable, and compatible with other Microsoft software products. The loader format is identical to that of the MACRO-80 assembler, COBOL-80 compiler and FORTRAN-80 compiler, so programs written in any one of these four languages can be loaded and linked together. The compiler can also provide a formatted listing of the machine code that is generated.

Compiled programs are fast and compact due to extensive optimizations performed during compilation:

- Expressions are reordered to minimize temporary storage and (wherever possible) to transform floating point division into multiplication.
- Constant multiplications are distributed to allow more complete constant folding.
- Constants are folded wherever possible. The expression reordering finds "hidden" constant operations.
- Peephole optimizations are performed, including strength reduction.

- The code generator is template-driven, allowing optimal sequences to be generated for the most commonly used operations.
- String operations and garbage collection are extremely fast.

Compiled BASIC-80 programs are the ideal end product for BASIC applications' programmers. The machine code for any application program may be placed on a diskette, ROM, or other media. The program not only runs faster than with the interpreter, but the BASIC source program need not be distributed. Thus the original application program is protected from unauthorized alteration.

Language Features

The Microsoft BASIC-80 Compiler supports all the commercial language features of Microsoft BASIC-80, except those commands that are not usable in the compiler environment (i.e., direct mode commands such as LOAD, AUTO, SAVE, EDIT, etc.). That means you get the BASIC language compatible with other Microsoft BASIC packages.



In addition, the compiler supports double-precision transcendental functions (SIN, COS, TAN, ATN, LOG, EXP, SQR), %INCLUDE, CHAIN and COMMON. The %INCLUDE compiler directive brings another source file into the compilation without retyping the main source file.

BRUN Runtime Module

The BRUN runtime module contains the most common runtime routines needed for most programs. Using the BRUN module provides faster link loading of program modules and allows the user to link much

larger programs because the runtime routine library does not reside in memory during linking. The executable files saved on disk are also much smaller since the BRUN module exists separately.

Utility Software Package

The BASIC-80 package includes the Microsoft Utility Software Package. The Utility Software Package includes the MACRO-80 macro assembler, the LINK-80 linking loader and the CREF-80 Cross-Reference Facility. Refer to the description of the Microsoft Utility Software Package for full details.

SPECIFICATIONS

Operating Environment

The BASIC Compiler requires a minimum of 34K bytes of memory (exclusive of the operating system). Microsoft recommends that 48K bytes be available for compiling medium to large programs. The compiler itself occupies about 28K bytes. At runtime, the BRUN module occupies approximately 15.5K bytes. If, as an option, the BRUN module is not used, the runtime library occupies 8K-18K bytes.

Required Hardware

- Intellec Microcomputer Development System
- iPDS (Personal Development System)
- minimum of 1 diskette drive

Required Software

- CP/M* Operating System

Documentation Package

One copy of each manual is supplied with the software package.

Description

- BASIC Compiler User's Manual
- BASIC-80 Reference Manual
- BASIC Reference Book
- Microsoft Utility Software Manual

(Specify by Alpha Character when ordering.)

ORDERING INFORMATION

Order Code

SD124CPM80F

Description

Microsoft BASIC-80 Compiler Software Package, CP/M version (iPDS Format)

SUPPORT:

Intel offers several levels of support for this product, depending on the system configuration in which it is used. Please consult the price list for a detailed description of the support options available.

An Intel Software License required
*Microsoft is a trademark of Microsoft, Inc

*CP/M is a registered trademark of Digital Research, Inc
*MP/M-II is a trademark of Digital Research, Inc

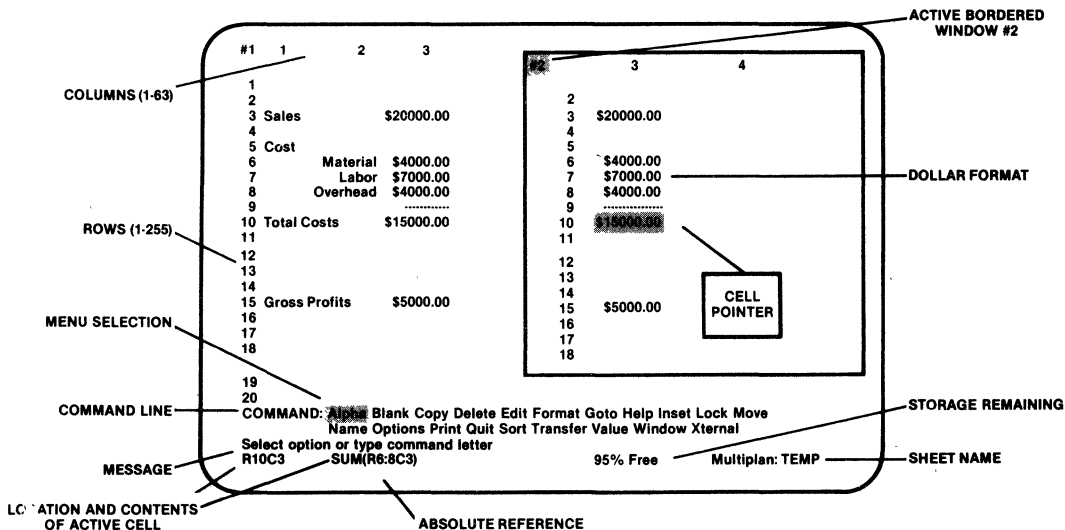


MICROSOFT* MULTIPLAN* SPREADSHEET

- Simplifies the design and use of very large spreadsheets, and multiple inter-related spreadsheets
- Automatically updates subtotals, totals, percentages, growth curves, etc
- Can perform multiple iterations to solve closed-loop problems
- Formulas automatically revised when reordering rows and columns in displays
- Can be used in time, monetary, and inventory budgeting
- Wide array of sophisticated functions to simplify formulas
- Cells and areas can be named for clarity
- Can reference and update several inter-related spreadsheets at once
- Simple to use, intuitive commands. Single keystroke command entry
- "Windows" allow several portions of large sheets to be viewed at once
- Contains the features of the most popular spreadsheet programs, as well as its contribution of new features

Multiplan is a productivity tool designed to help the user to analyze data in spreadsheet format. As an aid to both business and personal needs, Multiplan is an extremely powerful modeling and planning tool.

Multiplan is easy to learn and use, yet its versatility is enhanced by the skill of the user. Multiplan allows the user to operate in as intuitive a way as possible, and its widespread capabilities allow accomplishment of a variety of tasks. Advanced users are unencumbered by simplifying features, and have enough power to satisfy their needs.



Typical Multiplan Screen Display

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products: BXP, CREDIT, i, iCE, iCS, Im, Insite, Inte', INTEL, Inteleview, Intelink, Intellec, iMMX, iOSP, iPDS, iRMX, iSBC, iSBX, Library Manager, MCS, MULTIMODULE, Megachassis, Micromainframe, MULTIBUS, Multichannel, Plug-A-Bubble, PROMPT, Promware, RUPi, RMX/80, System 2000, UPI, and the combination of iCS, iRMX, iSBC, iSBX, ICE, i2ICE, MCS, or UPI and numerical suffix. Intel Corporation Assumes No Responsibility for the use of Any Circuitry Other Than Circuitry Embodied in an Intel product. No Other Patent Licenses are implied. © INTEL CORPORATION, 1983

*Microsoft & Multiplan are trademarks of Microsoft Corp

FEATURES

- Names can be used to express "cells" (worksheet elements), or groups of cells. These names, in turn, can be used as parts of formulas and commands. Named areas can be combined in various ways for ease of use.
- A wide range of functions unique to Multiplan is available in addition to the functions typical to the most popular spreadsheet programs. These functions allow the user to select windows, sort data, draw from other worksheets, and a number of other important operations.
- Expressions can be clarified by the use of names as in "PROFIT = SALES - COSTS" rather than "R12C1 = R1C3 - R5C12".
- Active sheets can draw data automatically from inactive "supporting" sheets through the use of named cells and areas. This unique feature allows the user to streamline the processing of data, and to generate an entire pad full of interrelated spreadsheets.
- Multiplan offers a worksheet size of up to 255 rows by 63 columns, a broad worksheet simulator in which words, numbers, and formulas may be entered into information cells. Added to the access of data in inactive sheets, this large sheet size allows the user to perform very rigorous analyses in a minimum amount of time.
- With Multiplan the user gains the capability to plan against several different situations to allow comparison of one set of circumstances against another. A good example of this would be the generation of several sheets, one based on steady growth versus others based on several potential problems. This way, contingency planning will become less tedious and more effective.
- By altering a single critical number, the impact on other dependent numbers will be automatically updated to help the user observe sensitivities and interdependencies. This helps the user to plan resources efficiently, and schedule more effectively.
- Multiplan overcomes the limitations of paper worksheets by allowing the user to instantly move, insert, or delete entire rows or columns of data. The remaining rows, columns, or free space will expand or contract automatically as necessary, thereby eliminating the costly and tiresome work of typing or hand-printing the worksheet over and over.
- All commands can be invoked by a single keystroke and selections are menu driven. Multiplan even offers proposed responses to commands, to encourage its use by even the most unskilled user. Multiplan's commands, prompts, and messages, as well as the screen and keyboard, communicate with each other and the user directly and naturally to allow the untrained user to accomplish objectives easily.
- A special edit area helps the user to make additions and deletions quickly and easily.
- Up to eight windows are available to allow users to view different parts of a very large worksheet simultaneously. The windows can be aligned, scrolled together, opened, or closed at will.
- An iteration option allows the simulation of closed-loop problems involving mutually interdependent formulas. The number of iterations can be chosen, or iterations can continue until a given constraint is met.
- Formulas can be moved from one worksheet location to another without having to be rewritten by the user.
- Reference to a particular cell need not be in absolute terms, but can be expressed as a location relative to other cells. A formula containing this sort of relative reference may be copied into other cells and will be automatically changed to reflect its new position.
- The sheet display may be redesigned or formatted in various ways without affecting the data stored in Multiplan. Thus, the same data can be presented in different order in different reports with a minimum of effort.

Commands

The following is a brief list of commands available under Multiplan. All of these commands are invoked by the single keystroke of their first letter (i.e. "C" for Copy or "F" for Format) with the exception of eXternal, which is invoked by typing an "X."

Several of the commands offer a number of selections of operational modes, which are displayed when the command is invoked. In order to choose a mode, either press the TAB key until the cursor rests over the selected mode, then hit RETURN, or type the first letter of the selected mode, then hit RETURN.

For more detailed descriptions of the commands, please see the Multiplan User's Manual.

ALPHA

Replaces the contents of the active cell with a character string. If the active cell already contains a string, that string is the proposed response of the command, so that it can be edited.

BLANK

Deletes contents of all specified cells. Names are not affected; if a cell was referred to by a name before use of this command, that name will still apply.

COPY

Presents a choice of three ways of copying the contents of some cells into other cells. To duplicate one cell across several to its right, choose Right. To duplicate one cell across several below it, choose Down. To copy any cell or cells to any others, choose From.

DELETE

Presents a two-way choice to delete cells. To delete a row or rows, choose R. To delete a column or columns, choose C. To blank out the cells, use the Blank command.

EDIT

Makes contents of the active cell available for editing. Place the cell pointer on the cell to be edited and press E. The cell's contents are then

placed on the command line for modification. The edit cursor is placed at the end of the current contents rather than highlighting the whole command, as is done for other defaults. If the cell contains a string, it is presented in double quotes. After having edited the cell's contents, press RETURN to put the changed contents back in the cell (or press ABORT to cancel any changes).

FORMAT

Presents a choice of three kinds of format adjustment. To set a specific format for a cell or group of cells, choose Cells. To set the width of a column or columns, choose Width. To set the default format—the format that applies wherever a specific format hasn't been set—choose Default.

GOTO

Presents a choice of ways to move the cell pointer over the sheet. To display a specific row and column, choose Row-col. To display a named area, choose Name.

HELP

Provides helpful information about Multiplan. When help is requested, the spreadsheet is replaced by text from the HELP file and the HELP command menu appears on the screen. Help is available in the areas of Applications, Commands, Editing, Formulas, and the Keyboard. The spreadsheet display is reinstated when the RESUME subcommand is entered.

INSERT

Presents a choice of ways to insert new cells into the sheet. To insert new rows choose Row. To insert new columns choose Column.

LOCK

Provides two ways to lock cells in protection against accidental change. Either individual cells or all cells containing formulas can be moved, deleted, formatted or sorted after having been locked, but their contents cannot be changed.

MOVE

Presents a choice of ways to move cells around the sheet. To move whole rows, choose Row. To move whole columns, choose Column.

Table 1. Multiplan Commands

ALPHA	— Replaces cell contents with a character string.
BLANK	— Clears cell contents.
COPY DOWN	— Used to fill a column with identical values.
COPY FROM	— Duplicates one or a number of cells to another location.
COPY RIGHT	— Used to make a row of identical values.
DELETE COLUMN	— Removes columns from the spreadsheet.
DELETE ROW	— Removes rows from the spreadsheet.
EDIT	— Allows editing of the contents of a single cell.
FORMAT CELLS	— Used to help align cells in a column.
FORMAT WIDTH	— Limits the width of all cells in a given column.
FORMAT DEFAULT CELLS	— Sets formats for all previously unformatted cells.
FORMAT DEFAULT WIDTH	— Sets formats for all previously unformatted columns.
FORMAT OPTIONS COMMAS	— Displays numbers with commas separating every third digit.
FORMAT OPTIONS FORMULAS	— Displays formulas instead of their values.
GOTO ROW-COL	— Moves the cell pointer to the specified row and column.
GOTO NAME	— Moves the cell pointer to the named area.
GOTO WINDOW	— Places the specified cell within the given window.
HELP APPLICATIONS	— Illustrates solutions to a number of common problems.
HELP COMMANDS	— Lists and describes all commands.
HELP EDITING	— Describes Editing functions.
HELP FORMULAS	— Gives Formula construction rules.
HELP KEYBOARD	— Explains special functions of the keyboard.
HELP NEXT	— Gives the next screenful of HELP text.
HELP PREVIOUS	— Gives the previous screenful from HELP call.
HELP RESUME	— Returns to the spreadsheet from HELP call.
HELP START	— Begins the HELP tutorial.
INSERT COLUMN	— Used to add a column to an existing spreadsheet.
INSERT ROW	— Used to add a row to an existing spreadsheet.
LOCK CELLS	— Protects the indicated cell from alteration.
LOCK FORMULAS	— Locks out alteration of all cells containing formulas or text.
MOVE COLUMN	— Changes the order of the columns on the sheet.
MOVE ROW	— Changes the order of the rows on the sheet.
NAME	— Assigns a name to a cell or number of cells.
OPTIONS	— Allows the user to disallow recalculation upon every change of a cell value, to mute the audible alarm, or to enable the iteration option.
PRINT FILE	— Outputs the spreadsheet to a diskette file.
PRINT MARGINS	— Sets up the margins on the printed output.
PRINT OPTIONS	— Allows optional printing modes to be used.
PRINT PRINTER	— Prints the spreadsheet on the system's printer.
QUIT	— Ends the Multiplan session without saving the active sheet.
SORT	— Sorts a range of rows to put values in a specified column into ascending or descending numerical order.
TRANSFER CLEAR	— Clears the active sheet.
TRANSFER DELETE	— Deletes the specified file.
TRANSFER LOAD	— Loads a sheet from the disk file.
TRANSFER OPTIONS	— Modifies the context of the following transfer operation.
TRANSFER RENAME	— Renames the active sheet.
TRANSFER SAVE	— Saves the active sheet on diskette.
VALUE	— Enters a value or formula into the active cell.
WINDOW BORDER	— Changes the border of the specified window.
WINDOW CLOSE	— Removes a window from the screen.
WINDOW LINK	— Sets or breaks link for synchronized scrolling between windows.
WINDOW SPLIT HORIZONTAL	— Horizontally divides a window into two windows.
WINDOW SPLIT VERTICAL	— Vertically divides one window into two windows.
WINDOW SPLIT TITLES	— Divides one window into two or four which scroll together.
XTERNAL COPY	— Copies data from an inactive sheet to the active sheet.
XTERNAL LIST	— Displays the relationships between the active sheet and the other sheets.
XTERNAL USE	— Sets a substitute name for a supporting sheet.

Commands (Continued)

NAME

Assigns a name to a cell or area of cells. The name defined may then be used wherever a reference to that cell or area is needed in a command or formula.

OPTIONS

The Options command can be used to set and reset various options provided with Multiplan.

The Recalc option controls how often Multiplan performs formula calculations. If the option is on, Multiplan recalculates all formulas whenever a cell is changed. If the option is off, recalculation is done only when the Recalc control key is pressed or during Transfer Save.

The Recalc option has an effect on how quickly Multiplan finishes entering a new value in a cell. The length of time Multiplan takes to recalculate the sheet depends on how many cells are in use, and on the complexity of the formulas in them. When you want to make a number of entries on a busy sheet, turn the Recalc option off to get the quickest response. Turn it on again when you are interested in seeing the effect of each change.

The Mute option silences Multiplan's audible alarm.

The Iterate option gives the user a means of solving problems which involve circular or "closed loop" references. Whereas formulas which count on each other's results (i.e., $A = B + C$, $B = A + C$) are disallowed in other spreadsheet programs, Multiplan allows spreadsheets with such references to be reiterated upon in an orderly manner either until a maximum number of iterations has been reached, or until a cell has reached a predetermined value.

PRINT

Presents a choice of four actions related to printing the active sheet. To begin printing, choose Go. To put printable output in a disk file, choose File. To set the margins that will be used on the printed output, choose Margins. To fix the part of the worksheet to be printed, or to insert a control line at the top of the output, choose Options.

QUIT

Ends the Multiplan session without saving the ac-

tive sheet. Multiplan requests confirmation; if it is given, Multiplan terminates, returning control to the computer operating system. The active sheet is lost unless it has previously been saved.

SORT

Reorders the rows on the spreadsheet so that the data in a specified column appears in ascending or descending numerical order. The column to be sorted may contain numbers, text, or other values, and if such values are mixed, they are presented in ascending order numerically, alphabetically and by error value, after which any blank cells follow.

TRANSFER

Offers a choice of five commands, which affect an entire sheet.

To load a saved sheet, replacing the active sheet, choose Load.

To save the active sheet in a disk file, choose Save.

To give the active sheet a new name, choose Rename.

To clear the active sheet, deleting all its contents, and restoring all its default settings, choose Clear.

To delete the disk copy of the active sheet, choose Delete.

VALUE

This command is used to enter a formula or number into the active cell. VALUE may either be selected from the command menu or by typing a numerical value, a mathematical symbol, or a left parentheses.

WINDOW

Presents a choice of four things that can be done with windows.

To open a new window by splitting the active window horizontally or vertically, or to open a window used strictly for titles, choose Split.

To close a window by removing it from the screen, choose Close.

To synchronize scrolling of windows, choose Link.

To move a window to a particular part of the sheet, choose Home.

To add or remove a decorative border around a window, choose Border.

XTERNAL

Presents a choice of actions relating to the use of data from other sheets in the formulas of the active sheet.

To copy data, or blocks of data from an inactive spreadsheet to the active sheet, choose Copy.

To display the relationships between the active sheet and other sheets, showing which sheets support (provide values for) the active one and which sheets depend on (use values from) the active sheet, choose List.

To assign a substitute name for an inactive sheet, specify Use.

Table 2. Multiplan Functions

ABS	— Calculates the absolute value of an argument.
AND	— True if, and only if, all values are true; otherwise returns false.
ATAN	— Gives the arctangent of an argument.
AVERAGE	— Returns the average of all cells referenced by up to 5 arguments.
COLUMN	— Gives the current column number.
COS	— Calculates an argument's cosine.
COUNT	— Finds the number of cells fitting the referenced criteria.
DOLLAR	— Formats numbers as dollar amounts.
EXP	— This is the inverse natural logarithm of the argument.
FALSE	— Returns the logical false value.
FIXED	— Rounds the first argument to the precision specified by the second.
IF	— Returns value specified after "THEN" if argument is true, or the "ELSE" specified value if false.
INDEX	— Returns the value of the cell in a named area offset by an index value.
INT	— Truncates the argument's fractional part.
ISERROR	— Returns true if, and only if, the argument is an error value.
ISNA	— Returns true if, and only if, the argument is an #N/A value.
LEN	— Gives the number of characters in the argument's string.
LN	— Calculates the natural logarithm of its argument.
LOG10	— Returns the common logarithm of its argument.
LOOKUP	— Used to search for dependent variables in a lookup table.
MAX	— Finds the largest numeric value in an area of cells.
MID	— Produces the middle characters of a string.
MIN	— Finds the smallest numeric value in an area of cells.
MOD	— Gives the remainder of the integer division of the two arguments.
NA	— Returns the #N/A value.
NOT	— Gives the logical inverse of the argument.
NPV	— Calculates the net present value of a constant annuity.
OR	— True if, and only if, any of the arguments are true; otherwise returns a false.
PI	— Returns Pi (3.14159...).
REPT	— Forms a string consisting of a repeated substring.
ROUND	— Rounds the first argument to the precision specified by the second.
ROW	— Gives the current row number.
SIGN	— Performs the Signum function on the argument.
SIN	— Returns the sine of the argument.
SQRT	— Calculates the square root of the argument.
STDEV	— Calculates the standard deviation of the arguments.
SUM	— Adds the sum of all cells in a specified area.
TAN	— Calculates the tangent of the argument.
TRUE	— Returns the logical true value.
VALUE	— Used to extract numbers from strings.



BENEFITS

Unlike other spreadsheet programs, Multiplan allows the user to create and view as many as eight different windows within the screen display area. Complete control is allowed over each window, allowing windows without borders, and the freezing and scrolling of title columns and rows.

Multiplan allows formulas to describe the contents of any cell. Formulas are written in a method similar to standard programming languages, and are evaluated according to priority of functions, a unique feature among spreadsheet programs. Parentheses are allowed to clarify the order of calculation. Formulas can use a string of characters as a variable name, and variables may be either numerical data, or strings of characters which may be manipulated to concatenate words and phrases. These are all unusually powerful and intuitively easy-to-learn features many of which are unique to Multiplan.

Multiplan gives the user an unusual amount of flexibility in rearranging the format or layout of a spreadsheet with its three forms of addressing: absolute, relative, or symbolic (by name). Any of the three can be combined in any order to produce the exact results needed in any case.

One of the features that sets Multiplan apart from other spreadsheet programs is the ability to name all cells. The NAME command allows the naming of single cells, an area of cells of any shape, or even a list of unconnected areas of cells. That

name can then be used in functions, or even as a response in a command. NAME also allows the user to review all cell names in their proper position on the screen in order to reduce confusion.

Multiplan commands can be entered by single letters on the command lines, after which the program will fill in the rest of the command. This speeds the user through complex operations without leaving any doubt about their functions. Versatile commands handle not only single data cells, rows, or columns as do other spreadsheet programs, but these commands allow Multiplan to move multiple rows or columns, or insert, delete, or handle any rectangular area. All relative references are automatically adjusted to account for these changes.

Multiplan automatically updates all entries affected by a change in a single cell, without requiring the user to command it to do so. This feature allows the user to fiddle with numbers and test for sensitivities and trouble spots.

Another unique benefit of Multiplan is its ability to employ values from one sheet in the formulas of another. This "sheet linkage" can be used to construct a hierarchy of worksheets, with detailed worksheets feeding their totals to a summary worksheet. When a detail sheet is updated and saved on diskette, the dependent summary sheet will be automatically updated the next time it is loaded.

SPECIFICATIONS

Operating Environment

REQUIRED HARDWARE:

Multiplan requires a minimum system which contains at least:

- 64K bytes of RAM
- 8080/8085 CPU
- Console with absolute cursor positioning
- One Diskette drive

OPTIONAL HARDWARE:

- Line printer

REQUIRED SOFTWARE:

- CP/M* Operating System

Documentation Package

Multiplan users manual

Shipping Media

(Specify by Alphabetical Character when ordering)

- A - Single density IBM 3740/1 compatible 8" diskette
- B - Double density IBM 3740/1 compatible 8" diskette
- F - IPDS™ compatible 5-1/4" diskette

*CP/M is a registered trademark of Digital Research, Inc.



ORDERING INFORMATION

Order Code

SD109CPM80A
SD109CPM80B
SD109CPM80F

Shipping Media

A—Single-density 8" diskette
B—Double-density 8" diskette
F—iPDS Format 5¼" diskette

Product Description

Multiplan spreadsheet program for use under CP/M* on 8080/8085 based small computers.

SUPPORT

Intel offers several levels of support for this product, depending on the system configuration in which it is used. Please consult the price list for a detailed description of the support options available.



PASCAL 80 SOFTWARE PACKAGE

- Offers a Superset of Standard Pascal
- Provides Highly Structured Language with Powerful Data Type Definitions to Suit Applications
- Compiles Pascal Source Code into Intermediate Code to Optimize Execution Speed and Storage
- Executes Compiler and Interprets the Intermediate Code on Intellec® Microcomputer Development Systems
- Provides a Utility to Produce Relocatable Object Modules Compatible with Other Intel® Languages
- Can Call Routines Written in PL/M 80, FORTRAN 80, or 8080/8085 Macro Assembler
- Allows Modular Breakdown of Large Programs and Separate Compilation of Individual Modules
- Gives Application Control Over Run-Time Errors by Providing User-Declared Error Procedures

PASCAL 80 Software Package consists of a compiler and an interactive Run-Time System designed to provide the Pascal programming language as a software development tool for Intellec Development System Users.

Pascal is a highly-structured, block-oriented programming language that is now gaining wide acceptance as a powerful software development tool. Its rigid structure encourages and enforces good programming techniques, which, combined with a high level of readability, helps produce more reliable software.

Standard Intel development tools, such as CREDIT editor can be used to create and modify Pascal source programs. The compiler compiles this source and creates a P-Code file. The Run-Time System executes this P-Code in an interpretive manner under ISIS-II.

*Pascal language as defined in *PASCAL User Manual and Report*, Second Edition, Kathleen Jensen and Niklaus Wirth.



LANGUAGE FEATURES

Data Structures

Pascal allows the user to define labels, constants, data types, variables, procedures, and functions.

Variable Types

Variables can be defined according to the following system-defined data types: boolean, integer, real, character, array, record, string, set, file, and pointer.

User-Defined Types

New types can be defined by the user for added flexibility.

File Handling Procedures

Pascal provides procedures to allow a user's program to interface with the ISIS-II file manager. Routines provided are: RESET, REWRITE, CLOSE, PUT, GET, SEEK, and PAGE.

Input/Output Procedures

Routines are provided to interact with the console or an ISIS file. These procedures are: READ, WRITE, READLN, WRITELN, plus BUFFER and BLOCK Read and Write.

Dynamic Memory Allocation

The procedures NEW, MARK, and RELEASE allow the user to obtain and release memory space at run-time for dynamically allocating variable storage.

String Handling

Pascal provides powerful tools for defining and manipulating strings and character arrays. These facilities enable concatenation of strings, character and pattern scans, insertion, deletion, and pointer manipulation.

Recursion

Pascal allows a PROCEDURE definition to include a call to itself, a powerful construct in many mathematical algorithms.

PROGRAM TRACING FACILITY

The PASCAL 80 System incorporates a program tracing facility which allows for selectively monitoring the execution of a Pascal program. When the TRACE flag is set, the line number of each program statement being executed is output to the console.

The TRACE flag may be manipulated in two ways:

- The TRACEON command (of the Run-Time System) will set the flag, and the TRACEOFF command will reset the flag.
- Pressing the Interrupt 4 switch on the Intellec System front panel will toggle the TRACE flag; i.e., the flag will be set if it was reset, and vice-versa.

COMPILER DIRECTIVES (PARTIAL LIST)

Compiler Command Line Directives

NOLIST

No list file is produced; used for fast compilation of "clean" programs.

NOCODE

No code file is produced; used for syntax error checking.

ERRLIST

List file is limited to only those Pascal lines that contain errors, along with the error messages produced.

LIST (file-name)

Specifies the name of the list file.

CODE (file-name)

Specifies the name of the code file.

NOECHO

Error lines are echoed on the console unless this directive is specified.

Embedded Compiler Directives

\$C text

Causes text to appear in code file (allows for comments, copyrights, etc.).

\$I+

Causes checking for I/O completion after each I/O transfer. Failure results in a run-time error. (\$I- causes no checking, and no errors on I/O failure.)

\$R+

Causes Range Checking to occur, so that an out-of-range value causes a Run-Time error. (\$R- suppresses generation of code for Range Checking.)

\$O+

Causes the compiler to operate in overlay mode. Overlays allow less source code to reside in memory. (\$O- causes no overlays, which decreases compile time, since there are fewer disk accesses.)

\$T+

Causes the compiler to generate tracing instructions to be used by the TRACE facility. (\$T- suppresses tracing instructions.)

BENEFITS

Brings Pascal to Intel Microcomputer Development Systems:

- Pascal is a block-structured, highly-readable programming language, suitable for a wide-range of applications.

- Pascal is being acclaimed as the programming language of the future; it is being taught in many colleges and universities around the country.

- PASCAL 80 Run-Time System provides great ease in programming formatted I/O operations.

PASCAL 80 provides a portable language for application programs running under ISIS-II.

PASCAL 80 can be used to evaluate complicated algorithms using a natural language.

PASCAL 80 compiler generates intermediate Pseudo-code.

- P-code is optimized for speed and storage space.

- P-code is approximately 50% to 70% smaller than corresponding machine code.

- P-code is machine independent, providing code portability to any CPU.

Makes the Intel Development System a more valuable tool. Extension of software support to include Pascal makes software development and resource management more flexible.

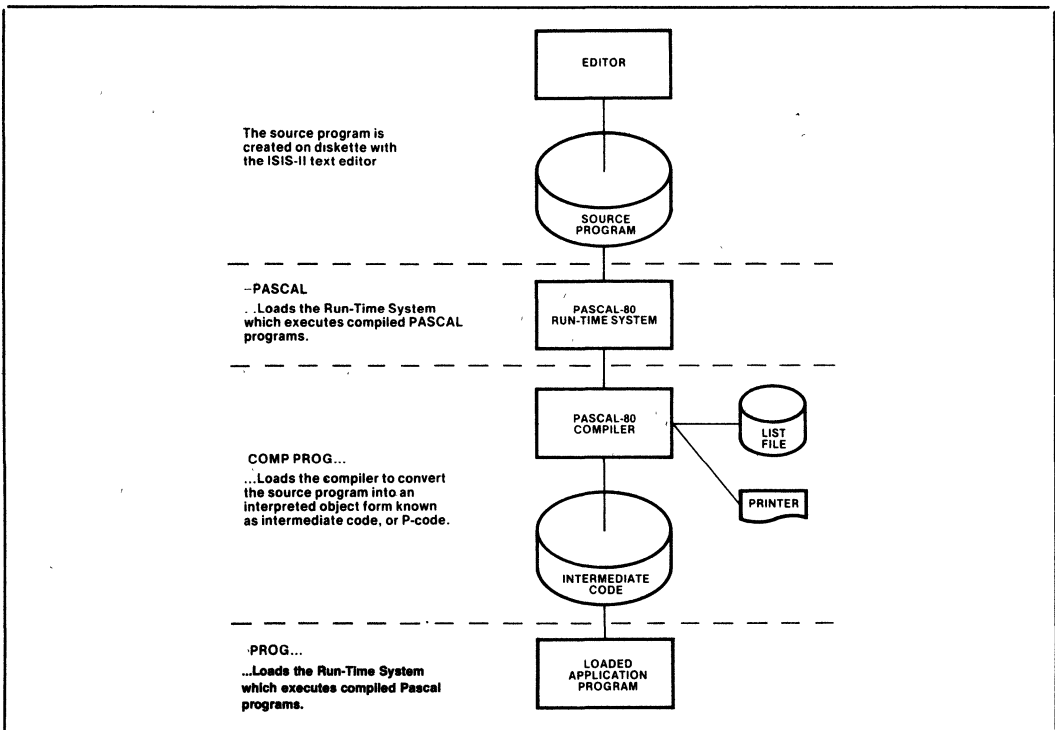


Figure 1. Program Development Cycle

Table 1. Sample Program Listing Showing Nesting Levels

BUFFER.PAS Program Listing					
Line	Seg	Proc	Lev	Disp	
1	1	1		1	program example;
2	1	1		3	
3	1	1		3	{ Example using bufferread and bufferwrite with break characters }
4	1	1		3	
5	1	1		3	var buffer: string;
6	1	1		44	disk storage: file;
7	1	1		64	break char;
8	1	1		65	new len, len: integer;
9	1	1		67	buff array: packed array[0..80] of char;
10	1	1		108	
11	1	1	0	0	begin
12	1	1	1	0	rewrite(disk storage, 'data');
13	1	1	1	27	writeln('Input a line of text: ');
14	1	1	1	68	readln(buffer);
15	1	1	1	87	len := bufferwrite(disk storage, buffer[1], length(buffer));
16	1	1	1	109	repeat
17	1	1	2	109	reset(disk storage);
18	1	1	2	116	writeln, writeln;
19	1	1	2	132	write('Input break char [cntrl Z to stop]: ');
20	1	1	2	179	readln(break);
21	1	1	2	197	if not eof(input) then
22	1	1	3	208	begin
23	1	1	4	208	new len = bufferread(disk storage, buff array, len, ord(break)),
24	1	1	4	226	writeln('The buffer read. ');
25	1	1	4	262	writeln(copy(buffer, 1, abs(new len)));
26	1	1	4	292	writeln('Length. ', abs(new len):0);
27	1	1	4	331	if new len < 0 then writeln('Break char not found');
28	1	1	3	378	end.
29	1	1	1	378	until eof(input);
30	1	1	0	388	end

SPECIFICATIONS

Operating Environment

REQUIRED HARDWARE

- Intellec® Microcomputer Development System
- Model 800
- Series II Model 220, Model 230, Model 240
- 64KB of Memory
- Dual-Diskette Drives
- Single- or Double-Density*
- System Console
- Intel® CRT or non-Intel® CRT

*Recommended.

REQUIRED SOFTWARE

- ISIS-II Diskette Operating System
- Single- or Double-Density

OPTIONAL SOFTWARE

ISIS-II CREDIT™ (CRT-Based Text Editor)

Documentation Package

PASCAL 80 User's Guide (9801015-01)

PASCAL User Manual and Report, Second Edition,
Kathleen Jensen and Niklaus Wirth

Shipping Media

- Flexible Diskettes
- Single- and Double-Density

ORDERING INFORMATION

Part Number	Description
MDS-381*	PASCAL 80 Software Package
	Requires Software License

*MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Sciences Corporation.

SUPPORT CATEGORY: Level D



WordStar* WORD PROCESSING SOFTWARE

- **Powerful, reliable, and user-friendly word processing software package**
- **Six on-screen menus and ten Help menus provide quick command reference**
- **Printout enhancements provide numerous combined print functions**
- **Simple formatting commands including Hyphen-Help**
- **Streamlines text entry**
- **Horizontal scrolling for wide pages**
- **Wordwrap removes need to worry about right margin**
- **On-screen formatting displays text exactly as it will be printed**
- **All functions easily controlled despite differences in printers and consoles**

WordStar, a popular word processing program written for use under the CP/M⁺ operating system, gives screen editing capabilities in an easy to learn and use format. The program is in use by programmers, and engineers for documentation and program entry, as well as managers and secretaries.

With WordStar, the user can easily make insertions and deletions, move or copy blocks of text, and search for and replace a string of text. WordStar will automatically reformat text upon command as these editing functions are performed.

Documents produced by WordStar can include any combination desired of pagination (page numbers), right and left justification, subscripts, superscripts, underlining, boldface type, overstrikes, crossouts, and even accents for use in foreign languages. Commands for all of these are entered with simple control-character keystrokes which are well documented in the program's six help menus.

All WordStar commands are easily executed using the CTRL key and the standard typewriter keys. Using the CTRL key, the function of standard keys can be changed to perform useful editing commands. The cursor-movement diamond (a group of standard keys on the keyboard) allows fast access to any area of text.

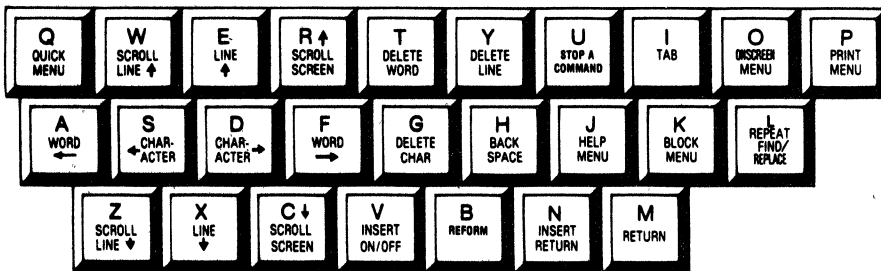


Figure 1. WordStar Keyboard Functions

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products: BXP CREDIT, i, ICE, iCS, i'm, Insite, Int, i, INTEL, InteleVision, Intelligent Identifier™, Intelligent Programming™, Intellink, Intellec, iMMX, iOSP, iPDS, iRMX, iSBC, iSBX, Library Manager, MCS, MULTIMODULE, Megachassis, Micromainframe, MULTIBUS, Multichannel, Plug-A-Bubble, PROMPT, Promware, RUPi, RMX/80, System 2000, UPI, and the combination of iCS, iRMX, iSBC, iSBX, ICE, i² ICE, MCS, or UPI and a numerical suffix. Intel Corporation Assumes No Responsibility for the use of Any Circularity Other Than Circularity Embodied in an Intel Product. No Other Patent Licenses are Implied. © INTEL CORPORATION, 1983

*WordStar MailMerge and SpellStar are trademarks of MicroPro International.

†CP/M is a registered trademark of Digital Research Inc.

FEATURES

WordStar is designed to be simple for the novice to use, while remaining sophisticated enough to be appealing to even the most advanced user.

Standard typewriter keys are combined with the "Control" key to provide a wide variety of editing functions (Fig. 1). All cursor control is localized to the ten keys in the "Cursor-Movement Diamond" (Fig. 2), and the on-screen menu details the functions of the other keys, so the user can quickly find functions without memorizing them.

Wordwrap is a feature of WordStar that allows the typist to entirely disregard margins. When typed characters go beyond the right margin, WordStar brings the last full word down to the next line automatically. The only time the Return key needs to be used is between paragraphs. Margins can be automatically right and left justified both during and after entry.

Horizontal Scrolling give the flexibility in creating documents too wide to fit on the video screen. When Wordwrap is disabled and a line is being typed beyond normal screen width, the displayed lines are automatically scrolled offscreen to the left. A single keystroke can be used to move the lines back to their normal position. Editing functions can also use Horizontal Scrolling to examine and modify any part of a wide document.

The On-Screen Formatting feature displays the text on the screen as it will appear when it is printed. This allows the changing of margins, spacing, and other format variables without requiring the use of a number of intermediate printouts.

Hyphen-Help aids in reformatting by positioning the cursor over a word requiring hyphenation at the end of a line, and allowing the user to select a hyphenation point or decide not to hyphenate. Hyphens entered this way are "soft", and will not be printed if the document is reformatted and the hyphen is no longer required. Permanent or "hard" hyphens are inserted while typing and will always be printed.

WordStar's Find and Replace command allows the text to be scanned for a specified character string. Once the string is found it will be replaced quickly with the updated information. Options with this command allow the user to perform functions like finding the "nth" occurrence, performing the operation "n" times, replacing the string without verification by the

user each time, searching backward, or to compensate for differences in upper and lower case letters (i.e., at the beginning of a sentence).

Entire blocks of text can be marked at their beginning and ending, then moved to a new area as easily as moving the cursor. Different block control commands allow the duplication and deletion of blocks as well.

Column Move assists in the creation and editing of tables of data. With Column Move, a column can be taken from one table and moved to another table or to another place in the same table. Columns can also be easily duplicated or deleted.

Over 20 Page Formatting commands enable a range of functions from producing automatic page headings to overriding built-in parameters for line height and character width. Margins can be set and number of lines typed per page can be dictated via these very simple commands. These page commands are especially useful in long documents.

Decimal Tab is a feature that assists in aligning figures into columns. When a number is entered into a decimal tab position, it will be automatically aligned so that its decimal point is directly below the decimal point of the number on the line above.

Files can be combined with each other to form derivative documents. One file can be inserted at any point of another, beginning middle or end, with equal simplicity.

Print controls, single letters entered while editing to enhance the printout, permit the user of underscore, boldface, underlining, double-strike, superscript, subscript, overprint, and nearly any combination of the above. This facilitates the generation of mathematical formulas with subscripts and superscripts, and allows the text to include foreign words and phrases with accents above and below certain letters. Alternate character pitch, for italics, and even ribbon color selection can be controlled by WordStar if these options are available on the printer in use.

Can be used with MailMerge* to generate chained printing combining form letters with mailing lists. MailMerge allows names to be drawn from the address and inserted into the text of the form letter.

SpellStar* may also be used with WordStar to check the spelling in a document against both a 20,000-word standard dictionary and a user-generated supplemental dictionary which can be used to store names, buzz words, or abbreviations.

WordStar is easily adapted to nearly any video terminal and document printer, despite the wide variation of options and communication standards used by these devices. At installation time the user is prompted through a series of questions which configure that WordStar installation to fit the hardware at hand.

When an existing file is edited and saved, any previous version of the file is saved under the original filename as a .BAK extension (i.e., after updating a file entitled "LETTER" there would be two files, LETTER and LETTER.BAK on the diskette). When a document is to be updated, its latest extension is automatically used as input. Whenever a new .BAK extension is created, the older .BAK version is destroyed.

WordStar allows documents of almost any size to be entered and edited. A Memory Management feature automatically transfers text to and from mass storage if the document is too large to be held in main memory at one time.

There are six Main Menus (Fig. 3A-3F) and ten Help Menus to guide even the most inexperienced user through a WordStar editing session. The Main or

On-Screen Menus are displayed at the top of the screen along with the text being edited. Should the user desire to fill the entire screen with text, the menu displays can be turned on and off as desired.

The ten Help Menu guide the user through the use of all editing functions from Moving Text to Paragraph Reform.

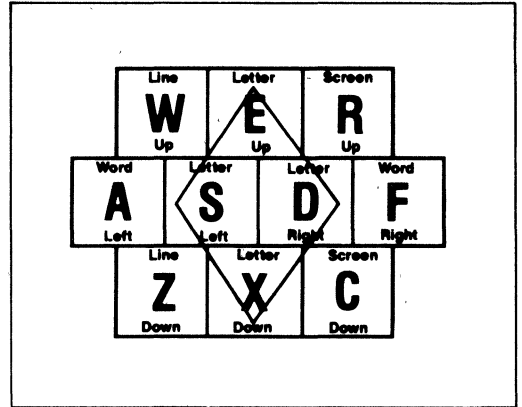


Figure 2. The Cursor Movement Diamond

```

A:TEST.DOC PAGE 1 LINE 1 COL 1 INSERT ON
<<< MAIN MENU >>>
* * Cursor Movement * *|* Delete *| * Miscellaneous *| * Other Menus *
^S char left ^D char right |^G char |^I Tab ^B Reform | (from Main only)
^A word left ^F word right |DEL chr lf|^V Insert On or Off|^J Help ^K Block
^E line up ^X line down |^T word rt|^L Find/Replce again|^Q Quick ^P Print
* * Scrolling * *|^Y line |RETURN End paragraph|^O Onscreen
^Z line up ^W line down | |^N Insert a RETURN|
^C screen up ^R screen down| |^U Stop a command |
L-----R

```

Figure 3A. Main Menu: guide to the most frequently used commands. This menu—and all other menus—can be called up at any time, or dropped to allow full-screen viewing of the text.

```

^J A:TEST.DOC PAGE 1 LINE 1 COL 1 INSERT ON
<<< HELP MENU >>>
H Display and set the help level | S Status line | * Other Menus *
B Paragraph reform (CTRL B command) | R Ruler line | (from Main only)
F Flags in rightmost column of screen | M Margins and tabs | ^J Help ^K Block
D Dot commands, print ctrl(P command) | P Place markers | ^Q Quick ^P Print
| V Moving text | ^O Onscreen
| |Space bar returns
| |y to Main Menu.
L-----R

```

Figure 3B. Help Menu: a directory of commands that control help levels and show reference information.

```

^Q      A:TEST.DOC  PAGE 1 LINE 1 COL 1          INSERT ON
      <<<  Q U I C K  M E N U  >>>
* * Cursor Movement * *|* Delete *| * Miscellaneous * | * Other Menus *
S left side  D right side |Y line rt|F Find text in file | (from Main only)
E top of scrn X bottom scrn|DEL lin lf|A Find and Replace |^J Help ^K Block
R top of file C end of file|* * * *|L Find misspelling|^Q Quick ^P Print
B top of block K end of block          |Q Repeat command or | ^O Onscreen
0-9 marker  Z up      W down          | key until space |Space bar returns
V last Find or block                  | bar or other key |you to Main Menu.
L---!----!----!----!----!----!----!----!----!----!-----R
  
```

Figure 3C. Quick Menu: expanded cursor movement, deletion, find/replace commands, and place marker commands.

```

^K      A:TEST.DOC  PAGE 1 LINE 1 COL 1          INSERT ON
      <<<  B L O C K  M E N U  >>>
* Saving Files * |* Block Operations *| * File Operations *| * Other Menus *
S Save and resume|B Begin K End |R Read P Print | (from Main only)
D Save—done      |H Hide / Display |O Copy E Rename|^J Help ^K Block
X Save and exit  |C Copy Y Delete|J Delete          |^Q Quick ^P Print
Q Abandon file  |V Move W Write |* Disk Operations *|^O Onscreen
* Place Markers *|N Column off (ON)|L Change logged disk|Space bar returns
0-9 Set/hide # 0-9|          |F Directory on (OFF)|you to Main Menu.
L---!----!----!----!----!----!----!----!----!----!-----R
  
```

Figure 3D: Block Menu: instructions for using block and place markers, saving and printing a file, and inserting other files.

```

^O      A:TEST.DOC  PAGE 1 LINE 1 COL 1          INSERT ON
      <<<  O N S C R E E N  M E N U  >>>
* Margins & Tabs * |* Line Functions *| * More Toggles * | * Other Menus *
L Set left margin |C Center text      |J Justify off (ON)| (from Main only)
R Set right margin |S Set line spacing |V Vari-tabs off (ON)|^J Help ^K Block
X Release margins |          |H Hyph-help off (ON)|^Q Quick ^P Print
I Set N Clear tab| * Toggles * |E Soft hyph on (OFF)|^O Onscreen
G Set paragraph tab|W Wr d wrap off (ON)|D Prnt disp off (ON)|Space bar returns
F Ruler from line |T Rlr line off (ON)|P Pge break off (ON)|you to Main Menu.
L---!----!----!----!----!----!----!----!----!----!-----R
  
```

Figure 3E. Onscreen Menu: functions that perform onscreen document formatting (such as line spacing, tabs, margins, justification, and wordwrap).

```

^P      A:TEST.DOC  PAGE 1 LINE 1 COL 1          INSERT ON
      <<<  PRINT  MENU  >>>
*Special Effects*| * Special Effects * |* Printing Changes *| * Other Menus *
(begin and end) | (one time each) |A Alternate pitch | (from Main only)
B Bold  D Double|H Overprint character|N Standard pitch |^J Help ^K Block
S Underscore |O Non-break space |C Printing pause |^Q Quick ^P Print
X Strikeout |F Phantom space |Y Other ribbon color|^O Onscreen
V Subscript |G Phantom rubout | * User Patches *|Space bar returns
T Superscript |RETURN Overprint line|Q(1) W(2) E(3) R(4) |you to Main Menu.
L-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----R
  
```

Figure 3F. Print Menu: special print control characters including subscripts, superscripts, boldface, double strike, and strikeout.

BENEFITS

WordStar is an advanced word-processing program that can turn any CP/M based personal computer into a sophisticated yet easy to learn and use text processor. It takes very little time for even the least-trained user to learn to productively generate documentation with WordStar.

The simplifying features of WordStar do not detract from its acceptance by advanced users. Menus and other features are designed to be unobtrusive when they are not needed. WordStar's sophistication means that it will not run out of horsepower as the

user progresses, but will always be an appealing and highly productive tool.

With WordStar there is no question about the appearance of the printed output, since the text can be displayed on the screen exactly as it is to be printed.

Time savings when using WordStar will be considerable. Generation of new text is easier than by handwritten/typed means. When WordStar is used for program editing it supplies powerful features unavailable in other editors. With WordStar, both code and documentation can be generated at the same time within the same environment.

Table 1.

EDITING COMMAND INDEX	
^	hold CTRL key, type letter
^A	cursor left word
^B	reform paragraph
^C	scroll up screenful
^D	cursor right character
^E	cursor up line
^F	cursor right word
^G	delete character right
^H	cursor left character
^I	tab
^J	help PREFIX
^K	editing PREFIX
^L	find/replace again
^M	(Same as RETURN)
^N	insert hard carriage return
^O	formatting PREFIX
^P	print control PREFIX
^Q	editing PREFIX
^R	scroll down screenful
^S	cursor left character
^T	delete word right
^U	interrupt
^V	insert on/off
^W	scroll down line
^X	cursor down line
^Y	delete line
^Z	scroll up line
^JB	explain reform
^JD	summarize print directives
^JF	explain Flags
^JH	set Help level
^JI	command index
^JM	explain tabs and Margins
^JP	explain Place markers
^JR	explain Ruler line
^JS	explain Status line
^JV	explain moving text
^K0-^K9	set/hide marker 0-9
^KB	mark/hide Block beginning
^KC	Copy block
^KD	Done edit (save)
^KE	rName file
^KF	File directory on/off
^KH	Hide/display marked block
^KJ	delete additional file
^KK	mark block end
^KL	change Logged disk
^KN	column mode on/off
^KO	cOpy file
^KP	Print
^KQ	abandon edit
^KR	Read additional file
^KS	Save and reedit
^KV	moVe block
^KW	Write block to additional file
^KX	save and eXit
^KY	delete block
^OC	Center cursor line
^OD	print control display on/off
^OE	soft hyphen Entry on/off
^OF	margins & tabs from line
^OG	paraGraph tab
^OH	Hyphen-Help on/off
^OI	set tab stop
^OJ	Justification on/off
^OL	set left margin
^ON	clear tab stop
^OP	Page break display on/off
^OR	set Right margin
^OS	set line Spacing
^OT	ruler display on/off
^OW	Wordwrap
^PA-^PZ	enter ^A-^Z
^PM	make next line overprint
^PO	enter non-break space
^Q0-^Q9	cursor to marker 0-9
^QA	find and replace
^QB	cursor Block beginning
^QC	cursor end file
^QD	cursor right end line
^QE	cursor top screen
^QF	Find
^QK	cursor block end
^QL	find misspelling
^QP	cursor Previous position
^QQ	repeat next command
^QR	cursor beginning of file
^QS	cursor left Side screen
^QV	cursor source
^QW	continuous downward scroll
^QX	cursor bottom of screen
^QY	delete to end line
^QZ	continuous upward scroll
^Qdel	delete to beginning line
^DEL	delete character left
^ESCAPE	error release
^LINE FEED	(same as ^J)
^RETURN	hard carriage return
^TAB	tab

Table 1. (Continued)

NO-FILE COMMANDS			
D	open a Document file	O	cOpy file
E	rEname file	P	Print
F	File directory on/off	R	Run program
H	set Help level	S	run SpellStar (optional)
L	change Logged disk	X	eXit to operating system
M	run MailMerge (optional)	Y	delete file
N	open a Non-document file		

SPECIFICATIONS

OPERATING ENVIRONMENT

Hardware Required

8080 or 8085 CPU
 5¼" or 8" Diskette drive
 Printer
 64K Bytes of memory
 Console with absolute cursor addressing
 Note Intellec Series II and III require iMDX-511

Optional hardware

Additional mass storage

Software Required

CP/M 2.2 operating system

DOCUMENTATION PACKAGE

Wordstar Training Guide
 Wordstar Operator's Guide

 Wordstar General Information Manual
 Wordstar Reference Manual
 Wordstar Installation Manual

SUPPORT

Intel offers several levels of support for this product, depending on the system configuration in which it is used. Please consult the price list for a detailed description of the support options available.

Intel software license is required.

ORDERING INFORMATION

Description

WordStar word processing software package for use under the CP/M operating system

Order Code	Shipping Media
------------	----------------

SD111CPM80ASU	A—Single-density 8" diskette
SD111CPM80BSU	B—Double-density 8" diskette
SD111CPM80FSU	F—iPDS Format 5¼" diskette



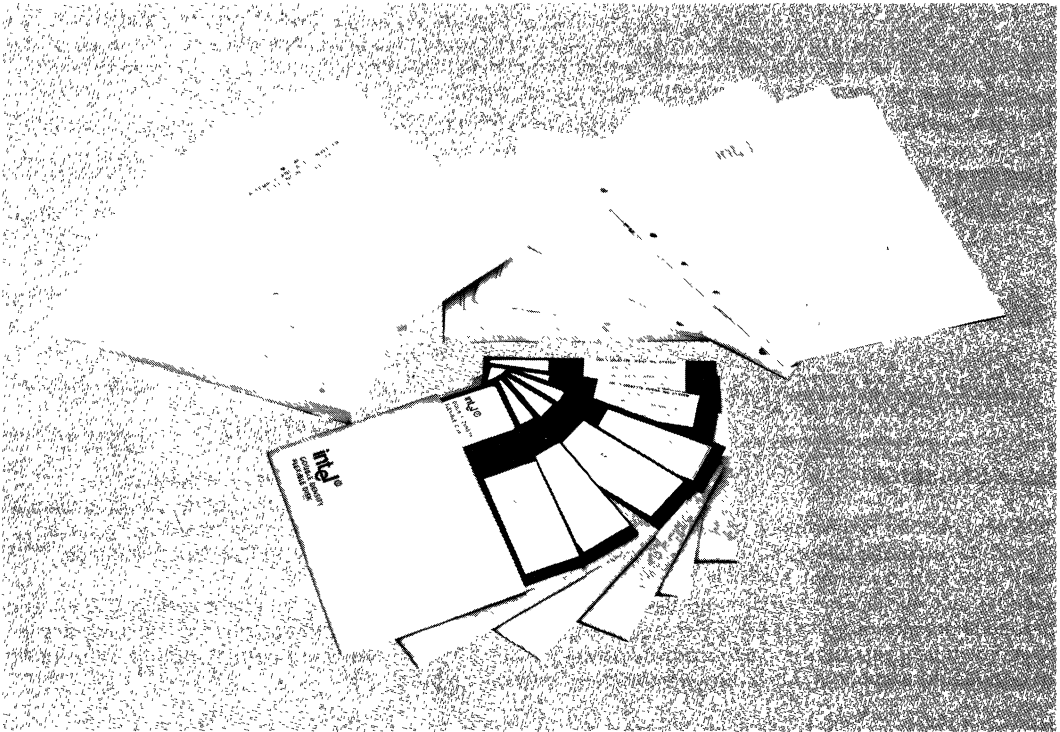
iAPX 86,88 SOFTWARE DEVELOPMENT PACKAGES FOR SERIES II/PDS

- **PL/M 86/88 High Level Programming Language**
- **ASM 86/88 Macro Assembler for iAPX 86,88 Assembly Language Programming**
- **LINK 86/88 and LOC 86/88 Linkage and Relocation Utilities**
- **CONV 86/88 Converter for Conversion of 8080/8085 Assembly Language Source Code to iAPX 86, 88 Assembly Language Source Code**
- **OH 86/88 Object-to-Hexadecimal Converter**
- **LIB 86/88 Library Manager**

The iAPX 86,88 Software Development Packages for Series II provide a set of software development tools for the iAPX 86/88 CPUs and the iSBC 86/12A single board computer. The packages operate under the ISIS-II operating system on Intel Microcomputer Development Systems—Model 800, Series II or the Personal Development System (PDS)—thus minimizing requirements for additional hardware or training for Intel Microcomputer Development System users.

These packages permit 8080/8085 users to efficiently upgrade existing programs into iAPX 86/88 code from either 8080/8085 assembly language source code or PL/M 80 source code.

For the new Intel Microcomputer Development System user, the packages operating on a PDS or an Intel Series II, such as a Model 235, provide total iAPX 86,88 software development capability.





PL/M 86/88 COMPILER FOR SERIES II/PDS

- **Language is Upward Compatible from PL/M 80, Assuring MCS-80/85™ Design Portability**
- **Supports 16-bit Signed Integer and 32-bit Floating Point Arithmetic in Accordance with IEEE Proposed Standard**
- **Easy-to-Learn, Block-Structured Language Encourages Program Modularity**
- **Produces Relocatable Object Code Which is Linkable to All Other 8086 Object Modules**
- **Supports Full Extended Addressing Features of the iAPX 86/10 and 88/10 Microprocessors (Up to 1 Mbyte)**
- **Code Optimization Assures Efficient Code Generation and Minimum Application Memory Utilization**

Like its counterpart for MCS-80/85 program development, PL/M 86/88 is an advanced, structured high-level programming language. The PL/M 86/88 compiler was created specifically for performing software development for the Intel iAPX 86,88 Microprocessors.

PL/M 86/88 has significant new capabilities over PL/M 80 that take advantage of the new facilities provided by the iAPX 86,88 microsystem, yet the PL/M 86/88 language remains compatible with PL/M 80.

With the exception of hardware-dependent modules, such as interrupt handlers, PL/M 80 applications may be recompiled with PL/M 86/88 with little need for modification. PL/M 86/88, like PL/M 80, is easy to learn, facilitates rapid program development, and reduces program maintenance costs.

PL/M is a powerful, structured, high-level system implementation language in which program statements can naturally express the program algorithm. This frees the programmer to concentrate on the logic of the program without concern for burdensome details of machine or assembly language programming (such as register allocation, meanings of assembler mnemonics, etc.).

The PL/M 86/88 compiler efficiently converts free-form PL/M language statements into equivalent 86/88 machine instructions. Substantially fewer PL/M statements are necessary for a given application than if it were programmed at the assembly language or machine code level.

The use of PL/M high-level language for system programming, instead of assembly language, results in a high degree of engineering productivity during project development. This translates into significant reductions in initial software development and follow-on maintenance costs for the user.

FEATURES

Major features of the Intel PL/M 86/88 compiler and programming language include:

Block Structure

PL/M source code is developed in a series of modules, procedures, and blocks. Encouraging program modularity in this manner makes programs more readable, and easier to maintain and debug. The language becomes more flexible by clearly defining the scope of user variables (local to a private procedure, global to a public module, for example).

The use of procedures to break down a large problem is paramount to productive software development. The PL/M 86/88 implementation of a block

structure allows the use of REENTRANT which is especially useful in system design.

Language Compatibility

PL/M 86/88 object modules are compatible with object modules generated by all other 86/88 translators. This means that PL/M programs may be linked to programs written in any other 86/88 language.

Object modules are compatible with ICE-88 and ICE-86 units; DEBUG compiler control provides the In-Circuit Emulators with symbolic debugging capabilities.

PL/M 86/88 Language is upward-compatible with PL/M 80, so that application programs may be easily ported to run on the iAPX 86 or 88.

Supports Five Data Types

PL/M makes use of five data types for various applications. These data types range from one to four bytes, and facilitate various arithmetic, logic, and addressing functions:

- Byte: 8-bit unsigned number
- Word: 16-bit unsigned number
- Integer: 16-bit signed number
- Real: 32-bit floating point number
- Pointer: 16-bit or 32-bit memory address indicator

Another powerful facility allows the use of BASED variables that map more than one variable to the same memory location. This is especially useful for passing parameters, relative and absolute addressing, and memory allocation

Two Data Structuring Facilities

In addition to the five data types and based variables, PL/M supports two data structuring facilities. These add flexibility to the referencing of data stored in large groups.

- Array: Indexed list of same type data elements
- Structure: Named collection of same or different type data elements
- Combinations of Each: Arrays of structures or structures of arrays

8087 Numerics Support

PL/M programs that use 32-bit REAL data may be executed using the Numeric Data Processor for improved performance. All floating-point operations supported by PL/M may be executed on the 8087 NDP, or the 8087 Emulator (a software module) provided with the package. Determination of use of the chip or emulator takes place at link-time, allowing compilations to be run-time independent.

Built-In String Handling Facilities

The PL/M 86/88 language contains built-in functions for string manipulation. These byte and word functions perform the following operations on character strings: MOVE, COMPARE, TRANSLATE, SEARCH, SKIP, and SET.

Interrupt Handling

PL/M has the facility for generating interrupts to the iAPX 86 or 88 via software. A procedure may be defined with the INTERRUPT attribute, and the compiler will automatically initialize an interrupt vector at the appropriate memory location. The compiler will also generate code to save and restore the processor status, for execution of the user-defined interrupt handler routine. The procedure SET\$INTERRUPT, the function retuning an INTERRUPT\$PTR, and the PL/M statement CAUSE\$INTERRUPT all add flexibility to user programs involving interrupt handling.

Segmentation Control

The PL/M 86/88 compiler takes full advantage of program addressing with the SMALL, COMPACT, MEDIUM, and LARGE segmentation controls. Programs with less than 64KB total code space can exploit the most efficient memory addressing schemes, which lowers total memory requirements. Larger programs can exploit the flexibility of extended one-megabyte addressing.

Code Optimization

The PL/M 86/88 compiler offers four levels of optimization for significantly reducing overall program size.

- Combination or "folding" of constant expressions; and short-circuit evaluation of Boolean expressions.
- "Strength reductions" (such as a shift left rather than multiply by 2); and elimination of common sub-expressions within the same block.
- Machine code optimizations; elimination of superfluous branches; re-use of duplicate code; removal of unreadable code.
- Byte comparisons (rather than 20-bit address calculations) for pointer variables; optimization of based-variable operations.

Compiler Controls

The PL/M 86/88 compiler offers more than 25 controls that facilitate such features as:

- Conditional compilation
- Intra- and Inter-module cross reference
- Corresponding assembly language code in the listing file
- Setting overflow conditions for run-time handling

BENEFITS

PL/M 86/88 is designed to be an efficient, cost-effective solution to the special requirements of iAPX 86 or 88 Microsystem Software Development, as illustrated by the following benefits of PL/M use:

Low Learning Effort

PL/M 86/88 is easy to learn and to use, even for the novice programmer.

Earlier Project Completion

Critical projects are completed much earlier than otherwise possible because PL/M 86/88, a structured high-level language, increases programmer productivity.

Lower Development Cost

Increases in programmer productivity translate immediately into lower software development costs

because less programming resources are required for a given programmed function.

Increased Reliability

PL/M 86/88 is designed to aid in the development of reliable software (PL/M 86/88 programs are simple statements of the program algorithm). This substantially reduces the risk of costly correction of errors in systems that have already reached full production status, as the more simply stated the program is, the more likely it is to perform its intended function.

Easier Enhancements and Maintenance

Programs written in PL/M tend to be self-documenting, thus easier to read and understand. This means it is easier to enhance and maintain PL/M programs as the system capabilities expand and future products are developed.

IAPX 86,88 MACRO ASSEMBLER FOR SERIES II/PDS

- **Powerful and Flexible Text Macro Facility with Three Macro Listing Options to Aid Debugging**
- **Highly Mnemonic and Compact Language, Most Mnemonics Represent Several Distinct Machine Instructions**
- **“Strongly Typed” Assembler Helps Detect Errors at Assembly Time**
- **High-Level Data Structuring Facilities Such as “STRUCTURES” and “RECORDS”**
- **Over 120 Detailed and Fully Documented Error Messages**
- **Produces Relocatable and Linkable Object Code**

ASM 86/88 is the “high-level” macro assembler for the iAPX 86,88 assembly language. ASM 86/88 translates symbolic 86/10, 88/10 assembly language mnemonics into 86/10, 88/10 relocatable object code.

ASM 86/88 should be used where maximum code efficiency and hardware control is needed. The iAPX 86,88 assembly language includes approximately 100 instruction mnemonics. From these few mnemonics the assembler can generate over 3,800 distinct machine instructions. Therefore, the software development task is simplified, as the programmer need know only 100 mnemonics to generate all possible 86/10, 88/10 machine instructions. ASM 86/88 will generate the shortest machine instruction possible given no forward referencing or given explicit information as to the characteristics of forward referenced symbols.

ASM 86/88 offers many features normally found only in high-level languages. The iAPX 86,88 assembly language is strongly typed. The assembler performs extensive checks on the usage of variables and labels. The assembler uses the attributes which are derived explicitly when a variable or label is first defined, then makes sure that each use of the symbol in later instructions conforms to the usage defined for that symbol. This means that many programming errors will be detected when the program is assembled, long before it is being debugged on hardware.

FEATURES

Major features of the Intel iAPX 86,88 assembler and assembly language include:

Powerful and Flexible Text Macro Facility

- Macro calls may appear anywhere
- Allows user to define the syntax of each macro
- Built-in functions
 - conditional assembly (IF-THEN-ELSE, WHILE)
 - repetition (REPEAT)
 - string processing functions (MATCH)
 - support of assembly time I/O to console (IN, OUT)
- Three Macro Listing Options include a GEN mode which provides a complete trace of all macro calls and expansions

High-Level Data Structuring Capability

- STRUCTURES: Defined to be a template and then used to allocate storage. The familiar dot notation may be used to form instruction addresses with structure fields.
- ARRAYS: Indexed list of same type data elements.
- RECORDS: Allows bit-templates to be defined and used as instruction operands and/or to allocate storage.

Fully Supports iAPX 86,88 Addressing Modes

- Provides for complex address expressions involving base and indexing registers and (structure) field offsets.
- Powerful EQU facility allows complicated expressions to be named and the name can be used as a synonym for the expression throughout the module.

Powerful STRING MANIPULATION INSTRUCTIONS

- Permit direct transfers to or from memory or the accumulator.
- Can be prefixed with a repeat operator for repetitive execution with a count-down and a conditional test.

Over 120 Detailed Error Messages

- Appear both in regular list file and error print file.
- User documentation fully explains the occurrence of each error and suggests a method to correct it.

Support for ICE-86™ Emulation and Symbolic Debugging

- Debug options for inclusion of symbol table in object modules for In-Circuit Emulation with symbolic debugging.

Generates Relocatable and Linkable Object Code—Fully Compatible with LINK 86/88, LOC 86/88 and LIB 86/88

- Permits ASM 86/88 programs to be developed and debugged in small modules. These modules can be easily linked with other ASM 86/88 or PL/M 86/88 object modules and/or library routines to form a complete application system.

BENEFITS

The iAPX 86,88 macro assembler allows the extensive capabilities of the 86/88 CPU's to be fully exploited. In any application, time and space critical routines can be effectively written in ASM 86/88. The 86,88 assembler outputs relocatable and linkable object modules. These object modules may be easily combined with object modules written in PL/M 86/88—Intel's structured, high-level programming language. ASM 86/88 compliments PL/M 86/88 as the programmer may choose to write each module in the language most appropriate to the task and then combine the modules into the complete applications program using the iAPX 86,88 relocation and linkage utilities.

CONV 86/88 MCS[®]-80/85 to iAPX 86,88 ASSEMBLY LANGUAGE CONVERTER UTILITY PROGRAM

- **Translates 8080/8085 Assembly Language Source Code to iAPX 86,88 Assembly Language Source Code**
- **Automatically Generates Proper ASM 86/88 Directives to Set Up a "Virtual 8080" Environment that is Compatible with PL/M 86/88**
- **Provides a Fast and Accurate Means to Convert 8080/8085 Programs to the iAPX 86/88 Facilitating Program Portability**

In support of Intel's commitment to software portability, CONV 86/88 is offered as a tool to move 8080/8085 programs to the iAPX 86/88. A comprehensive manual, "MCS-86 Assembly Language Converter Operating Instructions for ISIS-II Users," covers the entire conversion process. Detailed methodology of the conversion process is fully described therein.

- CONV 86/88 will accept as input an error-free 8080/8085 assembly-language source file and optional controls, and produce as output, optional PRINT and OUTPUT files.
- The PRINT file is a formatted copy of the 8080/8085 source and the 86/88 source file with embedded caution messages.
- The OUTPUT file is an 86/88 source file.
- CONV 86/88 issues a caution message when it detects a potential problem in the converted 86/88 code.
- A transliteration of the 8080/8085 programs occurs, with each 8080/8085 construct mapped to its exact 86/88 counterpart:

Registers
Condition flags
Instruction
Operands
Assembler directives
Assembler control lines
Macros

Because CONV 86/88 is a transliteration process, there is the possibility of as much as a 15%–20% code expansion over the 8080/8085 code. For compactness and efficiency it is recommended that critical portions of programs be re-coded in iAPX 86,88 assembly language.

Also, as a consequence of the transliteration, some manual editing may be required for converting instruction sequences dependent on:

- instruction length, timing, or encoding
- interrupt processing*
- PL/M parameter passing conventions*

*Mechanical editing procedures for these are suggested in the converter manual.

The accompanying figure illustrates the flow of the conversion process. Initially, the abstract program may be represented in 8080/8085 or iAPX 86,88 assembly language to execute on that respective target machine. The conversion process is porting a source destined for the 8080/8085 to the 86/88 via CONV 86/88.

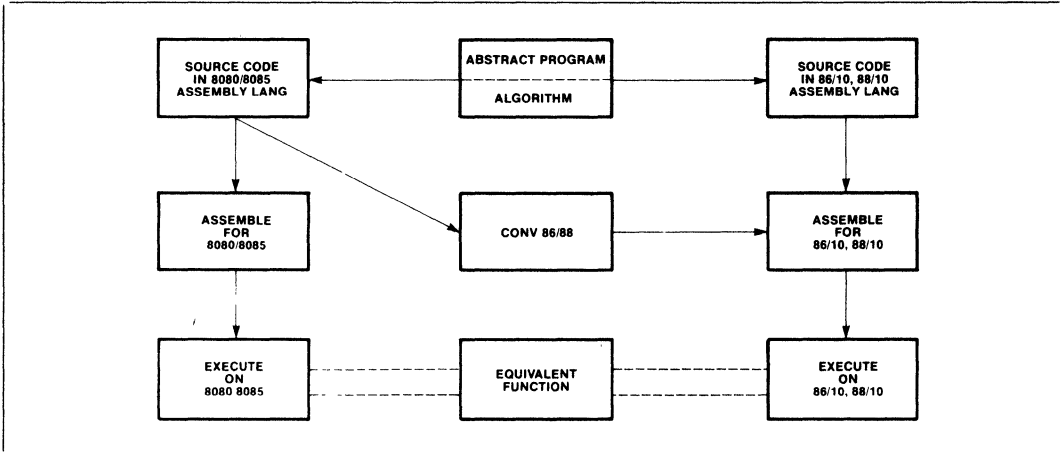


Figure 1. Porting 8080/8085 Source Code to the iAPX 86/10 and 88/10

LINK 86/88

- Automatic Combination of Separately Compiled or Assembled iAPX 86, 88 Programs Into a Relocatable Module
- Automatic Selection of Required Modules from Specified Libraries to Satisfy Symbolic References
- Extensive Debug Symbol Manipulation, Allowing Line Numbers, Local Symbols, and Public Symbols to be Purged and Listed Selectively
- Automatic Generation of a Summary Map Giving Results of the LINK 86/88 Process
- Abbreviated Control Syntax
- Relocatable Modules may be Merged into a Single Module Suitable for Inclusion in a Library
- Supports "Incremental" Linking
- Supports Type Checking of Public and External Symbols

LINK 86/88 combines object modules specified in the LINK 86/88 input list into a single output module. LINK 86/88 combines segments from the input modules according to the order in which the modules are listed.

LINK 86/88 will accept libraries and object modules built from PL/M 86/88, ASM 86/88, or any other translator generating Intel's iAPX 86/88 Relocatable Object Modules.

Support for incremental linking is provided since an output module produced by LINK 86/88 can be an input to another link. At each stage in the incremental linking process, unneeded public symbols may be purged.

LINK 86/88 supports type checking of PUBLIC and EXTERNAL symbols reporting an error if their types are not consistent.

LINK 86/88 will link any valid set of input modules without any controls. However, controls are available to control the output of diagnostic information in the LINK 86/88 process and to control the content of the output module.

LINK 86/88 allows the user to create a large program as the combination of several smaller, separately compiled modules. After development and debugging of these component modules the user can link them together, locate them using LOC 86/88 and enter final testing with much of the work accomplished.

LIB 86/88

- **LIB 86/88 is a Library Manager Program which Allows You to:**
 - Create Specially Formatted Files to Contain Libraries of Object Modules**
 - Maintain These Libraries by Adding or Deleting Modules**
 - Print a Listing of the Modules and Public Symbols in a Library File**
- **Libraries Can be Used as Input to LINK 86/88 Which Will Automatically Link Modules from the Library that Satisfy External References in the Modules Being Linked**
- **Abbreviated Control Syntax**

Libraries aid in the job of building programs. The library manager program LIB 86/88 creates and maintains files containing object modules. The operation of LIB 86/88 is controlled by commands to indicate which operation LIB 86/88 is to perform. The commands are:

CREATE: creates an empty library file
ADD: adds object modules to a library file
DELETE: deletes modules from a library file
LIST: lists the module directory of library files
EXIT: terminates the LIB 86 program and returns control to ISIS-II

When using object libraries, the linker will call only those object modules that are required to satisfy external references, thus saving memory space.

LOC 86/88

- **Automatic Generation of a Summary Map Giving Starting Address, Segment Addresses and Lengths, and Debug Symbols and their Addresses**
- **Automatic and Independent Relocation of Segments. Segments May Be Relocated to Best Match Users Memory Configuration**
- **Extensive Capability to Manipulate the Order and Placement of Segments in IAPX 86/88 Memory**
- **Extensive Debug Symbol Manipulation, Allowing Line Numbers, Local Symbols, and Public Symbols to be Purged and Listed Selectively**
- **Abbreviated Control Syntax**

Relocatability allows the programmer to code programs or sections of programs without having to know the final arrangement of the object code in memory.

LOC 86/88 converts relative addresses in an input module to absolute addresses. LOC 86/88 orders the segments in the input module and assigns absolute addresses to the segments. The sequence in which the segments in the input module are assigned absolute addresses is determined by their order in the input module and the controls supplied with the command.

LOC 86/88 will relocate any valid input module without any controls. However, controls are available to control the output of diagnostic information in the LOC 86/88 process, to control the content of the output module, or both.

The program you are developing will almost certainly use some mix of random access memory (RAM), read-only memory (ROM), and/or programmable read-only memory (PROM). Therefore, the location of your program affects both cost and performance in your application. The relocation feature allows you to develop your program on the Intel development system and then simply relocate the object code to suit your application.

OH 86/88

- Converts an iAPX 86/88 Absolute Object Module to Symbolic Hexadecimal Format

■ Facilitates Preparing a File for Later Loading by a Symbolic Hexadecimal Loader, such as the iSBC™ Monitor SDK-86 Loader, or Universal PROM Mapper
- Converts an Absolute Module to a More Readable Format that can be Displayed on a CRT or Printed for Debugging

The OH 86/88 utility converts an 86/88 absolute object module to the hexadecimal format. This conversion may be necessary to format a module for later loading by a hexadecimal loader such as the iSBC 86/12 monitor or Universal PROM Mapper. The conversion may also be made to put the module in a more readable format than can be displayed or printed.

The module to be converted must be in absolute format; the output from LOC 86/88 is in absolute format.

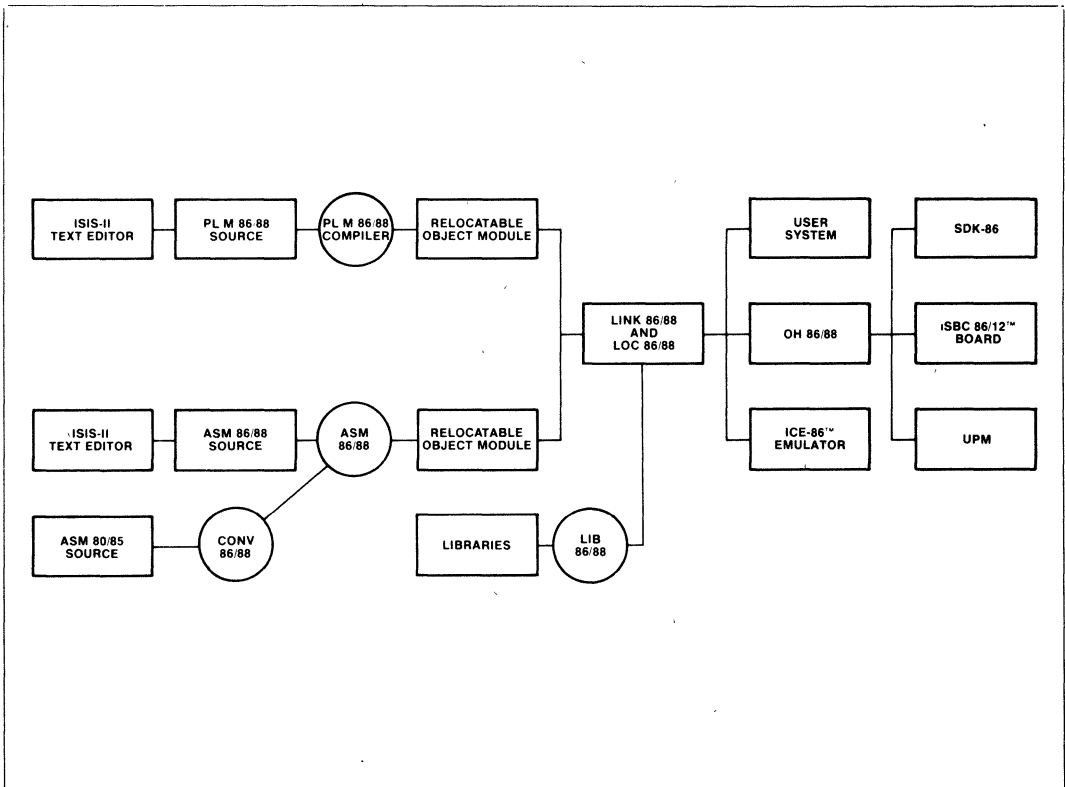


Figure 2. iAPX 86,88 Software Development Cycle

SPECIFICATIONS

Operating Environment

Intel Microcomputer Development Systems
Intel Personal Development System

Documentation

- PL/M-86 Programming Manual*
- ISIS-II PL/M-86 Compiler Operator's Manual*
- MCS-86 User's Manual*
- MCS-86 Software Development Utilities Operating Instructions for ISIS-II Users*
- MCS-86 Macro Assembly Language Reference Manual*
- MCS-86 Macro Assembler Operating Instructions for ISIS-II Users*
- MCS-86 Assembly Language Converter Operating Instructions for ISIS-II Users*
- Universal PROM Programmer User's Manual*

ORDERING INFORMATION

iAPX 86,88 Software Development Packages for Series II:

Part No.	Description
MDS-308*	Assembler and Utilities Package
MDS-309*	PL/M compiler and Utilities Package
MDS-311*	PL/M compiler, Assembler, and Utilities Package
All Packages Require Software Licenses	

SUPPORT:

Hotline Telephone Support, Software Performance Reports (SPR), Software Updates, Technical Reports, Monthly Newsletters are available

*MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Sciences Corporation.



86/88/186/188 SOFTWARE PACKAGES

FORTRAN 86/88 Software Package

- Features High-Level Language Support for Floating-Point Calculation, Transcendentals, Interrupt Procedures, and run-time exception handling
- Meets ANS FORTRAN 77 Subset Language Specifications
- Supports Complex Data Types

PASCAL 86/88 Software Package

- Resident on iAPX 86 Based Intel Microcomputer Development Systems
- Object Compatible and Linkable with PL/M 86/88, ASM 86/88 and FORTRAN 86/88
- Supports Large Array Operation

PL/M 86/88/186/188 Software Package

- Advanced Structured System Implementation Language for Algorithm Development
- Supports 16-bit Signed Integer and 32-bit Floating Point Arithmetic in Accordance with IEEE Proposed Standard
- Easy-to-Learn Block-Structured Language Encourages Program Modularity

iC-86 C Compiler for the 8086

- Implements Full C Language
- Produces High Density Code Rivaling Assembler
- Supports Intel Object Module Format (OMF)

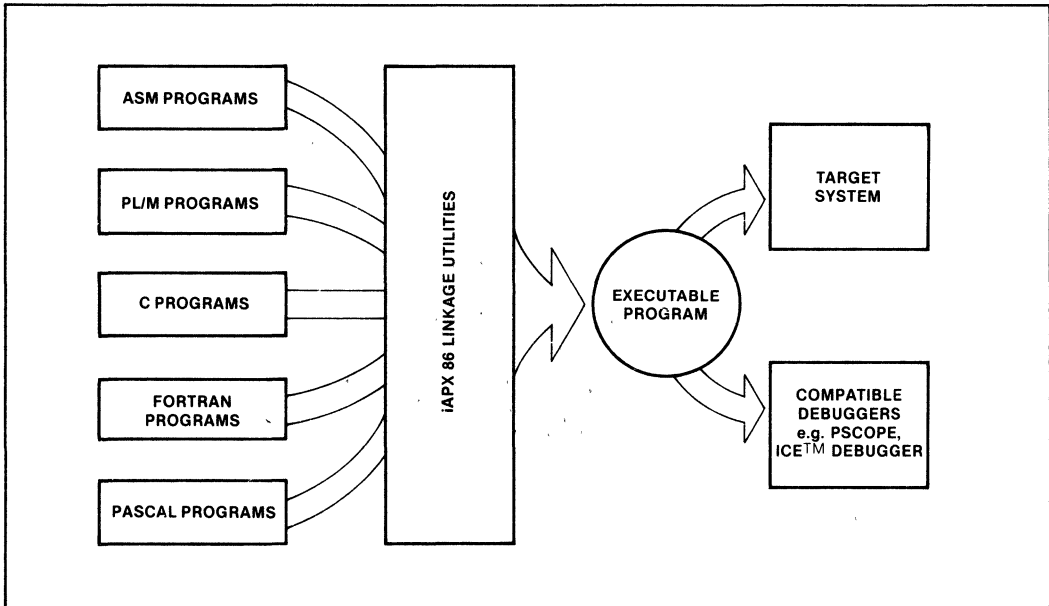


Figure 1. Program modules compiled with any of the iAPX 86 languages may be linked together. Each language is compatible with Intel's debug tools.

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. Information Contained Herein Supercedes Previously Published Specifications On These Devices From Intel.

INTEL CORPORATION, 1983



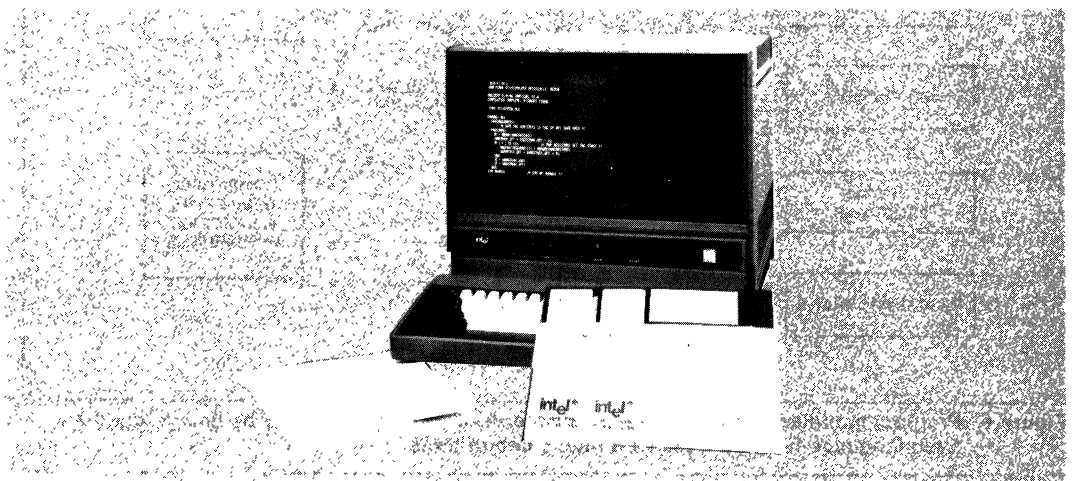
FORTRAN 86/88 SOFTWARE PACKAGE

- Features high-level language support for floating-point calculations, transcendentals, interrupt procedures, and run-time exception handling
- Meets ANS FORTRAN 77 Subset Language Specifications
- Supports iAPX 86/20, 88/20 Numeric Data Processor for fast and efficient execution of numeric instructions
- Uses REALMATH Floating-Point Standard for consistent and reliable results
- Supports Arrays Larger Than 64K
- Unlimited User Program Symbols
- Offers powerful extensions tailored to microprocessor applications
- Offers upward compatibility with FORTRAN 80
- Provides FORTRAN run-time support for iAPX 86,88,186,188-based design
- Provides users ability to do formatted and unformatted I/O with sequential or direct access methods
- ICE™ Symbolic Debugging Fully Supported
- PSCOPE Source Level Debugging Fully Supported
- Supports complex data types

FORTRAN 86/88 meets the ANS FORTRAN 77 Language Subset Specification and includes many features of the full standard. Therefore, the user is assured of portability of most existing ANS FORTRAN programs and of full portability from other computer systems with an ANS FORTRAN 77 Compiler.

FORTRAN 86/88 programs developed and debugged on the Intel Microcomputer Development Systems may be tested with the prototype using ICE symbolic debugging, and executed on an RMX-86 operating system, or on a user's iAPX 86,88,186,188-based operating system.

FORTRAN 86/88 is one of a complete family of compatible programming languages for iAPX 86,88,186,188 development: PL/M, Pascal, FORTRAN, and Assembler. Therefore, users may choose the language best suited for a specific problem solution.



FEATURES

Extensive High-Level Language Numeric Processing Support

Single (32-bit), double (64-bit), and double extended precision (80-bit) floating-point data types

REALMATH Proposed IEEE Floating-Point Standard) for consistent and reliable results

Full support for all other data types: integer, logical, character

Ability to use hardware (iAPX 86/20, 88/20 Numeric Data Processor) or software (simulator) floating-point support chosen at link time

ANS FORTRAN 77 Standard

Intel® Microprocessor Support

FORTRAN 86/88 language features support of iAPX 86/20, 88/20 Numeric Data Processor

Compiler generates in-line iAPX 86/20, 88/20 Numeric Data Processor object code for floating-point arithmetic (See Figure 1)

Intrinsics allow user to control iAPX 86/20, 88/20 Numeric Data Processor

iAPX 86,88,186,188 architectural advantages used for indexing and character-string handling

Symbolic debugging of application using ICE emulators

Source level debugging using PSCOPE.

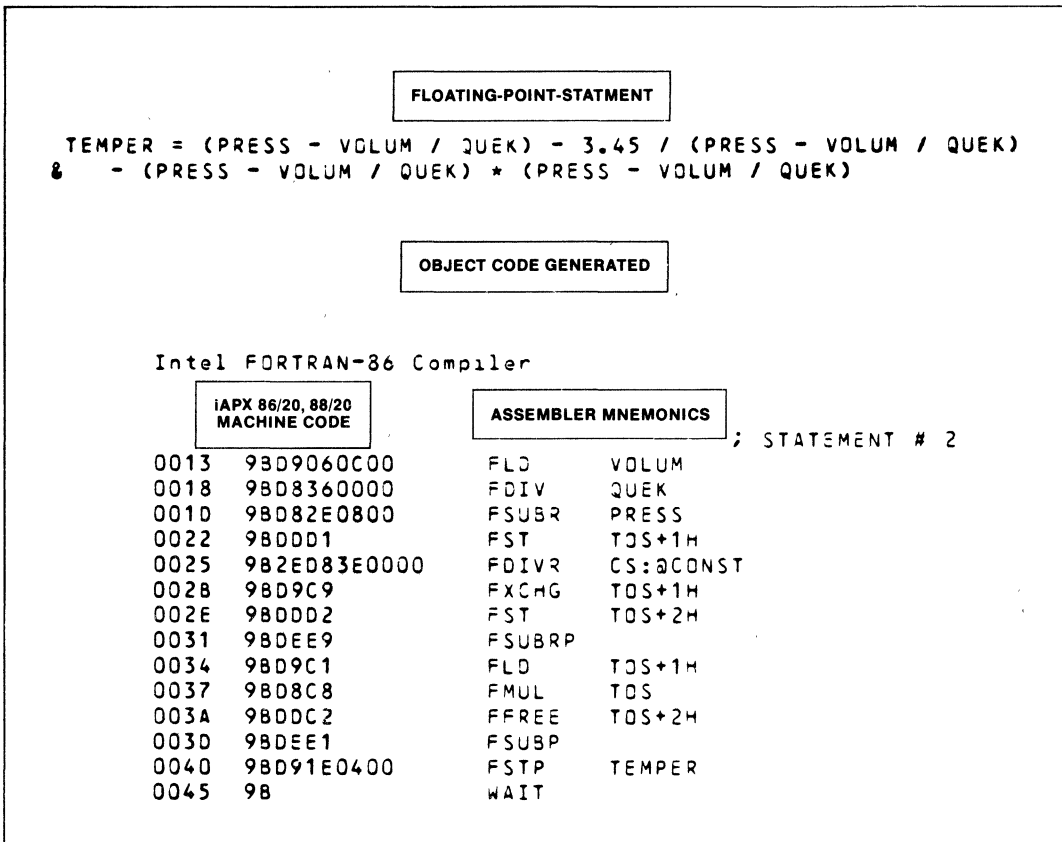


Figure 2. Object Code Generated by FORTRAN 86/88 for a Floating-Point Calculation Using iAPX 86/20, 88/20 Numeric Processor.

Microprocessor Application Support

- Direct byte- or word-oriented port I/O
- Reentrant procedures
- Interrupt procedures

Flexible Run-Time Support

Application object code may be executed in iAPX 86, 88, 186, 188-based environment of user's choice:

- a Series III or Series IV Intel Development System
- an iAPX 86, 88, 186, 188-based system with iRMX-86 Operating System
- an iAPX 86, 88, 186, 188-based system with user-designed Operating System

Run-time exception handling for fixed-point numerics, floating-point numerics, and I/O errors

Relocatable object libraries for complete run-time support of I/O and arithmetic functions. In-line code execution is generated for iAPX 86/20, 88/20 Numeric Data Processor

BENEFITS

FORTAN 86/88 provides a means of developing application software for the Intel iAPX 86, 88, 186, 188 products lines in a familiar, widely accepted, and industry-standard programming language. FORTAN 86, 88 will greatly enhance the user's ability to provide cost-effective software development for Intel microprocessors as illustrated by the following:

Early Project Completion

FORTAN is an industry-standard, high-level numerics processing language. FORTAN programmers can use FORTAN 86/88 on microprocessor projects with little retraining. Existing FORTAN software can be compiled with FORTAN 86/88 and programs developed in FORTAN 86/88 can run on other computers with ANS FORTAN 77 with little or no change. Libraries of mathematical programs using ANS 77 standards may be compiled with FORTAN 86/88.

Application Object Code Portability for a Processor Family

FORTAN 86/88 modules "talk" to the resident Intel development operating system using Intel's standard interface for all development-system software. This allows an application developed under the ISIS-II operating system to execute on iRMX/86, or a user-supplied operating system by linking in the iRMX/86 or other appropriate interface library. A standard logical-record interface enables communication with non-standard I/O devices.

Comprehensive, Reliable and Efficient Numeric Processing

The unique combination of FORTAN 86/88, iAPX 86/20, 88/20 Numeric Data Processor, and REALMATH (Proposed IEEE Floating-Point Standard) provide universal consistency in results of numeric computations and efficient object code generation.

SPECIFICATIONS

Operating Environment

Intel Microcomputer Development Systems (Series III/Series IV)

Documentation Package

FORTAN 86/88 User's Guide



ORDERING INFORMATION

Part Number Description

MDS*-315 FORTRAN 86/88 Software Package

Requires Software License

SUPPORT

Intel offers several levels of support for this product which are explained in detail in the price list. Please consult the price list for a description of the support options available.

*MDS is an ordering code only and is not used as a product name or trademark. MDS is a registered trademark of Mohawk Data Sciences Corporation.



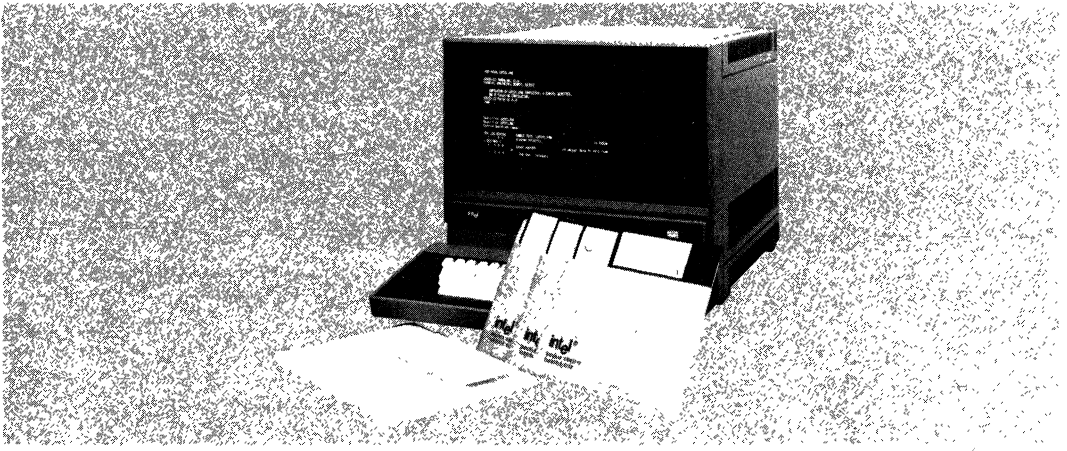
PASCAL 86/88 SOFTWARE PACKAGE

- Resident on iAPX 86 Based Intel Microcomputer Development Systems
- Object Compatible and Linkable with PL/M 86/88, ASM 86/88 and FORTRAN 86/88
- ICE™ Symbolic Debugging Fully Supported
- PSCOPE Source Level Debugging Fully Supported
- Implements REALMATH for Consistent and Reliable Results
- Supports large array operation
- Unlimited User Program Symbols
- Supports iAPX86/20, 88/20 Numeric Data Processors
- Strict Implementation of ISO Standard Pascal
- Useful Extensions Essential for Microcomputer Applications
- Separate Compilation with Type-Checking Enforced Between Pascal Modules
- Compiler Option to Support Full Run-Time Range-Checking

PASCAL 86/88 conforms to and implements the ISO Draft Proposed Pascal standard. The language is enhanced to support microcomputer applications with special features, such as separate compilation, interrupt handling and direct port I/O. To assist the development of portable software, the compiler can be directed to flag all non-standard features.

The PASCAL 86/88 compiler runs on Series III and Series IV Microcomputer Development Systems. A well-defined I/O interface is provided for run-time support. This allows a user-written operating system to support application programs as an alternate to the development system environment. Program modules compiled under PASCAL 86/88 are compatible and linkable with modules written in PL/M 86/88, ASM 86/88 or FORTRAN 86/88. With a complete family of compatible programming languages for the iAPX 86, 88, 186, 188 one can implement each module in the language most appropriate to the task at hand.

PASCAL 86/88 object modules contain symbol and type information for program debugging using ICE™ emulators and PSCOPE source language debugger. For final production version, the compiler can remove this extra information and code.



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. Information Contained Herein Supercedes Previously Published Specifications of These Devices from Intel.

©INTEL CORPORATION, 1983

JUNE 1984

FEATURES

Includes all the language features of Jensen & Wirth Pascal as defined in the ISO Draft Proposed Pascal Standard.

Supports required extensions for microcomputer applications.

- Interrupt handling
- Direct port I/O

Separate compilation extensions allow:

- Modular decomposition of large programs
- Linkage with other Pascal modules as well as PL/M 86/88/186/188, ASM 86/88/186/188 and FORTRAN 86/88.
- Enforcement of type-checking at LINK-time

Supports numerous compiler options to control the compilation process, to INCLUDE files, flag non-standard Pascal statements and others to control program listings and object modules.

Utilizes the IEEE standard for Floating-Point Arithmetic (the Intel REALMATH standard) for arithmetic operations.

Well-defined and documented run-time operating system interfaces allow the user to execute the applications under user-designed operating systems.

Predefined type extensions allow:

- Create precision in read, integer, and unsigned calculations.
- Means to check 8087 errors
- Circumvention of rigid type checking on calls to non-Pascal routines

BENEFITS

Provides a standard Pascal for iAPX 86, 88, 186, 188 based applications.

- Pascal has gained wide acceptance as the portable application language for microcomputer applications
- It is being taught in many colleges and universities around the world
- It is easy to learn, originally intended as a vehicle for teaching computer programming
- Improves maintainability: Type mechanism is both strictly enforced and user extendable
- Few machine specific language constructs

Strict implementation of the proposed ISO standard for Pascal aids portability of application programs. A compile time option checks conformance to the standard making it easy to write conforming programs.

PASCAL 86/88 extensions via predefined procedures for interrupt handling and direct port I/O make it possible to code an entire application in Pascal without compromising portability.

Standard Intel REALMATH is easy to use and provides reliable results, consistent with other Intel languages and other implementations of the IEEE proposed Floating-Point standard.

Provides run-time support for co-processors. All real-type arithmetic is performed on the 86/20 numeric data processor unit or software emulator. Run-time library routines, common between Pascal and other Intel languages (such as FORTRAN), permit efficient and consistently accurate results.

Extended relocation and linkage support allows the user to link Pascal program modules with routines written in other languages for certain parts of the program. For example, real-time or hardware dependent routines written in ASM 86/88/186/188 or PL/M 86/88/186/188 can be linked to Pascal routines, further extending the user's ability to write structured and modular programs.

PASCAL 86/88 programs "talk" to the resident operating system using Intel's standard interface for translated programs. This allows users to replace the development operating system by their own operating systems in the final application.

PASCAL 86/88 takes full advantage of iAPX 86, 88, 186, 188 high level language architecture to generate efficient machine code

Compiler options can be used to control the program listings and object modules. While debugging, the user may generate additional information such as the symbol record information required and useful for debugging using PSCOPE or ICE emulation. After debugging, the production version may be streamlined by removing this additional information.



SPECIFICATIONS

Operating Environment

Documentation Package

REQUIRED HARDWARE

PASCAL 86 User's Guide

Intel Microcomputer Development Systems (Series III, Series IV)

ORDERING INFORMATION

Part Number Description

MDS*-314 PASCAL 86/88 Software Package

Requires software license.

*MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Science.

SUPPORT:

Hotline Telephone Support, Software Performance Report (SPR), Software Updates, Technical Reports, and Monthly Technical Newsletters are available.



PL/M 86/88/186/188 Software Package

- **Systems Programming Language for the iAPX 86/88/186/188 Processors**
- **Language Is Upward Compatible from PL/M 80, Assuring MCS®-80/85 Design Portability**
- **Advanced Structured System Implementation Language for Algorithm Development**
- **Supports 16-Bit Signed Integer and 32-Bit Floating Point Arithmetic in Accordance with IEEE Proposed Standard**
- **Easy-to-Learn Block-Structured Language Encourages Program Modularity**
- **Improved Compiler Performance Now Supports More User Symbols and Faster Compilation Speeds**
- **Produces Relocatable Object Code Which Is Linkable to All Other 8086 Object Modules**
- **Code Optimization Assures Efficient Code Generation and Minimum Application Memory Utilization**
- **Built-In Syntax Checker Doubles Performance for Compiling Programs Containing Errors**
- **Resident on iAPX 86 Intel Microcomputer Development Systems**

PL/M 86 is an advanced, structured, high-level systems programming language. The PL/M 86 compiler was created specifically for performing software development for the Intel 8086, 8088, 80186 and 80188 Microprocessors. PL/M was designed so that program statements naturally express the program algorithm. This frees the programmer to concentrate on the logic of the program without concern for burdensome details of machine or assembly language programming (such as register allocation, meanings of assembler mnemonics, etc.).

The PL/M 86 compiler efficiently converts free-form PL/M language statements into machine instructions. Substantially fewer PL/M statements are necessary for a given application than if it were programmed at the assembly language or machine code level.

The use of PL/M high-level language for system programming, instead of assembly language, results in a high degree of engineering productivity during project development. This translates into significant reductions in initial software development and follow-up maintenance costs for the user.



NOTE The Inteltec® Development System pictured here is not included with the PL/M 86/88 Software package but merely depicts a language in its operating environment. The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products: BXP, CREDIT, i, iCE, iCS, iM, iMsite, Intel, INTEL, Intelevison, Intelink, Inteltec, iMMX, iOSP, iPDS, iRMX, iSBC, iSBX, Library Manager, MCS, MULTIMODULE, Megachassis, Micromainframe, MULTIBUS, Multichannel, Plug-A-Bubble, PROMPT, Promware, RUPI, RMX/80, System 2000, UPI, and the combination iCS, iRMX, iSBC, iSBX, ICE, i2ICE, MCS, or UPI and numerical suffix. Intel Corporation Assumes No Responsibility for the use of Any Circuitry Other Than Circuitry Embodied in an Intel product. No Other Patent Licenses are implied. ©INTEL CORPORATION, 1983

MAY 1983

FEATURES

Major features of the Intel PL/M 86 compiler and programming language include:

Block Structure

PL/M source code is developed in a series of modules, procedures, and blocks. Encouraging program modularity in this manner makes programs more readable, and easier to maintain and debug. The language becomes more flexible, by clearly defining the scope of user variables (local to a private procedure).

The use of procedures to break down a large problem is paramount to productive software development. The PL/M 86 implementation of a block structure allows the use of REENTRANT (recursive) procedures, which are especially useful in system design.

Language Compatibility

PL/M 86 object modules are compatible with object modules generated by all other iAPX 86 translators. This means that PL/M programs may be linked to programs written in any other iAPX 86 language.

Object modules are compatible with In-Circuit Emulators; DEBUG compiler control provides the In-Circuit Emulators with symbolic debugging capabilities.

PL/M 86 Language is upward compatible with PL/M 80, so that application programs may be easily ported to run on the iAPX 86.

Supports Seven Data Types

PL/M makes use of seven data types for various applications. These data types range from one to four bytes, and facilitate various arithmetic, logic, and addressing functions:

- Byte: 8-bit unsigned number
- Word: 16-bit unsigned number
- DWORD: 32-bit unsigned number
- Integer: 16-bit signed number
- Read: 32-bit floating point number
- Pointer: 16-bit or 32-bit memory address indicator
- Selector: 16-bit base portion of a pointer

Another powerful facility allows the use of BASED variables that map more than one variable to the same memory location. This is especially useful for passing parameters, relative and absolute addressing, and memory allocation.

Two Data Structuring Facilities

In addition to the five data types and based variables, PL/M supports two data structuring facilities. These help the user to organize data into logical groups.

- Array: Indexed list of same type data elements
- Structure: Named collection of same or different type data elements
- Combinations of Each: Arrays of structures or structures of arrays

8087 Numerics Support

PL/M programs that use 32-bit REAL data may be executed using the Numeric Data Processor for improved performance. All floating-point operations supported by PL/M may be executed on the iAPX 86/20 or 88/20 NDP, or the 8087 Emulator (a software module) provided with the package. Determination of use of the chip or Emulator takes place at linktime, allowing compilations to be run-time independent.

Built-In String Handling Facilities

The PL/M 86 language contains built-in functions for string manipulation. These byte and word functions perform the following operations on character strings: MOVE, COMPARE, TRANSLATE, SEARCH, SKIP, and SET.

Interrupt Handling

PL/M has the facility for handling interrupts. A procedure may be defined with the INTERRUPT attribute, and the compiler will automatically initialize an interrupt vector at the appropriate memory location. The compiler will also generate code to save and restore the processor status, for execution of the user-defined interrupt handler routine. The procedure SET\$INTERRUPT, the function returning an INTERRUPT\$PTR, and the PL/M statement CAUSE\$INTERRUPT all add flexibility to user programs involving interrupt and handling.

Compiler Controls

Including several that have been mentioned, the PL/M 86 compiler offers more than 25 controls that facilitate such features as:

- Conditional compilation
- Including additional PL/M source files from disk
- Corresponding assembly language code in the listing file
- Setting overflow conditions for run-time handling

Segmentation Control

The PL/M 86 compiler takes full advantage of program addressing with the SMALL, COMPACT, MEDIUM, and LARGE segmentation controls. Programs with less than 64KB total code space can exploit the most efficient memory addressing schemes, which lowers total memory requirements. Larger programs can exploit the flexibility of extended one-megabyte addressing.

Code Optimization

The PL/M 86 compiler offers four levels of optimization for significantly reducing overall program size.

- Combination or “folding” of constant expressions; and short-circuit evaluation of Boolean expressions
- “Strength reductions” (such as a shift left rather than multiply by 2); and elimination of common sub-expressions within the same block
- Machine code optimizations; elimination of superfluous branches; re-use of duplicate code; removal of unreachable code
- Byte comparisons (rather than 20-bit address calculations) for pointer variables; optimization of based-variable operations

Error Checking

The PL/M 86 compiler has a very powerful feature to speed up compilations. If a syntax or program error is detected, the compiler will skip the code generation and optimization passes. This usually yields a 2X performance increase for compilation of programs with errors.

A fully detailed set of programming and compilation errors is provided by the compiler.

```

M DO, /* Beginning of module */
SORTPROC PROCEDURE (PTR, COUNT, RECSIZE, KEYINDEX) (PUBLIC);
  DECLARE PTR POINTER, (COUNT, RECSIZE, KEYINDEX) INTEGER
/* Parameters
PTR is pointer to first record
COUNT is number of records to be sorted
RECSIZE is number of bytes in each record—max is 128
KEYINDEX is byte position within each record of a BYTE scalar
to be used as sort key */
  DECLARE (RECORD BASED PTR) (1) BYTE
  CURRENT (128) BYTE
  (I J) INTEGER.
SORT DO J 1 TO COUNT-1
  CALL MOVB(@RECORD(J*RECSIZE) @CURRENT RECSIZE)
  I J
FIND DO WHILE I 0
  AND RECORD((I 1)*RECSIZE KEYINDEX)
  CURRENT(KEYINDEX)
  CALL MOVB(@RECORD((I 1)*RECSIZE)
  @RECORD(I*RECSIZE)
  RECSIZE)
  I I 1
  END FIND
  CALL (MOVB) (@CURRENT @RECORD(I*RECSIZE) RECSIZE)
END SORT
END SORTPROC
END M
  'End of module'
  
```

PUBLIC and EXTERNAL attributes promote program modularity

Based Variables allow manipulation of external data by passing the base of the data structure (a pointer). This minimizes the STACK space used for parameter passing and the execution time to perform many STACK operations

The AT operator returns the address of a variable, instead of its contents. This is very useful in passing pointers for based variables

One of several PL/M built-in procedures for string manipulation

Figure 3. Sample PL/M 86 Program.



BENEFITS

PL/M 86 is designed to be an efficient, cost-effective solution to the special requirements of iAPX 86 Microsystem Software Development, as illustrated by the following benefits of PL/M use:

Cost-Effective Alternative to Assembly Language

PL/M 86 programs are code efficient. PL/M 86 combines all of the benefits of a high-level language (ease of use, high productivity) with the ability to access the iAPX 86 architecture. Consequently, for the development of systems software, PL/M 86 is the cost-effective alternative to assembly language programming.

Low Learning Effort

PL/M is easy to learn and to use, even for the novice programmer.

Earlier Project Completion

Critical projects are completed much earlier than otherwise possible because PL/M 86, a structured high-level language, increases programmer productivity.

Lower Development Cost

Increases in programmer productivity translate immediately into lower software development costs because fewer programming resources are required for a given programmed function.

Increased Reliability

PL/M 86 is designed to aid in the development of reliable software (PL/M 86 programs are simple statements of the program algorithm). This substantially reduces the risk of costly correction of errors in systems that have already reached full production status, as the more simply stated the program is, the more likely it is to perform its intended function.

Easier Enhancements and Maintenance

Programs written in PL/M tend to be self-documenting, thus easier to read and understand. This means it is easier to enhance and maintain PL/M programs as the system capabilities expand and future products are developed.

SPECIFICATIONS

Operating Environment

REQUIRED HARDWARE:

Intel Microcomputer Development Systems (Series III/Series IV)

Documentation Package

PL/M-86 User's Guide for 8086-based Development Systems (121636)

SUPPORT:

Hotline Telephone Support, Software Performance Reporting (SPR), Software Updates, Technical Reports, Monthly Newsletter available.

ORDERING INFORMATION

Part Number	Description
MDS-313*	PL/M 86 Software Package

Requires Software License

MDS is an ordering code only and is not used as a product name or trademark. MDS is a registered trademark of Mohawk Data Sciences Corporation.



iC-86 C COMPILER FOR THE 8086

- Implements full C Language
- Produces high density code rivaling assembler
- Supports Intel Object Module Format (OMF)
- Runs under the Intel UDI on Intel Development Systems and iRMX™ 86
- Available for the VAX/VMS* Operating System
- Supports both small and large models of computation
- Supports PSCOPE-86 and I²ICE™
- Supports IEEE Floating Point Math with 8087 coprocessor
- Supports Bit Fields
- Supports full standard I/O Library (STDIO)
- Written in C

The C Programming Language was originally designed in 1972 and has become increasingly popular as a systems development language. C is not a "very high level" language and is not tied to any specific application area. Although it is used for writing operating systems, it has been used equally well to write numerical, text-processing and data base programs. C combines the flexibility and programming speed of a higher level language with the efficiency and control of assembly language.

Intel iC-86 brings the full power of the C programming language to 8086 and 8088 based microprocessor systems.

Intel iC-86 supports the full C language as described in the Kernighan and Ritchie book, "The C Programming Language," (Prentice-Hall, 1978). Also included are the latest enhancements to the C language: structure assignments, functions taking structure arguments and returning structures, and the "void" and "enum" data types.

C is rapidly becoming the standard microprocessor system implementation language because it provides:

1. the ability to manipulate the fundamental objects of the machine (including machine addresses) as easily as assembly language.
2. the power and speed of a structured language supporting a large number of data types, storage classes, expressions and statements,
3. processor independence (most programs developed for other processors can be easily transported to the 8086), and
4. code that rivals assembly language in efficiency

INTEL iC-86 COMPILER DESCRIPTION

The iC-86 compiler operates in four phases: preprocessor, parser, code generator, and optimizer. The preprocessor phase interprets directives in C source code, including conditional compilations (`# define`). The parser phase converts the C program into an intermediate free form and does all syntactic and

semantic error checking. The code generator phase converts the parser's output into an efficient intermediate binary code, performs constant folding, and features an extremely efficient register allocator, ensuring high quality code. The optimizer phase converts the output of the code generator into

relocatable Intel Object Module Format (OMF) code, without creating an intermediate assembly file. Optionally, the iC-86 compiler can produce a symbolic assembly like file. The iC-86 optimizer eliminates common code, eliminates redundant loads and stores, and resolves span dependencies (shortens branches) within a program.

The iC-86 runtime library consists of a number of functions which the C programmer can call. The runtime system includes the standard I/O library

(STDIO), conversion routines, routines for manipulating strings, special routines to perform functions not available on the 8086 (32-bit arithmetic and emulated floating point), and (where appropriate) routines for interfacing with the operating system.

iC-86 uses Intel's linker and locator and generates debug records for symbols and lines on request, permitting access to Intel's PSCOPE AND I²ICET[™] to aid in program testing.

FEATURES

Support for Small and Large Models

Intel iC-86 supports both the SMALL and LARGE modes of segmentation. A SMALL model program can have up to 64K bytes of code and 64K bytes of data, with all pointers occupying two bytes. Because two byte pointers permit the generation of highly compact and efficient code, this model is recommended for programs that can meet the size restrictions. The LARGE segmentation model is used by programs that require access to the full addressing space of the 8086/8088 processors. In this model, each source file generates a distinct pair of code and data segments of up to 64K bytes in length. All pointers are four bytes long.

Preprocessor Directives

#define—defines a macro

#include—includes code outside of the program source file

#if—conditionally includes or excludes code

Other preprocessor directives include **#undef**, **#ifdef**, **#ifndef**, **#else**, **#endif**, and **#line**.

Statements

The C language supports a variety of statements:

Conditionals: IF, IF-ELSE

Loops: WHILE, DO-WHILE, FOR

Selection of cases: SWITCH, CASE, DEFAULT

Exit from a function: RETURN

Loop control: CONTINUE, BREAK

Branching: GOTO

Expressions and Operators

The C language includes a rich set of expressions and operators.

Primary expression: invoke functions, select ele-

ments from arrays, and extract fields from structures or unions

Arithmetic operators: add, subtract, multiply, divide, modulus

Relational operators: greater than, greater than or equal, less than, less than or equal, not equal

Unary operators: indirect through a pointer, compute an address, logical negation, ones complement, provide the size in bytes of an operand.

Logical operators: AND, OR

Bitwise operators: AND, exclusive OR, inclusive OR, bitwise complement

Data Types and Storage Classes

Data in C is described by its type and storage class. The type determines its representation and use, and the storage class determines its lifetime, scope, and storage allocation. The following data types are fully supported by iC-86.

char

an 8 bit signed integer

int

a 16 bit signed integer

short

same as int (on the 8086)

long

a 32 bit signed integer

unsigned

a modifier for integer data types (char, int, short, and long) which doubles the positive range of values

float

a 32 bit floating point number which utilizes the 8087 or a software floating point library

double

a 64 bit floating point number

void

a special type that cannot be used as an operand in expressions; normally used for functions called only for effect (to prevent their use in contexts where a value is required).

enum

an enumerated data type

These fundamental data types may be used to create other data types including: arrays, functions, structures, pointers, and unions.

The storage classes available in iC-86 include:

register

suggests that a variable be kept in a machine register, often enhancing code density and speed

extern

a variable defined outside of the function where it is declared; retaining its value throughout the entire program and accessible to other modules

auto

a local variable, created when a block of code is entered and discarded when the block is exited

static

a local variable that retains its value until the termination of the entire program

typedef

defines a new data type name from existing data types

BENEFITS

Faster Compilation

Intel iC-86 compiles C programs substantially faster than standard C compilers because it produces Intel OMF code directly, eliminating the traditional intermediate process of generating an assembly file.

Portability of Code

Because Intel iC-86 supports the STDIO and produces Intel OMF code, programs developed on a variety of machines can easily be transported to the 8086.

Rapid Program Development

Intel iC-86 provides the programmer with detailed error messages and access to PSCOPE-86 and i2ICE™ to speed program development.

Full Manipulation of the 8086

Intel iC-86 enables the programmer to utilize features of the C language to control bit fields, pointers, addresses and register allocation, taking full advantage of the fundamental concepts of the 8086.

SPECIFICATIONS

Operating Environment

The iC-86 compiler runs host resident on both the Intel Series III Microcomputer Development System under ISIS-II and on the System 86/330 under the iRMX™ 86 operating system. iC-86 can also run as a cross compiler on a VAX 11/780 computer under the VMS operating system 128 KBytes of User Memory is required on all versions. Specify desired version when ordering.

Required Hardware

Development System Version

—Intellec® Microcomputer Development System; Series III or Series IV

—Dual Diskette Drives, Single or Double Density
—System Console; CRT or Hardcopy Interactive Device

iRMX 86 version:

—Any iAPX 86/88, iSBC® 86/88, iTPS 86/XXX, or SYS 86/3XX based system capable of running the iRMX 86 Operating System

VAX version:

—Digital Equipment Corporation VAX 11/780 or compatible computer

Optional Hardware

ISIS-II version:

- ICE-86, I²ICE-86

iRMX 86 version:

- Numeric Data Processors for support of the REALMATH standard

VAX version:

- None

Required Software

ISIS-II version:

- ISIS-II Diskette Operating System
- Series III or Series IV Operating System

iRMX 86 version:

- iRMX 86 Realtime Multiprogramming Operating System
- iRMX 860 Utilities Package

VAX version:

- VMS Operating System

Optional Software

Development System Version:

- None

iRMX 86 version:

- None

VAX version:

- MDS*-384 Kit-Mainframe Link for distributed development, or iMDX-394 Asynchronous Communications Link.
- VAX iAPX 86/88/186 MACRO Assembler and utilities package (iMDX-341VX)

Documentation Package

The C Programming Language by Kernighan and Ritchie (1978 Prentice-Hall)

iC-86 User Manual

Shipping Media

Development System Version:

- Two single and one double density ISIS-II format 8" diskettes, one 5 1/4" Series IV Format

iRMX 86 version:

- Double Density iRMX 86 format 8" diskette
- Double Density iRMX 86 format 5 1/4" diskette

VAX version:

- 1600 bpi, 9 track Magnetic tape

ORDERING INFORMATION

Order Code	Description
iMDX-317	iC-86 Compiler for ISIS-II
iRMX-866	iC-86 Compiler for iRMX 86
iMDX-347	iC-86 Cross Compiler for VAX/VMS

Intel Software License required

SUPPORT

Intel offers several levels of support for this product which are explained in detail in the price list. Please consult the price list for a description of the support options available.

*MDS is an ordering code only and is not used as a product name or trademark. MDS is a registered trademark of Mohawk Data Sciences Corporation.

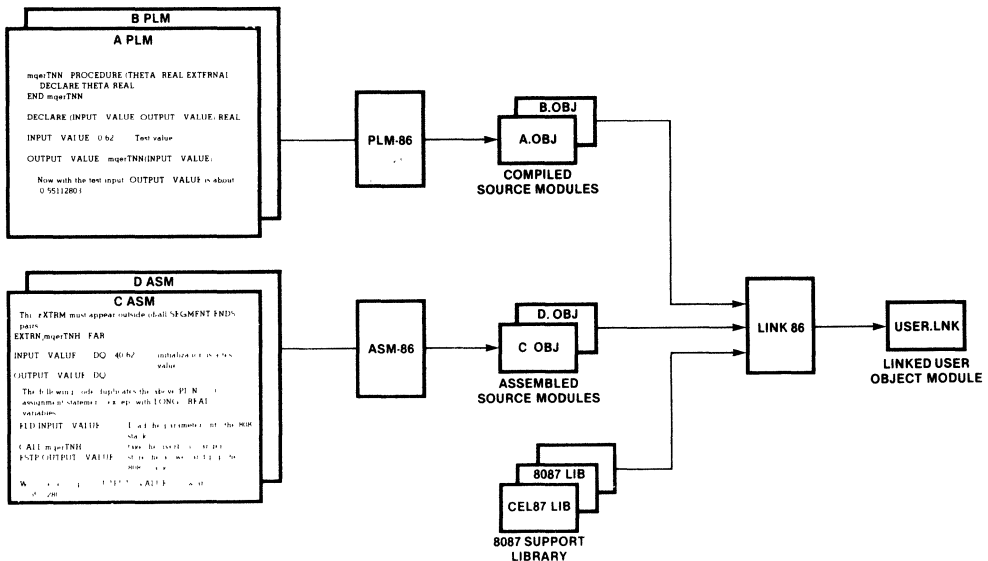


8087 SUPPORT LIBRARY

- Library to support floating point arithmetic in PL/M-86 and ASM-86
- Full 8087 Software Emulator for software debugging without the 8087 component
- Common elementary function library provides trigonometric, logarithmic and other useful functions
- Accurate, verified and efficient implementation of algorithms for functions
- Decimal conversion module supports binary-decimal conversions
- Supports proposed IEEE Floating Point Standard for high accuracy and software portability
- Error-handler module simplifies floating point error recovery

The 8087 Support Library provides PL/M-86 and ASM-86 users with the equivalent numeric data processing capability of Fortran-86. With the Library, it is easy for PL/M-86 and ASM-86 programs to do floating point arithmetic. Programs can link in modules to do trigonometric, logarithmic and other numeric functions, and the user is guaranteed accurate, reliable results for all appropriate inputs. The 8087 Support Library implements Intel's REALMATH standard and also supports the proposed IEEE Floating Point Standard. Consequently, by using this Library, the PL/M-86 user not only saves software development time, but is guaranteed that the numeric software meets industry standards and is portable—his software investment is maintained.

The 8087 Support Library consists of the common elementary function library, the decimal conversion module, the error handler module, the full 8087 Software emulator and interface libraries to the 8087 and to the 8087 emulator.



CEL87.LIB

THE COMMON ELEMENTARY FUNCTION LIBRARY

CEL87.LIB contains commonly used floating point functions. It is used along with the 8087 numeric coprocessor or the 8087 emulator and it provides a complete package of elementary functions, giving valid results for all appropriate inputs. This library provides PL/M-86 and ASM-86 users all the math functions supported intrinsically by the Fortran-86. Following is a summary of CEL87 functions, grouped by functionality.

Rounding and Truncation Functions:

mqrIEX, mqrIE2, and mqrIE4 round a real number to the nearest integer; to the even integer if there is a tie. The answer returned is real, a 16-bit integer or a 32-bit integer respectively.

mqrIAX, mqrIA2, mqrIA4 round a real number to the nearest integer, to the integer away from zero if there is a tie; the answer returned is real, a 16-bit integer or a 32-bit integer, respectively.

mqrICX, mqrIC2, mqrIC4 truncate the fractional part of a real input; the answer is real, a 16-bit integer or a 32-bit integer, respectively.

Logarithmic and Exponential Functions:

mqrLGD computes decimal (base 10) logarithms.

mqrLGE computes natural (base e) logarithms.

mqrEXP computes exponentials to the base e.

mqrY2X computes exponentials to any base.

mqrYI2 raises an input real to a 16-bit integer power.

mqrYI4 is as mqrYI2, except to a 32-bit integer power.

mqrYIS is as mqrYI2, but it accommodates PL/M-86 users.

Trigonometric and Hyperbolic Functions:

mqrSIN, mqrCOS, mqrTAN compute sine, cosine, and tangent.

mqrASN, mqrACS, mqrATN compute the corresponding inverse functions.

mqrSNH, mqrCSH, mqrTNH compute the corresponding hyperbolic functions.

mqrAT2 is a special version of the arc tangent function that accepts rectangular coordinate inputs.

Other Functions:

mqrDIM is FORTRAN's positive difference function.

mqrMAX returns the maximum of two real inputs.

mqrMIN returns the minimum of two real inputs.

mqrSGH combines the sign of one input with the magnitude of the other input.

mqrMOD computes a modulus, retaining the sign of the dividend.

mqrRMD computes a modulus, giving the value closest to zero.

DCON87.LIB

THE DECIMAL CONVERSION LIBRARY

DCON87.LIB is a library of procedures which convert binary representations of floating point numbers and ASCII-encoded string of digits.

The binary-to-decimal procedure mqcBIN DECLOW accepts a binary number in any of the formats used for the representation of floating point numbers in the 8087. Because there are so many output formats for floating point numbers, mqcBIN_DECLOW does not attempt to provide a finished, formatted text string. Instead, it provides the "building blocks" for you to use to construct the output string which meets your exact format specification.

The decimal-to-binary procedure `mqcDEC_BIN` accepts a text string which consists of a decimal number with optional sign, decimal point, and/or power-of-ten exponent. It translates the string into the caller's choice of binary formats.

Decimal-to-binary procedure `mqcDELOW_BIN` is provided for callers who have already broken the decimal number into its constituent parts.

The procedures `mqcLONG_TEMP`, `mqcSHORT_TEMP`, `mqcTEMP_LONG`, and `mqcTEMP_SHORT` convert floating point numbers between the longest binary format, `TEMP_REAL`, and the shorter formats.

EH87.LIB THE ERROR HANDLER MODULE

EH87.LIB is a library of five utility procedures which a user can utilize for writing trap handlers. Trap handlers are called when an unmasked 8087 error occurs.

The 8087 error reporting mechanism can be used not only to report error conditions, but also to let software implement IEEE standard options not directly supported by the chip. The three such extensions to the 8087 are: normalizing mode, non-trapping not-a-number (NaN), and non-ordered comparison. The utility procedures support these extra features.

DECODE is called near the beginning of the trap handler. It preserves the complete state of the 8087, and also identifies what function called the trap handler, and returns available arguments and/or results. DECODE eliminates much of the effort needed to determine what error caused the trap handler to be called.

NORMAL provides the "normalizing mode" capability for handling the "D" exception. By calling NORMAL in your trap handler, you eliminate the need to write code in your application program which tests for non-normal inputs.

SIEVE provides two capabilities for handling the "I" exception. It implements non-trapping NaN's and non-ordered comparisons. These two IEEE standard features are useful for diagnostic work.

ENCODE is called near the end of the trap handler. It restores the state of the 8087 saved by DECODE, and performs a choice of concluding actions, by either retrying the offending function or returning a specified result.

FILTER calls each of the above four procedures. If your error handler does nothing more than detect fatal errors and implement the features supported by SIEVE and NORMAL, then your interface to EH87.LIB can be accomplished with a single call to FILTER.

E8087 THE FULL 8087 EMULATOR

E8087 is an object module that functionally emulates the 8087 coprocessor chip. It is ideal for use during prototyping and debugging floating point programs. However, the target system should use the 8087 component because it executes 1000 times faster and uses significantly less memory.

E8087.LIB, 8087.LIB, 87NULL.LIB INTERFACE LIBRARIES

E8087. LIB, 8087.LIB and 87NULL. LIB libraries configure a user's application program for his run-time environment: running with the emulator, with the 8087 component or without floating point arithmetic, respectively.

SPECIFICATIONS

TARGET ENVIRONMENT

8086/8088 Based Microcomputer System

Required Software

For Series II:
8086/8088 Software Development Package

DEVELOPMENT ENVIRONMENT

Required Hardware

All Intel Microcomputer Development Systems (Series II, Series III/Series IV)

Documentation Package

Numeric Support Library Manual

*Recommended

ORDERING INFORMATION

Part Number	Description
MDS*-319	8087 Support Library

Requires Software License

SUPPORT

Intel offers several levels of support for this product which are explained in detail in the price list. Please consult the price list for a description of the support options available.

*MDS is an ordering code only and is not used as a product name or trademark. MDS is a registered trademark of Mohawk Data Sciences Corporation.



8087 SOFTWARE SUPPORT PACKAGE

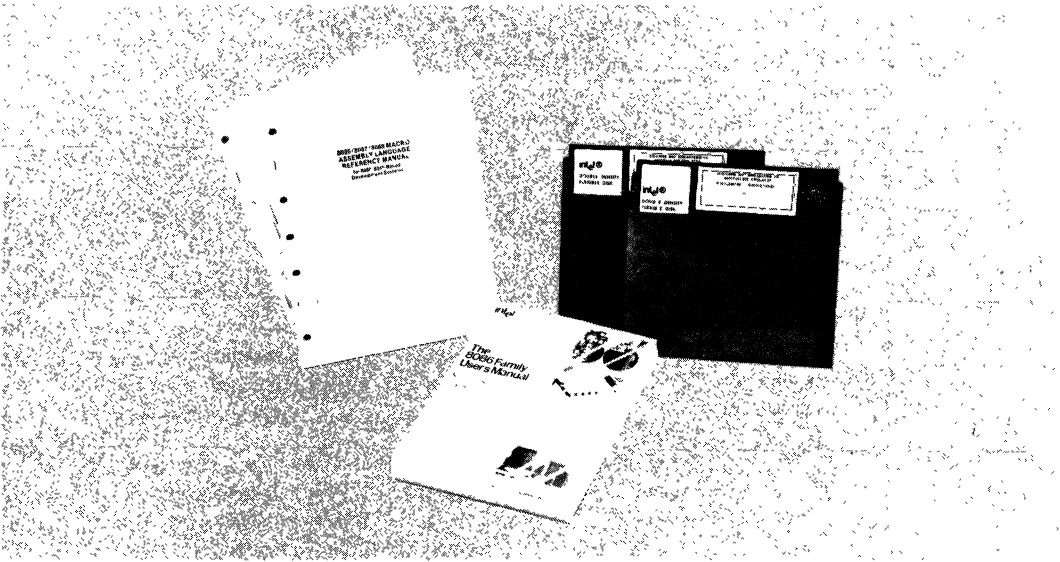
- **Program Generation for the 8087 Numeric Data Processor on 8080/8085 Based Intel Microcomputer Development Systems**
- **Consists of: 8086/8087/8088 Macro Assembler, 8087 Software Emulator**
- **Macro Assembler Generates Code for 8087 Processor or Emulator, While Also Supporting the 8086/8088 Instruction Set**
- **8087 Emulator Duplicates Each 8087 Floating-Point Instruction in Software, for Evaluation of Prototyping, or for Use in an End Product**
- **Macro Assembler and 8087 Emulator are Fully Compatible with Other 8086/8088 Development Software**
- **Implementation of the IEEE Proposed Floating-Point Standard (the Intel® Realmath Standard)**

The 8087 Software Support Package is an optional extension of Intel's 8086/8088 Software Development Package.

The 8087 Software Support Package consists of the 8086/8087/8088 Macro Assembler, and the Full 8087 Emulator. The assembler is a functional superset of the 8086/8088 Macro Assembler, and includes instructions for over sixty new floating-point operations, plus new data types supported by the 8087.

The 8087 Emulator is an 8086/8088 object module that simulates the environment of the 8087, and executes each floating-point operation using software algorithms. This emulator functionally duplicates the operation of the 8087 Numeric Data Processor.

Also included in this package are interface libraries to link with 8086/8087/8088 object modules, which are used for specifying whether the 8087 Processor or the 8087 Emulator is to be used. This enables the run-time environment to be invisible to the programmer at assembly time.



The following are trademarks of Intel Corporation and may be used only to identify Intel products: BXP, CREDIT, Intellec, Multibus, i, iSBC, Multimodule, ICE, iSBX, PROMPT, iRMX, iCS, Library Manager, Promware, Insite, MCS, RMX, Intel, Megachassis, UPI, Intelevison, Micromap, μ Scope and the combination of iCE, iCS, iSBC, iSBX, MCS, or RMX and a numerical suffix.

© INTEL CORPORATION, 1983

FUNCTIONAL DESCRIPTION

8086/8087/8088 Macro Assembler

The 8086/8087/8088 Macro Assembler translates symbolic macro assembly language instructions into appropriate machine instructions. It is an extended version of the 8086/8088 Macro Assembler, and therefore supports all of the same features and functions, such as limited type checking, conditional assembly, data structures, macros, etc. The extensions are the new instructions and data types to support floating-point operations. Realmath floating-point instructions (see Table 1) generate code capable of being converted to either 8087 instructions or interrupts for the 8087 Emulator. The Processor/Emulator selection is made via interface libraries at LINK-time. In addition to the new

floating-point instructions, the macro assembler also introduces two new 8087 data types: QWORD (8 bytes) and TBYTE (ten bytes). These support the highest precision of data processed by the 8087.

Full 8087 Emulator

The Full 8087 Emulator is a 16-kilobyte object module that is linked to the application program for floating-point operations. Its functionality is identical to the 8087 chip, and is ideal for prototyping and debugging floating-point applications. The Emulator is an alternative to the use of the 8087 chip, although the latter executes floating-point applications up to 100 times faster than an 8086 with the 8087 Emulator. Furthermore, since the 8087 is a "co-processor," use of the chip will allow many operations to be performed in parallel with the 8086.

Table 1. 8087 Instructions

Arithmetic Instructions

Addition	
FADD FADDP FIADD	Add real Add real and pop Integer add
Subtraction	
FSUB FSUBP FISUB FSUBR FSUBRP FISUBR	Subtract real Subtract real and pop Integer subtract Subtract real reversed Subtract real reversed and pop Integer subtract reversed
Multiplication	
FMUL FMULP FIMUL	Multiply real Multiply real and pop Integer multiply
Division	
FDIV FDIVP FIDIV FDIVR FDIVRP FIDIVR	Divide real Divide real and pop Integer divide Divide real reversed Divide real reversed and pop Integer divide reversed
Other Operations	
FSQRT FSCALE FPREM FRNDINT EXTRACT FABS FCHS	Square root Scale Partial remainder Round to integer Extract exponent and significand Absolute value Change sign

Processor Control Instructions

FINIT/FNINIT	Initialize processor
FDISI/FNDISI	Disable interrupts
FENI/FNENI	Enable interrupts
FLDCW	Load control word
FSTCW/FNSTCW	Store control word
FSTSW/FNSTSW	Store status word
FCLEX/FNCLEX	Clear exceptions
FSTENV/FNSTENV	Store environment
FLDENV	Load environment
FSAVE/FNSAVE	Save state
FRSTOR	Restore state
FINCSTP	Increment stack pointer
FDECSTP	Decrement stack pointer
FFREE	Free register
FNOP	No operation
FWAIT	CPU wait

Comparison Instructions

FCOM	Compare real
FCOMP	Compare real and pop
FCOMPP	Compare real and pop twice
FICOM	Integer compare
FICOMP	Integer compare and pop
FTST	Test
FXAM	Examine

Table 1. 8087 Instructions (cont'd)
Transcendental Instructions

FPTAN	Partial tangent
FPATAN	Partial arctangent
F2XM1	$2^x - 1$
FYL2X	$Y \bullet \log_2 X$
FYL2XP1	$Y \bullet \log_2(X + 1)$

Constant Instructions

FLDZ	Load +00
FLD1	Load +10
FLDPI	Load π
FLDL2T	Load $\log_2 10$
FLDL2E	Load $\log_2 e$
FLDLG2	Load $\log_{10} 2$
FLDLN2	Load $\log_e 2$

Data Transfer Instructions

Real Transfers	
FLD	Load real
FST	Store real
FSTP	Store real and pop
FXCH	Exchange registers
Integer Transfers	
FILD	Integer load
FIST	Integer store
FISTP	Integer store and pop
Packed Decimal Transfers	
FBLD	Packed decimal (BCD) load
FBSTP	Packed decimal (BCD) store and pop

SPECIFICATIONS
Operating Environment
REQUIRED HARDWARE

- Intel Microcomputer Development Systems
- Series II
- Personal Development System
- Series IV

REQUIRED SOFTWARE

8086/8088 Software Development Package

Documentation Package

8086/8087/8088 Macro Assembly Language Reference Manual for 8080/8085-Based Development Systems

8086/8087/8088 Macro Assembler Operating Instructions for 8080/8085-Based Development Systems

The 8086 Family Users Manual Supplement for the 8087 Numeric Data Processor

ORDERING INFORMATION
Part Number Description

MDS*-387 8087 Software Support Package

Requires Software License

SUPPORT

Intel offers several levels of support for this product which are explained in detail in the price list. Please consult the price list for a description of the support options available.

*MDS is an ordering code only and is not used as a product name or trademark. MDS is a registered trademark of Mohawk Data Sciences Corporation.

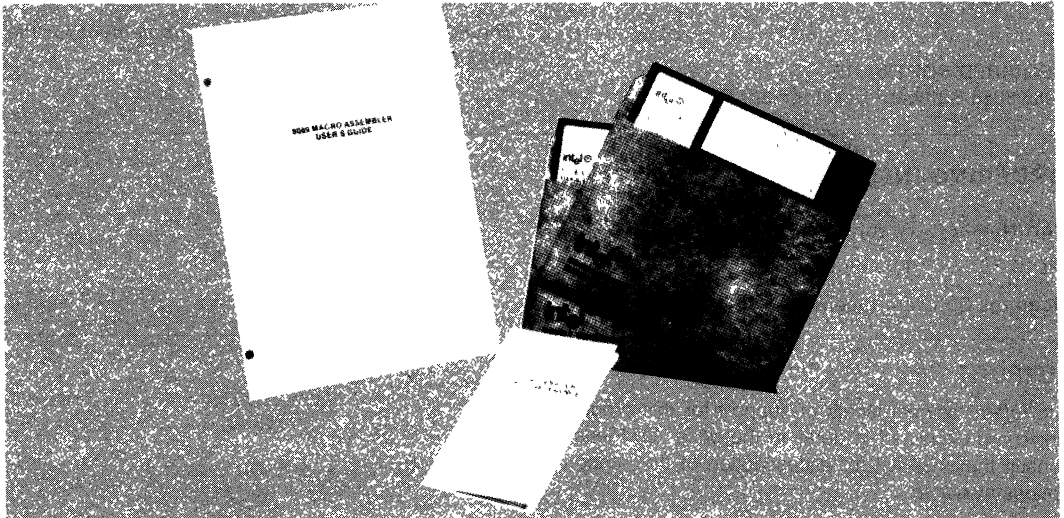


8089 IOP SOFTWARE SUPPORT PACKAGE #407200

- Program Generation for the 8089 I/O Processor on the Intellec® Microcomputer Development System
- Contains 8089 Macro Assembler, plus Relocation and Linkage Utilities
- Relocatable Object Module Compatible with All iAPX 86 and iAPX 88 Object Modules
- Fully Supports Symbolic Debugging with the RBF-89 Software Debugger
- Supports 8089-Based Addressing Modes with a Structure Facility that Enables Easy Access to Based Data
- Powerful Macro Capabilities
- Provides Timing Information in Assembly Listing
- Fully Detailed Set of Error Messages

The IOP Software Support Package extends Intellec Microcomputer Development System support to the 8089 I/O Processor. The macro assembler translates symbolic 8089 macro assembly language instructions into relocatable machine code. The relocation and linkage utilities provide compatibility with iAPX 86, iAPX 88, and 8089 modules, and make structured, modular programming easier.

The macro assembler also provides symbolic debugging capability when used with the RBF-89 software debugger. 8089 program modularity is supported with inter-segment jumps and calls. The macro assembler also provides instruction cycle counts in the listing file, for giving the programmer execution timing information. The programs in the 8089 Software Support Package run on any Intellec Series II or Model 800 with 64K bytes of memory.



The following are trademarks of Intel Corporation and may be used only to identify Intel products: BXP, CREDIT, Intellec, Multibus, i, iSBC, Multimodule, ICE, iSBX, PROMPT, iCS, iRMX, Library Manager, Promware, Insite, MCS, RMX, Intel, Megachassis, UPI, Intelevison, Micromap, μ Scope and the combination of iCE, iSBC, iSBX, MCS, or RMX and a numerical suffix

© INTEL CORPORATION 1983

MAY 1983

ORDER NUMBER:210853-002

Table 1. Sample Program Listing

8089 MACRO ASSEMBLER

I319-11 8089 MACRO ASSEMBLER X105 ASSEMBLY OF MODULE TASK
 OBJECT MODULE PLACED IN F1 TASK OBJ
 ASSEMBLER INVOKED BY: as899 f1:task a89 gen macro debug pagewidth(132) print:(f1:taskx.lst)

LOC	OBJECT CODE	TIMING	INC	MAC	LINE	SOURCE
					1
					2	*
					3	* 8089 TASK PROGRAM *
					4	*
					5
					6	b
0000					7	name TASK
					8	TASK segment
					9	a
					10	
					11	In the first part of this sample program data is moved from
					12	8086 system RAM to memory local to the 8089 IOP. In the second
					13	part, the data is moved from the local memory to a data port
					14	also in the 8089 I/O space
					15	data@port@8251 equ bc888h ;8251 DP on 8089 local bus
0000					16	command@port@8251 equ bc881h ;8251 CP on 8089 local bus
0001					17	buffer@8889 equ b288h ;RAM buffer in 8089 I/O space
					18	
					19	extrn buffer@8886 ;RAM buffer in 8086 system memory
					20	extrn y ;location of the buffer count
					21	
					22	%define (macro_1)
					23	(mov gb,buffer@8889 ; Move buffer address into GB
					24	lpd gc,y ; Load pointer to count into GC
					25	movb bc,[gc] ; Move byte count into BC
					26)
					27	%define (macro_2(param_1, param_2)) local loop
					28	(inc %param_1 ; Increment pointer into source
					29	dec %param_2 ; Decrement byte count
					30	jnz %param_2,%loop ; Loop back if byte count > 0
					31)
0000	1180 00000000	38	46		32	ONE: lpd ga,buffer@8886 ; Load register GA with address
					33	%macro_1
0006	3130 0002	47	71	+1	34	mov gb,buffer@8889 ; Move buffer address into GB
000A	5180 00000000	77	117	+1	35	lpd gc,y ; Load pointer to count into GC
0010	6002	92	139	+1	36	movb bc,[gc] ; Move byte count into BC
					37	%macro_2
0012	0090 00CD	124	185		38	loop00: movb [gb],[ga] ; Move byte from 8086 to 8089 buffer
0016	0030	134	202		39	inc ga ; Increment pointer into 8086 buffer
					40	%macro_2(gb,gc)
0018	2030	144	219	+1	41	inc %PARAM_1 ; Increment pointer into source
					42	gb ; Increment pointer into source
001A	403C	154	236	+1	43	dec %PARAM_2 ; Decrement byte count
					44	gc ; Decrement byte count
001C	4040 F3	173	259	+1	45	jnz %PARAM_2
					46	gc,%LOOP
					47	LOOP00 ; Loop back if byte count > 0
					48	%macro_2
					49	inc %PARAM_1 ; Increment pointer into source
					50	gb ; Increment pointer into source
001F	1130 00C0	191	284		51	TWO: mov ga,data@port@8251 ; load GA with address of 8251 DP
0023	1130 01C0	209	309		52	mov gc,command@port@8251 ; load GC with address of 8251 CP
					53	%macro_1
0027	3130 0002	227	334	+1	54	mov gb,buffer@8889 ; Move buffer address into GB
0028	5180 00000000	262	398	+1	55	lpd gc,y ; Load pointer to count into GC
0031	6002	281	402	+1	56	movb bc,[gc] ; Move byte count into BC
					57	%macro_2
0033	000A FD	306	434		58	loopB1: jnbt [gc],B,loopB1 ; loop until 8251 transmit ready
0036	0091 00CC	338	488		59	movb [ga],[gb] ; move message into buffer
					60	%macro_2(gb,gc)
0034	2030	348	497	+1	61	inc %PARAM_1 ; Increment pointer into source
					62	gb ; Increment pointer into source
003C	403C	358	514	+1	63	dec %PARAM_2 ; Decrement byte count
					64	gc ; Decrement byte count
003E	4040 F2	377	537	+1	65	jnz %PARAM_2
					66	gc,%LOOP
					67	LOOPB1 ; Loop back if byte count > 0
					68	%macro_2
0041	2040	319	562		69	hlt
0043					70	TASK ends
					71	END

ASSEMBLY COMPLETE, NO ERRORS FOUND

FUNCTIONAL DESCRIPTION

The IOP Software Support Package contains:

- ASM89 —The 8089 Macro Assembler.
- LINK86 —Resolves control transfer references between 8089 object modules, and data references in 8086, 8088, and 8089 modules.
- LOC86 —Assigns absolute memory addresses to 8089 object modules.
- OH86 —Converts absolute object modules to hexadecimal format.
- UPM —The Universal PROM Mapper, which supports PROM programming in all iAPX 86/11 and iAPX 88/11 applications.

ASM89 translates symbolic 8089 macro assembly language instructions into the appropriate machine codes. The ability to refer to both program and data addresses with symbolic names makes it easier to develop and modify programs, and avoids the errors of hand translation.

The powerful macro facility allows frequently used code sequences to be referred to by a single name,

so that any changes to that sequence need to be made in only one place in the program. Common code sequences that differ only slightly can also be referred to with a macro call, and the differences can be substituted with macro parameters.

ASM89 provides symbolic debugging information in the object file. The RBF-89 debugger makes use of this information, so the programmer can symbolically debug 8089 programs. ASM89 also provides cycle counts for each instruction in the assembly listing file (see Table 1). These cycle counts help the programmer determine how long a particular routine or code sequence will take to execute on the 8089.

ASM89 provides relocatable object module compatibility with the 8086 and 8088 microprocessors. This object module compatibility, along with the 8086/8088 relocation and linkage utilities, facilitates the designing of iAPX 86/11 and iAPX 88/11 systems.

ASM89 fully supports the based addressing modes of the 8089. A structure facility allows the user to define a template that enables accessing of based data symbolically.

SPECIFICATIONS

Operating Environment

Intel Microcomputer Development Systems (Model 800, Series II, Series III, Series IV)

Support

Hotline Telephone Support, Software Performance Report (SPR), Software Updates, Technical Reports, and Monthly Technical Newsletters are available.

Documentation Package

- 8089 Macro Assembler User's Guide (9800938)*
- 8089 Macro Assembler Pocket Reference (9800936)*
- MCS-86 Software Development Utilities Operating Instructions for ISIS-II Users (9800639)*
- Universal PROM Programmer User's Manual (9800819)*

Shipping Media

—Single and Double Density Diskettes

ORDERING INFORMATION

Part Number Description

MDS*-312 8089 IOP Software Support Package

Requires Software License

*MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Sciences Corporation



iAPX 286 SOFTWARE DEVELOPMENT PACKAGE

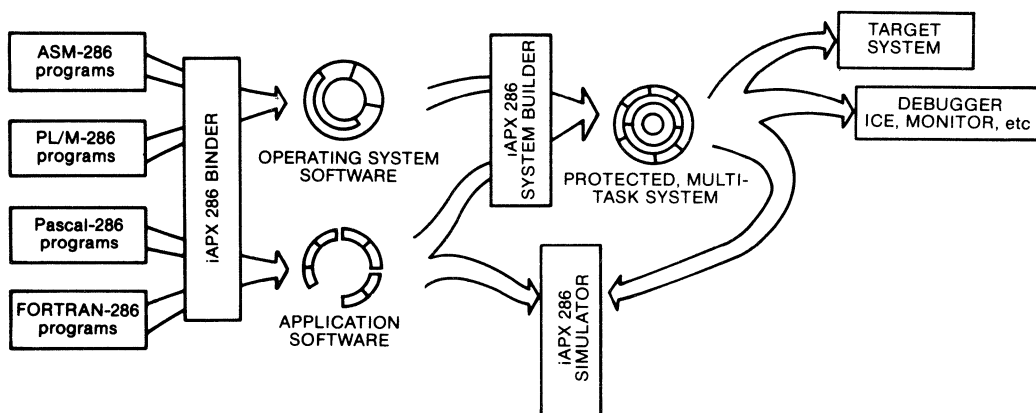
- **Complete System Development Capability for High-Performance iAPX 286 Applications.**
- **Allows creation of Multi-User, Virtual Memory, and Memory-Protected Systems.**
- **Macro Assembler for Machine-Level Programming.**
- **System Utilities for Program Linkage and System Building.**
- **Software Simulator for Execution and Symbolic Debugging on Intel Development System.**
- **Package Supports Program Development with PL/M-286, Pascal-286, and FORTRAN 286.**
- **Extends Existing Intellec® Development Systems to Provide Broad Support for the iAPX 286 Microprocessor.**

The iAPX 286 is a 16-bit microprocessor system with 32-bit virtual addressing, integrated memory protection, and instruction pipelining for high performance. The iAPX 286 Software Development Package is a cohesive set of software design aids for programming the iAPX 286 microprocessor system. The package enables system programmers to design protected, multi-user and multi-tasking operating system software, and enables application programmers to develop tasks to run on a protected operating system.

The iAPX 286 Software Development package contains a macro assembler, a program binder (for linking separately compiled modules together), a system builder (for configuring protected multiple-task systems), and a software simulator (for execution and symbolic debugging).

The memory protection features of the iAPX 286 architecture are invisible to application programmers, who use language translators and the program binder. System programmers may use special memory protection features in ASM-286 or PL/M 286, and use the system builder for initializing and managing protection features. The Simulator duplicates the operation of the 80286 CPU, as well as the floating point operations of the 80287.

All the utilities in the Software Development Package run on the Intel Microcomputer Development Systems (Series III/ Series IV)



The iAPX 286 Software Development Package keeps the protection mechanism invisible to the application programmer, yet easy to configure for the system programmer.

IAPX 286 MACRO ASSEMBLER

- **Instruction Set and Assembler Mnemonics Are Upward Compatible with ASM-86/88.**
- **Powerful and Flexible Text Macro Facility.**
- **Type-Checking at Assembly Time Helps Reduce Errors at Run-Time.**
- **Structures and RECORDS Provide Powerful Data Representation.**
- **"High-Level" Assembler Mnemonics Simplify the Language.**
- **Supports Full Instruction Set of the iAPX 286/20, Including Memory Protection and Numerics.**

ASM-286 is the "high-level" macro assembler for the iAPX 286 assembly language. ASM-286 translates symbolic assembly language mnemonics into relocatable object code. The assembler mnemonics are a superset of ASM-86/88 mnemonics; new ones have also been added to support the new iAPX 286 instructions. The segmentation directives have been greatly simplified.

The iAPX 286 assembly language includes approximately 150 instruction mnemonics. From these few mnemonics the assembler can generate over 4,000 distinct machine instructions. Therefore, the software development task is simplified, as the programmer need know only 150 mnemonics to generate all possible machine instructions. ASM-286 will generate the shortest machine instruction possible (given explicit information as to the characteristics of any forward referenced symbols).

The powerful macro facility in ASM-286 saves development and maintenance time by coding common program sequences only once. A macro substitution is made each time the sequence is to be used. This facility also allows for conditional assembly of certain program sequences.

ASM-286 offers many features normally found only in high-level languages. The assembly language is strongly typed, which means it performs extensive checks on the usage of variables and labels. This means that many programming errors will be detected when the program is assembled, long before it is being debugged.

ASM-286 object modules conform to a thorough, well-defined format used by all 286 high-level languages and utilities. This makes it easy to call (and be called from) HLL object modules.

Key Benefit:

For programmers who wish to use assembly language, ASM-286 provides many powerful "high-level" capabilities that simplify program development and maintenance.

iAPX 286 BINDER

- **Links Separately Compiled Program Modules Into an Executable Task.**
- **Makes the iAPX 286 Protection Mechanism Invisible to Application Programmers.**
- **Works with PL/M-286, Pascal-286, FORTRAN-286 and ASM-286 Object Modules.**
- **Performs Incremental Linking with Output of Binder and Builder.**
- **Resolves PUBLIC/EXTERNAL Code and Data References, and Performs Intermodule Type-Checking.**
- **Provides Print File Showing Segment Map, Errors and Warnings.**
- **Assigns Virtual Addresses to Tasks in the 2³² Address Space.**
- **Generates Linkable or Loadable Module for Debugging.**

BND-286 is a utility that combines iAPX 286 object modules into executable tasks. In creating a task, the Binder resolves Public and External symbol references, combines segments, and performs address fix-ups on symbolic code and data.

The Binder takes object modules written in ASM-286, PL/M-286, Pascal-286 or FORTRAN-286, and generates a loadable module (for execution or debugging), or a linkable module (to be re-input to the Binder later; this is called incremental binding). The binder accepts library modules as well, linking only those modules required to resolve external references. BND-286 generates a print file displaying a segment map, and error messages.

The Binder will be used by system programmers and application programmers. Since application programmers need to develop software independent of any system architecture, the 286 memory protection mechanism is "hidden" from users of the Binder. This allows application tasks to be fully debugged before becoming part of a protected system. (A protected system may be debugged, as well.) System protection features are specified later in the development cycle, using the 286 System Builder. It is possible to link operating system services required by a task using either the Binder or the Builder. This flexibility adds to the ease of use of the 286 utilities.

Key Benefit:

The Binder is the only utility an application programmer needs to develop and debug an individual task. Users of the Binder need not be concerned with the architecture of the target machine, making application program development for the 286 very simple.

iAPX 286 MAPPER

- **Flexible Utility to Display Object File Information.**
- **MAP-286 Selectively Purges Symbols from a Load Module.**
- **Provides Inter-Module Cross-Referencing for Modules Written in All Languages.**
- **Mapper Allows Users to Display:**

Protection Information:	Debug Information:
SEGMENT TABLES	MODULE NAMES
GATE TABLES	PROGRAM SYMBOLS
PUBLIC ADDRESSES	LINE NUMBERS

Key Benefit:

A cross-reference map showing references *between* modules simplifies debugging; the map also lists and controls all symbolic information in one easy-to-read place.

iAPX 286 LIBRARIAN

- **Fast, Easy Management of iAPX 286 Object Module Libraries.**
- **Only Required Modules Are Linked, When Using the Binder or Builder.**
- **Librarian Allows Users to:**
 - Create Libraries
 - Add Modules
 - Replace Modules
 - Delete Modules
 - Copy Modules from Another Library
 - Save Library Module to Object File
 - Create Backup
 - Display Module Information
(creation date, publics, segments)

Key Benefit:

Program libraries improve management of program modules, and reduce software administrative overhead.

iAPX 286 SYSTEM BUILDER

- **Supports Complete Creation of Protected, Multi-task Systems.**
- **Resolves PUBLIC/EXTERNAL Definitions (between protection levels).**
- **Supports Memory Protection by Building System Tables, Initializing Tasks, and Assigning Protection Rights to Segments.**
- **Creates a Memory Image of a 286 System for Cold-start Execution.**
- **Target System may be Boot-loadable, Programmed into ROM, or Loaded From Mass-store.**
- **Generates Print File with Command Listing and System Map.**

BLD-286 is the utility that lets system programmers configure multi-tasking, protected systems from an operating system and discrete tasks. The Builder generates a cold-start execution module, suitable for ROM-based or disk-based systems.

The Builder accepts input modules from iAPX 286 translators or the iAPX 286 Binder. It also accepts a "Build File" containing definitions and initial values for the 286 protection mechanism—descriptor tables, gates, segments, and tasks. BLD-286 generates a Loadable or bootloadable output module, as well as a print file with a detailed map of the memory-protected system.

Using the Builder command Language, system programmers may perform the following functions:

- Assign physical addresses to segments; also set segment access rights and limits.
- Create Call, Trap, and Interrupt "Gates" (entry-points) for inter-level program transfers.
- Make gates available to tasks; this is an easier way to define program interfaces than using interface libraries.
- Create Global (GDT), Interrupt (IDT), and any Local (LDT) Descriptor Tables.
- Create Task State Segments and Task Gates for multi-task applications.
- Resolve inter-module and inter-level references, and perform type-checking.
- Automatically select required modules from libraries.
- Configure the memory image into partitions in the address space.
- Selectively generate an object file and various sections of the print file.

Key Benefit:

Allows a system programmer to define the configuration of a protected system in *one* place, with one easy-to-use Utility. This specification may then be adopted by all project members, using either the Builder *or just the Binder*. The flexibility simplifies program development for all users.



iAPX 286 SIMULATOR

- Supports Symbolic Debugging of Complete, Protected 286 Systems.
- Allows 286 Program Execution and Debugging in Absence of iAPX 286 Hardware Execution Vehicle.
- Functionally Duplicates the Operation of the iAPX 286 Microprocessor, Including Memory Protection.
- Executes Full Instruction Set, Including 80287 Numerics.
- Symbolic Access to Program Variables as well as Descriptor Tables.
- Two Execution Timers for Program Benchmarking and Interrupt Simulation.
- UDI File System Support for User Program.

SIM-286 is an 8086-resident program designed to support development of iAPX 286 O.S. kernels, systems, and applications. All of these may be developed and debugged without the use of a 286 hardware execution vehicle.

The Simulator consists of a human interface layer, and software executors for the 80286 CPU and 80287 Numeric Data Processor. The human interface receives commands with symbolic names, and passes control to the executor as though it were a 286-resident monitor.

SIM-286 lets designers manipulate a 286 program using the symbolic names given for code and data. It also lets users symbolically examine and modify the protection features (such as system tables, access rights, etc.), if it is desired.

SIM-286 contains two instruction timers. One may be set and incremented during execution; this allows program sequences to be benchmarked in clock cycles and microseconds. The second, an interval timer, may be set to generate interrupts every η clock cycles, to simulate event-driven processing. These timers are extremely useful for developing system kernels.

For programs that make operating system calls for file I/O, SIM-286 provides access to these services through the Universal Development Interface.

Key Benefit:

Symbolic system debugging (for protected 286 software) may be performed in the absence of a 286-based target.

SPECIFICATIONS

OPERATING ENVIRONMENT

Intel Microcomputer Development Systems
(Series III/Series IV)

DOCUMENTATION

- ASM 286 Language Reference Manual
- ASM 286 Macro Assembler Operating Instructions
- iAPX 286 Utilities User's Guide

- iAPX 286 System Builder User's Guide
- iAPX 286 Simulator User's Guide
- Pocket Reference for all the above:
 - ASM 286
 - Utilities
 - SIM 286

SUPPORT:

Hotline Telephone Support, Software Performance Report (SPR), Software Updates, Technical Reports, and Monthly Technical Newsletters are available.

ORDERING INFORMATION

Product Code	Description
IMDX-321	iAPX 286 Software Development Package



PL/M 286 SOFTWARE PACKAGE

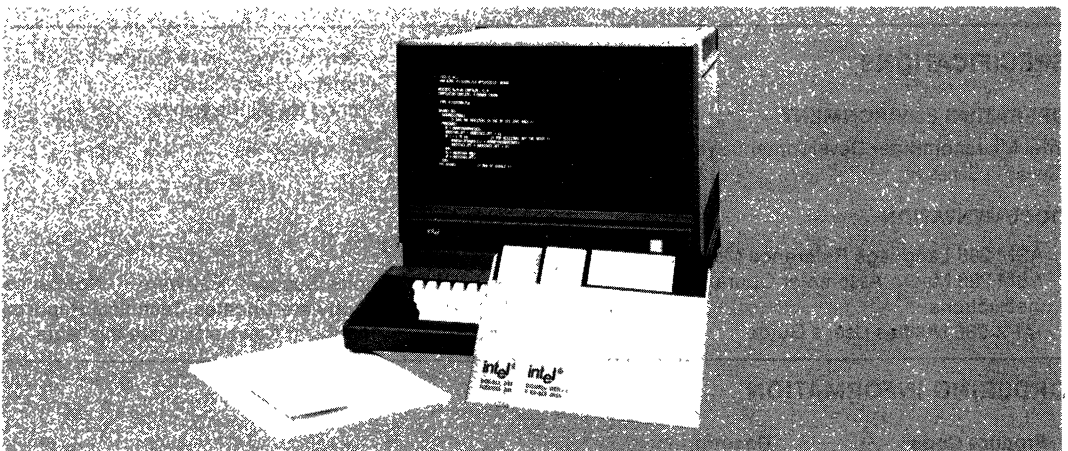
- **Systems programming language for the protected virtual address mode iAPX 286**
- **Upward compatible with PL/M 86 and PL/M 80 assuring software portability**
- **Enhanced to support design of protected, multi-user, multi-tasking, virtual memory operating system software**
- **Advanced, structured system implementation language for algorithm development**
- **Produces relocatable object code which is linkable to object modules generated by all other iAPX 286 language translators**
- **Multiple levels of optimization**
- **Resident on Intel microcomputer development systems (Series III, IV)**

PL/M 286 is a powerful, structured, high-level system implementation language for the development of system software for the protected virtual address mode iAPX 286. PL/M 286 has been enhanced to utilize iAPX 286 features—memory management and protection—for the implementation of multi-user, multi-tasking virtual memory operating systems.

PL/M 286 is upward compatible with PL/M 86 and PL/M 80. Existing systems software can be re-compiled with PL/M 286 to execute in protected virtual address mode on the iAPX 286.

PL/M 286 is the high-level alternative to assembly language programming on the iAPX 286. For the majority of iAPX 286 system programs, PL/M 286 provides the features needed to access and to control efficiently the underlying iAPX 286 hardware and consequently it is the cost-effective approach to develop reliable, maintainable system software.

The PL/M 286 compiler has been designed to efficiently support all phases of software development. Features such as a built-in syntax checker, multiple levels of optimization, virtual symbol table and four models of program size and memory usage for efficient code generation provide the total program development support needed.



FEATURES

Major features of the Intel PL/M 286 compiler and programming language include:

Structured Programming

PL/M source code is developed in a series of modules, procedures, and blocks. Encouraging program modularity in this manner makes programs more readable, and easier to maintain and debug. The language becomes more flexible by clearly defining the scope of user variables (local to a private procedure, for example).

The use of modules and procedures to break down a large problem leads to productive software development. The PL/M 286 implementation of block structure allows the use of REENTRANT procedures, which are especially useful in system design.

Language Compatibility

PL/M 286 object modules are compatible with object modules generated by all other 286 translators. This means that PL/M programs may be linked to programs written in any other 286 language.

Object modules are compatible with In-Circuit Emulators; DEBUG compiler control provides the In-Circuit Emulators with full symbolic debugging capabilities.

PL/M 286 language is upward compatible with PL/M 86 and PL/M 80 so that application programs may be easily ported to run on the protected mode iAPX 286.

Supports Seven Data Types

PL/M makes use of seven data types for various applications. These data types range from one to four bytes and facilitate various arithmetic, logic, and addressing functions:

- Byte: 8-bit unsigned number
- Word: 16-bit unsigned number
- Dword: 32-bit unsigned number
- Integer: 16-bit signed number
- Real: 32-bit floating-point number
- Pointer: 16-bit or 32-bit memory address indicator
- Selector: 16-bit pointer base.

Another powerful facility allows the use of BASED variables which permit run-time mapping of var-

iables to memory locations. This is especially useful for passing parameters; relative and absolute addressing, and dynamic memory allocation.

Two Data Structuring Facilities

In addition to the seven data types and based variables, PL/M supports two powerful data structuring facilities. These help the user to organize data into logical groups.

- Array: Indexed list of same type data elements
- Structure: Named collection of same or different type data elements
- Combinations of both: Arrays of structures or structures of arrays.

Numerics Support

PL/M programs that use 32-bit REAL data are executed using the 80287 Numeric Data Processor for high performance. All floating-point operations supported by PL/M are executed on the 80287 according to the IEEE floating-point standard. PL/M 286 programs can use built-in functions and predefined procedures—INIT\$REAL\$MATH\$UNIT, SET\$REAL\$MODE, GET\$REAL\$ERROR, SAVE\$REAL\$STATUS, RESTORE\$REAL\$STATUS—to control the operation of the 80287 within the scope of the language.

Built-In String Handling Facilities

The PL/M 286 language contains built-in functions for string manipulation. These byte and word functions perform the following operations on character strings: MOVE, COMPARE, TRANSLATE, SEARCH, SKIP, and SET.

Built-In Port I/O

PL/M 286 directly supports input and output from the iAPX 286 ports for single BYTE and WORD transfers. For BLOCK transfers, PL/M 286 programs can make calls to predefined procedures.

Interrupt Handling

PL/M 286 has the facility for generating and handling interrupts on the iAPX 286. A procedure may be defined as an interrupt handler through use of the INTERRUPT attribute. The compiler will then generate code to save and restore the processor status on each execution of the user-defined

interrupt handler routine. The PL/M statement CAUSE\$INTERRUPT allows the user to trigger a software interrupt from within the program.

Protection Model

PL/M 286 supports the implementation of protected operating system software by providing built-in procedures and variables to access the protection mechanism of the iAPX 286. Predefined variables—TASK\$REGISTER, LOCAL\$TABLE, MACHINE\$STATUS, etc.—allow direct access and modification of the protection system. Untyped procedures and functions—SAVE\$GLOBAL\$TABLE, RESTORE\$GLOBAL\$TABLE, SAVE\$INTERRUPT\$TABLE, RESTORE\$INTERRUPT\$TABLE, CLEAR\$TASK\$SWITCHED\$FLAG, GET\$ACCESS\$RIGHTS, GET\$SEGMENT\$LIMIT, SEGMENT\$READABLE, SEGMENT\$WRITABLE, ADJUST\$RPL—provide all the facilities needed to implement efficient operating system software.

Compiler Controls

The PL/M 286 compiler offers controls that facilitate such features as:

- Optimization
- Conditional compilation
- The inclusion of additional PL/M source files from disk
- Cross-reference of symbols
- Optional assembly language code in the listing file
- The setting of overflow conditions for run-time handling.

Addressing Control

The PL/M 286 compiler uses the SMALL, COMPACT, MEDIUM, and LARGE controls to generate optimum addressing instructions for programs. Programs of any size can be easily modularized into "subsystems" to exploit the most efficient memory addressing schemes. This lowers total memory requirements and improves run-time execution of programs.

Code Optimization

The PL/M 286 compiler offers four levels of optimization for significantly reducing overall program size.

- Combination or "folding" of constant expressions; and short-circuit evaluation of Boolean expressions

- "Strength reductions": a shift left rather than multiply by 2; and elimination of common sub-expressions within the same block
- Machine code optimizations; elimination of superfluous branches; reuse of duplicate code; removal of unreachable code
- Optimization of based-variable operations and cross-statement load/store.

Error Checking

The PL/M 286 compiler has a very powerful feature to speed up compilations. If a syntax or program error is detected, the compiler will skip the code generation and optimization passes. This usually yields a 2X performance increase for compilation of programs with errors.

A fully detailed and helpful set of programming and compilation error messages is provided by the compiler and user's guide.

BENEFITS

PL/M 286 is designed to be an efficient, cost-effective solution to the special requirements of protected mode iAPX 286 Microsystem Software Development, as illustrated by the following benefits of PL/M use:

Low Learning Effort

PL/M 286 is easy to learn and use, even for the novice programmer.

Earlier Project Completion

Critical projects are completed much earlier than otherwise possible because PL/M 286, a structured high-level language, increases programmer productivity.

Lower Development Cost

Increases in programmer productivity translate immediately into lower software development costs because less programming resources are required for a given programmed function

Increased Reliability

PL/M 286 is designed to aid in the development of reliable software (PL/M 286 programs are simple statements of the program algorithm). This substantially reduces the risk of costly correction of errors in

systems that have already reached full production status, as the more simply stated the program is, the more likely it is to perform its intended function.

Easier Enhancements and Maintenance

Programs written in PL/M tend to be self-documenting, thus easier to read and understand. This means it is easier to enhance and maintain PL/M programs as the system capabilities expand and future products are developed.

Cost-Effective Alternative to Assembly Language

PL/M 286 programs are code efficient. PL/M 286 combines all of the benefits of a high-level language (ease of use, high productivity) with the ability to access the iAPX 286 architecture. This includes language features for control of the iAPX 286 protection mechanism. Consequently, for the development of systems software, PL/M 286 is the cost-effective alternative to assembly language programming.

SPECIFICATIONS

Operating Environment

Intel Microcomputer Development System (Series III/Series IV)

Documentation Package

PL/M 286 User's Guide

ORDERING INFORMATION

Part Number	Description
iMDX 323	PL/M 286 Software Package

Requires Software License

SUPPORT:

Hotline Telephone Support, Software Performance Report (SPR), Software Updates, Technical Reports, and Monthly Technical Newsletters are available.

iSDM™ 286 iAPX 286 SYSTEM DEBUG MONITOR

- Development support for iSBC® 286- and iAPX 286-based applications
- Real Address Mode (RAM) and Protected Virtual Address Mode (PVAM) support
- Support of MULTIBUS® I and MULTIBUS® II environments
- Powerful debugging commands, including single step CPU operation
- For MULTIBUS® II, software configuration of system boards at start-up and automatic configuration of memory boards
- Universal Development Interface (UDI) support via development system connection
- Command execution, including program load capability from Intellec® Series III or Series IV Development Systems
- Supports 80287 Numeric Processor Extension (NPX) for high-speed math applications

The Intel iSDM™ 286 System Debug Monitor package contains the necessary software, cables, EPROMs, and documentation required to interface an iSBC® 286 board or iAPX 286 application to an Intellec® Series III or Series IV through a high-speed link. The System Debug Monitor supports an OEM's choice of MULTIBUS® I or MULTIBUS II environments, and the iRMX™ 86 Real-Time Multitasking Operating System or a custom operating system. The monitor contains debugging tools that examine CPU registers, memory content, CPU descriptor tables, and other crucial environmental details. The Monitor also allows programs to access files on the development system via the internal UDI support and the serial communication link.



The following are trademarks of Intel Corporation and may be used only to describe Intel products: Intel, ICE, iMMX, iRMX, iSBC, iSBX, iSXM, MULTIBUS, MULTICHANNEL and MULTIMODULE. 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. Information contained herein supercedes previously published specifications on these devices from Intel.

FUNCTIONAL DESCRIPTION

Overview

The iSDM 286 System Debug Monitor provides programmers of iAPX 286-based applications with the debugging tools needed to test new applications ranging from single-user systems to complex operating systems executing in either a MULTIBUS I or MULTIBUS II environment. Programmers are given direct access to both the Real Address (RAM) and Protected Virtual Address (PVAM) modes of the CPU via a simple terminal interface or via an Intellec Series III or Series IV Development System.

Powerful Debugging Commands

The iSDM 286 Monitor contains a powerful set of user functions, including commands to:

- Examine and modify CPU registers
- Examine, modify, and move memory locations
- Symbolic reference to variable names
- Find and compare memory contents
- Set program breakpoints
- Bootstrap load application software from iRMX 86 file compatible peripherals (requires the iRMX 86 Operating System for Bootstrap Loader)
- Single-step CPU operation
- Switch from Real Address Mode to Protected Virtual Address Mode

Formatted Displays

The iSDM 286 Monitor formats all iAPX 286 predefined data structures into clearly understandable displays. This display gives programmers a formatted view of such CPU structures as LDTs, GDTs, IDTs, Segment Selectors, and Task State Segments—not just a series of unconnected digits.

Universal Development Interface (UDI)

Via the Universal Development Interface (UDI), the iSDM 286 Monitor can support the execution of iRMX 86, Series III, Series IV, or any other UDI-based applications. The Monitor emulates many of the UDI calls (RAM or PVAM), and passes all requests for a file system to the host development system. UDI applications, such as compilers and other programs available from Independent Software Vendors, can be tested in the target iAPX 286 environment immediately.

MULTIBUS® II Software Configuration of System Boards

The MULTIBUS II Interconnect Space Registers allow the software to configure boards, eliminating much of the need for jumpers and wire wraps. The iSDM 286 Monitor can initialize these registers at configuration time using user-defined variables. The Monitor can also automatically configure memory boards, defining the addresses for each board sequentially in relation to the board's physical placement in the card cage. This feature allows for the swapping, adding, and deleting of memory boards on a dynamic basis.

Command Execution

Commands to the iSDM 286 Monitor are entered interactively via a standalone terminal, an Intellec® Series III or a Series IV Development System. The target application hardware is connected to the terminal or development system via a serial link. Figure 1 shows a typical MULTIBUS I environment and Figure 2 shows a typical MULTIBUS II environment. All control operations and UDI file manipulations occur over the serial link through the cables supplied. More than one channel can be configured for the communication since the Monitor scans all configured channels to determine which channel is in use.

Numeric Data Processor Support

In addition to executing 80287 Numeric Processor Extension (NPX) applications with full NPX performance, programmers may examine and modify NPX registers using decimal and real number format. Any location in memory known to contain numeric values in standard real format (IEEE-P754) may be examined or modified using normal decimal notation. In this manner, programmers may feel confident that correct and meaningful numbers are available to applications without having to encode and decode complex real, integer, and BCD hexadecimal formats.

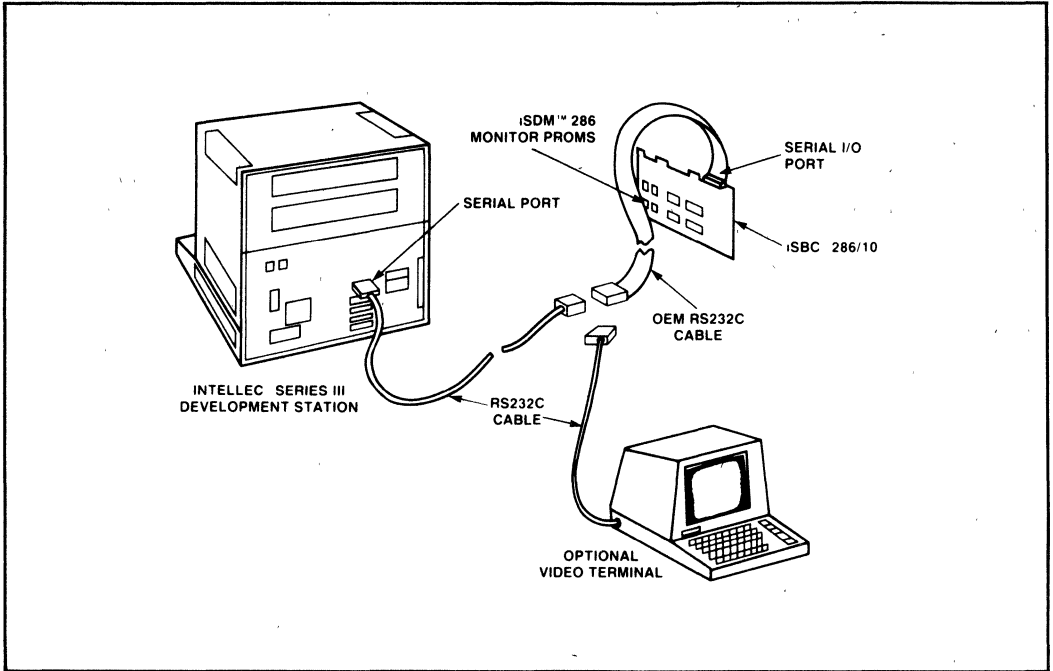


Figure 1. Typical MULTIBUS® I Environment

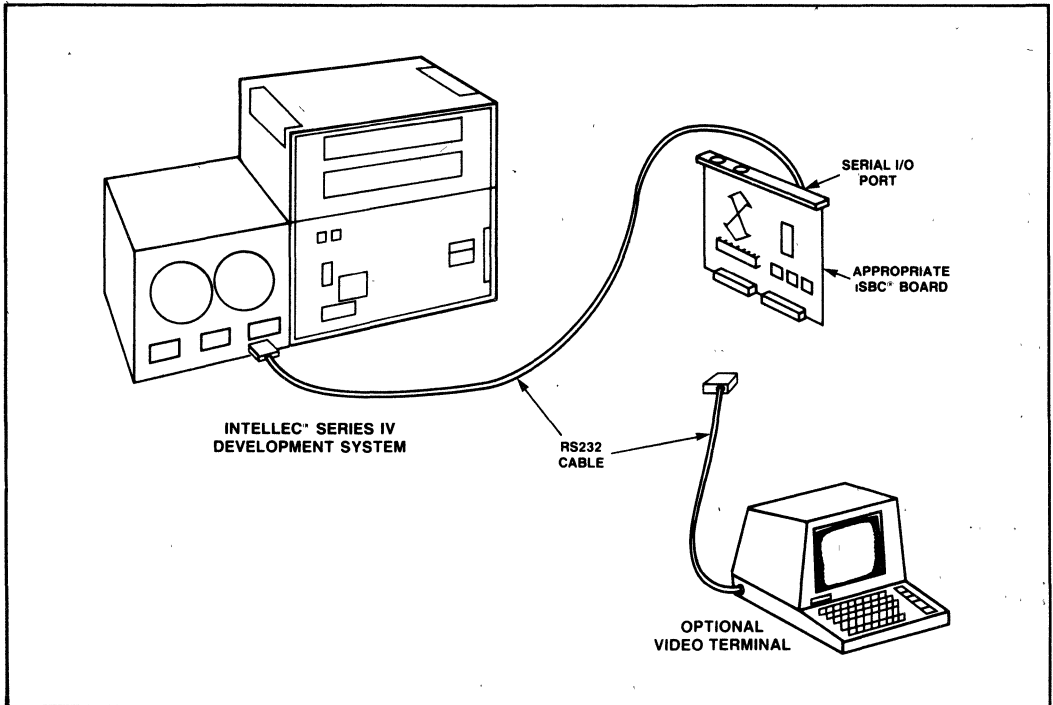


Figure 2. Typical MULTIBUS® II Environment

SPECIFICATIONS

Development System Environment

Intellec Series III or Series IV Development System with 128K of memory and 1 disk drive.

Target System Environment

Any iAPX 286 system with at least 4K of read-write memory starting at location 0H and 32K of read-only memory starting at location 0FF8000H.

Serial communication with a stand-alone terminal or development system requires either a 8274 USART and 8253 or 8254 PIT, or an 82530 SCC.

Monitor EPROMs are supplied for locations 0FF8000H through 0FFFFFFH.

ORDERING INFORMATION

The iSDM 286 System Debug Monitor package includes cables, EPROMs, software, and a reference

manual. The software is provided on a double-density, single-sided ISIS-formatted 8" diskette for Series III Development System use and on a double-density, double-sided iRMX-formatted 5¼" diskette for Series IV Development System use.

The OEM license option listed here allows users to incorporate iSDM 286 into their applications. Each use requires payment of an Incorporation Fee.

ORDER CODE: iSDM 286 RO.

The iSDM 286 RO product also includes 90 days of support services that includes the Software Problem Report service.

Another licensing option includes prepayment of all future incorporation fees.

As with all Intel software, purchase of any of these options requires the execution of a standard Intel Master Software license. The specific rights granted to users depends on the specific option and the license signed.



80287 SUPPORT LIBRARY

- Library to support floating point arithmetic in Pascal-286, PL/M-286 and ASM-286
- Common elementary function library provides trigonometric, logarithmic and other useful functions
- Decimal conversion module supports binary-decimal conversions
- Error-handler module simplifies floating point error recovery
- Supports proposed IEEE Floating Point Standard for high accuracy and software portability

The 80287 Support Library provides Pascal-286, PL/M-286 and ASM-286 users with numeric data processing capability. With the Library, it is easy for programs to do floating point arithmetic. Programs can bind in library modules to do trigonometric, logarithmic and other numeric functions, and the user is guaranteed accurate, reliable results for all appropriate inputs. Figure 1 below illustrates how the 80287 Support Library can be bound with PL/M-286 and ASM-286 user code to do this. The 80287 Support Library supports the proposed IEEE Floating Point Standard. Consequently, by using this Library, the user not only saves software development time, but is guaranteed that the numeric software meets industry standards and is portable—the software investment is maintained.

The 80287 Support Library consists of the common elementary function library (CEL287.LIB), the decimal conversion library (DC287.LIB), the error handler module (EH287.LIB) and interface libraries (80287.LIB, NUL287.LIB).

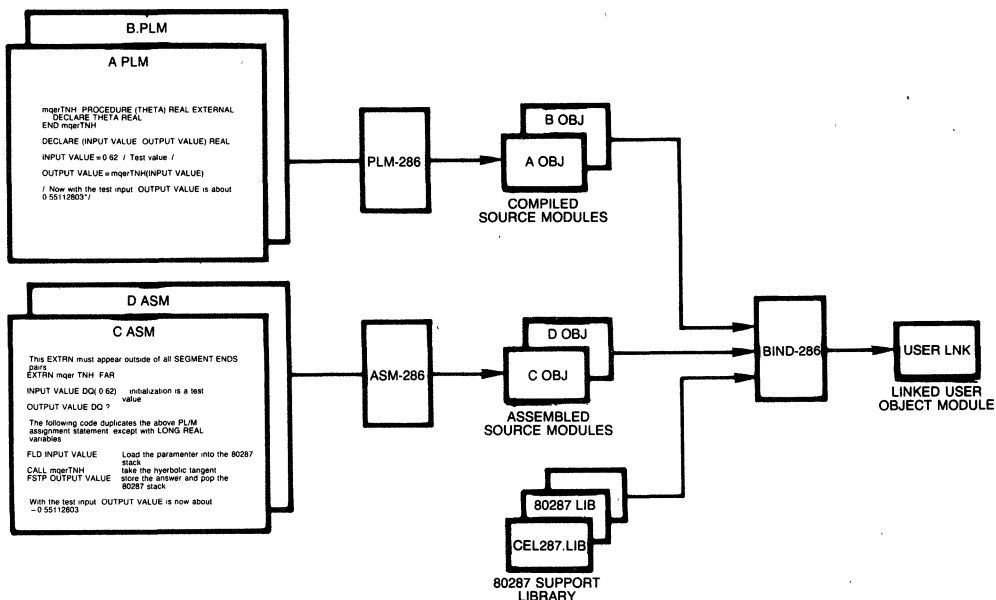


Figure 1. Use of 80287 Support Library with PL/M-286 and ASM-286.

CEL287.LIB THE COMMON ELEMENTARY FUNCTION LIBRARY

FUNCTIONS

CEL287.LIB contains commonly used floating point functions. It is used along with the 80287 numeric coprocessor. It provides a complete package of elementary functions, giving valid results for all appropriate inputs. Following is a summary of CEL287 functions, grouped by functionality.

Rounding and Truncation Functions:

mqrEX, **mqrE2**, and **mqrE4**. Round a real number to the nearest integer; to the even integer if there is a tie. The answer returned is real, a 16-bit integer or a 32-bit integer respectively.

mqrAX, **mqrA2**, **mqrA4**. Round a real number to the nearest integer, to the integer away from zero if there is a tie; the answer returned is real, a 16-bit integer or a 32-bit integer, respectively.

mqrCX, **mqrC2**, **mqrC4**. Truncate the fractional part of a real input; the answer is real, a 16-bit integer or 32-bit integer, respectively.

Logarithmic and Exponential Functions:

mqrLGD computes decimal (base 10) logarithms.
mqrLGE computes natural (base e) logarithms.
mqrEXP computes exponentials to the base e.

mqrY2X computes exponentials to any base.
mqrY12 raises an input real to a 16-bit integer power.
mqrY14 is as **mqrY12**, except to a 32-bit integer power.
mqrYIS is as **mqrY12**, but it accommodates PL/M-286 users.

Trigonometric and Hyperbolic Functions:

mqrSIN, **mqrCOS**, **mqrTAN** compute sine, cosine, and tangent.
mqrASN, **mqrACS**, **mqrATN** compute the corresponding inverse functions.
mqrSNH, **mqrCSH**, **mqrTNH** compute the corresponding hyperbolic functions.
mqrAT2 is a special version of the arc tangent function that accepts rectangular coordinate inputs.

Other Functions:

mqrDIM is FORTRAN's positive difference function.
mqrMAX returns the maximum of two real inputs.
mqrMIN returns the minimum of two real inputs.
mqrSGH combines the sign of one input with the magnitude of the other input.
mqrMOD computes a modulus, retaining the sign of the dividend.
mqrRMD computes a modulus, giving the value closest to zero.

DC287.LIB THE DECIMAL CONVERSION LIBRARY

DC287.LIB is a library of procedures which convert binary representations of floating point numbers and ASCII-encoded string of digits.

The binary-to-decimal procedure **mqcBIN_DECLOW** accepts a binary number in any of the formats used for the representation of floating point numbers in the 80287. Because there are so many output formats for floating point numbers, **mqcBIN_DECLOW** does not attempt to provide a finished, formatted text string. Instead, it provides the "building blocks" for you to use to construct the output string which meets your exact format specification.

The decimal-to-binary procedure **mqcDEC_BIN** accepts a text string which consists of a decimal number with optional sign, decimal point, and/or power-of-ten exponent. It translates the string into the caller's choice of binary formats.

Decimal-to-binary procedure **mqcDECLOW_BIN** is provided for callers who have already broken the decimal number into its constituent parts.

The procedures **mqcLONG_TEMP**, **mqcSHORT_TEMP**, **mqcTEMP_LONG**, and **mqcTEMP_SHORT** convert floating point numbers between the longest binary format, **TEMP_REAL**, and the shorter formats.

EH287.LIB THE ERROR HANDLER MODULE

EH287.LIB is a library of five utility procedures for writing trap handlers. Trap handlers are called when an unmasked 80287 error occurs.

The 80287 error reporting mechanism can be used not only to report error conditions, but also to let software implement IEEE standard options not directly supported by the chip. The three such extensions to the 80287 are: normalizing mode, non-trapping not-a-number (NaN), and non-ordered comparison. The utility procedures support these extra features.

DECODE is called near the beginning of the trap handler. It preserves the complete state of the 80287, and also identifies what function called the trap handler, and returns available arguments and/or results. DECODE eliminates much of the effort needed to determine what error caused the trap handler to be called.

NORMAL provides the "normalizing mode" capability for handling the "D" exception. By calling NORMAL

in your trap handler, you eliminate the need to write code in your application program which tests for non-normal inputs.

SIEVE provides two capabilities for handling the "I" exception. It implements non-trapping NaN's and non-ordered comparisons. These two IEEE standard features are useful for diagnostic work.

ENCODE is called near the end of the trap handler. It restores the state of the 80287 saved by DECODE, and performs a choice of concluding actions, by either retrying the offending function or returning a specified result.

FILTER calls each of the above four procedures. If your error handler does nothing more than detect fatal errors and implement the features supported by SIEVE and NORMAL, then your interface to EH287.LIB can be accomplished with a single call to FILTER.

80287.LIB, NUL287.LIB INTERFACE LIBRARIES

80287.LIB and NUL287.LIB libraries configure a user's application program for his run-time environ-

ment; running with the 80287 component or without floating point arithmetic, respectively.

SPECIFICATIONS

Operating Environment

Intel Microcomputer Development Systems (Series III, Series IV)

Documentation Package

80287 Support Library Reference Manual

Related Software

A 80287 software emulator is available as part of the 8086 software toolbox (IMDX364)

ORDERING INFORMATION

Part Number	Description
iMDX329	80287 Support Library
Requires Software License	

SUPPORT

Intel offers several levels of support for this product which are explained in detail in the price list. Please

consult the price list for a description of the support options available.



PASCAL-286 SOFTWARE PACKAGE

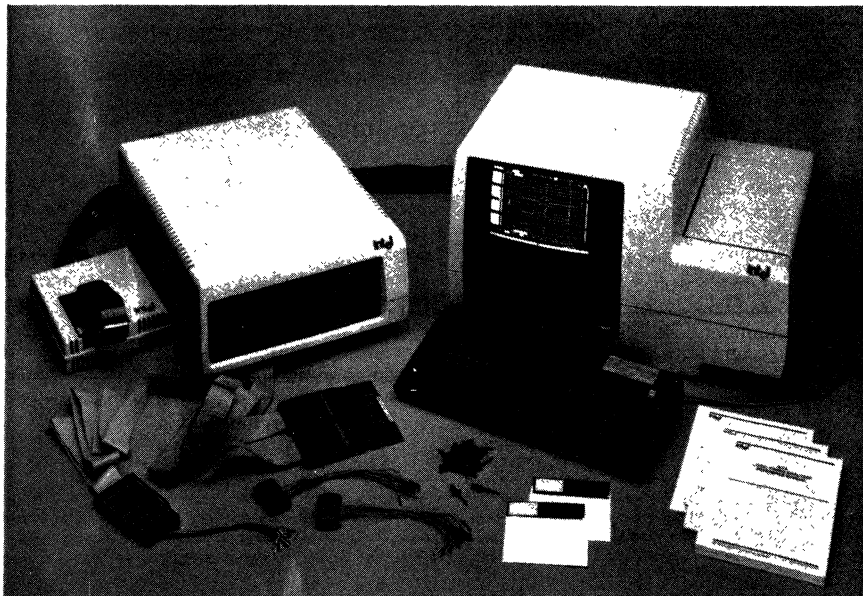
- High-level programming language for the protected virtual mode iAPX 286
- Implements ISO standard Pascal. Many useful extensions may be enabled via a compiler switch
- Upward compatible with Pascal-86 for software portability
- Produces relocatable object code which is linkable to object modules generated by other iAPX 286 translators
- Supports full symbolic debugging with iAPX 286 software and ICE™ debuggers
- Fully supports the 80287 numeric processor using the IEEE floating point standard

Pascal-286 is a powerful, structured, applications programming language for the protected virtual address mode of the iAPX 286. Pascal-286 is upward compatible with Pascal-86 so that 8086 Pascal source code can be ported to the iAPX 286 in protected mode.

Pascal-286 implements strict ISO standard Pascal, but with many useful extensions. These include separate compilation of modules, interrupt handling, port I/O, and 80287 numerics support. A control is provided in the compiler to flag all non-ISO features used.

Pascal-286 produces relocatable object code which can be linked with object code produced by other iAPX 286 translators such as ASM-286 and PL/M-286. Thus, a combination of translators can be used to provide great programming flexibility.

Type and symbol information needed by software and in-circuit debuggers is added to the object code by the Pascal-286 compiler. This information can be stripped off by the compiler or linker for the final production version.



Intel Corporation Assumes No Responsibility for the use of Any Circuitry Other Than Circuitry Embodied in an Intel Product. No Other Patent Licenses are implied. ©INTEL CORPORATION, 1982. Note: The development system pictured here is not included in the Pascal-286 software package, but merely depicts the language in its operating environment.

FEATURES

Conforms to ISO Standard Pascal

Pascal has gained wide acceptance as a portable language for microcomputer applications. However, portability can result only if standards are adhered to. Pascal-286 is a strict implementation of ISO standard Pascal. Extensions are provided to make the language more powerful for microprocessor applications. All extensions are clearly highlighted in the documentation. In addition, the compiler provides a control to flag any non ISO feature used. Pascal-286 will evolve to track future enhancements to standard Pascal.

Upward Compatible with Pascal-86

The Pascal-286 compiler produces object code for the protected virtual address mode of the iAPX 286 language. However, no 286 architecture specific features have been added to the Pascal-286 language. This makes Pascal-286 source code upward compatible with Pascal-86, which allows for porting of 8086 software to the protected 286 with relative ease.

Compatible With Other iAPX 286 Translators

All Intel iAPX 286 translators output object code in a standardized format. This allows 286 programs to be written in a mixture of languages. Systems routines which need access to architectural features can be coded in PL/M-286 or ASM-286. Pascal-286 may be better suited for the applications routines. The systems and application routines can then be combined using the 286 linker (BIND-286).

Standardized Run Time Support

Programs compiled with Pascal-286 can be moved from the development host environment to the target environment with ease. This is the result of standardizing run-time operating system interfaces required by the compiled program into a well defined and well documented set of routines. After programs are developed on a development host, they can then be executed in the target using the same set of system interfaces.

Extensions for Microprocessor Programming

Pascal-286 provides extensions that make it powerful for microprocessor applications. Built-in procedures allow I/O directly from the ports of the iAPX 286. This speeds up I/O as it is done by direct communication with the microprocessor. Interrupt processing is also supported by built in procedures. Examples are: ENABLEINTERRUPTS, DISABLEINTERRUPTS, CAUSEINTERRUPT. Many built in procedures and variables are provided for communicating with the 80287 for numeric computations.

Compiler Controls

The Pascal-286 compiler provides many controls which can be used at invocation time to enhance programming flexibility. Examples are: CODE/NOCODE, DEBUG/NODEBUG, INCLUDE (file), LIST/NOLIST, OPTIMIZE (n), EXTENSIONS/NOEXTENSIONS. All controls have default values that are active unless the opposite is specified during invocation. Thus, for most compiles, no controls need be specified.

Support for IEEE Standard Numerics

Pascal-286 provides full support for the 80287 numerics co-processor. All floating point operations are done according to the IEEE floating point standard. The benefits are predictable, accurate and consistent results. Built-in procedures to support the 80287 include GET8087ERRORS and MASK 8087ERRORS. A full set of 80287 library routines are supplied with the compiler.

Optimizations

The Pascal-286 compiler produces highly optimized code, both in size and execution time. This is achieved by:

- Use of powerful iAPX 286 instructions, in particular, for string handling, 80287 numerics and subroutine linkage
- Short circuit evaluation of boolean expressions, constant folding and strength reduction of multiplications and additions
- Elimination of superfluous branches, optimization of span dependent jumps



SPECIFICATIONS

Operating Environment

Intel 8086 based microcomputer
Development systems (Series III, Series IV)

Documentation Package

Pascal-286 User's Guide
Pascal-286 Pocket Reference

ORDERING INFORMATION

Part Number

Description

iMDX-324 Pascal-286 Software Package

Requires Software License

Support

Hotline service, SPR (Software Performance Reports),
Updates and technical newsletters are available.



VAX*/VMS* RESIDENT SOFTWARE DEVELOPMENT PACKAGES FOR iAPX 286

- Hosted on DEC VAX* Minicomputer Under the VMS* Operating System
- Packages include PL/M-286, BUILD-286, BIND-286, LIB-286 and MAP-286
- Allows Development of System and Application Software for the Protected Virtual Address Mode of the iAPX-286
- Compatible with Corresponding Intel Development System Resident Products

These packages provide the capability of developing software on a VAX*/VMS* host for the iAPX-286 in protected virtual address mode. With these packages a user can assemble and compile 286 programs, configure system and application software and create and manage 286 object libraries. Figure 1 illustrates the process of 286 software development on VAX*/VMS* hosts.

Two packages are available:

1. A PL/M-286 package which contains the PL/M-286 compiler and run time support libraries.
2. An ASM-286 package which contains the iAPX-286 Assembler (ASM-286) and programming utilities. These utilities include the iAPX-286 System Builder (BLD-286), the System Binder (BND-286), a Library Utility (LIB-286) and an Object Map Utility (MAP-286).

These packages are compatible with corresponding products which are hosted on Intel development systems. Correspondence can be established via version numbers. For example, BND-286 V2.0 offers the same set of features on VAX/VMS and Intel development systems.

Owing to this compatibility, iAPX-286 software developed on VAX/VMS can be linked to iAPX-286 software from development systems. Moreover, iAPX-286 programs developed on the VAX can then be downloaded to development systems and debugged using 286 debuggers like the I²ICE™-286 system.

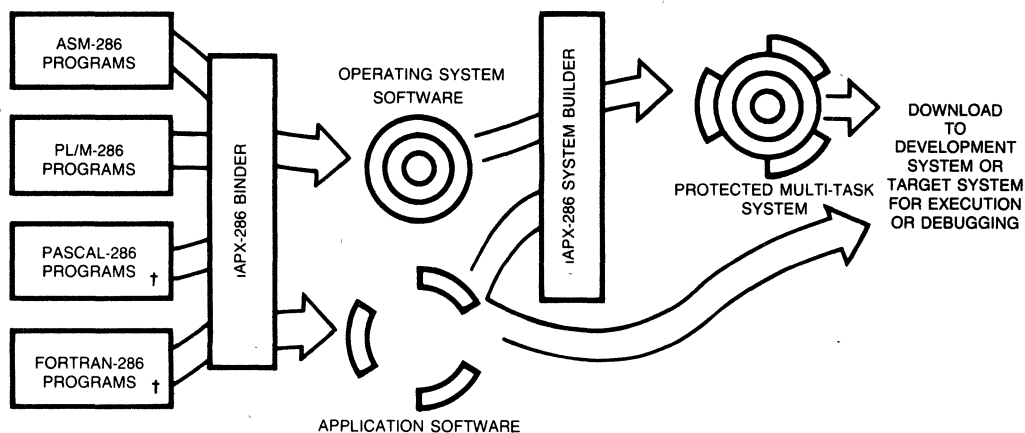


Figure 1: 286 Software Development on VAX*/VMS*

*VAX, VMS are trademarks of Digital Equipment Corporation †Currently Available on Intel Development Systems Only

VAX*/VMS* RESIDENT PL/M-286

- **Systems Programming Language for the protected virtual address mode iAPX-286**
- **Produces relocatable object code linkable to object modules generated by other Intel 286 language translators**
- **Enhanced to support design of protected, multi-user, multi-tasking, virtual memory operating system software**
- **Upward compatible with PL/M-86 and PL/M-80 to allow software portability**
- **Provides multiple levels of optimization to produce efficient code**
- **Compatible with development system resident PL/M-286**

PL/M-286 is a powerful, structured, high-level system implementation language for the development of system software for the protected virtual address mode iAPX-286. PL/M-286 has been enhanced to utilize iAPX-286 features--memory management and protection--for the implementation of multi-user, multi-tasking virtual memory operating systems.

PL/M-286 is upward compatible with PL/M-86 and PL/M-80. Existing systems software can be re-compiled with PL/M-286 to execute in protected virtual address mode on the iAPX-286.

PL/M-286 is the high-level alternative to assembly language programming on the iAPX-286. For the majority of iAPX-286 system programs, PL/M-286 provides the features needed to access and to control efficiently the underlying iAPX-286 hardware, and consequently it is the cost-effective approach to develop reliable, maintainable system software.

The PL/M-286 compiler has been designed to efficiently support all phases of software development. Features such as built-in syntax checker, multiple levels of optimization, virtual symbol table and four models of program size and memory usage for efficient code generation provide the total program development support needed. The compiler also provides complete symbolic debug capability to the various 286 debuggers and emulators.

VAX/VMS resident PL/M-286 is completely feature compatible with development system resident PL/M-286 with the same version number.

VAX*/VMS* RESIDENT iAPX-286 MACRO ASSEMBLER

- Supports full Instruction Set of the iAPX-286 including memory protection and numerics (with 80287)
- Structures and RECORDS provide powerful data representation
- Type checking at assembly time helps reduce errors at run-time
- Powerful and flexible Text Macro facility
- Upward compatible with ASM-86/88/186
- Compatible with development system resident iAPX-286 Macro Assembler

ASM-286 is the "high-level" macro assembler for the iAPX-286 assembly language. ASM-286 translates symbolic assembly language mnemonics into relocatable object code. The assembler mnemonics are a superset of ASM-86/88 mnemonics; new ones have also been added to support the new iAPX-286 instructions. The segmentation directives have been greatly simplified.

The iAPX-286 assembly language includes approximately 150 instruction mnemonics. From these few mnemonics the assembler can generate over 4,000 distinct machine instructions. Therefore, the software development task is simplified, as the programmer need know only 150 mnemonics to generate all possible machine instructions. ASM-286 generates the shortest machine instruction possible (given explicit information as to the characteristics of any forward referenced symbols).

The powerful macro facility in ASM-286 saves development and maintenance time by coding common program sequences only once. A macro substitution is made each time the sequence is to be used. This facility also allows for conditional assembly of certain program sequences.

ASM-286 offers many features normally found only in high-level languages. The assembly language is strongly typed, which means it performs extensive checks on the usage of variables and labels. This means that many programming errors will be detected when the program is assembled, long before it is being debugged.

ASM-286 object modules conform to a thorough, well-defined format used by 286 high-level languages and utilities. This makes it easy to call (and be called from) HLL object modules.

ASM-286 also provides support for the 80287 numerics co-processor. The complete instruction set of the 80287 is available through high-level mnemonics.

VAX/VMS resident ASM-286 is completely feature compatible with development system resident ASM-286 with the same version number.

VAX*/VMS* RESIDENT iAPX-286 SYSTEM BUILDER

- A tool for configuring multi-tasking protected, virtual memory systems software for the iAPX-286
- Links separately compiled modules. Resolves EXTERNAL/PUBLIC definitions
- Creates a memory image of a 286 system for cold start execution
- Target system may be bootloadable, programmed into ROM or loaded from mass storage
- Generates print file with command listing and system map
- Compatible with development system resident iAPX-286 System Builder

BLD-286 is the iAPX-286 System Builder. It allows systems programmers to configure multi-tasking and memory protected iAPX-286 software. The configuration is specified by the user in a "Build file" using a symbolic meta-language. BLD-286 thus provides the programmer a high-level symbolic interface to the multi-tasking and memory protection features of the iAPX-286 architecture.

BLD-286 accepts as inputs object modules from the iAPX-286 translators, the iAPX-286 Binder and itself (for incremental building). Using the programmer's specifications in the Build File, it produces a bootloadable or loadable module as well as a print file with a map of the configured module.

Using the builders command language, system programmers may perform the following functions:

- Assign physical addresses to segments; also set segment access rights and limits.
- Create Call, Trap, and Interrupt "Gates" (entry-points) for inter-level program transfers.
- Make gates available to tasks; this is an easier way to define program interfaces than using interface libraries.
- Create Global (GDT), Interrupt (IDT), and any Local (LDT) Descriptor Tables.
- Create Task State Segments and Task Gates for multi-tasking applications.
- Resolve inter-module and inter-level references, and perform type-checking.
- Automatically select required modules from libraries.
- Configure the memory image into partitions in the address space.
- Selectively generate an object file and various sections of the print file.

VAX/VMS BLD-286 is completely feature compatible with development system resident BLD-286 with the same version number.

VAX*/VMS* RESIDENT iAPX-286 BINDER

- **Links separately compiled program modules into an executable task**
- **Makes the iAPX-286 protection mechanism invisible to application programmers**
- **Assigns virtual addresses to tasks**
- **Performs incremental linking with output of Binder and Builder**
- **Resolves PUBLIC/EXTERNAL code and data references, and performs intermodule type-checking**
- **Provides print file showing segment map, errors and warnings**
- **Generates linkable or loadable module for debugging**
- **Compatible with development system resident iAPX-286 Binder.**

BND-286 is a utility that combines iAPX-286 object modules into executable tasks. In creating a task, the Binder resolves Public and External symbol references, combines segments, and performs address fix-ups on symbolic code and data.

The Binder takes object modules, produced by the 286 translators, and generates a loadable module (for execution or debugging), or a linkable module (to be re-input to the Binder later; this is called incremental binding). The binder accepts library modules as well, linking only those modules required to resolve external references. BND-286 generates a print file displaying a segment map, and error messages.

The Binder is useful for system as well as application programmers. Since application programmers need to develop software independent of any system architecture, the 286 memory protection mechanism is "hidden" from users of the Binder. This allows application tasks to be fully debugged before becoming part of a protected system. (A protected system may be debugged, as well.) System protection features are specified later in the development cycle, using the 286 System Builder. It is possible to link operating system services required by a task using either the Binder or the Builder. This flexibility adds to the ease of use of the 286 utilities.

VAX/VMS resident BND-286 is completely feature compatible with development system resident BND-286 with the same version number.

VAX*/VMS* RESIDENT iAPX-286 LIBRARIAN

- **Allows creation and management of iAPX-286 object libraries**
- **Library functions include Create, Delete, Add, Replace, Copy, Save, Backup and Display**
- **Only required modules linked in when using Binder or Builder**
- **Compatible with development system resident iAPX-286 Librarian**

LIB-286 is the iAPX-286 Librarian. It can be used to create and manage iAPX-286 Object Libraries. By placing often used object modules into libraries, the administrative overhead of managing software modules can be reduced.

VAX/VMS based LIB-286 is completely feature compatible with development system resident LIB-286 with the same version number.



VAX*/VMS* RESIDENT iAPX-286 MAPPER

- Flexible Utility to display object file information in symbolic form
- Compatible with development system resident iAPX-286 Mapper

MAP-286 is a cross reference utility for iAPX-286 object modules. It provides a symbolic listing of the EXTERNAL and PUBLIC symbols in the specified object modules.

VAX/VMS resident MAP-286 is completely feature compatible with development system resident MAP-286 with the same version number.

SPECIFICATIONS

Operating Environment

DEC VAX* 11/780 or compatible model running VMS* operating system V3.4 (or upward compatible versions)

Documentation

Installation guide and user's manuals for the software are supplied with the products.

SUPPORT

Hotline Telephone Support, Software, Performance Report (SPR) Software Updates, Technical Reports and Monthly Newsletters are available.

ORDERING INFORMATION

Product Code	Description
iMDX-371VX	ASM-286, BLD-286, BND-286, LIB-286, MAP-286
iMDX-373VX	PL/M-286

VAX/VMS are trademarks of Digital Equipment Corporation



VAX*/VMS* RESIDENT iAPX-86/88/186 SOFTWARE DEVELOPMENT PACKAGES

- **Executes on DEC VAX* Minicomputer under VMS* Operating System to translate PL/M-86, Pascal-86 and ASM-86 Programs for iAPX-86, 88 and 186 Microprocessors.**
- **Packages include Pascal-86; PL/M-86; ASM-86; Link and Relocation Utilities; OH-86 Absolute Object Module to Hexadecimal Format Converter; and Library Manager Program.**
- **Output linkable with Code Generated on Intellec® Development Systems.**

The VAX/VMS Resident Software Development Packages contain software development tools for the iAPX-86, 88, and 186 microprocessors. The package lets the user develop, compile, maintain libraries, and link and locate programs on a VAX running the VMS operating system. The translator output is object module compatible with programs translated by the corresponding version of the translator on an Intellec Development System.

Three packages are available:

1. An ASM-86 Assembler Package which includes the Assembler, the Link Utility, the Locate Utility, the absolute object to hexadecimal format conversion utility and the Library Manager Program.
2. A PL/M-86 Compiler Package which contains the PL/M-86 Compiler and Runtime Support Libraries.
3. A Pascal-86 Compiler Package which contains the Pascal-86 Compiler and Runtime Support Libraries.

The VAX/VMS resident development packages and the Intellec Development System development packages are built from the same technology base. Therefore, the VAX/VMS resident development packages and the Intellec Development System development packages are very similar.

Version numbers can be used to identify features correspondence. The VAX/VMS resident development packages will have the same features as the Intellec Development System product with the same version number.

Support for the iAPX-186 processor will be provided as an update to the iAPX-86, 88 software.

The object modules produced by the translators contain symbol and type information for programming debugging using ICE™ translators and/or the PSCOPE debugger. For final production version, the compiler can remove this extra information and code.

*VAX, DEC, and VMS are trademarks of Digital Equipment Corporation.

VAX*-PL/M-86/88/186 SOFTWARE PACKAGE

- **Executes on VAX* Minicomputer Under the VMS* Operating System**
- **Supports 16-Bit Signed Integer and 32-Bit Floating Point Arithmetic in Accordance with IEEE Proposed Standard**
- **Easy-To-Learn Block-Structured Language Encourages Program Modularity**
- **Produces Relocatable Object Code Which is Linkable to All Other Intel 8086 Object Modules, Generated on Either a VAX* or Intellec® Development Systems**
- **Code Optimization Assures Efficient Code Generation and Minimum Application Memory Utilization**
- **Built-In Syntax Checker Doubles Performance for Compiling Programs Containing Errors**
- **Source Input/Object Output Compatible with PL/M-86 Hosted on an Intellec® Development System**
- **ICE™, PSCOPE Symbolic Debugging Fully Supported**

Like its counterpart for MCS®-80/85 program development, and Intellec® hosted iAPX-86 program development, VAX-PL/M-86 is an advanced, structured high-level programming language. The VAX-PL/M-86 compiler was created specifically for performing software development for the Intel iAPX-86, 88, and 186 Microprocessors.

PL/M is a powerful, structured, high-level system implementation language in which program statements can naturally express the program algorithm. This frees the programmer to concentrate on the logic of the program without concern for burdensome details of machine or assembly language programming (such as register allocation, meanings of assembler mnemonics, etc.).

The VAX-PL/M-86 compiler efficiently converts free-form PL/M language statements into equivalent iAPX-86/88/186 machine instructions. Substantially fewer PL/M statements are necessary for a given application than if it were programmed at the assembly language or machine code level.

The use of PL/M high-level language for system programming, instead of assembly language, results in a high degree of engineering productivity during project development. This translates into significant reductions in initial software development and follow-on maintenance costs for the user.

VAX*-PASCAL-86/88 SOFTWARE PACKAGE

- Executes on VAX* Minicomputer Under the VMS* Operating System
- Produces Relocatable Object Code Which Is Linkable to All Other Intel 8086 Object Modules, Generated on Either a VAX* or Inteltec® Development Systems
- ICE™, PSCOPE Symbolic Debugging Fully Supported
- Implements REALMATH for Consistent and Reliable Results
- Supports iAPX-86/20, 88/20 Numeric Data Processors
- Strict Implementation of ISO Standard Pascal
- Useful Extensions Essential for Micro-computer Applications
- Separate Compilation with Type-Checking Enforced Between Pascal Modules
- Compiler Option to Support Full Run-Time Range-Checking
- Source Input/Object Output Compatible with Pascal-86 Hosted on a Inteltec Development System

VAX-PASCAL-86 conforms to and implements the ISO Pascal standard. The language is enhanced to support microcomputer applications with special features, such as separate compilation, interrupt handling and direct port I/O. Other extensions include additional data types not required by the standard and miscellaneous enhancements such as an allowed underscore in names, an OTHERWISE clause in CASE construction and so forth. To assist the development of portable software, the compiler can be directed to flag all non-standard features

The VAX-PASCAL-86 compiler runs on the Digital Equipment Corporation VAX under the VMS Operating System. A well-defined I/O interface is provided for run-time support. This allows a user-written operating system to support application programs on the target system as an alternate to the development system environment. Program modules compiled under PASCAL-86 are compatible and linkable with modules written in PL/M-86, and ASM-86. With a complete family of compatible programming languages for the iAPX-86, 88, and 186 one can implement each module in the language most appropriate to the task at hand.

VAX*-iAPX-86/88/186 MACRO ASSEMBLER

- Executes on VAX* Minicomputer Under The VMS* Operating System
- Produces Relocatable Object Code Which Is Linkable to All Other Intel iAPX-86/88/186 Object Modules, Generated on Either a VAX* or Intellec® Development Systems
- Powerful and Flexible Text Macro Facility with Three Macro Listing Options to Aid Debugging
- Highly Mnemonic and Compact Language, Most Mnemonics Represent Several Distinct Machine Instructions
- "Strongly Typed" Assembler Helps Detect Errors at Assembly Time
- High-Level Data Structuring Facilities Such as "STRUCTURES" and "RECORDS"
- Over 120 Detailed and Fully Documented Error Messages
- Produces Relocatable and Linkable Object Code
- Source Input/Object Output Compatible with ASM-86 hosted on an Intellec Development System

VAX-ASM-86 is the "high-level" macro assembler for the iAPX-86/88/186 assembly language. VAX-ASM-86 translates symbolic iAPX-86/88/186 assembly language mnemonics into iAPX-86/88/186 relocatable object code

VAX-ASM-86 should be used where maximum code efficiency and hardware control is needed. The iAPX-86/88/186 assembly language includes approximately 100 instruction mnemonics. From these few mnemonics the assembler can generate over 3,800 distinct machine instructions. Therefore, the software development task is simplified, as the programmer need know only 100 mnemonics to generate all possible iAPX-86/88/186 machine instructions. VAX-ASM-86 will generate the shortest machine instruction possible given no forward referencing or given explicit information as to the characteristics of forward referenced symbols

VAX-ASM-86 offers many features normally found only in high-level languages. The iAPX-86/88/186 assembly language is strongly typed. The assembler performs extensive checks on the usage of variable and labels. The assembler uses the attributes which are derived explicitly when a variable or label is first defined, then makes sure that each use of the symbol in later instructions conforms to the usage defined for that symbol. This means that many programming errors will be detected when the program is assembled, long before it is being debugged on hardware.

VAX*-LIB-86

- **Executes on VAX* Minicomputer Under the VMS* Operating System**
- **VAX*-LIB-86 is a Library Manager Program which Allows You to:
Create Specifically Formatted Files to Contain Libraries of Object Modules
Maintain These Libraries by Adding or Deleting Modules
Print a Listing of the Modules and Public Symbols in a Library File**
- **Libraries Can be Used as Input to VAX*-LINK-86 Which Will Automatically Link Modules from the Library that Satisfy External References in the Modules Being Linked**
- **Abbreviated Control Syntax**

Libraries aid in the job of building programs. The library manager program VAX-LIB-86 creates and maintains files containing object modules. The operation of VAX-LIB-86 is controlled by commands to indicate which operation VAX-LIB-86 is to perform. The commands are:

CREATE: creates an empty library file
 ADD: adds object modules to a library file
 DELETE: deletes modules from a library file
 LIST: lists the module directory of library files
 EXIT: terminates the LIB-86 program and returns control to VMS

When using object libraries, the linker will call only those object modules that are required to satisfy external references, thus saving memory space.

VAX-OH-86

- **Executes on VAX* Minicomputer Under the VMS* Operating System**
- **Converts an iAPX 86/88/186 Absolute Object Module to Symbolic Hexadecimal Format**
- **Facilitates Preparing a file for Loading by Symbolic Hexadecimal Loader (e.g. iSBC™ Monitor SDK-86 Loader), or Universal PROM Mapper**
- **Converts an Absolute Module to a More Readable Format that can be Displayed on a CRT or Printed for Debugging**

The VAX-OH-86 utility converts an 86/88 absolute object module to the hexadecimal format. This conversion may be necessary for later loading by a hexadecimal loader such as the iSBC 86/12 monitor or the Universal PROM Mapper. The conversion may also be made to put the module in a more readable format that can be displayed or printed.

The module to be converted must be in absolute form; the output from VAX-LOC-86 is in absolute format.

VAX*-LINK-86

- Executes on VAX* Minicomputer Under the VMS* Operating System
- Automatic Combination of Separately Compiled or Assembled 86/88/186 Programs Into a Relocatable Module, Generated on Either a VAX or an Intellec® Development System
- Automatic Selection of Required Modules from Specified Libraries to Satisfy Symbolic References
- Extensive Debug Symbol Manipulation, allowing Line Numbers, Local Symbols, and Public Symbols to be Purged and Listed Selectively
- Automatic Generation of a Summary Map Giving Results of the LINK-86 Process
- Abbreviated Control Syntax
- Relocatable modules may be Merged into a Single Module Suitable for Inclusion in a Library
- Supports "Incremental" Linking
- Supports Type Checking of Public and External Symbols

VAX-LINK-86 combines object modules specified in the VAX-LINK-86 input list into a single output module. VAX-LINK-86 combines segments from the input modules according to the order in which the modules are listed.

VAX-LINK-86 will accept libraries and object modules built from VAX-PL/M-86, VAX-PASCAL-86, VAX-ASM-86, or any other Intel translator generating 8086 Relocatable Object Modules, such as the Series III resident translators.

Support for incremental linking is provided since an output module produced by VAX-LINK-86 can be an input to another link. At each stage in the incremental linking process, unneeded public symbols may be purged.

VAX-LINK-86 supports type checking of PUBLIC and EXTERNAL symbols reporting a warning if their types are not consistent.

VAX-LINK-86 will link any valid set of input modules without any controls. However, controls are available to control the output of diagnostic information in the VAX-LINK-86 process and to control the content of the output module.

VAX-LINK-86 allows the user to create a large program as the combination of several smaller, separately compiled modules. After development and debugging of these component modules the user can link them together, locate them using VAX-LOC-86 and enter final testing with much of the work accomplished.

VAX*-LOC-86

- **Executes on the VAX* Minicomputer Under the VMS* Operating System**
- **Automatic Generation of a Summary Map Giving Starting Address, Segment Addresses and Length, and Debug Symbols and their Addresses**
- **Extensive Capability to Manipulate the Order and Placement of Segments in 8086/8088 Memory**
- **Abbreviated Control Syntax**
- **Automatic and Independent Relocation of Independent Relocation of Segments. Segments May be Relocated to Best Match Users Memory Configuration**
- **Extensive Debug Symbol Manipulation, Allowing Line Numbers, Local Symbols, and Public Symbols to be Purged and Listed Selectively**

Relocatability allows the programmer to code programs or sections of programs without having to know the final arrangement of the object code in memory.

VAX-LOC-86 converts relative addresses in an input module in iAPX-86/88/186 object module format to absolute addresses. VAX-LOC-86 orders the segments in the input module and assigns absolute addresses to the segments. The sequence in which the segments in the input module are assigned absolute addresses is determined by their order in the input module and the controls supplied with the command.

VAX-LOC-86 will relocate any valid input module without any controls. However, controls are available to control the output of diagnostic information in the VAX-LOC-86 process, to control the content of the output module, or both.

The program you are developing will almost certainly use some mix of random access memory (RAM), read-only memory (ROM), and/or programmable read-only memory (PROM). Therefore, the location of your program affects both cost and performance in your application. The relocation feature allows you to develop your program and then simply relocate the object code to suit your application.

SPECIFICATIONS

Operating Environment

Required Hardware

VAX* 11/780, 11/782, 11/750, or 11/730
9 Track Magnetic Tape Drive, 1600 BPI

Required Software

VMS Operating System V3.0 or Later. All of the development packages are delivered as unlinked VAX object code which can be linked to VMS as designed for the system where the development package is to be used. VMS command files to perform the link are provided.

Documentation Package

iAPX-86, 88 Development Software Installation Manual and User's Guide for VAX/VMS, Order number 121950-001

Shipping Media

9 Track Magnetic Tape 1600 bpi

ORDERING INFORMATION

Part Number Description

iMDX-341VX	VAX-ASM-86, VAX-LINK-86, VAX-LOC-86, VAX-LIB-86, VAX-OH-86, Package
iMDX-343VX	VAX-PLM-86 Package
IMDX-344VX	VAX-PASCAL-86 Package

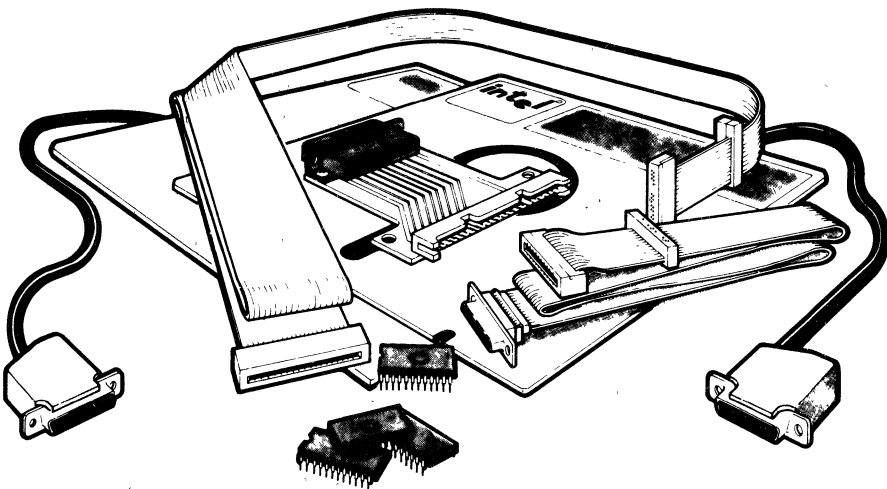
REQUIRES SOFTWARE LICENSE



iSDM™ 86 SYSTEM DEBUG MONITOR

- Supports target system debugging for iSBC® /iAPX 86, 88, 186 and 188-based applications
- Provides interactive debugging commands including single-step code execution and symbolic displays of results
- Supports 8087 Numeric Processor Extension (NPX) for high-speed math applications
- Allows building of custom commands through the Command Extension Interface (CEI)
- Supports application access to ISIS-II files
- Provides program load capability from an Intellect® Development System
- Contains configuration facilities which allow an applications bootstrap from iRMX™ 86 and 88 file compatible peripherals
- Modular to allow use from an Intellect Development System or from a stand-alone terminal

The Intel iSDM™ 86 System Debug Monitor package contains the necessary hardware, software, cables, EPROMs and documentation required to interface, through a serial or parallel connection, an iSBC® 86/05, 86/12A, 86/14, 86/30, 88/25, 88/40, 88/45, 186/03, 186/51, 188/48, or iAPX 86, 88, 186 or 188 target system to an MDS 800, Series II, Series III, or Series IV Intellect® Microcomputer Development System for execution and interactive debugging of applications software on the target system. The Monitor can: load programs into the target system; execute the programs instruction by instruction or at full speed; set breakpoints; and examine/modify CPU registers, memory content, and other crucial environmental details. Additional custom commands can be built using the Command Extension Interface (CEI). The Monitor supports the OEM's choice of the iRMX™ 86 Operating System, the iRMX 88 Real-Time Multi-tasking Executive or a custom system for the target application system. OEM's may utilize any iRMX 86, 88 supported target system peripheral for a bootstrap of the application system or have full access to the ISIS-II files of the Intellect System.



The following are trademarks of Intel Corporation and may be used only to describe Intel products: Intel, ICE, iMMX, iRMX, iSBC, iSBX, iSXM, MULTIBUS, Multichannel and MULTIMODULE. 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. Information contained herein supersedes previously published specifications on these devices from Intel.

FUNCTIONAL DESCRIPTION

Overview

The iSDM 86 Monitor extends the software development capabilities of the Intellec system so the user can effectively develop applications to ensure timely product availability.

The iSDM 86 package consists of four parts:

- The loader program
- The iSDM 86 Monitor
- The Command Extension Interface (CEI)
- The ISIS-II Interface

The user can use the iSDM 86 package to load programs into the target system from the development system, execute programs in an instruction-by-instruction manner, and add custom commands through the command extension interface. The user also has the option of using just the iSDM 86 Monitor and the CEI in a stand-alone application, without the use of an Intellec development system.

Powerful Debugging Commands

The iSDM 86 Monitor contains a powerful set of commands to support the debugging process. Some of the

features included are: bootstrap of application software; selective execution of program modules based on break-points or single stepping requests; examination, modification and movement of memory contents; examination and modification of CPU registers, including NPX registers. All results are displayed in clearly understandable formats. Refer to Table 1 for a more detailed list of the iSDM 86 monitor commands.

Numeric Data Processor Support

Arithmetic applications utilizing the 8087 Numeric Processor Extension (NPX) are fully supported by the iSDM 86 Monitor. In addition to executing applications with the full NPX performance, users may examine and modify the NPX's registers using decimal and real number format.

This feature allows the user to feel confident that correct and meaningful numbers are entered for the application without having to encode and decode complex real, integer, and BCD hexadecimal formats.

Command Extension Interface (CEI)

The Command Extension Interface (CEI) allows the addition of custom commands to the iSDM 86 Monitor commands. The CEI consists of various procedures that can be used to generate custom commands. Up to three custom commands (or sets of commands) can be added

Table 1. Monitor Commands

Command	Function
B	Bootstrap application program from target systems peripheral device
C	Compare two memory blocks
D	Display contents of memory block
E*	Exit from loader program to ISIS-II Interface
F	Find specified constant in a memory block
G	Execute application program
I	Input and display data obtained from input port
L*	Load absolute Intellec® object file into target system memory
M	Move contents of memory block to another location
N	Display and execute single instruction
O	Output data to output port
P	Print values of literals
R*	Load and execute absolute Intellec® object file in target system memory
S	Display and (optionally) modify contents of memory
T*	Transfer block of memory to an Intellec® file
U,V,W	User defined custom commands extensions
X	Examine and (optionally) modify CPU and NPX registers

* Commands require an attached Series II/Series III.

to the monitor without programming new EPROMs or changing the monitor's source code.

ISIS-II Interface

The ISIS-II interface consists of libraries which contain interfaces to ISIS-II I/O calls. A program running on an iAPX 86, 88, 186 or 188-based system can use the ISIS-II interface and access the individual ISIS-II I/O calls. The interface allows the inclusion of these calls into the program; however, most of the calls require a Series II/ Series III system. Table 2 contains a summary of the major I/O calls and parameters.

Program Load Capability

The iSDM 86 loader allows the loading of iAPX 86, 88, 186 or 188-based programs into the target system. It executes on a Intellec Microcomputer Development System and communicates with the target system through a serial or a parallel load interface. If a Series II/ Series III/ Series IV system containing an Intel I/O expansion board is being used, the board can be used as a fast parallel load interface, freeing up the UPP port for application use.

Configuration Facility

The monitor contains a full set of configuration facilities which allow it to be carefully tailored to the requirements

of the target system. Pre-configured EPROM-resident monitors are supplied by Intel for the iSBC 86/05, 86/12A, 86/14, 86/30, 88/45, 186/03, 186/51, and 188/48 boards. The monitor must be configured by the user for the iSBC 88/25, 88/40 boards and for other iAPX 86, 88, 186, 188 applications. iRMX 86 and iRMX 88 system users may use the configuration facilities to include the iAPX 86, 88 Bootstrap Loader (V5.0 or newer) in the monitor.

Variety of Connections Available

The physical interface between the Intellec Microcomputer Development System and the target system can be established in one of three ways. The systems can be connected via a serial link, a parallel link or a fast parallel link. The fast parallel link requires the use of an iSBC 108(A), 116(A), 517 or 519 I/O expansion board in the Intellec system and is only available for connections with the Series II/ Series III/ Series IV systems. The cabling arrangement is different depending upon the development system being used. Figure 1 displays the cable connections needed between an Intellec Series III system and a target system for a serial interface.

The iSDM 86 Monitor does not require the use of a development system. The monitor can be used by simply attaching a stand-alone terminal to the target system. Figure 1 also displays the cable connections needed for this arrangement.

Table 2. Routines for ISIS-II Services Available to Target System Applications

Routine	Target System Function
ATTRIB	Changes to ISIS-II file attribute
CI	Returns a character input from the console
CO	Transfers a character for console output
CLOSE	Closes an opened ISIS-II file
DELETE	Deletes the specified ISIS-II file
DQ\$CFG	Returns information about monitor's communication link and type
ERROR	Displays an error message on the Intellec® console
EXIT	Exits to the target system monitor
LOAD	Loads target system memory with ISIS-II object code file
OPEN	Opens an ISIS-II file for access
READ	Reads up to 4096 bytes from an ISIS-II file to memory
RENAME	Renames an ISIS-II disk file
SEEK	Seeks to the specified ISIS-II file location
WRITE	Writes up to 4096 bytes from memory to an ISIS-II file

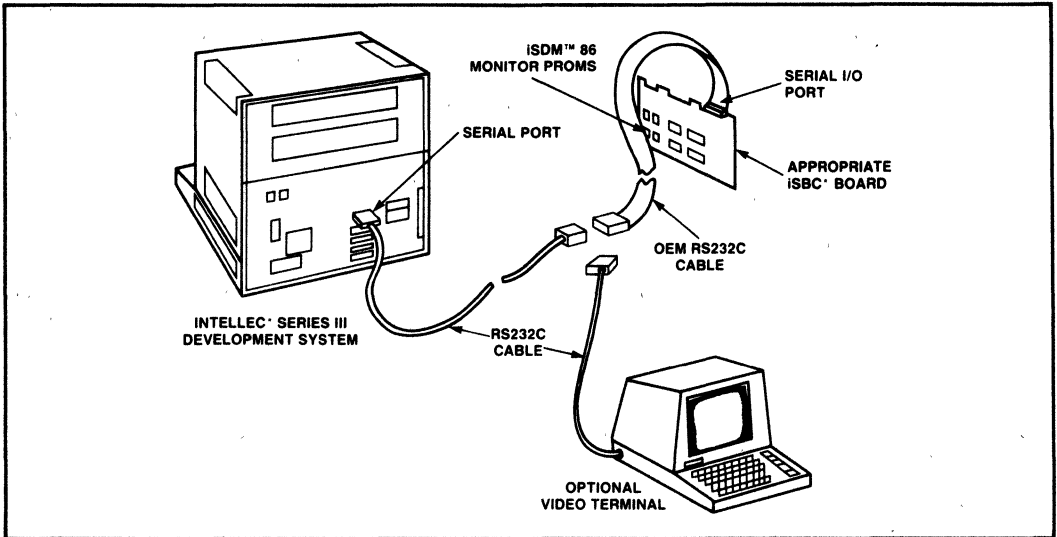


Figure 1. Typical iSDM™ 86 Serial Connection Environment

SPECIFICATIONS

Development System Environment

The Intellec Microcomputer Development System may be utilized for application program development and, if used, requires the following to support the iSDM 86 package:

- 48 Kbytes memory
- Double density or single density diskette subsystem
- ISIS-II Operating System and associated language translators

iAPX 86, 88, 186, 188 TARGET SYSTEM ENVIRONMENT

To support the iSDM 86 package, the target system must contain the following:

- 2K read-write memory beginning at location 0H
- 16K read-only memory beginning at location FC000H
- For Parallel link:
 - 8255A Programmable Peripheral Interface

- For Serial link:
 - 8251A USART or 8274 Multiprotocol Serial Controller, and 8253/4 or 80130 or iAPX 186/188 timer, or
 - 82530 Serial Communications Controller, including 82530 timer

Hardware

- Supported iSBC Microcomputers:

iSBC 86/05	Single Board Computer
iSBC 86/12A	Single Board Computer
iSBC 86/14	Single Board Computer
iSBC 86/30	Single Board Computer
iSBC 88/25	Single Board Computer
iSBC 88/40	Single Board Computer
iSBC 88/45	Single Board Computer
iSBC 186/03	Single Board Computer
iSBC 186/51	Single Board Computer
iSBC 188/48	Single Board Computer

- Supported iSBX MULTIMODULE™ Boards:

iSBX 350 Parallel I/O MULTIMODULE Board
iSBX 351 Serial I/O MULTIMODULE Board

iSDM™ 86 Package Contents
Cables:

- 1 — Parallel I/O Cable (upload/download)
- 2 — RS232 Cables

Adaptors:

- 1 — Parallel Status Adaptor
- 1 — Parallel Adaptor

I/O Drivers and Terminators:

- 4 — Pull-up Resistor Packs
- 4 — Pull-up/down Resistor Packs
- 4 — Line Driver Packs

Interface and Execution Software Diskettes:

- 1 — Single Density, ISIS Compatible
- 1 — Double Density, ISIS Compatible

System Monitor EPROMs:

Microcomputer	EPROM
iSBC® 86/05 iSBC® 86/12A iSBC® 86/14 iSBC® 86/30	Four 2732A EPROMs
iSBC® 88/45	Two 2764 EPROMs
iSBC® 186/03 iSBC® 186/51	Two 2764 EPROMs
iSBC® 188/48	Two 2764 EPROMs

Reference Manual (Supplied):

146165-001 — iSDM 86 System Debug Monitor Reference Manual

ORDERING INFORMATION
Part Number Description

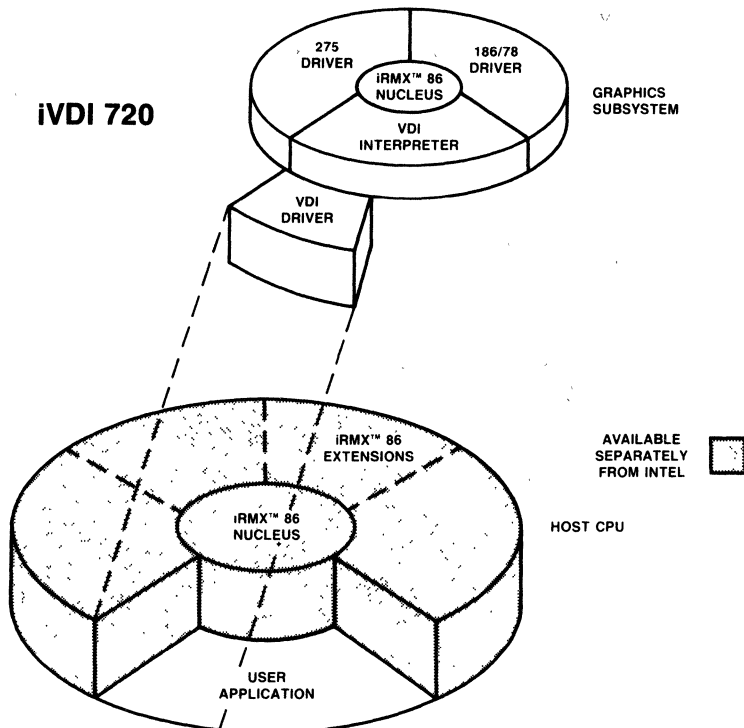
iSDM 86	<p>Intellec to target system interface and target system monitor, suitable for use on iSBC 86, 88, 186, 188 computers, or other iAPX 86, 88, 186, 188 microcomputers. Package includes cables, EPROMs, software and operator manual.</p> <p>The iSDM 86 package includes SPR Service for 90 days after shipment.</p> <p>As with all Intel Software, purchase of any of these options requires execution of a standard Intel Master Software License.</p>
iSDM 86 RO	Object Software
iSDM 86 BSR	Machine Readable Source

iVDI 720 GRAPHICS VIRTUAL DEVICE Interpreter

- Provides standardized decoding of high-level graphics commands
 - Full iRMX™ 86 compatibility operating system (Rel. 6)
 - Standardized input & output drivers
 - Compact for EPROM installation
- Support for iSBX™ 275 and iSBC® 186/78 Graphics hardware modules
 - Procedural interface from Pascal 86, PL/M 86 and Fortran 86
 - Compatible with (proposed) ANSI X3H33 specification
 - Virtual Device Metafile interpreter

The Intel iVDI 720 Graphics Virtual Device Interpreter provides both a powerful library of high-level commands, and the drivers necessary to support the iSBX™ 275 or iSBC® 186/78 graphics modules in an iRMX™ 86 (Release 6) environment. It allows the OEM to quickly tailor an Intel system for application into the rapidly growing graphics marketplace, especially low-cost CAD/CAE, CAM, and process control. Individual single-board computer (SBC) modules may also be configured from Intel's broad product family.

For intra-systems graphics control, iVDI 720 is the most powerful and efficient product available that brings (proposed) ANSI X3H33 compatibility to an iRMX 86 operating system environment.



The following are trademarks of Intel Corporation and may only be used to describe Intel Products: Intel, ICE, iRMX, iSBC, iSBX, iSXM, MULTIBUS, MULTICHANNEL, MULTIMODULE, and ICS. Intel Corporation assumes no responsibility for the use of any circuitry other than the circuitry embodied in an Intel Product. No other circuit patent licenses are implied

FUNCTIONAL DESCRIPTION
Graphics Standard Software

The iVDI 720 Graphics Virtual Device Interpreter implements the proposed ANSI standard on any Intel-based graphics system running under the iRMX 86 operating system, release 6. The proposed standard is a significant advancement in graphics software. It creates a predictable environment for the input and output of high-level commands between the user and system, or among the graphics peripherals attached to the system, such as a mouse, tablet, printer or plotter. The software supports two environments: stand-alone and distributed, depending on the hardware configuration.

All elements of iVDI 720 can run as tasks of the operating system or as part of the graphics application program, hence a stand-alone partitioning of graphics activities such as with the iSBX 275 MULTIMODULE™ attached to a general purpose CPU board like the iSBC 86/30. In a distributed environment, the device driver runs under the iRMX 86 operating system and the remaining application code and VDI interpreter are exercised by a separate processor dedicated to graphics activities. The iSBC 186/78 subsystem was designed especially for the distributed solution.

iSBC® 186/78 Graphics Subsystem Support

By virtue of its on-board, high integration microprocessor (the Intel 80186), the iSBC 186/78 subsystem is an excellent platform on which to perform graphics routines in a distributed environment. This is particularly important in multi-user systems where one iSBC 186/78 subsystem can be dedicated to each user. (see figure 1)

The compact coding of the iVDI 720 Graphics Virtual Device Interpreter lends itself to EPROM installation on the iSBC 186/78 subsystem. The host CPU board is thereby off-loaded from graphics activities so it can direct more global system level operations such as database management or network communications.

iSBX™ 275 Graphics MULTIMODULE™ Support

In single-user applications or where graphics activities are not the major focus of the system, the iSBX 275 MULTIMODULE shares the CPU on the host processor board through the iSBX expansion bus. The subsystem formed in this manner supports either monochrome or eight colors and is a very cost-effective solution. Like the iSBC 186/78 subsystem,

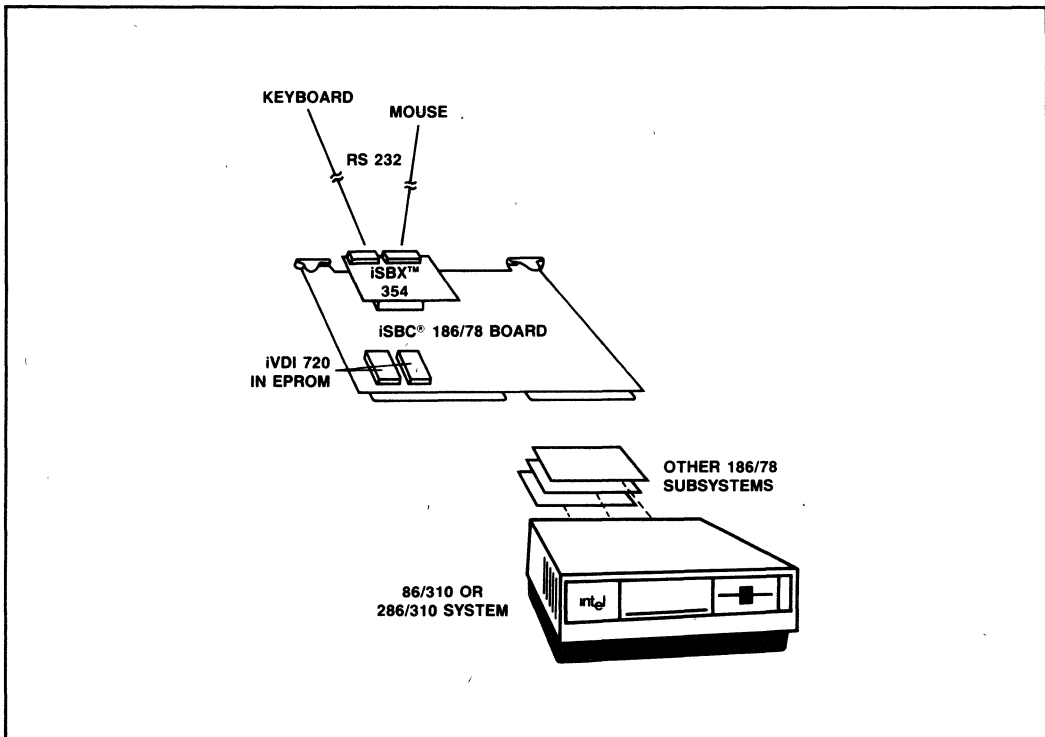


Figure 1. Multi-User Example

this expansion module is based on the Intel 82720 Graphics Display Controller (GDC) component. (see figure 2)

For example using an iSBC 86/30 CPU board, the iVDI 720 library can be installed in EPROM to simplify the application and provide higher performance execution.

82720 Component Designs

The Intel 82720 GDC is an intelligent graphics controller component designed to operate as the heart of a raster-scan computer graphics display system. The 82720 performs all the basic timing needed to generate the raster display and manage the display memory. In addition, it supports several high-level graphics figure drawing functions. The Intel 82720 is an alternative to the NEC 7220 component.

VDI COMMAND LIBRARY

In addition to providing driver support for Intel's growing family of graphics modules, the iVDI 720 Graphics Virtual Device Interpreter decodes a wealth of high-level commands to streamline the development of application code for a variety of graphics devices.

The proposed ANSI standard provides multiple encodings of high-level text and graphics commands and capabilities. The iVDI 720 software decodes a binary representation of these proposed commands, along with the Virtual Device Metafile (VDM) routines that allow consistent formatting and storage of VDI encoded images.

In addition to a full set of inquiry functions, many additional high-level commands are supported in the iVDI 720 software. (See Table 1)

These features are configurable as defined in the iVDI 720 Software Reference Manual. However, they are typically device dependent and therefore reflect the users application. Consequently, the reference manual should be consulted to assure compatibility.

DEVELOPMENT ENVIRONMENT

Intel's family of development systems and their extensions are highly recommended for both the development of iVDI 720 and related application code. Languages that are supported include Fortran 86, Pascal 86 and PL/M 86. All iVDI 720 commands can be called from any of these programming languages through the PL/M 86 procedural interface that is integral to the iVDI 720 product.

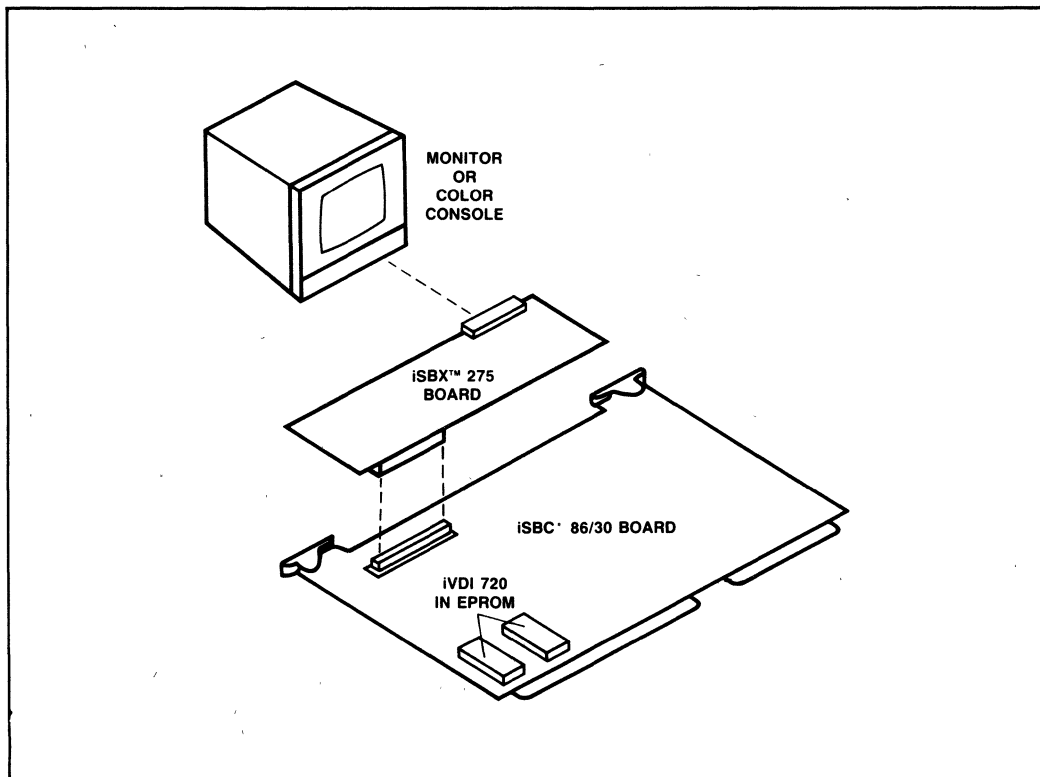


Figure 2. Single-User Example

Table 1. iVDI 720 Command Library

Graphical Elements: Polyline Polymarker Polygon Circle Arc Arc Close (Pie or Chord) Text Append Text Cell Array		Control & Descriptor Elements: Begin & End Metafile Begin & End Picture Background Color VDC Extent Clip Rectangle Clip Indicator Clear Surface Defaults Replacement Set Device Viewpoint Color Direct Precision Scaling Mode Color Specification Marker Size Mode Mode	
Attribute Elements: Aspect Source Flags Bundled & Individual Character Orientation Attributes Character Path Character Height Character Spacing Character Expansion Text Alignment Factor Perimeter Type & Color Interior Style Hatch Fill Marker Type & Color Pattern Fill Line Type & Color Pattern Definition Set Color Table Text Precision Pattern Size String Pattern Reference Point Character Text Color Stroke		Input Elements: Initialize Locator Initialize String Sample Locator Sample String Request Locator Request Locator Set Prompt State Set Echo State Release Input Device Set Input Device Mode	

SPECIFICATIONS

ANSI X3H33 VDI Specification

The American National Standards Institute (ANSI) administers the standard specification. Requests for information should be directed to:

X3 Secretariat
 Computer Business Equipment Manufacturers Association (CBEMA)
 311 First Street, NW
 Washington, D.C. 20001

Intel is heavily involved in the development of the ANSI X3H33 Virtual Device Interface standard. We will endeavor to bring to our user base the latest revisions through phased introductions and updates. Consequently, it is strongly advised that implementers of iVDI 720 also subscribe to the update service (VDI 720 WX, see below).

iVDI 720 Specifications

Code size — 80 Kbytes in distributed mode (using the iSBC 186/78 subsystem), including the iRMX 86 nucleus

Code size — 64 Kbytes in stand-alone mode (using the iSBX 275 MULTIMODULE)

Source-code language — PL/M 86

Related Literature

Reference material may be ordered from any Intel sales representative, distributor office or from Intel Literature Department, 3065 Bowers Avenue, Santa Clara, Calif., 95051.

146717 — iVDI 720 Software Reference Manual

210506 — iSBX 275 Video Graphics Controller Data Sheet

231035 — iSBC 186/78 Video Graphics Subsystem Data Sheet

146666 — iSBC 186/78 Video Graphics Subsystem Hardware Reference Manual

210655 — 82720 GDC Component Data Sheet

9803126 — iRMX 86 Configuration Guide

Ordering Information

Intel makes available a variety of licensing programs to the iVDI 720 Graphics Virtual Device Interpreter which allow different plans for incorporation of the Intel software into the final product. The Intel Master software Agreement should be consulted to determine which plan is best suited for the particular application and production environment.

The iVDI 720 Graphics Virtual Device Interpreter comes in three formats as shown below, along with

source listings and update services. The iRMX 86-Real-time Multitasking Operating System is available separately.

iVDI 720RO OEM license (8 inch single-sided/double dense ISIS and iRMX plus 5¼ inch double-sided/double density iRMX formats are supplied)

iVDI 720RF Incorporation fee payment

iVDI 720WX Object code update

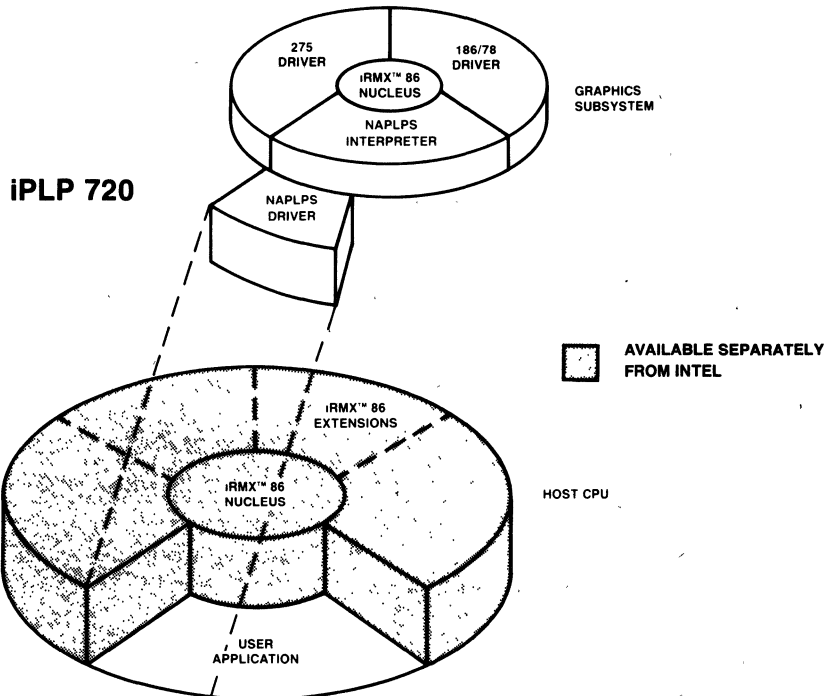


iPLP 720 NAPLPS Interpreter

- Provides decoding of NAPLPS (Videotex) commands
- Multiple font sizes and definitions
- Full iRMX™ 86 compatibility
- Compact for EPROM installation
- Compatible with ANSI BSR X3.110 -1983
- Driver support for Intel Graphics hardware modules
- Complete library of high-level frame management instructions
- Resolution independent presentation
- Simplified communication of Graphic images between systems

The Intel iPLP 720 NAPLPS (North American Presentation Level Protocol Syntax) Interpreter provides both a powerful library of high-level commands, and the drivers necessary to support the iSBX™ 275 or iSBC 186/78 Graphics Modules in an iRMX™ 86 Operating System (Release 5 or later) environment. It allows the OEM to quickly tailor an Intel system for application into the rapidly growing Videotex marketplace. Individual iSBC® modules may also be configured from Intel's broad product family, and iPLP 720 will also be convenient for the those implementing custom, component designs based on the Intel 82720 Graphics Display Controller (GDC).

Regardless of the hardware configuration, iPLP 720 is the most powerful and efficient product available that brings full ANSI X3L2 Videotex compatibility to an iRMX 86 environment.



The following are trademarks of Intel Corporation and may be used only to describe Intel products: Intel, ICE, iMMX, iRMX, iSBC, iSBX, iSXM, MULTIBUS, Multichannel and MULTIMODULE. 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. Information contained herein supercedes previously published specifications on these devices from Intel.

FUNCTIONAL DESCRIPTION

NAPLPS Interpreter

The iPLP 720 software implements the NAPLPS videotex standard on any 82720-based graphics application using the iRMX 86 Operating System, Release 5 or later. The NAPLPS Standard is a significant advancement in device-level graphics software. It creates a predictable environment for development of presentation frames along with a transmission scheme to dramatically improve performance when moving graphics images from one system to another. The software supports two environments: stand-alone and distributed, depending on the hardware configuration.

All elements of iPLP 720 can run as tasks of the operating system or as part of the graphics application program, hence a stand-alone partitioning of graphics activities such as with the ISBX 275 MULTIMODULE™. In a distributed environment, the device driver runs under the iRMX 86 Operating System and the remaining controller code and NAPLPS interpreter are exercised by a separate processor dedicated to graphics activities. The iSBC 186/78 subsystem was designed especially for the distributed solution.

This architecture also allows iPLP 720 to run under non-iRMX environments.

iSBC® 186/78 Video Graphics Subsystem Support

By virtue of its on-board, high integration microprocessor (the Intel 80186), the iSBC 186/78 subsystem is an excellent platform on which to perform graphics routines in a distributed environment. This is particularly important in multi-user systems where one iSBC 186/78 subsystem can be dedicated for each user. The host CPU board is thereby off-loaded to direct more global system level operations such as database management or network communications.

The combination of the iSBC 186/78 subsystem and iPLP 720 interpreter meets all requirements of ANSI's NAPLPS Standard Reference Model (SRM).

ISBX™ 275 Graphics Controller Support

In single-user applications or where graphics activities are not the major focus of the system, the iSBC 275 MULTIMODULE shares the CPU on the host processor board through the iSBX Expansion Bus. The subsystem formed in this manner supports either monochrome or eight colors and is a very cost-

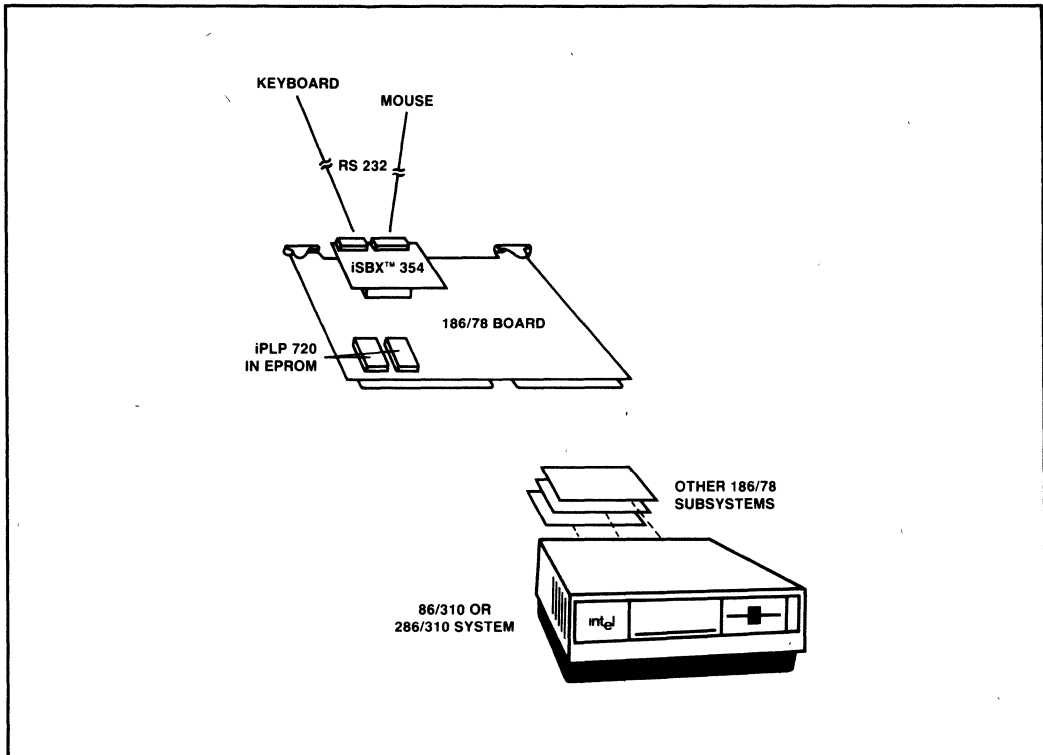


Figure 1. Multi-User Example

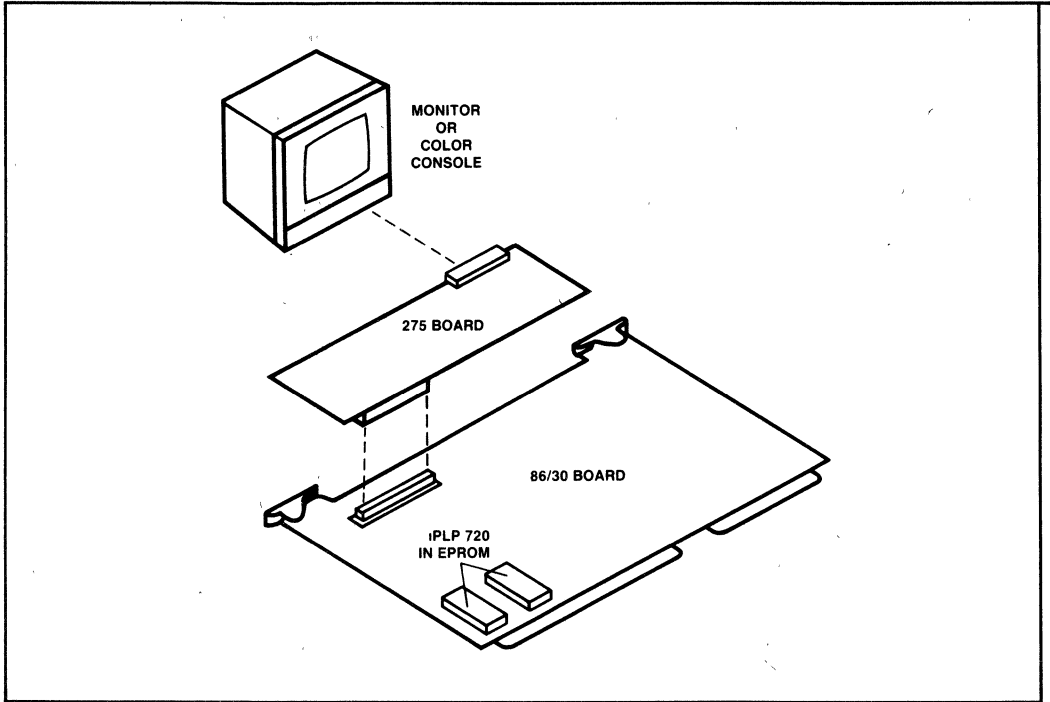


Figure 2. Single-User Example

effective solution. Like the iSBC 186/78 subsystem, this expansion module is based on the Intel 82720 component.

82720 Component Designs

The Intel 82720 GDC is an intelligent graphics controller component designed to operate as the heart of a raster-scan computer graphics display system. The 82720 performs all the basic timing needed to generate the raster display and manage the display memory. In addition, it supports several high-level graphics figure drawing functions. The Intel 82720 is an alternative to the NEC 7220 component. Custom hardware designs based on either component are supported by iPLP 720 running in an iRMX 86 environment.

NAPLPS COMMAND LIBRARY

In addition to providing driver support for Intel's growing family of graphics products, the iPLP 720 NAPLPS Interpreter decodes a wealth of high level commands to streamline the development of application code for videotex environments.

The NAPLPS standard provides character set encodings of high level text and graphics commands and capabilities. The iPLP 720 NAPLPS Interpreter implements every one of these character encodings.

The following table lists the major high level commands which are encoded in the NAPLPS standard:

Table 1. iPLP 720 Command Library

Geometric Drawing Primitives:	
Point	Incremental Point
Line	Incremental Line
Polygon	Incremental Polygon
Arc	Rectangle
Text:	
Character Rotation	Cursor Styles
Character Path Movement	Word or Character Wrap Around
Inter-character Spacing	Inter-row Spacing
Character Field Dimensions	Scrolling
Texture:	
Line Texture	
Highlighting	
Programmable Texture Patterns	
Texture Mask Size	
Miscellaneous Functions:	
Logical Pel Size	Set Color
Macro Definitions	Mosaic Sets
Blink Processes	Wait Intervals
Dynamically Redefinable Character Sets	

Other features are configurable as defined in the standard, however they are typically device dependent and therefore reflect the users application. Consequently, the standards document should be consulted to assure compatibility.

ANSI X3L2 NAPLPS Specification

The development of any standard is an evolutionary process. Consequently, the specification used as a reference model for the design of iPLP 720 was published in October 1982. Updates to iPLP 720 will render it fully compatible with the final specification. The American National Standards Institute (ANSI) administers the standard specification and makes it available to all interested parties. Requests for copies should be directed to:

X3 Secretariat
Computer Business Equipment Manufacturers
Association (CBEMA)
311 First Street, NW
Washington, D.C. 20001

Reference document: BSR X3.110

iPLP 720 Specifications

Code size — 90 Kbytes (four 27256 EPROMs) in distributed mode;
— 72 Kbytes in stand-alone mode

Source-code language — PL/M 86

Related Literature

Reference material may be ordered from any Intel sales representative, distributor office, or from Intel Literature Department, 3065 Bowers Avenue, Santa Clara, Calif., 95051.

- 146144** — iPLP 720 Software Reference Manual
- 210506** — iSBX™ 275 Video Graphics Controller Data Sheet
- 231035** — iSBC® 186/78 Video Graphics Subsystem Data Sheet
- 146666-001** — 186/78 Reference Manual.
- 210655** — 82720 GDC Component Data Sheet
- 9803126** — iRMX™ 86 Configuration Guide
- 145412** — Intel's Guide to Understanding the ANSI Videotex/Teletex Standard

ORDERING INFORMATION

Intel makes available a variety of licenses to the iPLP 720 NAPLPS interpreter that allow different plans for incorporation of the Intel software into the final product. The Intel Master Software Agreement should be consulted to determine which is best suited for the particular application and production environment.

The iPLP 720 NAPLPS Interpreter comes in two formats as shown below, along with yearly update serv-

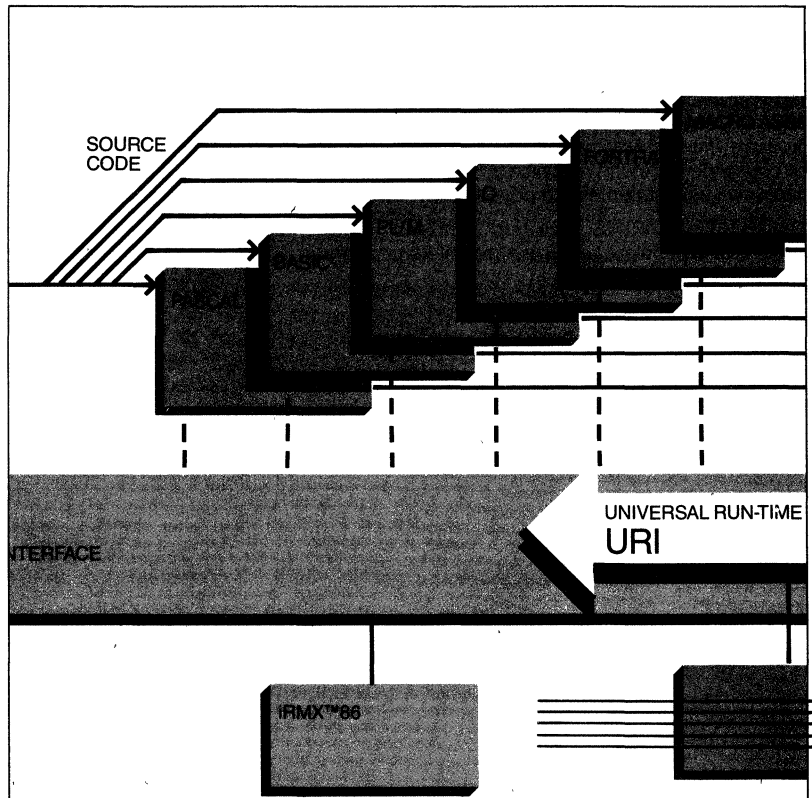
ices. The iRMX 86 Real-time Multitasking Operating Systems are available separately.

- | | |
|------------|---|
| iPLP 720RO | OEM License, single density (both ISIS and iRMX formats are included) |
| iPLP 720RF | Incorporation fee payment |
| iPLP 720WX | Object code update |



iRMX™ LANGUAGE

- Industry-standard languages and utilities for developing applications on iRMX-based systems. Includes FORTRAN, Pascal, C, BASIC, PL/M, assembler, text editor
- Complete set of utilities to create and manage object modules
- Mix languages on single application system with UDI standard
- Intel 8087 and 80287 math coprocessor support
- 8086 and 80286 compatibility
- Worldwide post-sales service and support organization



Full Language Support for iRMX™ -Based Systems

Intel's iRMX™ 86 based systems are completely supported by a wide variety of popular languages and utilities with which to build fast, real-time, multi-tasking applications. Included are the latest versions of FORTRAN, Pascal, BASIC, C, PL/M and Assembler for Intel's iAPX 86 and iAPX 286 processors. Previously developed applications using any of these languages port easily to iRMX-based systems with minimal source code modifications.

In addition to the wealth of languages available, iRMX-based systems are complemented by utilities with which to create and manage object modules. This latitude in configurability allows programmers to team their efforts in order to achieve a shorter development time than would otherwise be possible.

Because the high-level languages are actually resident on the iRMX-based system, OEMs can pass application software directly on to end users. End users may then tailor the OEM's system to better meet application needs by writing programs using the same languages.

Language-Independent Application Development

Intel's Universal Development Interface (UDI) and Object Module Format (OMF) enable several users to write different modules of an application, in different languages, then link them together.

The OMF provides users with the ability to mix languages on a single application system, affording the luxury of choosing exactly the right language tools for specific pieces of the application, rather than compromising specialized tasks for the sake of one, project-wide language.

iRMX languages are fully compatible with the Intel Series III and Series IV Development Systems, should the user choose to develop applications on a specialized development system. Applications are easily moved to the final target system for test, debug and minor redevelopment.

Fast, Lean Programs for Rapid Processing

The iRMX language products enable programmers to write the smallest, fastest programs available in high-level languages, due to the compiler's superior ability to optimize code.

It is also possible to make iRMX operating system calls directly from FORTRAN, PASCAL and PL/M. This means that application developers can take full advantage of the iRMX multi-tasking capability, whereby multiple applications execute concurrently on the operating system. Multi-tasking, a requirement of most real-time systems, is sometimes as necessary in application software development as in an operating system environment.

Standardized REALMATH Support

All the iRMX languages (except BASIC and C) support the REALMATH floating point standard. This ensures universal consistency in numeric computation results and enables the user to take advantage of the Intel iAPX 86/20 and iAPX 88/20 Numeric Data Processor or iSBC® 337 MULTIMODULE™ boards, which boost performance two to four times over that possible on a mini-computer.

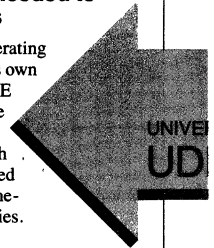
All the Utilities Needed to Link Languages

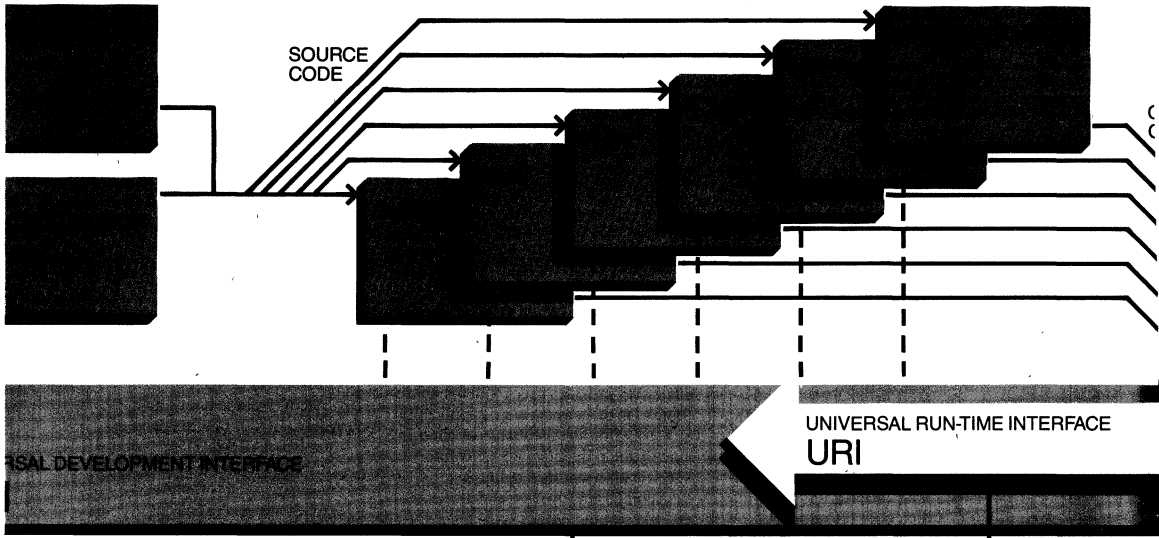
Utilities for iRMX operating systems include Intel's own EDIT, LINK, LOCATE and LIBRARIAN. The iRMX EDIT program meets the needs of both novice and sophisticated users with powerful line-oriented editing facilities.

Using the iRMX LINK program, users may link individually compiled object modules to form a single, relocatable object module. This provides the ability to merge work from several programmers into one cohesive application system.

The iRMX LOCATE utility maps relocatable object code into the processor memory segments, allowing user definition of module/memory type allocation. For example, often-used portions of an application may be mapped to (P)ROM.

The LIBRARIAN object code library manager affords easy creation, collection and maintenance of related object code to reduce the overhead of separately maintained modules.



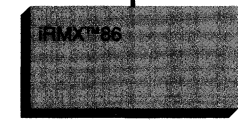


Finally, the iRMX Assembler for the iAPX 86 and iAPX 286 processors generate extremely efficient code and invoke 8086/8087 machine instructions.

iRMX™ 86 Pascal

iRMX Pascal meets the proposed ISO language standard and implements several microcomputer extensions. A compile-time option checks conformance to the standard, making it easy to write uniform code. Industry-standard specifications contribute to portability of application programs and provide greater reliability.

iRMX 86 Pascal supports extensions, such as an interrupt-handler and direct



port I/O extension, that allow programs to be written specifically for microcomputers. Separate module compilation allows linkage of Pascal modules with modules written in other high-level languages.

For more information on iRMX 86 Pascal see the Pascal 86 Software Package data sheet.

iRMX™ 86 FORTRAN

The iRMX 86 FORTRAN compiler provides total compatibility with FORTRAN



66 language standards, plus most new features provided by the FORTRAN 77 language standard. iRMX 86 FORTRAN includes extensions specifically for microcomputer application development. Programming is simplified by relocatable object libraries, which provide run-time support for execution time activities.

iRMX 86 FORTRAN supports the 8087 math coprocessor for the most powerful

microcomputer solution available in number-intensive applications. For more information on iRMX 86 FORTRAN see the FORTRAN 86 Software Package data sheet.

iRMX™ 86 PL/M

PL/M offers full access to micro-computer architecture while simultaneously offering all the benefits of a high-level language. Invented by Intel in 1976, PL/M 80 was the first microcomputer-specific, block-structured, high-level language available. Since then, thousands of users have generated code for millions of microcomputer-based systems using PL/M 80 and PL/M 86.

Software written for 8-bit processors (PL/M 80) are easily ported to the more powerful 16-bit (PL/M 86) environment. The same portability will be available for future VLSI.

For more information about iRMX 86 PL/M see the PL/M 86/88 Software Package data sheet.

iRMX™ 86 BASIC

Intel's offering of Microsoft BASIC is a standardized version of the most popular high-level language in the world. Existing BASIC programs are easily ported to iRMX-based systems. BASIC is an excellent pass-through language by which an OEM can offer customers the ability to write and modify their own applications.

iRMX™ 86 C Compiler

The popular new programming language, C (Mark William's Company version), is fully supported on iRMX-based systems. iRMX 86 C offers both small and large

segmentation models, enabling applications to be written efficiently. The iRMX 86 C compiler combines assembly language efficiency with high-level language convenience; it can manipulate on a machine-address level while maintaining the power and speed of a structured language.

The iRMX 86 C compiler affords easy portability of existing C programs to iRMX-based systems. For more information on the iRMX C compiler see the iRMX 86 C Software Package data sheet.

iRMX™ 86 Text Editor

The iRMX 86 Text Editor is screen-oriented, menu-driven and easy to learn. Guided by the menu of commands always before him, the user can edit text and programs easily and efficiently.

iRMX 86 Text Editor allows the simultaneous edit of two files. This allows easy transference of text between files and use of existing material in the creation of new files. Creating macros, strings of frequently-used commands, is also very simple. The editor "remembers" the selected commands and allows the user to re-use them repeatedly.

Worldwide Service and Support

All iRMX systems are completely supported by Intel's worldwide staff of trained hardware and software engineers. iRMX Language customers receive a warranty that includes Hotline Support, Software Updates, and Subscription Service.

Complete documentation is provided for all operating system and application software languages, as well as for system hardware components. An Intel system is not a collection of hardware and software pieces as much as a cohesive whole that is supported and serviced as such.

Intel Has Total Solutions for Real-Time Systems

iRMX 86 is the fastest, most powerful operating system available for multi-tasking, multi-user, real-time applications. Complemented by a wide range of industry-standard languages and utilities, the iRMX-based systems are highly flexible and configurable.

Application development for iRMX-based systems is possible at the board or the system level. OEMs can integrate functionality at the most profitable level of product design, using one system for both development and target use. Intel's choice of industry standard high-level languages enables the end user to extend OEM-provided functionality even further, if desired.

Who is better qualified to write and supply software for Intel VLSI than Intel? Today you have the ability to tap into hundreds of available application software packages, languages and utilities, peripherals and controllers and MULTIBUS® boards.

Tomorrow, and ten years down the road, you will be able to tap into the latest, high-performance VLSI—without losing today's software investment.



Specifications

<p>Required Hardware</p> <ul style="list-style-type: none"> • Any iAPX 86/286 based or iSBC 86/286 based system including Intel's System 86/300 and 286/300 family. In addition, object code from the compilers will run on iAPX 88 based systems. • 140KB of memory • Two iRMX 86 compatible floppy disks or one hard disk • One 8" double density or 5.25" double-density floppy disk drive for distribution of software • System console device 	<p>Required Software</p> <p>The iRMX 86 Operating System Release 6 or later including the nucleus, basic I/O system, extended I/O system and human interface</p> <p>Purchase of any RMX language requires signing of Intel's OEM License Agreement (OLA).</p>	
--	--	--

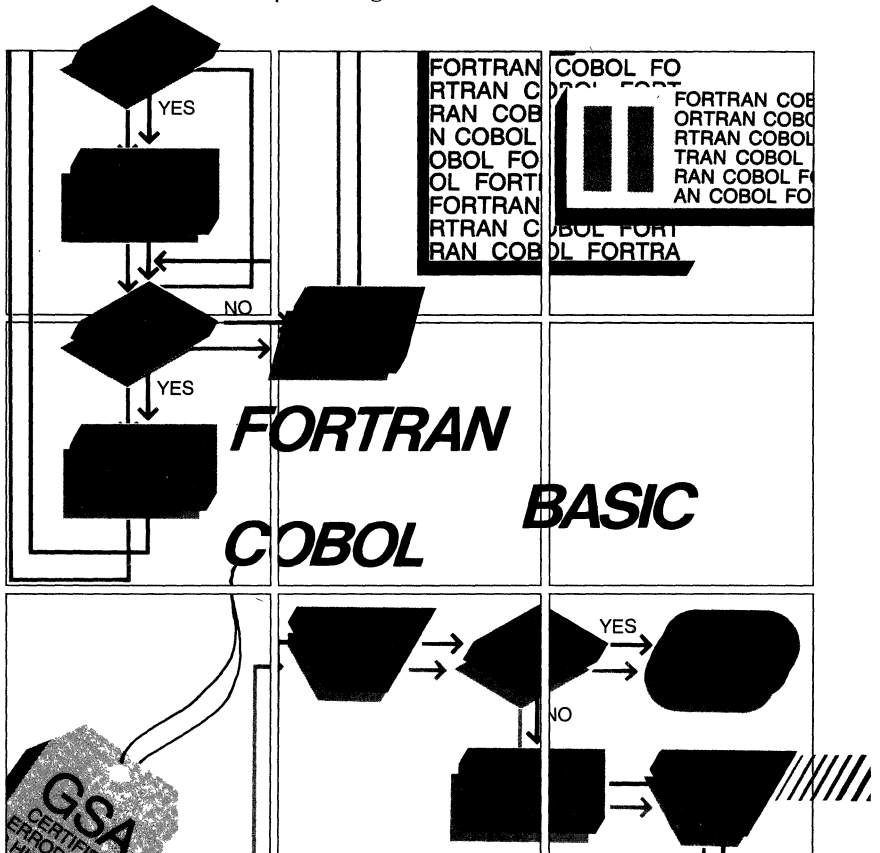
Ordering Information

Language	Order Code	Product Contents	Warranty
ASM 86, Utilities	RMX 860	Two 8" disk and two 5.25" diskettes Edit Reference Manual iAPX 86/88 Family Utilities User's Guide Macro Assembler Operating Instructions ASM 86 Language Reference Manual 8087 Support Library Reference Manual	90 days: Software Updates, Subscription Service, Hotline Support
Pascal	RMX 861	Two 8" diskettes and two 5.25" diskettes Pascal 86 User's Guide	90 days: Software Updates, Subscription Service, Hotline Support
FORTRAN	RMX 862	Two 8" diskettes and two 5.25" diskettes FORTRAN 86 User's Guide	90 days: Software Updates, Subscription Service, Hotline Support
PL/M	RMX 863	One 8" diskette and one 5.25" diskette PL/M 86 User's Guide	90 days: Software Updates, Subscription Service, Hotline Support
TX Editor	RMX 864	One 8" diskette and one 5.25" diskette TX Screen Echter User's Guide	90 days: Software Updates, Subscription Service
BASIC	RMX 865	One 8" diskette and one 5.25" diskette BASIC Reference Manual BASIC 86 User's Guide One 8" diskette and one 5.25" diskette	90 days: Software Updates, Subscription Service
C	RMX 866	One 8" diskette and one 5.25" diskette <i>C Programming Language</i> by Kernighan and Ritchie (Prentice-Hall) C 86 Compiler User's Guide	90 days: Software Updates, Subscription Service, Hotline Support



XENIX* LANGUAGES

- COBOL, BASIC and FORTRAN support for Xenix-based systems
- Conformation to international standards: ANSI 77 subset FORTRAN, ANSI X3.23 1974 COBOL to Federal High Level and ANSI X3.60—1978 subset BASIC
- Powerful microcomputer extensions to ANSI standards
- Easy porting of mainframe and minicomputer applications to micro environment
- Intel 80287 math coprocessor support
- Worldwide service and support organization



High-level Language Support for XENIX-Based Systems

Intel's XENIX operating system, available for component, board, or system-level integration, is a multi-user operating system well suited for both technical and commercial interactive applications. Typical applications include small business systems, software development/engineering workstations, distributed data processing and graphics.

For OEM and end-user application development on XENIX, Intel has provided three industry-standard, high-level languages—FORTRAN, COBOL and BASIC—with which to build microcomputer-based solutions for systems products or component and board-level applications. XENIX BASIC, FORTRAN and COBOL accommodate easy porting of existing mainframe and mini-based applications to the micro environment.

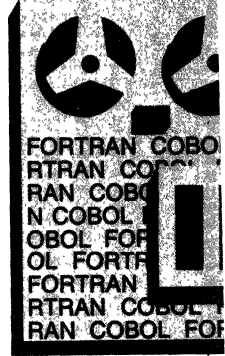
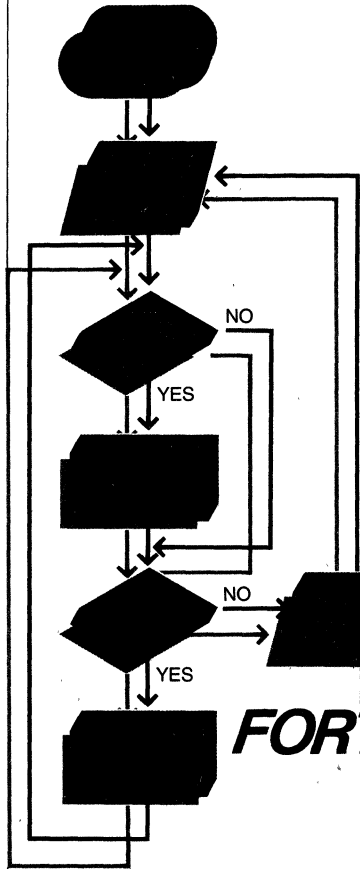
XENIX FORTRAN for Scientific and Technical Applications

FORTRAN is the most popular programming language for scientific and numerical applications. There are thousands of existing FORTRAN programs and subroutines written in mainframe and minicomputer environments, most of which can be ported to a micro environment via Intel's offering of Microsoft FORTRAN.

Compliance with the X3.9 1978 ANSI standard for FORTRAN at the subset level ensures portability with minimal source code modifications. By moving to a microcomputer-based system, you lose none of your mainframe and mini-developed software investment.

Speed and Accuracy Where They're Needed

Scientific, math-oriented applications usually require fast, highly accurate processing. XENIX FORTRAN delivers accuracy with double-precision arithmetic



which handles numbers containing 15 significant digits.

High speed results from XENIX FORTRAN support of the Intel 80287 floating point coprocessor, as well as from an extensive subroutine library, which includes subroutines for 16- and 32-bit integer arithmetic and 32- and 64-bit floating-point arithmetic. Because of XENIX FORTRAN's 80287 math coprocessor support, some programs written in XENIX FORTRAN will execute from two to four times faster than their minicomputer counterparts.

Calls to "C" and MS MACRO Assembler are possible, making it easy to interface non-standard peripherals to XENIX FORTRAN programs.

FORTRAN

XENIX COBOL for the Micro Environment

Intel's offering of Microfocus COBOL is a mainframe-caliber compiler for ANSI 1974 COBOL programs, enabling XENIX-based systems to compile and run existing COBOL programs with minimal source code modification. XENIX COBOL also contains features specifically aimed at facilitating the interactive

*XENIX is a trademark of Microsoft Corporation

BASIC

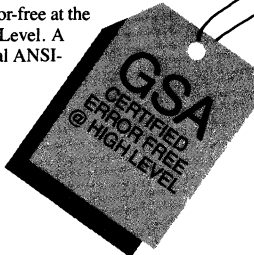
FORTRAN COBOL F
FORTRAN COBOL FO
FORTRAN COBOL FOR
FORTRAN COBOL FOR
FORTRAN COBOL FOR
FORTRAN COBOL FOR

program development of new applications in a microcomputer environment.

These features include a facility for dynamically loading sub-programs from disk as required which effectively removes limits on the size of the application code that can be run. XENIX COBOL augments the functionality of the ANSI standard with additional compiler features, such as interactive screen-handling, that further increase convenience and programmer productivity.

Users can license a separate run-time support package. This enables OEMs to pass COBOL applications onto customers at a much lower cost than that involved in transferring full COBOL packages.

XENIX COBOL is one of only eleven COBOL compilers in existence—and the only one for microcomputers—that has been GSA-certified as error-free at the High Level. A special ANSI-



defined communications module provides the user with a standard mechanism for program-to-program message-passing in multi-user networks such as those found in an "office of the future" settings.

Forms-2™ Support for Screen-Painting

XENIX COBOL supports FORMS-2, a powerful visual programming tool that speeds the creation of programs involving interactive screen-handling. In an extremely user-friendly environment, the user "paints" a form on the screen, and FORMS-2 generates the COBOL source code to support it. FORMS-2 results in greatly improved programmer productivity in a microcomputer, screen-building environment.

XENIX BASIC for Maximum Flexibility

Intel's offering of Microsoft BASIC opens a whole window of applications to the XENIX user. Since their BASIC is the same as that used on MS-DOS* based machines, most programs written for MS-DOS can now run on XENIX unchanged. When developing your own

programs, BASIC is simple and easy for quick prototyping, yet complete enough for total development. Conforming to the ANSI X3.60 1978 subset standard, BASIC also has powerful extensions, 16 significant digit Double Precision floating point arithmetic, 80287 support, and assembly languages routine calling capabilities. From using applications to designing your own programs BASIC is easy, complete, and extremely flexible.

Worldwide Service and Support

All XENIX systems are fully supported by Intel's worldwide staff of trained hardware and software engineers. Complete documentation is provided for all operating systems and application software languages, as well as for system hardware components. The XENIX and XENIX Languages warranty includes Hotline support, Software Updates, and Subscription Service.

Total Solutions for Interactive, Multi-User Applications

Intel's XENIX-based systems offer the most complete solutions for interactive, multi-user applications requiring fast, accurate throughput and a friendly programming environment. XENIX is complemented by industry-standard, high-level languages with which OEMs can create flexible and open end-user systems.

XENIX languages are completely portable—from one level of integration to another (chip to board to system).

Intel is paving the way into the future of VLSI and pioneering VLSI-based systems. We are committed to providing customers with smooth, uninterrupted application development on the latest VLSI-based systems - today and tomorrow.

COBOL

XENIX* LANGUAGES



Specifications

<p>Required Hardware:</p> <ul style="list-style-type: none"> • Any iAPX 286 based or iSBC® 286 based system including Intel's 286/300 family and iDIS systems • 196 KB memory • Two floppy disks or one hard disk • One 8" double-density or 5.25" double-density floppy disk drive for distribution of media 	<p>Required Software:</p> <ul style="list-style-type: none"> • Intel's XENIX 286 Operating System • Purchase of any XENIX Language requires signing of Intel's OEM License Agreement (OLA) 	
--	---	--

Ordering Information

Language	Order Code	Product Contents	Warranty
COBOL	XNX 2867	One 8" diskette and one 5.25" diskette Level II COBOL Language Reference Manual—122158 Level II COBOL Operating Guide—122159 Forms II Utility Manual—122160 Level II COBOL Pocket Guide—122161	90 days: Software Updates, Subscription Service
	XNX 2868	Incorporation Fee for passing through the COBOL Runtime System	
FORTRAN	XNX 2862	One 8" diskette and two 5.25" diskettes Fortran Reference Manual Fortran User's Guide	90 days: Software Updates, Subscription Service
BASIC	XNX 2865	One 8" diskette and one 5.25" diskette BASIC Reference Manual BASIC User's Guide	90 days: Software Updates, Subscription Service

FORMS-2 is a trademark of Micro Focus



2920 SOFTWARE SUPPORT PACKAGE

- Complete software design and development support for the 2920
- Extends Inteltec® Microcomputer Development System to support 2920 software development

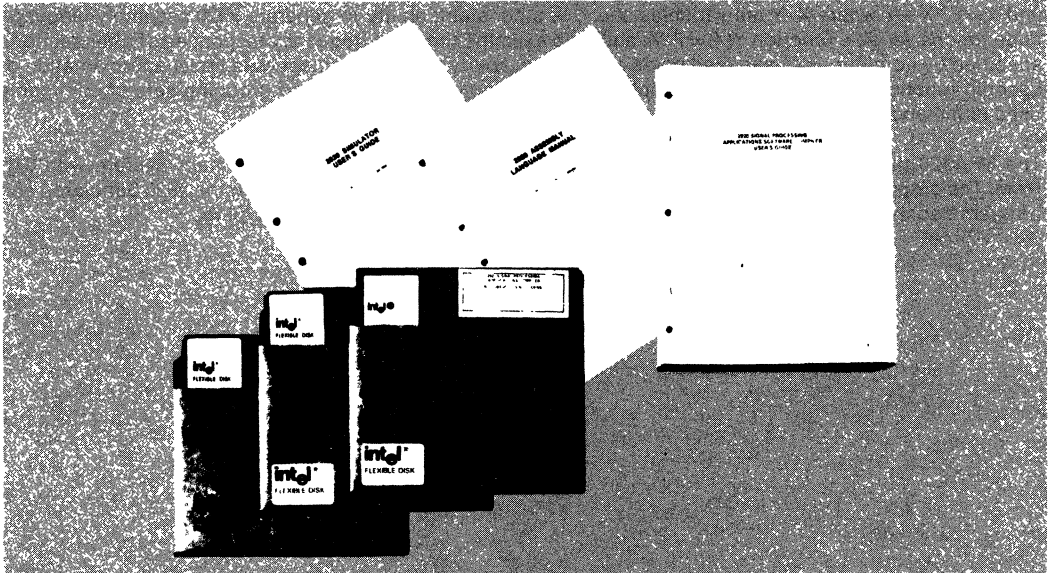
The 2920 Software Support Package furnishes a 2920 Signal Processing Applications Software/Compiler, 2920 Assembler, and 2920 Software Simulator. These three software design and development tools run on the Inteltec® Microcomputer Development System.

The 2920 Signal Processing Application Software/Compiler is an interactive tool for designing software to be executed on the 2920 Signal Processor. The compiler accepts English-like statements from the user and generates 2920 assembly language code.

The assembler translates symbolic 2920 assembly language programs into the machine operation code. The user can load the code into the simulator for 2920 simulation or to the Universal PROM Programmer for 2920 EPROM programming.

The simulator, operating entirely in software, allows the user to test and symbolically debug 2920 programs. The user can specify input signals, simulate program execution, set up breakpoints, display input and output, and display and alter the contents of the 2920 registers and memory locations. The simulator can also stop or trace the program and constructively give the user access to the key elements inside a 2920 for analyzing his program.

The compiler, assembler, and simulator enable the designer to develop and test an entire program without a complete prototype design. The 2920 designer works on the Inteltec® Microcomputer Development System rather than on a breadboard. The development system can program, store and recall programs or routines and aid in 2920 program design.



2920 Software Support Package

The following are trademarks of Intel Corporation and may be used only to identify Intel products: BXP, Inteltec, Multibus, i, iSBC, Multimodule, ICE, iSBX, PROMPT, iCS, Library Manager, Promware, Insite, MCS, RMX, Intel, Megachassis, UPI, Intelevison, Microamp, µScope and the combination of ICE, iCS, iSBC, iSBX, MCS, or RMX and a numerical suffix

2920 SIGNAL PROCESSING APPLICATIONS SOFTWARE/COMPILER

- **Compiler generates 2920 Assembly Language Code**
- **Extensive command set for designing electrical filters**
- **Graphics capability enhances analysis of filter response or piecewise linear function approximations**
- **Powerful MACRO capability for executing frequently used routines**
- **Interactive software support tool for 2920 Signal Processor**
- **Extends Intellec® Microcomputer Development System support of the 2920**
- **Contains MACRO library for several standard filters and signal processing functions**

The 2920 Signal Processing Applications Software/Compiler (SPAS20) is an interactive tool for designing software to execute on the 2920 Signal Processor.

The SPAS20 package can be visualized as being comprised of four inter-related sections: A compiler section, a filter design section, a curve fitting section, and a MACRO section.

Among the abilities of SPAS20 are: ability to generate 2920 assembly language code directly from specifications of signal processing building blocks such as filters and waveform generators; ability to generate 2920 assembly language code for several classes of algebraic equations such as $Y = C * X$, $Y = C * Y$, and $Y = C * X + Y$ where X , Y are variables and C is a constant; ability to generate 2920 assembly language code for one variable function $Y(X) = F(X)$; ability to examine time and frequency responses of filter sections specified by continuous or sampled poles and zeroes; ability to examine piecewise linear approximation of specific function; ability for users to implement more complex commands by grouping sets of commonly used commands into a MACRO.

The SPAS20 package runs under ISIS-II on any Intellec® Microcomputer Development System with 64K RAM. The output of SPAS20 can be assembled with the 2920 assembler, tested with the 2920 Simulator, and programmed into the 2920 chip with the Universal PROM Programmer for prototyping.

FUNCTIONAL DESCRIPTION

The 2920 Signal Processing Applications Software/Compiler gives the analog designer a "high level language" for his 2920 applications—it decreases the need to code 2920 assembly language. Furthermore, the compiler is interactive. This feature enables the designer to define a filter, or transfer function, graph their response, and change their parameters many times, without having to program and test in an actual 2920 implementation.

Once a filter is realized by moving poles and zeros in the continuous and sampled planes, the filter may be coded and written onto an ISIS file. Similarly, after a function $Y = F(X)$ has been defined, the code for a piecewise linear approximation can be stored onto an ISIS file. Several other file commands are available to store and retrieve command sequences for SPAS20 sessions.

SPAS20 Command Language

- DEFINE** This command defines a pole or zero by associating it with a number (i.e., POLE 3), and with real and imaginary coordinates in the continuous or sampled plane.
- This command also defines a symbol by associating a name with a numeric value, or a MACRO by providing a pointer to a specified command sequence.
- GRAPH/ OGRAPH** This command graphically displays the values of object(s) specified. For example, GRAPH GAIN and GRAPH PHASE are used to display filter response. The OGRAPH command will "overgraph" the new response over the old response, after any changes have been made. (You may also graph Group Delay, Step, and Impulse.)
- MOVE** Allows the definition of a pole or zero to be changed—its coordinates, its plane, or both.
- REMOVE** Deletes the definition of a pole, zero, symbol, or macro.
- HELP** Types an explanatory message on the console, pertaining to a command or its attributes.
- FIT** This command performs curve fitting, i.e. it approximates an arbitrary user supplied function with a piecewise linear function.

- DATA** This command allows for specification of a set of vertices (i.e. X – Y coordinate pairs) which determine a piecewise linear approximation of some defined function, filter response characteristics, etc.
- HOLD** Command to correct attenuation due to sample-and-hold distortion: if ON, it corrects absolute gain by $\sin(x)/x$ and phase by adding x , where $x=TS \cdot \text{FREQ} \cdot \pi$. It corrects group delay by subtracting $\pi \cdot TS$.
- EVALUATE** Gives the decimal numeric value of any expression.
- CODE** Creates 2920 assembly language code for given poles, and zeros, equations, and user defined functions.

The SPAS20 compiler also recognizes the following commands for file handling:

- PUT/ APPEND** Writes out objects (commands) to a specified file, either creating a new one or appending an existing one. This enables the user to store all or part of a SPAS20 session on a diskette to be brought back later with the INCLUDE command.
- DISPLAY** Copies the contents of a file to the console.
- INCLUDE** Executes a sequence of instructions from a diskette file as if they were typed in from the console.
- LIST** Creates a file containing all console interactions.
- In addition to naming macros for specific command sequences, compound and conditional commands may be formed using all of the above statements. These compound commands are:
- IF** Establishes conditional flow of control within a block of commands.
- REPEAT** Used for repetition of a block of commands; executes indefinitely or until a condition is met (using WHILE, UNTIL, and END statements).
- COUNT** Establishes the number of times a command sequence is to be executed, in a looping fashion.

SPAS20 MACRO Facility

A macro is a sequence of commands that is stored on a temporary diskette file. The command sequence is executed when the macro name is entered as a command. This saves repetitive entry of the sequence, and permits algorithms to be saved on diskette for future use. This SPAS20 facility allows you to do the following:

- Display the text of any macro.
- Define a macro, specifying its name and any parameters that are to be used by the block. This definition is followed by the contents of the macro (commands) and the EM statement to end its definition.
- Invoke a macro by entering its name and appropriate values for any parameters.
- List the names of all defined macros.
- Remove any or all macros.

Intel also supplies several MACRO library files containing the following commonly needed MACROS:

- Filter design MACROS
 - Butterworth filter
 - Chebyshev filter
 - Bilinear transform
 - Evaluate gain or phase of digital filter in parallel form
 - Time response simulation
- Function design MACROS
 - Code and error optimization
 - Calculate interstitial error
- MACROS for generation of 2920 code
 - Code for all-POLE filter
 - Input and A/D conversion
 - Multiplication
 - Division
 - Logarithm functions
 - Square-root functions
 - Sinewave oscillator

SAMPLE SPAS20 FILTER DESIGN SESSION

```

--:FI : SPAS20 . SFT
ISIS-II 2920 SIGNAL PROCESSING APPLICATIONS COMPILER. V2.0
*
*DEFINE POLE 1 = -707.707      ; CREATE A POLE IN CONTINUOUS S-PLANE
*
*P2      LIST ALL POLES AND ZEROS
POLE 1 = -707 00000.707 00000.CONTINUOUS
*
*FSCALE = 100.10000          ; ESTABLISHES FREQUENCY RANGE OF INTEREST
*
*YSCALE = -45.1              ; ESTABLISHES MAGNITUDE RESPONSE RANGE OF INTEREST
*
*GRAPH GAIN                  ; PLOT MAGNITUDE RESPONSE OF POLE PAIR

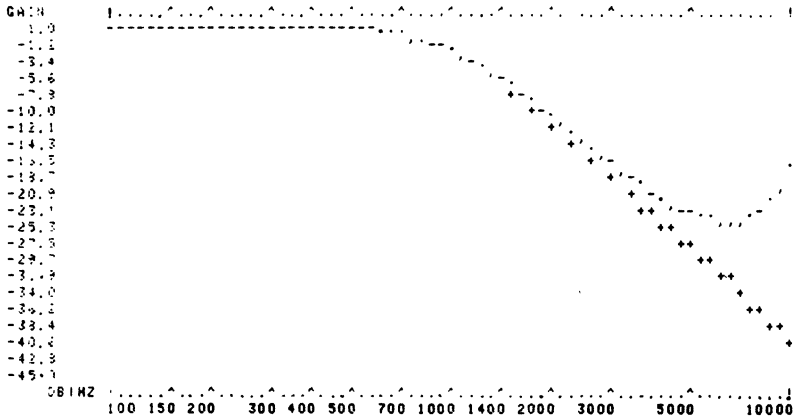
GAIN      1
          1.0
          -1.2
          -3.4
          -5.6
          -7.8
          -10.0
          -12.1
          -14.3
          -16.5
          -18.7
          -20.9
          -23.1
          -25.3
          -27.5
          -29.7
          -31.9
          -34.0
          -36.2
          -38.4
          -40.6
          -42.9
          -45.0
          DB1HZ
          100 150 200 300 400 500 700 1000 1400 2000 3000 5000 10000
**
** THE UNITS USED IN GRAPHING GAIN ARE SHOWN IN THE LOWER LEFT CORNER
** GAIN IN DECIBELS IS GRAPHED VERSES FREQUENCY IN HERTZ
**
** PREPARE TO MOVE TO THE DIGITAL DOMAIN.
** SAMPLE RATE MUST BE SPECIFIED
**
**TS = 1/13020      ; RATE FOR 192 INSTRUCTION PROGRAM AND 10MHZ CLOCK
TS = 7 6805004/10**5
  
```

SAMPLE SPAS20 FILTER DESIGN SESSION (Cont'd.)

```

*MOVE POLE TO Z          ; CONVERT FILTER TO DIGITAL VIA MATCHED-Z TRANSFORMATION
1 POLES/ZERPOES MOVED
*
*P                        ; LIST TRANSFORMED POLE
POLE 1 = 0 71092836.0 34118369.2
*
*. COMPARE RESPONSES OF THE ANALOG AND DIGITAL FILTERS BY GRAPHING THE
* NEW RESPONSE OVER THE OLD
*
*GRAPH GAIN

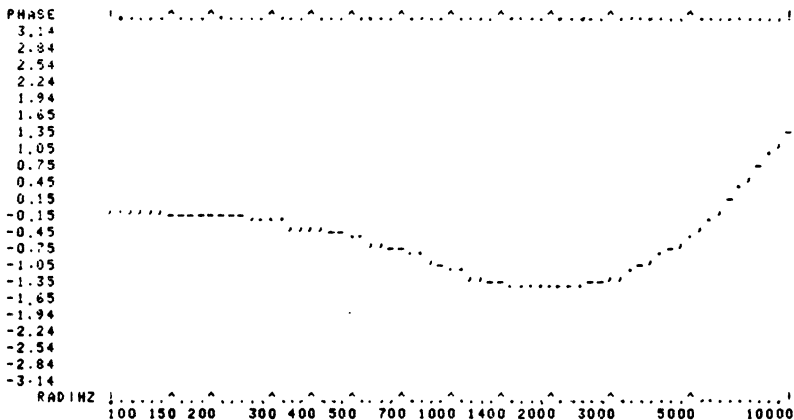
```



```

G*
*
*. PLUS SIGNS INDICATE OLD CURVE
*. NOTE THAT THE DIGITAL FILTER RESPONSE BEGINS TO INCREASE AGAIN
*. AT HALF THE SAMPLE RATE ( 6510 HZ )
*
*. THE PHASE CHARACTERISTICS OF THIS FILTER CAN BE EXAMINED
*
*YSCALE = -PI.PI          ; ESTABLISHES RANGE OF INTEREST
*
*GRAPH PHASE

```



```

P*
*
*PUT :F1:POLE PZ          ; SAVE THE POLE LOCATION IN A DISK FILE BACKUP
*
*CODE POLE 1 INST<11     ; GENERATE 2920 ASSEMBLY CODE FOR THIS FILTER
B:=1 33989*90 B2=-0 50541914

```

SAMPLE SPAS20 FILTER DESIGN SESSION (Cont'd.)

OPTIMIZED 2920 CODE IS NOW GENERATED TO SAVE SPACE, SOME OF THE SCREEN OUTPUT HAS BEEN DELETED NORMALLY ALL ATTEMPTS BY THE COMPILER TO GENERATE CODE ARE ECHOED ON THE SCREEN)

```

INST=10
POLE 1 = 0 71089458.0 34116779.2
BEST: PERROR = 3 3795874/10**5.1 58846567/10**5

; NOTE: MAKE SURE SIGNAL IS <0 74635571
LD> OUT2_P1,OUT1_P1,R00
; OUT2_P1=1 00000000*OUT1_P1
LD> OUT1_P1,OUT0_P1,R00
; OUT1_P1=1 00000000*OUT0_P1
SUB OUT0_P1,OUT1_P1,R05
; OUT0_P1=1 00000000*OUT0_P1-0 031250000*OUT1_P1
ADD OUT0_P1,OUT0_P1,R03
; OUT0_P1=1 12500000*OUT0_P1-0 035156250*OUT1_P1
ADD OUT0_P1,OUT1_P1,R02
; OUT0_P1=1 12500000*OUT0_P1+0 21484375*OUT1_P1
SUB OUT0_P1,OUT2_P1,R01
; OUT0_P1=1 12500000*OUT0_P1+0 21484375*OUT1_P1-0 50000000*OUT2_P1
SUB OUT0_P1,OUT2_P1,R08
; OUT0_P1=1 12500000*OUT0_P1+0 21484375*OUT1_P1-0 50390625*OUT2_P1
ADD OUT0_P1,OUT2_P1,R11
; OUT0_P1=1 12500000*OUT0_P1+0 21484375*OUT1_P1-0 50341796*OUT2_P1
SUB OUT0_P1,OUT2_P1,R09
; OUT0_P1=1 12500000*OUT0_P1+0 21484375*OUT1_P1-0 50537109*OUT2_P1
ADD OUT0_P1,INO_P1,R00
; OUT0_P1=1 12500000*OUT0_P1+0 21484375*OUT1_P1-0 50537109*OUT2_P1+1 00000000*INO_P1
*
* THE CODE COMMAND SPECIFIED THAT THE POLE PAIR BE CODED IN LESS THAN 11
* INSTRUCTIONS, SO 10 INSTRUCTIONS WERE GENERATED, WITH COMMENTS
* THE FINAL ERROR IN RADIUS AND ANGLE FOR THE POLE PAIR WAS OF THE
* ORDER OF 1/10**5 AS INDICATED ABOVE IN PERROR
* THIS OPTIMIZED 2920 ASSEMBLY CODE CAN NOW BE APPENDED TO A FILE
* WHICH MAY CONTAIN OTHER CODED FUNCTIONAL BLOCKS OF A 2920 PROGRAM
*
*EQUIT
    
```

SAMPLE SPAS20 CURVE FITTING SESSION

```

-; DEMONSTRATION OF THE SPAS20 CURVE-FITTING PACKAGE.
-
-SPAS20.SPT
ISIS-II 2920 SIGNAL PROCESSING APPLICATIONS SOFTWARE/COMPILER, V2.0
*LIST YCURFD,R29
*
* THE CURVE FITTING COMMANDS IN SPAS20 WILL GENERATE 2920 CODE TO CALCULATE
* SOME FUNCTION SUCH AS Y**3. Y**3 COULD BE COMPUTED ON THE 2920 CHIP
* WITH TWO MULTIPLIES USING ABOUT 18 INSTRUCTIONS AND THE DAR, HOWEVER IT
* WOULD TAKE UP THE DAR TOO LONG. THE CODE GENERATED BY THE CURVE FITTING
* COMMANDS DOES NOT USE THE DAR.
*
*CODE FIT YCURFD(X) = Y**3 ERPOK.05 ;ERROR BOUND OF .05
*
*CODE ; HERE IS THE CODE GENERATED.
LDA TEMP,X,R00
; TEMP=1.00000000*X
LDA YCURFD,Y,R01
; YCURFD=0.50000000*X
ADD YCURFD,X,R06
; YCURFD=0.51562500*X
ADD TEMP,X,R01
; TEMP=0.50000000*X+1.00000000*TEMP
ADD YCURFD,TEMP,R05
; YCURFD=1.00000000*YCURFD+0.031250000*TEMP
SUB YCURFD,TEMP,R02
; YCURFD=1.00000000*YCURFD-0.21875000*TEMP
ADD TEMP,X,R00
; TEMP=1.00000000*X+1.00000000*TEMP
ADD YCURFD,TEMP,R08
; YCURFD=1.00000000*YCURFD+0.0039062500*TEMP
SUB YCURFD,TEMP,R04
; YCURFD=1.00000000*YCURFD-0.0585937500*TEMP
LDA YCURFD,YCURFD,L02
; YCURFD=4.00000000*YCURFD-0.23437500*TEMP
*
*INST ; THE FUNCTION WAS CODED IN THIS MANY INSTRUCTIONS;
INST = 10.0000000
    
```

```

*ERROR ; THE CODE APPROXIMATES X**3 WITHIN THIS ERROR;
ERROR = 0.046875000
*
*DATA 0 THRU 1 ; EXAMINE THE PIECEWISE LINEAR FUNCTIONS VERTICES.
DATA 0.00000000 THRU 1.00000000 = 0.00000000 AT 0.00000000,&
0.065625012 AT 0.40000000,&
0.265625000 AT 0.66666669,&
0.953125000 AT 1.00000000
*
*GRAPH DATA(Y) ; THE DATA AREY APPROXIMATES THE FUNCTION AND CAN BE GRAPHED.
FUNCTION !.....!
0.95
0.91
0.46
0.82
0.77
0.73
0.68
0.64
0.59
0.54
0.50
0.45
0.41
0.36
0.32
0.27
0.23
0.19
0.14
0.09
0.05
0.00
!.....!
* 0.00 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.00
*GRAPH X**3 ; THE DIFFERENCE BETWEEN THE CODED AND THE ACTUAL APPEARS AS "+".
FUNCTION !.....!
1.00
0.95
0.90
0.86
0.81
0.76
0.71
0.67
0.62
0.57
0.52
0.48
0.43
0.38
0.33
0.29
0.24
0.19
0.14
0.10
0.05
0.00
!.....!
* 0.00 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.00
*GRA (X**3)-DATA(X) ; THE ERROR WILL BE GRAPHED.
FUNCTION !.....!
0.047
0.043
0.039
0.036
0.032
0.028
0.025
0.021
0.017
0.014
0.010
0.006
0.003
-0.001
-0.005
-0.008
-0.012
-0.016
-0.020
-0.023
-0.027
-0.031
!.....!
* 0.00 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.00
*EXIT ; THAT'S ALL FOLKS
-

```



2920 ASSEMBLER

2920 program development on Intel[®]
Microcomputer Development Systems

Produces Assembly Listing, Object Code
File, and Error Diagnostics

Translates symbolic assembly language
instructions into 2920 machine code

Output used for 2920 programming with
the Intel[®] PROM Programmer or the
2920 Simulator for program debug

The 2920 Assembler translates symbolic 2920 Assembly Language instructions into the appropriate machine operation codes. Through this facility, the programmer is able to symbolically program 2920 hardware operations. Compared to machine code, these symbolic references provide faster programming, easier debugging, and greater reliability.

The Assembler produces an object code file (executable machine code), a complete assembly listing, and error diagnostics. The object code output from the Assembler may be loaded directly into the Intel Universal PROM Programmer for programming the 2920 EPROM. The object code may also be loaded to the 2920 Simulator for 2920 system design and debug.

The 2920 Assembler runs under the ISIS-II Operating System on the Intel[®] Microcomputer Development Systems.

Sample 2920 Assembly Listing

ISIS-II 2920 ASSEMBLER X102

PAGE 1

ASSEMBLER INVOKED BY: AS2920 SAM ASM DEBUG

SAWTOOTH WAVE GENERATOR

LINE LOC OBJECT SOURCE STATEMENT

```

1          ;#TITLE('SAWTOOTH WAVE GENERATOR')
2          ;
3          ;
4 0 0000EF      INO          ; SAMPLE INPUT CHANNEL 0
5 1 0000EF      INO
6 2 0000EF      INO
7 3 008AEB      SUB Y,KP1,INO  ; SIMULTANEOUSLY CALCULATE SAWTOOTH
8 4 008A0A      SUB Y,KP1,R1,INO ; BY SUBTRACTING 3/16 FROM Y
9 5 0044EF      LDA DAR,Y,INO  ; ALSO CHECK SIGN BIT OF Y
10 6 7ABAEF     ADD Y,KP7,CNDS  ; IF Y NEGATIVE START NEXT TOOTH
11 7 6000EF     CVTS          ; CONVERT SAMPLED INPUT TO DIGITAL (SIGN BIT)
12 8 7082EF     LDA Y,KP0,CNDS  ; SUPPRESS SAWTOOTH IF INPUT WAS < 0
13 9 4044EF     LDA DAR,Y      ; PREPARE TO OUTPUT SAWTOOTH
14 10 4000EF    NOP           ; ANALOG LEVEL MUST SETTLE
15 11 4000EF    NOP
16 12 4000EF    NOP
17 13 8000EF    OUTO          ; OUTPUT SAWTOOTH
18 14 8000EF    OUTO
19 15 8000EF    OUTO
20 16 5000EF    EOP           ; PROGRAM WILL END IN THREE MORE INSTRUCTIONS
21 17 8000EF    OUTO
22 18 8000EF    OUTO
23 19 8000EF    OUTO
24
25          END

```

SYMBOL:

VALUE:

Y 0

```

ASSEMBLY COMPLETE
ERRORS = 0
WARNINGS = 0
RAMSIZE = 1
ROMSIZE = 20

```


2920 SIMULATOR

Speeds test and debug of 2920 programs

Output and internal data can be saved on disk for further analysis.

Simulates 2920 internal operation

Provides ability to set breakpoints and to collect trace information

Operates on Intellec® Microcomputer Development Systems

Easy-to-learn commands

Allows users to specify 2920 input signals, and display or alter ROM, RAM, and system variables

The 2920 Simulator is a software facility that provides testing and symbolic debugging of 2920 programs in an Intellec Microcomputer Development Systems environment. The 2920 designers have the capability to specify the 2920 input signals, to set breakpoints, to collect and display 2920 input, output, system variables, and ROM and RAM data values during simulation. The 2920 Simulator accepts the hex format object files produced by the 2920 assembler. Output values and internal trace data may be saved on ISIS-II disk files for further analysis.

Functional Description

2920 Input Signal Specification

The four analog signal inputs to the 2920 processor can be specified as algebraic combinations of basic functions of time. The basic functions are SIN, COS, EXP, LOG, SQR, SAW, SQW, ABS

2920 Simulation

The simulation of 2920 machine instructions is performed in software. All 2920 internal registers, memory, input values, output values, and other system variables can be examined and modified. The internal processing of the 2920 is simulated. Time constants for the sample and hold capacitors are assumed to be zero. Calculation of input signals is performed in single precision floating point. The speed of simulation varies with the complexity of the input signal, breakpoint setting, and trace condition. Exclusive of I/O time requirements, 2920 instructions will be simulated at a rate of approximately several hundred instructions per second.

Breakpoint Capabilities

After each instruction is simulated, the breakpoint is evaluated to determine whether to stop or continue simulation. Conditional breakpoints are also provided for debugging purposes. Simulation can be manually stopped at any time by pressing the ESC key on the Intellec console.

Trace Capabilities

Based on the qualifier's condition, trace data records can be collected during simulation. The trace data

records are stored in Intellec resident memory and are optionally written to the console for display or to a disk file for record.

Symbolic Debugging Capabilities

The 2920 Simulator allows the user to refer to program addresses symbolically. The user can load or save the symbols generated from the hex format object files or created during the debugging session. 2920 program memory in ROM can be disassembled, or filled with assembled instructions.

The 2920 Simulator is designed to provide users with powerful, easy-to-use commands. The user interfaces to the Simulator by entering commands to the Intellec console. The commands consist of one command line, terminated by one of the two line terminators — carriage return or line feed.

The 2920 Simulator offers two types of commands:

Simulation and Control Commands

Command	Operation
Simulate	Starts simulation of the input signals and the 2920 program in simulated ROM memory. Initial setting is "FOREVER."
Trace	Controls the trace selection. Initial setting is "TIME."
Qualifier	Sets qualifier condition during trace. Initial setting is "ALWAYS."
Breakpoint	Sets breakpoint condition during simulation. Initial setting is "NEVER."



Interrogation and Utility Commands

Table with 2 columns: Command and Operation. Rows include Display, Change, Base, Suffix, Load, Save, Define, Console, List, Exit, Evaluate, Remove, Help, Graphics On/Off, X Size.

Software Simulator Keyword References

Table with 2 columns: Keyword and Description. Rows include TIME, TQUAL, COUNT, BUFFERSIZE, TINST, SIZE, TPROG, VREF.

The above keyword references are designed to aid 2920 program debugging.

ISIS Compatibilities

The 2920 software simulator runs under the ISIS "submit" facility. The 2920 software simulator uses the ISIS-II line editing capabilities to correct errors in an input line on the Intellec console.

Keyword References

The 2920 Simulator provides users with keyword references to gain access to all of the numeric valued system variables including simulated 2920's memory, register, status flags and input/output. These keyword references can function as the evaluation command, display command, and change command.

2920 Processor Keyword References

Table with 2 columns: Keyword and Description. Rows include IN0-3, OUT0-7, IN, DAR, PC, CY, OVf, OVE.

Sample 2920 Simulation Session

```
-S42920.SFT
ISIS-II 2920 SIMULATOR, V1.1
*
*; THIS IS THE SIMULATION OF THE 'SAWTOOTH GENERATOR'
*
*LIST SRG.LOG ; LISTS THE SIMULATION SESSION TO AN ISIS FILE
*LOAD SRG.HEX ; LOAD THE OBJECT CODE INTO THE 2920 SIMULATOR
*ROM 0 TO 5 ; DISPLAY SRG PROGRAM
ROM 000 = LDA .K,KP5,R00,NOP
ROM 001 = ADD .K,KP1,R05,NOP
ROM 002 = LDA .K,.K,R02,NOP
ROM 003 = SUB .OSC,.K,R00,NOP
ROM 004 = LDA DAR,.OSC,R00,NOP
ROM 005 = ADD .OSC,KP4,L01,CNDS
*TPROC=1/1000 ; SET THE SAMPLE RATE
*TRA=PC,RAM .K ; SET THE ITEMS TO BE TRACED
*BASE=B ; DISPLAY THE RESULTS IN BINARY
*SIMULATE FROM 0 TILL COUNT=3 ; SIMULATE THREE INSTRUCTIONS TO VERIFY CONSTANT

PC RAM 0
SIMULATION BEGUN
1.00000000000000000000000000000000 0.10100000000000000000000000000000
2.00000000E+0 0.10100001000000000000000000000000
3.00000000E+0 0.00101000010000000000000000000000
SIMULATION TERMINATED
*QUALIFIER=PC=0 ; TRACE EVERY PROGRAM PASS
*TRACE=T,DAR,RAM .OSC ; SET THE ITEMS TO BE TRACED
*RAM .OSC=ONE ; INITIALIZE THE RAM LOCATION
**BREAKPOINT=T>.00132 ; SIMULATE FOR TWO CYCLES
*BASE=D ; SET THE BASE TO DECIMAL
*SIMULATE FROM 0 ; BEGIN SIMULATION
T DAR RAM 1
```



```

SIMULATION BEGUN
0.00010000      0.83984375      0.84277334
0.00020000      0.68359375      0.68554683
0.00030000      0.52734375      0.52832026
0.00040000      0.36718750      0.37109370
0.00050000      0.21093750      0.21386714
0.00060000      0.05468750      0.05664056
0.00070000     -0.10156250      0.89941396
0.00080000      0.73828125      0.74218745
0.00090000      0.58203125      0.58496089
0.00100000      0.42578125      0.42773833
0.00110000      0.26953125      0.27050776
0.00120000      0.10937500      0.11328119
0.00130000     -0.04687500      0.95605459

```

```

SIMULATION TERMINATED
*GRAPH ON          ; SWITCHES THE DISPLAY MODE TO GRAPHICS
*TRACE=T,0,DAR,RAI .OSC,-1,-1,1,1 ; SETS ITEMS TO BE TRACED
*RAM .OSC=ONF      ; INITIALIZE THE RAM LOCATION
*SIMULATE FROM 0

```

```

      T      -1      0      1      DAR      1      RAM 1      -1
SIMULATION BEGUN
->*          1          *          *
0 *          1          *          *
. *          1          *          *
0 *          1          *          *
0 *          1          *          *
0 *          1          *          *
1 *          2          1          *          3          *
0 *          1          *          *
0 *          1          *          *
0 *          1          *          *
0 *          1          *          *
*          2 1          *          3 *
SIMULATION TRMTERATED
*EXIT

```

SPECIFICATIONS

Operating Equipment

Required Hardware

Intellec[®] Microcomputer Development System
RUNNING ISIS

Required Software

ISIS-II Diskette Operating System

Optional Hardware

Line Printer
Universal PROM Programmer

Optional Software

FORTRAN-80 (Product Code MDS-301)

Documentation Package

2920 Assembly User's Guide (9800987)
2920 Simulator User's Guide (9800988)
2920 Signal Processing Application Compiler
User's Guide (121529)

Shipping Media

Flexible Diskettes

ORDERING INFORMATION

Product Code Description

MCI-20-SPS	2920 Software Support Package Includes 2920 Signal Processing Application Software/Compiler and 2920 Assembler/Simulator Software
------------	--



MCS[®]-48 DISKETTE-BASED SOFTWARE SUPPORT PACKAGE

- Extends Intellec microcomputer development system to support MCS-48 development
- MCS-48 assembler provides conditional assembly and macro capability
- Takes advantage of powerful ISIS-II file handling and storage capabilities
- Provides assembler output in standard Intel hex format

The MCS-48 assembler translates symbolic 8048 assembly language instructions into the appropriate machine operation codes, and provides both conditional and macroassembler programming. Output may be loaded either to an ICE-49 module for debugging or into the iUP Universal PROM Programmer for 8748 PROM programming. The MCS-48 assembler operates under the ISIS-II operating system on Intel Development systems.



FUNCTIONAL DESCRIPTION

The MCS-48 assembler translates symbolic 8048 assembly language instructions into the appropriate machine operation codes. The ability to refer to program addresses with symbolic names eliminates the errors of hand translation and makes it easier to modify programs when adding or deleting instructions. Conditional assembly permits the programmer to specify which portions of the master source document should be included or deleted in variations on a basic system design, such as the code required to handle optional external devices. Macro capability allows the programmer use of a single label to define a routine. The MCS-48 assembler will assemble the code required by the reserved routine whenever the macro label is inserted in the text. Output from the assembler is in standard Intel hex format. It may be either loaded directly to an in-circuit emulator (ICE-49) module for integrated hardware/software debugging, or loaded into the iUP Universal PROM Programmer for 8748 PROM programming. A sample assembly listing is shown in Table 1.

The MCS 48 assembler supports the 8048, 8049, 8050, 8020, 8021, 8022, 8041 and 8042. The MCS 48 assembler can also support CMOS versions of the 8048 family.

Table 1. Sample MCS-48 Diskette-Based

ISIS II 8048 MACROASSEMBLER V10		PAGE 1	
LOC	OBJ	SEQ	SOURCE STATEMENT
		1	DECIMAL ADDITION ROUTINE ADD BCD NUMBER
		2	AT LOCATION BETA TO BCD NUMBER AT ALPHA WITH
		3	RESULT IN ALPHA LENGTH OF NUMBER IS COUNT DIGIT
		4	PAIRS (ASSUME BOTH BETA AND ALPHA ARE SAME LENGTH
		5	AND HAVE EVEN NUMBER OF DIGITS OR MSD IS 0 IF
		6	ODD)
		7	INIT MACRO AUGND,ADDND CNT
		8	MOV R0 #AUGND
	L1	9	MOV R1 #ADDND
		10	MOV R2 #CNT
		11	ENDM
		12	
0001E		13	ALPHA EQU 30
0028		14	BETA EQU 40
0032		15	COUNT EQU 5
0100		16	ORG 100H
		17	INIT ALPHA BETA COUNT
0100	B81E	18+	MOV R0 #ALPHA
0102	B82B	19+ L1	MOV R1 #BETA
0104	B832	20+	MOV R2 #COUNT
0106	97	21	CLR C
0107	F0	22	LP MOV A @R0
0108	71	23	ADDC A @R1
0109	57	24	DA A
010A	A1	25	@R0 A
010B	1B	26	INC R0
010C	19	27	INC R1
010D	EA07	28	DJNZ R2 LP
			END
USER SYMBOLS			
ALPHA	0001E	BETA	0028
L1	0102	COUNT	0005
		LP	0107
ASSEMBLY COMPLETE NO ERRORS			
ISIS II ASSEMBLER SYMBOL CROSS REFERENCE V10		PAGE 1	
SYMBOL CROSS REFERENCE			
ALPHA	13H	17	
BETA	14H	17	
COUNT	15H	17	
INIT	7H	17	
L1	19H		
LP	22H	28	

SPECIFICATIONS

Operating Environment

- (All) Intel Microcomputer Development Systems (Series II, Series III/Series IV)
- Intel Personal Development System

Documentation Package

- Titles of: User Guides
- Operating Instructions
- Reference Manuals

Ordering Information

Part Number	Description
MDS-D48*	MCS-48 Disk Based Assembler
	Requires Software License

SUPPORT:

Hotline Telephone Support, Software Performance Reports (SPR), Software Updates, Technical Reports, Monthly Newsletters are available.

*MDS is an ordering code only and is not used as a product name or trademark. MDS is a registered trademark of Mohawk Data Sciences Corporation.



8051 SOFTWARE PACKAGES

PL/M51 Software Package Contains the following:

- **PL/M51 Compiler** which is designed to support all phases of software implementation
- **RL51 Linker and Relocator** which enables programmers to develop software in a modular fashion
- **LIB51 Librarian** which lets programmers create and maintain libraries of software object modules

8051 Software Development Package Contains the following:

- **8051 Macro Assembler** which gives symbolic access to 8051 hardware features
- **RL51 Linker and Relocator program** which links modules generated by the assembler
- **CONV51** which enables software written for the MCS[®]-48 family to be up graded to run on the 8051
- **LIB51 Librarian** which lets programmers create and maintain libraries of software object modules

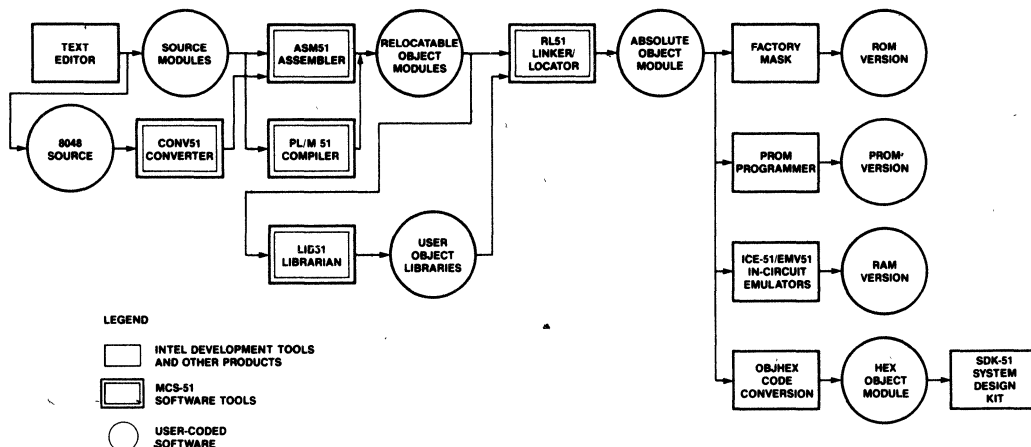


Figure 1. MCS[®]-51 Program Development Process

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. Information Contained Herein Supersedes Previously Published Specifications On These Devices From Intel.

PL/M 51 SOFTWARE PACKAGE

- High-level programming language for the Intel MCS[®]-51 single-chip microcomputer family
- Compatible with PL/M 80 assuring MCS[®]-80/85 design portability
- Enhanced to support boolean processing
- Tailored to provide an optimum balance among on-chip RAM usage, code size and code execution time
- Allows programmer to have complete control of microcomputer resources
- Produces relocatable object code which is linkable to object modules generated by all other 8051 translators
- Extends high-level language programming advantages to microcontroller software development
- Improved reliability, lower maintenance costs, increased programmer productivity and software portability
- Includes the linking and relocating utility and the library manager
- Supports all members of the Intel MCS[®]-51 architecture

PL/M 51 is a structured, high-level programming language for the Intel MCS-51 family of microcomputers. The PL/M 51 language and compiler have been designed to support the unique software development requirements of the single-chip microcomputer environment. The PL/M language has been enhanced to support Boolean processing and efficient access to the microcomputer functions. New compiler controls allow the programmer complete control over what microcomputer resources are used by PL/M programs.

PL/M 51 is largely compatible with PL/M 80 and PL/M 86. A significant proportion of existing PL/M software can be ported to the MCS-51 with modifications to support the MCS-51 architecture. Existing PL/M programmers can start programming for the MCS-51 with a small relearning effort.

PL/M 51 is the high-level alternative to assembly language programming for the MCS-51. When code size and code execution speed are not critical factors, PL/M 51 is the cost-effective approach to developing reliable, maintainable software.

The PL/M 51 compiler has been designed to support efficiently all phases of software implementation with features like a syntax checker, multiple levels of optimization, cross-reference generation and debug record generation.

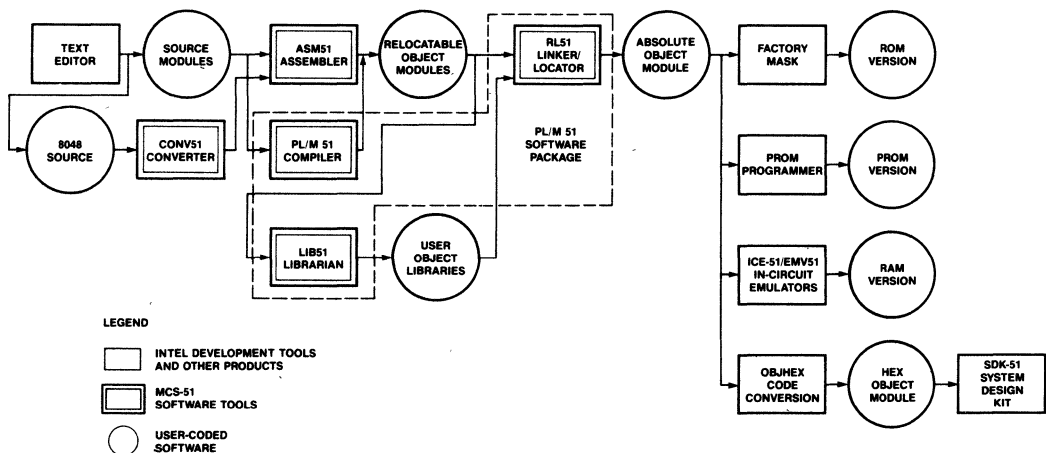


Figure 2. PL/M51 Software Package

PL/M 51 Compiler

FEATURES

Major features of the Intel PL/M 51 compiler and programming language include:

Structured Programming

PL/M source code is developed in a series of modules, procedures, and blocks. Encouraging program modularity in this manner makes programs more readable, and easier to maintain and debug. The language becomes more flexible, by clearly defining the scope of user variables (local to a private procedure, for example).

Language Compatibility

PL/M 51 object modules are compatible with object modules generated by all other MCS-51 translators. This means that PL/M programs may be linked to programs written in any other MCS-51 language.

Object modules are compatible with In-Circuit Emulators and Emulation Vehicles for MCS-51 processors; the DEBUG compiler control provides these tools with symbolic debugging capabilities.

Supports Three Data Types

PL/M makes use of three data types for various applications. These data types range from one to sixteen bits and facilitate various arithmetic, logic, and address functions:

- Bit: a binary digit
- Byte: 8-bit unsigned number or,
- Word: 16-bit unsigned number.

Another powerful facility allows the use of BASED variables that map more than one variable to the same memory location. This is especially useful for passing parameters, relative and absolute addressing, and memory allocation.

Two Data Structuring Facilities

PL/M 51 supports two data structuring facilities. These add flexibility to the referencing of data stored in large groups.

- Array: Indexed list of same type data elements
- Structure: Named collection of same or different type data elements
- Combinations of Both: Arrays of structures or structures of arrays.

Interrupt Handling

A procedure may be defined with the INTERRUPT attribute. The compiler will generate code to save and restore the processor status, for execution of the user-defined interrupt handler routines.

Compiler Controls

The PL/M 51 compiler offers controls that facilitate such features as:

- Including additional PL/M 51 source files from disk
- Cross-reference
- Corresponding assembly language code in the listing file

Program Addressing Control

The PL/M 51 compiler takes full advantage of program addressing with the ROM (SMALL/MEDIUM/LARGE) control. Programs with less than 2 KB code space can use the SMALL or MEDIUM option to generate optimum addressing instructions. Larger programs can address over the full 64 KB range.

Code Optimization

The PL/M 51 compiler offers four levels of optimization for significantly reducing overall program size.

- Combination or "folding" of constant expressions; "Strength reductions" (a shift left rather than multiply by 2)
- Machine code optimizations; elimination of superfluous branches
- Automatic overlaying of on-chip RAM variables
- Register history: an off-chip variable will not be reloaded if its value is available in a register.

Error Checking

The PL/M 51 compiler has a very powerful feature to speed up compilations. If a syntax or program error is detected, the compiler will skip the code generation and optimization passes. This usually yields a 2X performance increase for compilation of programs with errors.

A fully detailed set of programming and compilation error messages is provided by the compiler and user's guide.

BENEFITS

PL/M 51 is designed to be an efficient, cost-effective solution to the special requirements of MCS-51 Microsystem Software Development, as illustrated by the following benefits of PL/M use:

Low Learning Effort

PL/M 51 is easy to learn and to use, even for the novice programmer.

Earlier Project Completion

Critical projects are completed much earlier than otherwise possible because PL/M 51, a structured high-level language, increases programmer productivity.

Lower Development Cost

Increases in programmer productivity translate immediately into lower software development costs because less programming resources are required for a given programmed function.

Increased Reliability

PL/M 51 is designed to aid in the development of reliable software (PL/M programs are simple statements of the program algorithm). This substantially reduces the risk of costly correction of errors in systems that have already reached full production status, as the more simply stated the program is, the more likely it is to perform its intended function.

Easier Enhancements and Maintenance

Programs written in PL/M tend to be self-documenting, thus easier to read and understand. This means it is easier to enhance and maintain PL/M programs as the system capabilities expand and future products are developed.

RL51 Linker and Relocator

- **Links modules generated by the assembler and the PL/M compiler**
- **Locates the linked object to absolute memory locations**
- **Enables modular programming of software-efficient program development**
- **Modular programs are easy to understand, maintainable and reliable**

The MCS-51 linker and relocater (RL51) is a utility which enables MCS-51 programmers to develop software in a modular fashion. The utility resolves all references between modules and assigns absolute memory locations to all the relocatable segments, combining relocatable partial segments with the same name.

With this utility, software can be developed more quickly because small functional modules are easier to understand, design and test than large programs.

The total number of allowed symbols in user-developed software is very large because the assembler number of symbols' limit applies only per module, not to the entire program. Therefore programs can be more readable and better documented.

Modules can be saved and used on different programs. Therefore the software investment of the customer is maintained.

RL51 produces two files. The absolute object module file can be directly executed by the MCS-51 family. The listing file shows the results of the link/locate process.

LIB51 Librarian

The LIB51 utility enables MCS-51 programmers to create and maintain libraries of software object modules. With this utility, the customer can develop standard software modules and place them in libraries, which programs can access through a standard interface. When using object libraries, the linker will

call only object modules that are required to satisfy external references.

Consequently, the librarian enables the customer to port and reuse software on different projects—thereby maintaining the customer's software investment.

SPECIFICATIONS

Operating Environment

All Intel Microcomputer Development Systems or Intel Personal Development Systems

Documentation Package

PL/M 51 User's Guide
MCS-51 Utilities User's Guide

ORDERING INFORMATION

Part Number

iMDX 352
Requires Software License

Description

PL/M 51 Software
Package

SUPPORT:

Hotline Telephone Support, Software Performance Report (SPR), Software Updates, Technical Reports, and monthly Technical Newsletters are available.

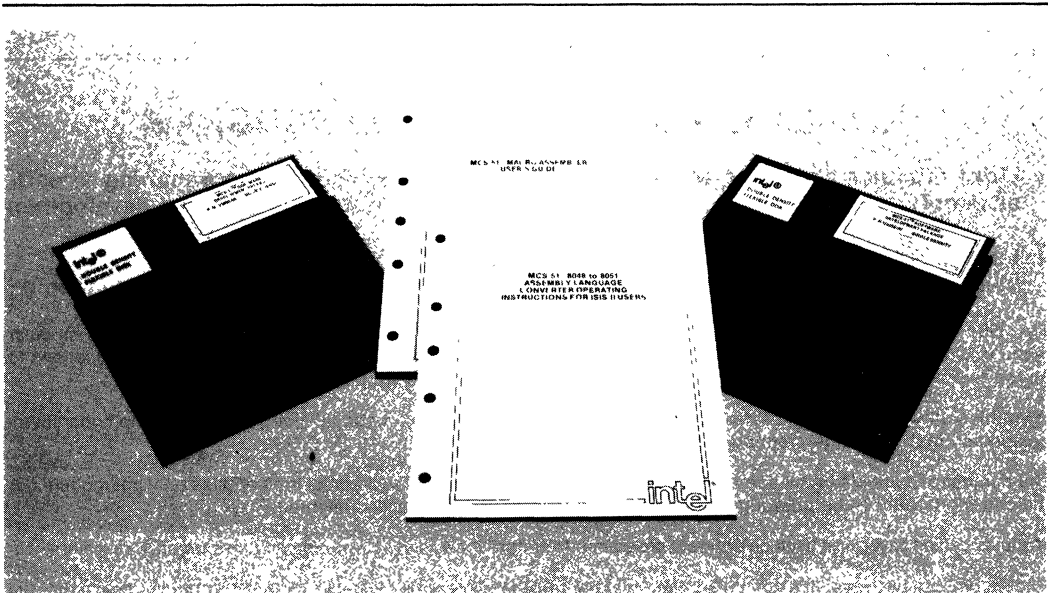
8051 SOFTWARE DEVELOPMENT PACKAGE

- Symbolic relocatable assembly language programming for 8051 microcontrollers
- Extends Intellec® Microcomputer Development System to support 8051 program development
- Produces Relocatable Object Code which is linkable to other 8051 Object Modules
- Encourage modular program design for maintainability and reliability
- Macro Assembler features conditional assembly and macro capabilities
- CONV51 Converter for translation of 8048 assembly language source code to 8051 assembly language source code
- Provides upward compatibility from the MCS-48™ family of single-chip microcontrollers

The 8051 software development package provides development system support for the powerful 8051 family of single chip microcomputers. The package contains a symbolic macro assembler and MCS-48 source code converter.

The assembler produces relocatable object modules from 8051 macro assembly language instructions. The object code modules can be linked and located to absolute memory locations. This absolute object code may be used to program the 8751 EPROM version of the chip. The assembler output may also be debugged using the ICE-51™ in-circuit emulator.

The converter translates 8048 assembly language instructions into 8051 source instructions to provide software compatibility between the two families of microcontrollers.



8051 MACRO ASSEMBLER

- Supports 8051 family program development on Intellec® Microcomputer Development Systems
- Gives symbolic access to powerful 8051 hardware features
- Produces object file, listing file and error diagnostics
- Object files are linkable and locatable
- Provides software support for many addressing and data allocation capabilities
- Symbolic Assembler supports symbol table, cross-reference, macro capabilities, and conditional assembly

The 8051 Macro Assembler (ASM51) translates symbolic 8051 macro assembly language modules into linkable and locatable object code modules. Assembly language mnemonics are easier to program and are more readable than binary or hexadecimal machine instructions. By allowing the programmer to give symbolic names to memory locations rather than absolute addresses, software design and debug are performed more quickly and reliably. Furthermore, since modules are linkable and relocatable, the programmer can do his software in modular fashion. This makes programs easy to understand, maintainable and reliable.

The assembler supports macro definitions and calls. This is a convenient way to program a frequently used code sequence only once. The assembler also provides conditional assembly capabilities.

Cross referencing is provided in the symbol table listing, showing the user the lines in which each symbol was defined and referenced.

ASM51 provides symbolic access to the many useful addressing features of the 8051 architecture. These features include referencing for bit and byte locations, and for providing 4-bit operations for BCD arithmetic. The assembler also provides symbolic access to hardware registers, I/O ports, control bits, and RAM addresses. ASM51 can support all members of the 8051 family.

Math routines are enhanced by the MULTiply and DIVide instructions.

If an 8051 program contains errors, the assembler provides a comprehensive set of error diagnostics, which are included in the assembly listing or on another file. Program testing may be performed by using the iUP Universal Programmer and iUP F87/51 personality module to program the 8751 EPROM version of the chip.

ICE51 and EMV51 are available for program debugging.

RL51 LINKER AND RELOCATOR PROGRAM

- Links modules generated by the assembler
- Locates the linked object to absolute memory locations
- Enables modular programming of software for efficient program development
- Modular programs are easy to understand, maintainable and reliable

The 8051 linker and relocator (RL51) is a utility which enables 8051 programmers to develop software in a modular fashion. The linker resolves all references between modules and the relocator assigns absolute memory locations to all the relocatable segments, combining relocatable partial segments with the same name.

With this utility, software can be developed more quickly because small functional modules are easier to understand, design and test than large programs.

The number of symbols in the software is very large because the assembler symbol limit applies only per module not the entire program. Therefore programs can be more readable and better documented.

Modules can be saved and used on different programs. Therefore the software investment of the customer is maintained.

RL51 produces two files. The absolute object module file can be directly executed by the 8051 family. The listing file shows the results of the link/locate process.

CONV51

8048 TO 8051 ASSEMBLY LANGUAGE CONVERTER UTILITY PROGRAM

- Enables software written for the MCS-48™ family to be upgraded to run on the 8051
- Maps each 8048 instruction to a corresponding 8051 instruction
- Preserves comments; translates 8048 macro definitions and calls
- Provides diagnostic information and warning messages embedded in the output listing

The 8048 to 8051 Assembly Language Converter is a utility to help users of the MCS-48 family of microcomputers upgrade their designs with the high performance 8051 architecture. By converting 8048 source code to 8051 source code, the software investment developed for the 8048 is maintained when the system is upgraded.

The goal of the converter (CONV51) is to attain functional equivalence with the 8048 code by mapping each 8048 instruction to a corresponding 8051 instruction. In some cases a different instruction is produced because of the enhanced instruction set (e.g., bit CLR instead of ANL).

Although CONV51 tries to attain functional equivalence with each instruction, certain 8048 code sequences cannot be automatically converted. For example, a delay routine which depends on 8048 execution speed would require manual adjustment. A few instructions, in fact, have no 8051 equivalent (such as those involving P4-P7). Finally, there are a few areas of possible intervention such as PSW manipulation and interrupt processing, which at least require the user to confirm proper translation. The converter always warns the user when it cannot guarantee complete conversion.

CONV51 produces two files. The output file contains the ASM51 source program produced from the 8048 instructions. The listing file produces correlated listings of the input and output files, with warning messages in the output file to point out areas that may require users' intervention in the conversion.

LIB51 LIBRARIAN

The LIB51 utility enables MCS-51 programmers to create and maintain libraries of software object modules. With this utility, the customer can develop standard software modules and place them in libraries, which programs can access through a standard interface. When using object libraries, the linker will call only object modules that are required to satisfy external references.

Consequently, the librarian enables the customer to port and reuse software on different projects—thereby maintaining the customer's software investment.

SPECIFICATIONS**OPERATING ENVIRONMENT**

All Intel Microcomputer Development Systems or Intel Personal Development System

Documentation Package:

MCS-51 Macro Assembler User's Guide
MCS-51 Utilities User's Guide for 8080/8085 Based Development System
MCS-51 8048-to-8051 Assembly Language Converter
Operating Instructions for ISIS-II Users

ORDERING INFORMATION

Part Number	Description
MCI-51-ASM	8051 Software Development Package

*Requires Software License

SUPPORT:

Hotline Telephone Support, Software Performance Reporting (SPR), Software Updates, Technical Reports, Monthly Newsletter available.



iRMX™ 51 REAL-TIME MULTITASKING EXECUTIVE

- Software tool for family of 8051 microcontroller based applications
- Real-time, multitasking executive
- Supports remote task communication
- Small — 2.2K Bytes
- Reliable
- Simple user interface
- Compatible with BITBUS™/Distributed Control Modules (iDCM) product line: iSBX™ 344 & iRCB 44/10 boards

The iRMX™ 51 Executive is a compact, easy to use, software tool for development and implementation of applications built on the high performance 8-bit family of 8051 microcontrollers. A few members of this expansive family are the 8051, 8044, and 8052 microcontrollers. Like the 8051 family, the iRMX 51 Executive incorporates many features that make it exceptionally well suited for real-time control applications requiring manipulation and scheduling of more than one job, and fast response to external stimuli.

The 8051 microcontroller family is the family of choice for applications such as: data acquisition and monitoring, process control, robotics, and machine control. Using the iRMX 51 Executive for a foundation can significantly reduce applications development time. Also, the iRMX 51 Executive fully supports Intel's BITBUS™ microcontroller interconnect expressly designed for reliable high performance real-time control.

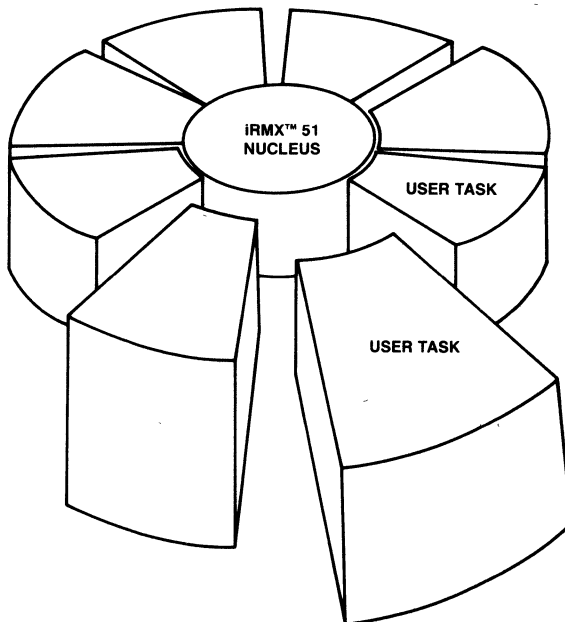


Figure 1. Structure Diagram

ARCHITECTURE

Real-time and Multitasking

Real-time control applications must be responsive to the external environment and typically involve the execution of more than one function (task or set of tasks) in response to different external stimuli. Control of an industrial drying process is an example. This process could require monitoring of multiple temperatures and humidity; control of fans, heaters, and motors that must respond accordingly to a variety of inputs. The iRMX 51 Executive fully supports applications requiring response to stimuli as they occur i.e. in real-time. This real-time response is supported for multiple tasks often needed to implement a control application.

Some of the facilities precisely tailored for development and implementation of real-time control application systems provided by the iRMX 51 Executive are: task management, interrupt handling, message passing, and when intergrated with communications support, message passing with different microcontrollers. Also, the iRMX 51 Executive is driven by events: interrupts, timers, and messages ensuring the application system always responds to the environment appropriately.

Task Management

A task is a program defined by the user to execute a particular control function or functions. Multiple programs or tasks may be required to implement a particular function such as 'control-

ling Heater 1'. The iRMX 51 Executive recognizes three different task states as one of the mechanisms to accomplish scheduling of up to eight tasks. Figure 2 illustrates the different task states and their relationship to one another.

The scheduling of tasks is priority based. The user can prioritize tasks to reflect their relative importance within the overall control scheme. For instance, if Heater 1 must go off line prior to Heater 2 then the task associated with Heater 1 shutdown could be assigned a higher priority ensuring the correct shutdown sequence. The RQ WAIT system call is also a scheduling tool. In this example the task implementing Heater 2 shutdown could include an instruction to wait for completion of the task that implements Heater 1 shutdown.

The iRMX 51 Executive allows for PREEMPTION of a task that is currently being executed. This means that if some external event occurs such as a catastrophic failure of Heater 1, a higher priority task associated with the interrupt, message, or timeout resulting from the failure will preempt the running task. Preemption ensures the emergency will be responded to immediately. This is crucial for real-time control application systems.

Interrupt Handling

The iRMX 51 executive supports sixteen interrupt sources as shown in Table 1. Four of these interrupt sources, excluding timer 0, can be as-

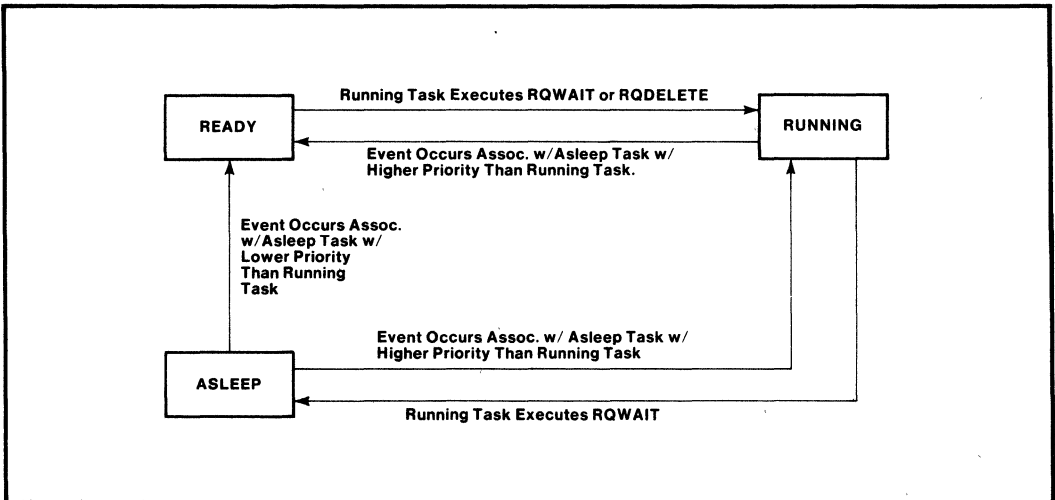


Figure 2. Task State Transition Diagram

signed to a task. When one of the interrupts occurs the task associated with it becomes a running task (if it were the highest priority task in a ready state). In this way, the iRMX 51 Executive responds to a number of internal and external stimuli including time intervals designated by the user.

Table 1. iRMX™ 51 Interrupt Sources

INTERRUPT SOURCE	INTERRUPT NUMBER
External Request 0	00H
Timer 0	01H
External Request 1	02H
Timer 1	03H
Internal Serial Port 1	04H
Reserved	05H
Reserved	06H
Reserved	07H
Reserved	08H
Reserved	09H
Reserved	0AH
Reserved	0BH
Reserved	0CH
Reserved	0DH
Reserved	0EH
Reserved	0FH

Message Passing

The iRMX 51 Executive allows tasks to interface with one another via a simple message passing facility. This message passing facility can be extended to different processors when communications support is integrated within a BITBUS/iDCM system, for example. This facility provides the user with the ability to link different functions or tasks. Linkage between tasks/functions is typically required to support development of complex control applications with multiple sensors (inputs variables) and drivers (output variables). For instance, the industrial drying process might require a dozen temperature inputs, six moisture readings, and control of: three fans, two conveyor motors, a dryer motor, and a pneumatic conveyor. The data gathered from both the temperature and humidity sensors could be processed. Two tasks might be required to gather the data and process it. One task could perform a part of the analysis, then include a pointer to the next task to complete

the next part of the analysis. The tasks could continue to move between one another.

REMOTE TASK COMMUNICATION

The iRMX 51 Executive system calls can support communication to tasks on remote controllers. This feature makes the iRMX 51 Executive ideal for applications using distributed architectures. Providing communication support saves significant application development time and allows for more effective use of this time. Intel's iDCM product line combines hardware and software to provide this function.

In an iDCM system, communication between nodes occurs via the BITBUS microcontroller interconnect. The BITBUS microcontroller interconnect is a high performance serial control bus specifically intended for use in applications built on distributed architectures. The iRMX 51 Executive provides BITBUS support.

BITBUS™/iDCM COMPATIBLE

A pre-configured version of the iRMX 51 Executive implements the BITBUS message format and provides all iRMX 51 facilities mentioned previously: task management, interrupt handling, and message passing. This version of the Executive is supplied in firmware on the iDCM Controller with the iDCM hardware products: the iSBX 344 BITBUS Controller MULTIMODULE and the iRCB 44/10 BITBUS Remote Controller boards. It is also supplied on diskette as part of the iRMX 510 iDCM Support Package to ease development of BITBUS systems.

SIMPLE USER INTERFACE

The iRMX 51 Executive's capabilities are utilized through system calls. These interfaces have been defined for ease of use and simplicity. Table 2 includes a listing of these interfaces and their functions. Note tasks may be created at system initialization or run-time using the CREATE TASK call.

Functions such as GET FUNCTION IDS, ALLOCATE/DEALLOCATE BUFFER, and SEND MESSAGE (Messages in the iRMX 51 Executive have a maximum size of 255 bytes.), support communication for distributed architectures. Architectures that define multiple remote stations requiring intelligent and dumb I/O manipulation. The remaining

Table 2. iRMX™ 51 System Interfaces

COMMAND	DESCRIPTION
RQ SEND MESSAGE	Sends a message (a command from the BITBUS master, a response from a slave, or a simple message between tasks on the same BITBUS component) to another task.
RQ WAIT	Waits for an interrupt, an event time-out, a message, or any combination of the three.
RQ CREATE TASK	Causes a new sequence of code to be run as an iRMX 51 task with a specific function identification code and priority.
RQ DELETE TASK	Stops the specified task and removes it from all execution lists.
RQ ALLOCATE	Allocates a fixed-length buffer from the on-chip, scratch-pad RAM for general use, or, in BITBUS applications, for a BITBUS message buffer.
RQ DEALLOCATE	Returns an on-chip buffer to the system.
RQ SET INTERVAL	Set the time interval to be used as a separate event-timer for the task.
RQ ENABLE INTERRUPT	Allow external interrupts to signal the microcontroller.
RQ DISSABLE INTERRUPT	Stops all external interrupts from signaling the microcontroller.
RQ GET FUNCTION ID	Provides a list of the 8 function identification codes representing the tasks currently operating on the microcontroller.

interfaces allow the user to specify the system's response to the external environment — a must for real-time control.

Another feature that eases application development is automatic register bank allocation. The Executive will assign tasks to register banks automatically unless a specific request is made. The iRMX 51 Executive keeps track of the register assignments allowing the user to concentrate on other activities.

The user configures an iRMX 51 system simply by: specifying the initial set of task descriptors and configuration values, and linking the system via the RL 51 Linker and Locator Program with user programs. The nature of the task descriptors allows the user to develop programs, locate them in off-chip ROM, and access them without writing additional code. Programs may be written in ASM 51 or PL/M 51. (Intel's 8051 Software Development Package contains both ASM 51 and RL 51. The iRMX 51 Executive supplies the configuration file and macro defining initial task descriptors.) Figure 3 shows the relationships that exist in the system generation process.

RELIABLE

Real-time control applications require reliability. The nucleus requires about 2K bytes of code space, 40 bytes on-chip RAM, & 218 bytes exter-

nal RAM. Streamlined code increases performance and reliability, and flexibility is not sacrificed as code may be added to either on-chip or external memory.

The iRMX 51 architecture and simple user interface further enhance reliability and lower cost. For example, the straightforward structure of the user interfaces, and the transparent nature of the scheduling process contribute to reliability of the overall system by minimizing programming effort. Also, modularity increases reliability of the system and lowers cost by allowing user tasks to be refined independent of the system. In this way, errors are identified earlier and can be easily corrected in each isolated module.

In addition, users can assign tasks a Function ID that allows tracking of the tasks associated with a particular control/monitoring function. This feature reduces maintenance and trouble shooting time thus increasing system run time and decreasing cost.

OPERATING ENVIRONMENT

The iRMX 51 Executive supports applications development based on any member of the high performance 8051 family of microcontrollers. The Executive is available on diskette with user linkable libraries or in the Distributed Control Modules (iDCM) controller preconfigured in on-

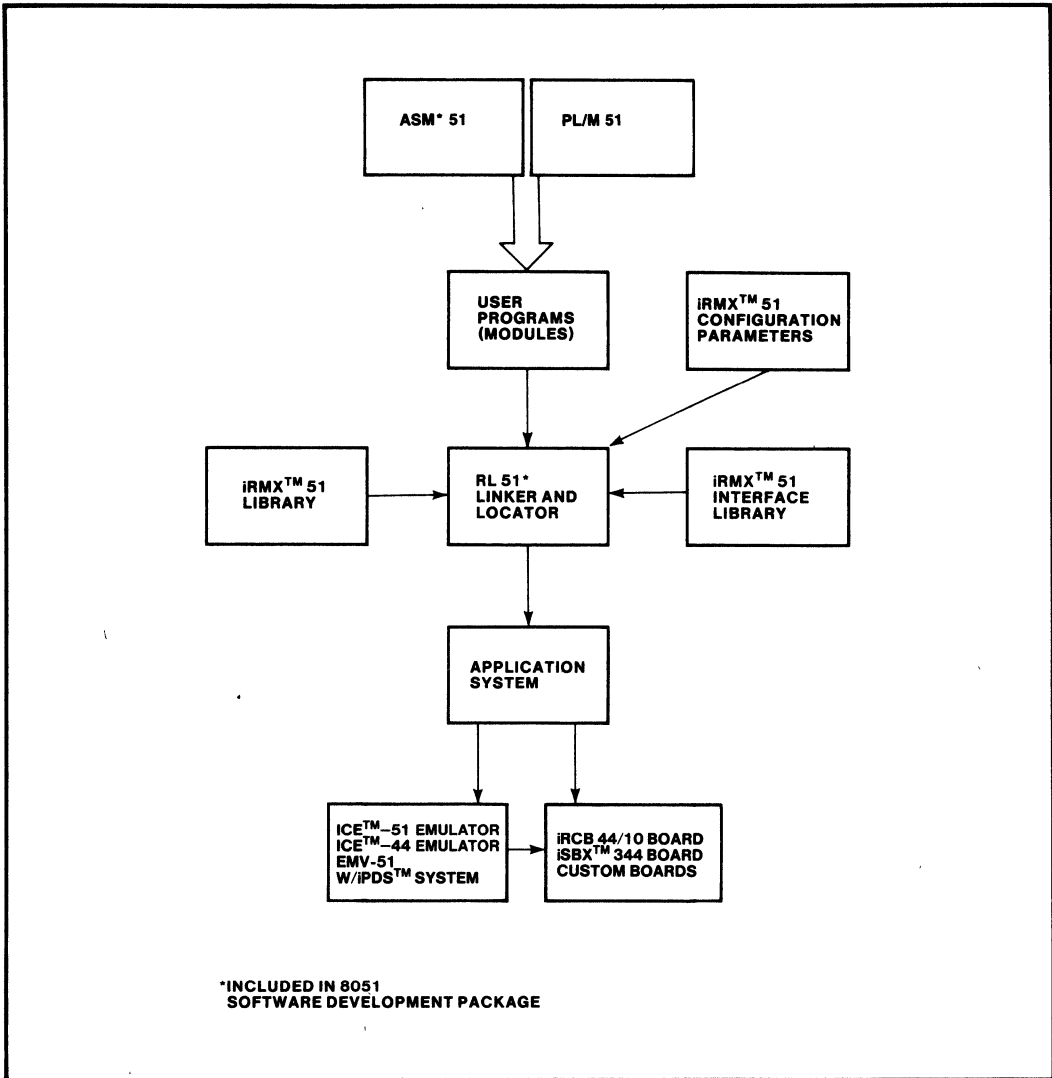


Figure 3. System Generation Process

chip ROM. (The iDCM controller is an 8044 component that consists of an 8051 microcontroller and SDLC controller on one chip with integral firmware.)

When in the iDCM environment (Figure 4), the iRMX 51 Executive can communicate with iRMX based systems like the System 286/310 or ISIS based systems like the Intel Portable Development System (iPDS) by using the iRMX 510 iDCM Support Package.

DEVELOPMENT ENVIRONMENT

Intel provides a complete development environment for the 8051 family of microcontrollers. This environment encompasses iDCM system (BITBUS based) applications also. Software development support consists of: the 8051 Software Development Package, and the iRMX 510 iDCM Support Package. Hardware tools consist of a variety of In Circuit Emulators (ICE), Intel's Portable Development System (iPDS) with EMV-51, and Intellec® Series II or III Development Systems.

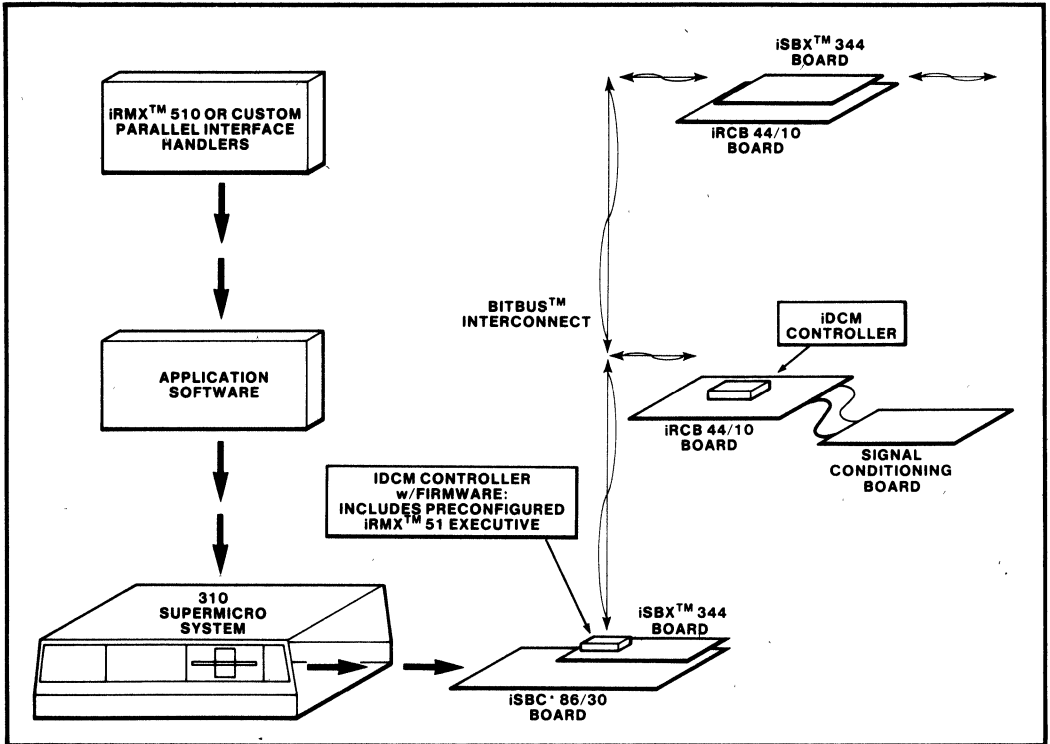


Figure 4. iDCM Operating Environment

SPECIFICATIONS

Supported Hardware

Microcontrollers

8051	80C51
8052	8044
8751	8744
8031	80C31
8032	8344

iDCM Product Line

iSBX 344 MULTIMODULE Board
 iRCB 44/10 Remote Controller Board

Compatible Software

IRMX™ 510 iDCM Support Package

Development Tools

ICE™ 51 or ICE 44 Emulators
 Intellec Series II or III Development System
 iPDS System w/EMV-51
 iRMX 510 iDCM Support Package
 8051 Software Development Package

Reference Manual (Supplied)

146312-001 — Guide to Using the Distributed Control Modules

Ordering Information

Part Number Description

IRMX 51BY	Executive for 8051 Family of Microcontrollers with Reference Manual. A, B, and F Media Formats Supplied
-----------	---



MCS[®]-96 SOFTWARE DEVELOPMENT PACKAGES

■ MCS[®]-96 Software Support Package

■ PL/M-96 Software Package

MCS[®]-96 SOFTWARE SUPPORT PACKAGE

■ Symbolic relocatable assembly language programming for the 8096 microcontroller family

■ System Utilities for Program Linking and Relocation

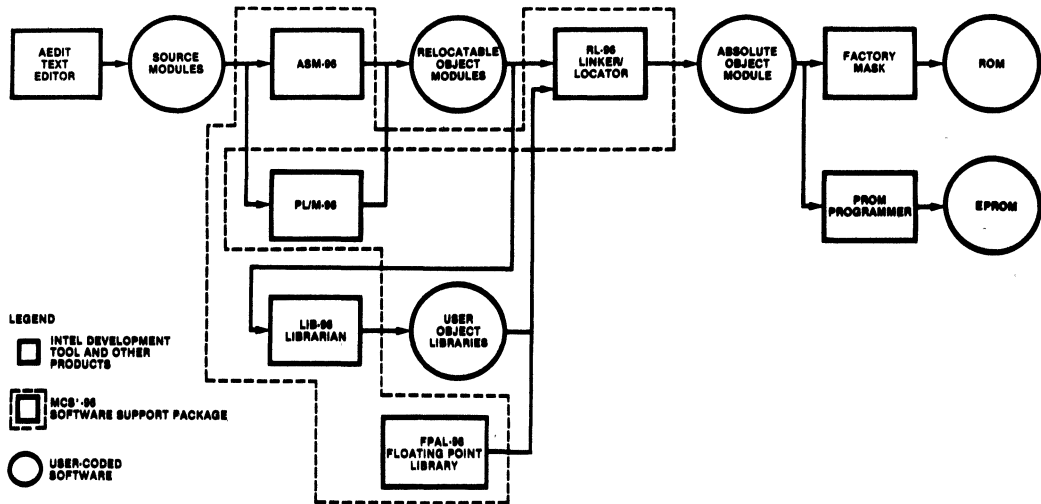
■ Extends Intellec[®] Microcomputer Development System to support MCS-96 program development

■ Encourages modular program design for maintainability and reliability

The MCS[®]-96 Software Support Package provides development system support for the MCS-96 family of 16-bit single chip microcomputers. The support package includes a macro assembler and system utilities.

The assembler produces relocatable object modules from MCS-96 macro assembly language instructions. The object modules then are linked and located to absolute memory locations.

The assembler and utilities run on the Intellec[®] Series III or equivalent Microcomputer Development System.



230613-1

Figure 1. MCS[®]-96 Software Development Process

8096 MACRO ASSEMBLER

- **Supports 8096 family program development on Intel[®] Microcomputer Development System**
- **Object files are linkable and locatable**
- **Gives symbolic access to powerful 8096 hardware features**
- **Symbolic Assembler supports macro capabilities, cross reference, symbol table and conditional assembly**

ASM-96 is the macro assembler for the MCS family of microcontrollers. ASM-96 translates symbolic assembly language mnemonics into relocatable object code. Since the object modules are linkable and locatable, ASM-96 encourages modular programming practices.

The macro facility in ASM-96 allows programmers to save development and maintenance time since common code sequences only have to be done once. The assembler also provides conditional assembly capabilities.

ASM-96 supports symbolic access to the many features of the 8096 architecture. An "include" file is provided with all of the 8096 hardware registers defined. Alternatively, the user can define any subset of the 8096 hardware register set.

Math routines are supported with mnemonics for 16 × 16-bit multiply or 32/16-bit divide instructions.

The assembler runs on a Series III/Series IV Intel[®] Development Systems for high performance.

RL96 LINKER AND RELOCATOR PROGRAM

- **Links modules generated by ASM-96 and PL/M-96**
- **Encourages modular programming for faster program development**
- **Locates the linked object module to absolute memory locations**
- **Automated selection of required modules from Libraries to satisfy symbolic references**

RL96 is a utility that performs two functions useful in MCS-96 software development:

- The link function which combines a number of MCS-96 object modules into a single program.
- The locate functions which assigns an absolute address to all relocatable addresses in the MCS-96 object module.

RL96 resolves all external symbol references between modules and will select object modules from library files if necessary.

RL96 creates two files:

- The program or absolute object module file that can be executed by the targeted member of the MCS-96 family.
- The listing file that shows the results of link/locate, including a memory map symbol table and an optional cross reference listing.

The relocater allows programmers to concentrate on software functionally and not worry about the absolute addresses of the object code. RL96 promotes modular programming. The application can be broken down into separate modules that are easier to design, test and maintain. Standard modules can be developed and used in different applications thus saving software development time.

FPAL96 FLOATING POINT ARITHMETIC LIBRARY

- Implements IEEE Floating Point Arithmetic
- Basic Arithmetic Operations
+, -, ×, /, Mod Plus Square Root
- Supports Single Precision 32 Bit Floating Point Variables
- Includes an Error Handler Library

FPAL96 is a library of single precision 32-bit floating point arithmetic functions. All math adheres to the proposed IEEE floating point standard for accuracy and reliability. An error handler to handle exceptions (for example, divide by zero) is included.

The following functions are included:

ADD	NEGATE
SUBTRACT	ABSOLUTE
MULTIPLY	SQUARE ROOT
DIVIDE	INTEGER
COMPARE	REMAINDER

LIB 96

The LIB 96 utility creates and maintains libraries of software object modules. The customer can develop standard modules and place them in libraries. Application programs can then call these modules using predefined interfaces.

LIB 96 uses the following set of commands:

- CREATE: Creates an empty library file.
- ADD: Adds object modules to a library file.
- DELETE: Deletes object modules from a library file.
- LIST: Lists the modules in the library file.
- EXIT: Terminates LIB 96

When using object libraries, RL96 will include only those object modules that are required to satisfy external references, thus saving memory space.

SPECIFICATIONS

Operating Environment

Required Hardware:
Intel Microcomputer Development System
— Series III/ Series IV

Documentation Package:

MCS-96 Macro Assembler User's Guide
MCS-96 Utilities User's Guide
MCS-96 Assembler and Utilities Pocket Reference Card
8096 Floating Point Arithmetic Library

ORDERING INFORMATION

Part Number

IMDX-355
Requires Software License

Description

MCS-96 Software Support Package

SUPPORT:

Hotline Telephone Support, Software Performance Report (SPR), Software Updates, Technical Reports, and Monthly Technical Newsletters are available.

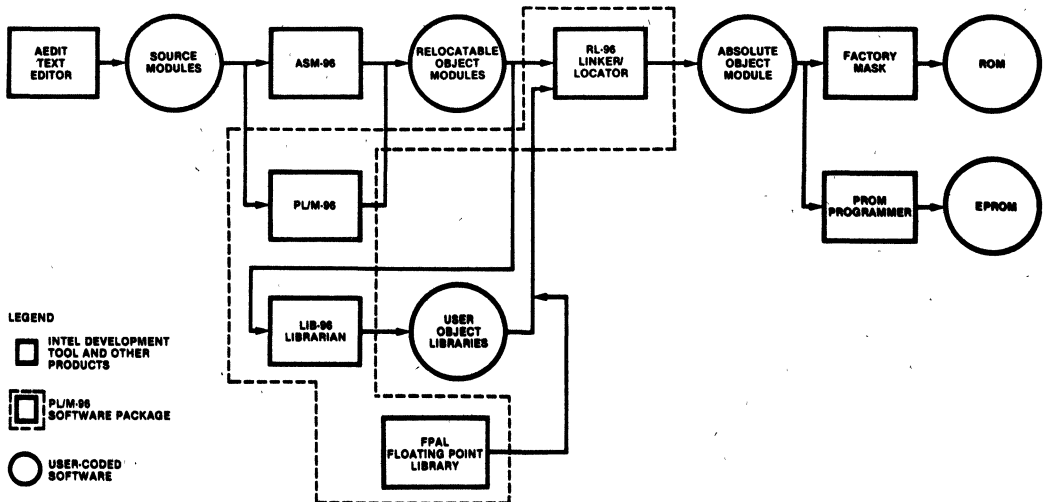
PL/M-96 SOFTWARE PACKAGE

- High level programming language for the Intel MCS[®]-96 microcontroller family
- Block structured language design encourages module programming
- Provides access to MCS[®]-96 on chip resources
- Produces relocatable object code which is linkable to object modules generated by other MCS[®]-96 translators
- Resident on IAPX-86 Intel microcomputer development systems for higher performance
- Includes a linking and relocating utility and the library manager
- IEEE Floating Point Library included for numeric support
- Compatible with PL/M-86 assuring design portability

PL/M-96 is a structured, high-level programming language useful for developing software for the Intel MCS-96 family of microcontrollers. PL/M-96 was designed to support the software requirements of advanced 16 bit microcontrollers. Access to the on chip resources of the MCS-96 has been provided in PL/M-96.

PL/M-96 is compatible with PL/M-86. Programmers familiar with PL/M will find they can program in PL/M-96 with little relearning effort.

The PL/M-96 compiler translates PL/M-96 high level language statements into MCS-96 machine instructions. By programming in PL/M an engineer can be more productive in the initial software development cycle of the project. PL/M can also reduce future maintenance and support cost because PL/M programs are easier to understand. PL/M-96 was designed to complement Intel's ASM-96.



230613-2

Figure 2. MCS[®]-96 Software Development Process

PL/M-96 COMPILER

FEATURES

Major features of the PL/M-96 compiler and programming language include:

Structured Programming

Programs written in PL/M-96 are developed as a collection of procedures, modules and blocks. Structured programs are easier to understand, maintain and debug. PL/M-96 programs can be made more reliable by clearly defining the scope of user variables (for example, local variables in a procedure). REENTRANT procedures are also supported by PL/M-96.

Language Compatibility

PL/M-96 object modules are compatible with all other object modules generated by Intel MCS-96 translators. Programmers may choose to link ASM-96 and PL/M-96 object modules together.

PL/M-96 object modules were designed to work with other Intel support tools for the MCS-96. The DEBUG compiler control provides these tools with symbolic information.

Data Types Supported

PL/M-96 supports seven data types for programmer flexibility in various logical, arithmetic and addressing functions. The seven data types include:

—BYTE:	8-bit unsigned number
—WORD:	16-bit unsigned number
—DWORD:	32-bit unsigned number
—SHORTINT:	8-bit signed number
—INTEGER:	16-bit signed number
—LONGINT:	32-bit signed number
—REAL:	32-bit floating point number

Another powerful feature are BASED variables. BASED variables allow the user to map more than one variable to the same memory location. This is especially useful for passing parameters, relative and absolute addressing, and memory allocation.

Data Structures Supported

Two data structuring facilities are supported by PL/M-96. The user can organize data into logical groups. This adds flexibility in referencing data.

- Array: Indexed list of same type data elements
- Structure: Named collection of same or different type data elements
- Combinations of Both: Arrays of structures or structures of arrays

Interrupt Handling

Interrupts are supported in PL/M-96 by defining a procedure with the INTERRUPT attribute. The compiler will generate code to save and restore the program status word when handling hardware interrupts of the MCS-96.

Compiler Controls

Compile time options increase the flexibility of the PL/M-96 compiler. These controls include:

- Optimization
- Conditional compilation
- The inclusion of common PL/M-96 source files from disk
- Cross reference of symbols
- Optional assembly language code in the listing file

Code Optimizations

The PL/M-96 compiler has four levels of optimization for reducing program size.

- Combination of constant expressions; "Strength reductions" (e.g.: a shift left rather than multiply by two)
- Machine code optimizations; elimination of superfluous branches; reuse of duplicate code, removal of unreachable code
- Overlaying of on chip RAM variables
- Optimization of based variable operations
- Use of short jumps where possible

Built in Functions

An extensive list of built in functions has been supplied as part of the PL/M-96 language. Besides TYPE CONVERSION functions, there are built in functions for STRING manipulations. Functions are provided for interrogating the MCS-96 hardware flags such as CARRY and OVERFLOW.

Error Checking

If the PL/M-96 compiler detects a programming or compilation error, a fully detailed error message is provided by the compiler. If a syntax or program error is detected, the compiler will skip the code generation and optimization passes. This powerful PL/M-96 feature can yield a two times increase in throughput when a user is in the initial program development cycle.

BENEFITS

PLM-96 is designed to be an efficient, cost-effective solution to the special requirements of MCS-96 Microcontroller Software Development, as illustrated by the following benefits of PL/M use:

Low Learning Effort

PL/M-96 is easy to learn and to use, even for the novice programmer.

Earlier Project Completion

Critical projects are completed much earlier than otherwise possible because PL/M-96, a structured high-level language, increases programmer productivity.

Lower Development Cost

Increases in programmer productivity translate immediately into lower software development costs because less programming resources are required for a given programmed function.

Increased Reliability

PL/M-96 is designed to aid in the development of reliable software (PL/M programs are simple statements of the program algorithm). This substantially reduces the risk of costly correction of errors in systems that have already reached full production status. The more simply the program is stated, the more likely it is to perform its intended function.

Easier Enhancements and Maintenance

Programs written in PL/M tend to be self-documenting, thus easier to read and understand. This means it is easier to enhance and maintain PL/M programs as the system capabilities expand and future products are developed.

RL96 LINKER AND RELOCATOR PROGRAM

- Links modules generated by ASM-96 and PL/M-96
- Encourages modular programming for faster program development
- Locates the linked object module to absolute memory locations
- Automated selection of required modules from Libraries to satisfy symbolic references

RL96 is a utility that performs two functions useful in MCS software development:

- The link function which combines a number of MCS object modules into a single program.
- The locate function which assigns an absolute address to all relocatable addresses in the MCS-96 object module.

RL96 resolves all external symbol references between modules and will select object modules from library files if necessary.

RL96 creates two files:

- The program or absolute object module file that can be executed by the targeted member of the MCS family.
- The listing file that shows the results of link/locate, including a memory map symbol table and an optional cross reference listing.

The relocator allows programmers to concentrate on software functionality and not worry about the absolute addresses of the object code. RL96 promotes modular programming. The application can be broken down into separate modules that are easier to design, test and maintain. Standard modules can be developed and used in different applications thus saving software development time.

FPAL96 FLOATING POINT ARITHMETIC LIBRARY

- Implements IEEE Floating Point Arithmetic
- Supports Single Precision 32 Bit Floating Point Variables
- Basic Arithmetic Operations
+, -, ×, /, Mod Plus Square Root
- Includes an Error Handler Library

FPAL96 is a library of single precision 32-bit floating point arithmetic functions. All math adheres to the proposed IEEE floating point standard for accuracy and reliability. An error handler to handle exceptions (for example, divide by zero) is included.

The following functions are included:

ADD	NEGATE
SUBTRACT	ABSOLUTE
MULTIPLY	SQUARE ROOT
DIVIDE	INTEGER
COMPARE	REMAINDER

LIB 96

The LIB 96 utility creates and maintains libraries of software object modules. The customer can develop standard modules and place them in libraries. Application programs can then call these modules using predefined interfaces.

LIB 96 uses the following set of commands:

- CREATE: Creates an empty library file
- ADD: Adds object modules to a library file
- DELETE: Deletes object modules from a library file
- LIST: Lists the modules in the library file
- EXIT: Terminates LIB 96

When using object libraries, RL96 will include only those object modules that are required to satisfy external references, thus saving memory space.

SPECIFICATIONS

Operating Environment

Required Hardware:
Intel Microcomputer Development System
— Series III/Series IV

Documentation Package:

PL/M-96 User's Guide
MCS-96 Utilities User's Guide
MCS-96 Assembler and Utilities Pocket
Reference Card
8096 Floating Point Arithmetic Library

ORDERING INFORMATION

Part Number

iMDX-356
Requires Software License

Description

PL/M-96 Software Package

SUPPORT:

Hotline Telephone Support, Software Performance Report (SPR), Software Updates, Technical Reports, and Monthly Technical Newsletters are available.



DEVELOPMENT PRODUCTIVITY TOOLS

INTRODUCTION

Improving an engineering team's productivity is a never ending task in today's competitive environments. Intel offers software tools and communication systems that optimize the usage of expensive engineering personnel and capital equipment. Software tools boost a programming team's productivity, thereby lowering development costs and shortening product development times. Communication software provides further productivity gains by linking multi-computer engineering environments into highly effective networks.

One software tool that substantially increases software productivity is PSCOPE, a source level symbolic debugger. The PSCOPE debugger allows the high-level language programmer to completely debug his code at the same level at which it was written. Breakpointing, tracing, and patching are all done in a faster and less error-prone manner than through obsolete machine-level debuggers. As software testing and maintenance consume a greater portion of development life-cycle time and cost, PSCOPE debugging can significantly improve programming efficiency.

Another set of valuable software tools are Intel's Program Management Tools (PMTs), which provide the essential ingredients to manage large software development projects. PMTs decrease the time spent on tracking program changes and manually generating new systems, thereby giving engineers more time for software design, development, and testing. PMTs consist of a Software Version Control System (SVCS), and an automated software generation facility (MAKE). Together these tools control, examine, and automate the management of a software system that may contain many versions consisting of numerous modules.

Intel's software toolboxes are collections of utilities that perform a variety of productivity-oriented functions. The ISIS-II Software Toolbox offers conditional submit file control tools, source management tools, and other tools that operate at the ISIS-II command level. The 8086 Software Toolbox is a collection of 16-bit software tools that are valuable for text formatting and preparation, software testing and performance analysis, 286/287 software development, and a multitude of other applications.

Intel also offers AEDIT, an advanced editor that significantly improves programmer productivity. AEDIT was designed with the programmer in mind, and offers full screen editing, the ability to edit two files at once, features for manipulating large blocks of text, and dynamic macro command definition.



PSCOPE HIGH-LEVEL PROGRAM DEBUGGER

- Source-Level Debugging for High Productivity
- Breakpoint, Single-Step and Execution Trace by Statement Numbers, Procedure Names and Labels
- High-Level Code Patching
- Compatible with Intel's I²ICE™ Integrated Instrumentation and In-Circuit Emulation System for Target System Debugging
- Native CPU Execution for iAPX 88 and 86 Architectures
- Supports PL/M, Pascal, and FORTRAN Program Debugging

PSCOPE is an interactive, symbolic debugger for high-level language programs. It allows users to scrutinize program execution at the source level, using high-level statement numbers, procedure and variable names and labels. This is typically a more productive way of debugging high-level language (HLL) programs than at the machine level.

Source-level debugging means that traditional functions, such as setting breakpoints or tracing execution flow, are more powerful in PSCOPE. For example, tracing procedure entry (or exit) points conveys much more information than tracing machine instructions. Single-step execution is more powerful, using statements and procedures, as well.

The productivity improvement from debugging in a high-level language is analogous to programming in a high-level language, when compared to assembly-level programming and debugging.

PSCOPE users may define high-level code patches, which are "compiled" and patched into the user's program. Code patches may be stored on disk, so they may be later incorporated into the program source file.

PSCOPE is an integral part of the advanced I²ICE Integrated Instrumentation and In-Circuit Emulation System. This allows a smooth migration from *program* debugging to *target system* debugging.

PSCOPE's symbol capacity is virtually unlimited. Symbols are paged to disk when necessary.

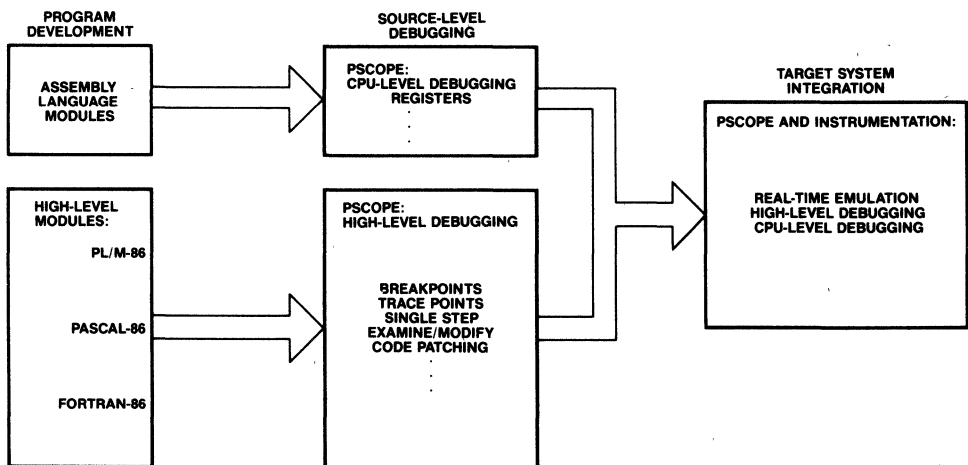


Figure 1. Debugging Methodology with PSCOPE

SAMPLE DEBUGGER SESSION

SÉRIES-III Pascal-86, V1.1

Source File: :F2:MAXMIN.PAS
Object File: :F2:MAXMIN.OBJ
Controls Specified: DEBUG.

STMT	LINE	NESTING	SOURCE TEXT: :F2:MAXMIN.PAS
1	1	0 0	program calc(input,output);
2	2	0 0	var a,b:integer;
3	4	0 0	procedure sum(x,y:integer);
4	5	1 0	var z:integer;
5	6	1 0	begin
5	7	1 1	z:=x*y;
6	8	1 1	writeln('The sum is ',z);
7	9	1 1	end;
8	11	0 0	procedure difference(x,y:integer);
9	12	1 0	var z:integer;
10	13	1 0	begin
10	14	1 1	z:=abs(x-y);
11	15	1 1	writeln('The difference is ',z);
12	16	1 1	end;
13	18	0 0	procedure maxmin(x,y:integer);
14	19	1 0	begin
14	20	1 1	if x<y then writeln('The maximum is ',y,
			The minimum is ',x);
16	21	1 1	if y<x then writeln('The maximum is ',x,
			The minimum is ',y);
18	22	1 1	if x=y then writeln ('The two inputs are equivalent ');
20	23	1 1	end;
21	25	0 0	begin
21	26	0 1	repeat (*forever*)
21	27	0 2	write('Input two integers ');
22	28	0 2	readln(a,b);
23	30	0 2	sum(a,b);
24	31	0 2	difference(a,b);
25	32	0 2	maxmin(a,b);
26	33	0 2	until l<0
27	34	0 2	end.

The program listing for the sample PSCOPE session illustrates the high-level nature of PSCOPE debugging. The program consists of the module CALC, the procedures SUM, DIFFERENCE, and MAXMIN, plus global and local variables. Users exercise and manipulate the program using these symbols. Code

patches, stepping, tracing, etc. are all done on line numbers, procedures, labels, and symbolic names. To debug a program, just PSCOPE and a listing are required—no linkage maps, core dumps, locate maps, etc. are necessary. This is how high-level debugging relates to high-level programming.

FEATURES

Unlimited Breakpoints

Breakpoints may be set on statement numbers, procedure names, or program labels. Any number of breakpoint registers may be defined.

High-Level Trace Points

Execution trace points are defined the same way as program breaks. Any number of trace points may be defined. A trace message is displayed when execution reaches a trace point.

Conditional Break and Trace

Any break or trace point may be defined to automatically call a *debugger procedure*, which will execute PSCOPE commands and/or evaluate predefined conditions. The operations will be performed, and the condition will determine if the break or trace will be done.

GO

The GO command initiates program execution from any starting point. A set of stopping points may be specified ("GO TIL"), and break/trace registers may be used ("GO USING").

Source-Level Stepping

A program may be executed, one high-level statement at a time, using the LSTEP command. Also, entire procedures may be treated as single statements during stepping (PSTEP); the procedures will be executed, but not stepped through.

Examine/Modify Data

PSCOPE allows users to symbolically examine (and change the value of) program variables and data structures. All PL/M and Pascal types are supported, including numerics, dynamic and stack variables, arrays, and fields within structures.

Virtual Symbol Table

All user-program symbols are stored in a virtual symbol table. This means symbols will be paged to disk, if necessary.

Help File

Many PSCOPE commands, facilities, and error messages have help information describing their use. The HELP command is used for learning the

PSCOPE command language, for quick reference of command syntax, and for learning the cause of command errors.

Debugger Procedures

PSCOPE has the facility for defining procedures in its command language. This block-structured command language allows users to *extend* the capability of the program under debug. Like macros with parameters, these procedures may also be used for generating compound and conditional debugger commands.

Code Patching

Program patches may be written in the debugger command language to augment or replace current program statements. These high-level code patches are much closer to actual program changes than machine-level patches, and are easy to use.

Built-in Editor

A menu-driven, CRT-oriented editor is built into PSCOPE. This is used for creating and editing program patches, debugger procedures, and command lines. One key is used to invoke the editor to alter the last command entered, or any debugger definition (literally, trace register, patch, etc.) may be edited selectively.

Debugger Command Language

GO/LSTEP/PSTEP—For controlling program execution.

DEFINE/DISPLAY/MODIFY/REMOVE—For manipulating debugger objects (such as break registers, patches, and procedures), or program objects (variables and data structures).

CALL/RETURN—For executing debugger procedures.

WRITE/CI—For console input and output.

DO/END—For defining command blocks.

REPEAT/COUNT—For repetition of commands or blocks.

IF/THEN/ELSE—For conditional execution of commands or blocks.

INCLUDE/PUT/APPEND—For saving/restoring commands and definitions to and from disk.

BENEFITS

Shortened Development Cycle

The ability to define debugger procedures and make code patches is very useful. It actually allows users to *extend* the capability of the program under debug. After debug sessions, users typically make program changes or enhancements. This involves the use of an editor, compiler and linkage tools that create a "new" load module for debugging. Since PSCOPE allows these changes and enhancements to be made *in the debugger*, the number of Edit/Compile/Link iterations is lowered. More confidence can be placed on a program during debugging, because its capabilities have been more fully exercised.

Improved Debugging Productivity

PSCOPE provides users with the same conceptual interface to program debugging that was used in program design. This includes the high-level language constructs such as statements, procedures, labels and symbolic variables and data structures. Functions such as program trace and single-step execution are more meaningful with statements and procedures than machine instructions; therefore the improvement in debugging productivity is analogous to the programming productivity using high-level languages.

More Reliable Software

Debugger procedures may be used to automate the software testing process. The procedure may repeatedly generate test values, execute the program with the input values, and record the results. Running more comprehensive tests, plus being able to "batch" the tests, yields more reliable software.

Easy to Learn and Use

An extensive command language, which is similar to block-structured languages such as PL/M and Pascal, is very easy to use in an interactive debug session. The HELP facility makes learning to use PSCOPE extremely fast as well. The "Literally" facility and debugger procedures also allow users to extend and tailor the command language to suit individual needs.

Improved Software Management

The use of debugger procedures allows parts of a software system to be debugged independently. Procedures can be substituted for program stubs, allowing programmers to debug different pieces of the system separately. This results in improved project management.

SPECIFICATIONS

Supports Intel's standard 86/88 languages:

- PL/M 86/88
- Pascal 86/88
- FORTRAN 86/88

PSCOPE runs on an Intellec® Series III or Series IV Microcomputer Development System, either stand-alone or in an NDS-II network configuration. A 512K application memory space is recommended for most applications.

ORDERING INFORMATION

Order Code	Description
iMDX-333	PSCOPE Program Debugger (for Series III and Series IV)
III-951A	PSCOPE Program Debugger and I ² ICE Base Software for Series III with 8" single density disk drive
III-951B	PSCOPE Program Debugger and I ² ICE Base Software for Series III with 8" double density disk drive
III-951C	PSCOPE Program Debugger and I ² ICE Base Software for Series IV with 5¼" double density disk drive

```

-run :fl:pscope
SLR1E5-III PSCOPE-86, V1.0
*
*define literally d = 'define'
*d literally l = 'literally'
*d l br = 'brkreg'
*d l tr = 'trcreg'
*
*
*load :fl:maxmin.86
*dir
DIR of :CALC
PJ_OUTPUT . . . . . TEXT (file)
PJ_INPJT . . . . . TEXT (file)
B . . . . . integer
A . . . . . integer
SUM . . . . . procedure
X . . . . . integer
Y . . . . . integer
Z . . . . . integer
DIFFERENCE . . . . . procedure
X . . . . . integer
Y . . . . . integer
Z . . . . . integer
MAXMIN . . . . . procedure
X . . . . . integer
Y . . . . . integer
*
*
*ptest
[Step at :CALC#21]
*ptest
      INPUT TWO INTEGERS:
[Step at :CALC#22]
*ptest
      (input)  19  4
[Step at :CALC#23]
*ptest
      THE SUM IS  76
[Step at :CALC#24]
*ptest
      THE DIFFERENCE IS  15
[Step at :CALC#25]
*ptest
      THE MAXIMUM IS  19
      THE MINIMUM IS  4
[Step at :CALC#21]
*
*
*define patch #5 til #6 = z=x+y
*go til #21
      INPUT TWO INTEGERS:
      (input)  19  4
      THE SUM IS  23
      THE DIFFERENCE IS  15
      THE MAXIMUM IS  19
      THE MINIMUM IS  4
[Break at #21]
*
*
*define proc PR1 = do
.*write 'the numbers and product are: ',a,b,a*b
.*write using ('0,>') 'break ? '
.*if CI == 'y' then return true
.** else return false endif
.*end
*
*d br B3 = #21 call PR1
*go using b3
      INPUT TWO INTEGERS:
      (input)  23  24
      THE SUM IS  47
      THE DIFFERENCE IS  1
      THE MAXIMUM IS  24
      THE MINIMUM IS  24
the numbers and the product are: +23 +24 +552
break ? y
[break at #21]
*
*
*exit
PSCOPE terminated

```

The Literally facility allows users to abbreviate, redefine and extend the command language to suit individual needs.

Any PL/M-86, Pascal-86 or FORTRAN-86 program may be loaded. All symbolic names may be displayed, in total or by type. Symbols defined at debug time may be displayed as well. All program types are supported, including numerics, user-defined types, and records. The symbols' types are displayed by the DIR command as well.

Several flavors of stepping are offered. This example illustrates PSTEP, a line-by-line step where procedures are executed as a single step. This program contains five steps in the main body, with three being procedure calls.

There appears to be a bug in the program, as the sum is displayed incorrectly. Looking at the program, we notice that X and Y were multiplied instead of added, at line #5. A code patch is defined, and the program executes correctly.

This illustrates the facility where a *debug procedure* (PR1) is called when reaching a breakpoint at line #21. Here, some values are displayed, and a condition is evaluated (in this case, a query to the user). Had the condition been false, program execution would continue with no break. The high-level constructs in the command language make this a very powerful facility.



PROGRAM MANAGEMENT TOOLS

- Increase Software Engineering Productivity
- Decrease Software Administration Overhead
- Allow Users to Control, Automate and Examine the Evolution of a Software Project
- Enhance the Capability of Networked (NDS-II) and Standalone Development Systems
- SVCS Simplifies Administration of Software Modules and Systems
- MAKE Automatically Generates New Releases of Software Systems
- Both Tools Easily Incorporated Into Existing Software Development Methodologies

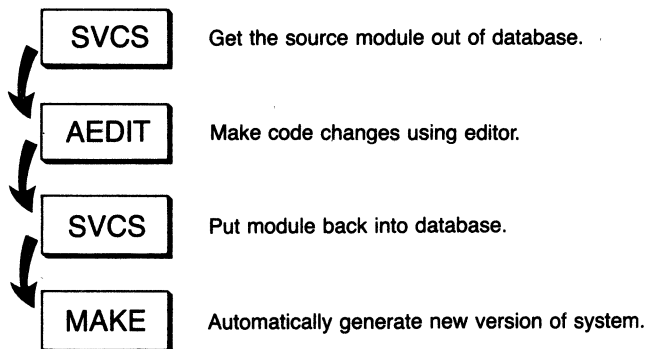
Intel's Program Management Tools (PMTs) provide the essential ingredients to manage large software development projects. PMTs decrease the time spent on tracking program changes and manually generating new systems, thereby giving engineers more time for software design, development, and testing.

PMTs consist of a "Software Version Control System" (SVCS), and an automated software generation facility (MAKE). Together these tools control, examine, and automate the management of a software system that may contain many versions consisting of numerous modules.

SVCS controls and documents software changes for all file types. SVCS handles storage and retrieval of different versions of a given module, controls update privileges, prevents different users from making changes independently, and requires all changes be thoroughly documented by recording who made what changes, when and why.

MAKE produces the specification of a "minimum-work" job required to generate a new system. This job (i.e. submit file) typically includes compiles and links of the latest versions of specified source and object modules. If a newer source module exists for any specified object module, MAKE will specify a compile of this module, replacing the older module in the completed program. Unnecessary links and compiles, however, are eliminated. MAKE does the minimum work required to ensure consistent, up-to-date software, thus saving many hours of compiles and links.

Incorporating PMTs into an existing project is easy. PMTs work with existing operating systems and software tools (editors, compilers, utilities) and require very little relearning. New users can quickly gain expertise in using PMTs by working through the examples contained in the PMT Tutorial Manual and Diskette, which are included with every PMT software package. Program Management Tools are ideal in a networked (NDS-II) environment, where multi-version software control is critical. PMTs are also extremely valuable on standalone systems (with Winchester disk) as well.



OPTIMAL CONTROL OF A SOFTWARE PROJECT.

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. Information Contained Herein Supersedes Previously Published Specifications On These Devices From Intel.

SOFTWARE VERSION CONTROL SYSTEM (SVCS)

- **Simplifies Administration of Software Modules and Systems**
- **Maintains Change History Information on Every Module**
- **Prevents Users From Accidentally Deleting System Software or Making Simultaneous Module Changes**
- **Offers an Effective Software Version Generation and Control Mechanism**

Intel's Software Version Control System (SVCS) is a utility that greatly simplifies software system housekeeping. SVCS automatically controls and documents software modules in a large project, eliminating costly manual administration by a project leader or librarian.

SVCS maintains a system database of software modules called units. Each unit is divided into four classes: Source, which contains the unit's source code; Object, which contains the unit's object code; History, which contains the unit's history file; and Composition, which can be arbitrarily used by the user.

Users interact with the database by using SVCS administrative and access commands. Project managers use administrative commands to create new system databases, add and delete database units, set unit access rights, and create and name new system variants. Programmers use SVCS access commands to check out and return database modules when making system changes. For every change made, SVCS records *what* changed, *who* changed it, *when* it was changed, and *why*.

SVCS variant generation and control enable project administrators to effectively create and identify new versions of software systems. Stable versions may be write protected and placed in the public domain, working versions may be identified and accessible only to programming personnel, and special versions may be created for customized releases. In addition, version control can minimize software archival, maintenance, and support administrative overhead.

AUTOMATED SOFTWARE GENERATION (MAKE)

- **Automatically Creates New Software Systems, Using the Latest Versions of Source Modules**
- **Automatically Determines Which Source Modules Need Recompiling**
- **Eliminates Unnecessary Compiles and Links**
- **Works Closely with SVCS for Generating Complete, Up-To-Date Systems**
- **Easily Adopted into Existing Development Methodologies**
- **Offers Many Powerful Macro Constructs**

MAKE is a utility that greatly simplifies the generation of software systems. MAKE produces a "minimum-work" submit file that can generate a complete, up-to-date system without any unnecessary compiles and links. MAKE can reduce system generation times from hours to minutes while concurrently minimizing administrative overhead.

MAKE accepts a text input file that instructs it how to generate a new software system. The input file specifies all modules required to generate the new system and includes a description of system dependencies. It also specifies specific system operations, such as compiles, links, SVCS operations, line-printer spoolings, and other system commands. MAKE uses this input file in conjunction with the time and date stamps on each module to determine the optimum system generation procedure that eliminates all unnecessary compiles and links.

Typically a MAKE input file is created once at the start of a project. Very occasionally during the life of the project it may need modification. A powerful set of macros makes the creation and subsequent modification of a generation procedure an easy task. Overall, the management of the MAKE input file is negligible compared to maintaining numerous submit files for system generations.

The close relationship between SVCS and MAKE help simplify the overall job of software control at all levels. For example, the very latest version of a source module may not be stable enough to be included in a generation. A less functional, but more reliable version may exist. Since SVCS keeps unique versions distinct, an SVCS-module containing the more reliable version may be specified in the MAKE input file.

BENEFITS: SVCS AND MAKE

Intel's Program Management Tools eliminate common problems such as:

"We've modified module F00, which has introduced a new set of problems. Now we can't restore it back to the earlier version."

"Module F002 has been modified; no one seems to know who changed it, or why."

"We often have several programmers making changes to the same modules. Trying to avoid simultaneous changes is a lot of effort, and we waste time synthesizing two sets of changes into one module."

"To ensure that we release up-to-date, correct software, we periodically go into "release mode" for a few days. Everyone stops work completely while we find the latest versions, and then start the generation from the ground up. It literally takes days, when we could be making productive changes."

SVCS and MAKE together provide a service that fits easily into your existing design methodology, and solves administrative problems such as those described above.

SPECIFICATIONS

Networked, Multi-User Software Control

NDS-II with at least one Intellec Microcomputer Development System
iNDX, ISIS-III(N) System Software

Standalone Use

Intellec Series III with Model 750 Winchester Disk or Intellec Series IV
SVCS and MAKE will not operate on ISIS-II local floppies or Model 740 Hard Disks.
SVCS and MAKE may be exported from any workstation in an NDS-II configuration.

Documentation

"A User's Guide to Program Management Tools" (121958)

SOFTWARE SUPPORT

This product includes a 90-day initial support consisting of new software releases, updates, subscription services (software performance reports and technical reports), and telephone hotline support. Additional software support services are available separately.

ORDERING INFORMATION

Part Number	Description
iMDX-332	Intel Program Management Tools



ISIS-II SOFTWARE TOOLBOX

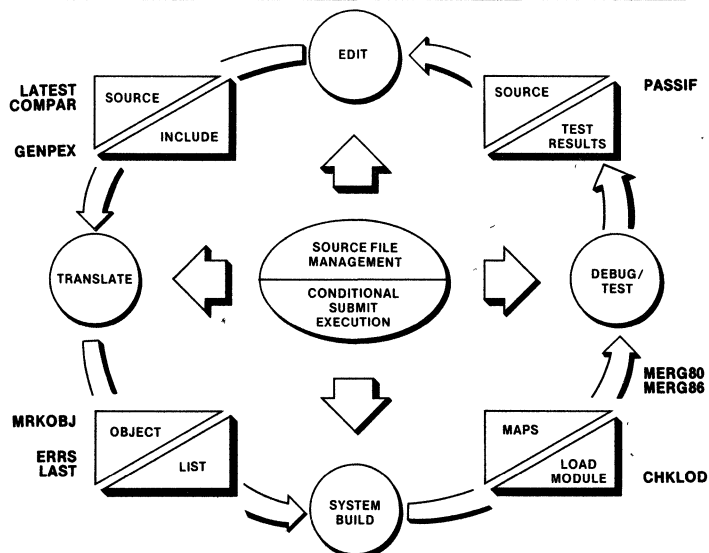
- Significantly Improves Programmer Productivity
- Provides Source File Management, Showing Source Changes, and Performing Version Control
- Collection of Utilities that Speed Up Software Design
- Provides Conditional Control and "Structured Programming" to Submit Files
- Enhances Capabilities of ISIS-II Operating System
- Runs on Model 800, Series II, and Series III Intellec® Development Systems
- Most Utilities will Operate on NDS-I Workstations, and Remote Hard Disks

The ISIS-II Software Toolbox is a collection of system utilities that perform a variety of "productivity-oriented" functions. There are two major subsets of Toolbox tools, in addition to numerous ad hoc utilities. These subsets provide Conditional Submit File Control and Source File Management.

The Conditional Submit File Control tools provide "structured programming" at the ISIS-II command level. Jumps, Calls, Returns, etc. are supported, as well as conditional command execution, based on assertions such as file existence, program errors, file matching, and string matching.

The Source Management Tools support version number tracking, and allow users to identify which versions of each source module were used to create a load module. There is also a tool which compares source files and reports all differences.

The tools outside of the two major subsets assist the programmer in some very specific development and debugging tasks. One tool manages all PUBLIC/EXTERNAL declarations in a system. Another merges the locate maps into a program listing, giving absolute symbolic debugging information. There's a directory sorter, a file compactor, and a tool to display just the last block of a file.



MANY TOOLS IN THE TOOLBOX ENHANCE SPECIFIC PHASES OF THE DEVELOPMENT CYCLE. OTHERS IMPROVE PRODUCTIVITY IN ALL PHASES.

FUNCTIONAL DESCRIPTION

Submit File Execution Control

IF/ELSE/ENDIF—conditional submit file execution based on file existence, program errors, pattern matching, plus several other conditions

GOTO—causes submit execution to resume at a specified label

RETURN—causes execution to return to the “submitter” (calling file)

EXIT—halts submit file execution

LOOP—forces execution to resume at the beginning of the submit file

RESCAN—allows submit execution to begin anywhere in file

NOTE—allows “progress report” notes to be placed in submit files

WAIT—displays a message, and waits for user input to continue or abort

STOPIF—halts submit file execution if specified listing contains errors

Source Management

XLATE2—submit-like tool with intelligent parameter substitution (for version control)

MRKOBJ—“marks” object modules with source version information

CHKLOD—lists source version data put in load modules by MRKOBJ

CLEAN—deletes all old versions off a specified disk

LATEST—displays latest version numbers of specified files

Operating System Functions

CONSOL—reassigns console input and console output as directed

DSORT*—alphabetically sorts floppy disk and hard disk directories

RELAB*—changes disk name to any other specified name

Program Development and Debugging

ERRS—fast display of program errors in PL/M 80, PL/M 86, and ASM 86 listings

MERG80—merges debug data from locate maps into PL/M 80 listings

MERG86—merges debug data from symbol maps into PL/M 86 and Pascal 86 listings

GENPEX—produces include file for PL/M external declarations (source level)

PASSIF—general purpose assertion checking, testing, and reporting tool

Text Processing

COMPAR—performs line-oriented text file comparison (shows source changes)

UPPER—changes all letters in an ASCII text file to uppercase

LOWER—changes all letters in an ASCII text file to lowercase

LAST—displays the last 512 bytes of a file

SORT—sophisticated line-oriented text file sorting tool

Disk Backup and File Processing

DCOPY—fast track-by-track diskette copying

HDBACK*—sophisticated hard disk to floppy disk backup program

PACK—compacts text files by removing strings of blanks

UNPACK—reconstitutes “packed” files

Disk Recovery

GANEF*—interactively reads and writes floppy or hard disk data blocks

Program Identification

WHICH—displays version number of Software Toolbox Programs

*These programs will not operate on the NDS-I remote hard disks.

ORDERING INFORMATION

Product Code	Description
MDS-363 ¹	ISIS-II SOFTWARE TOOLBOX

Requires software license.

SUPPORT CATEGORY: Level C

¹MDS is an ordering code only and is not used as a product name or trademark. MDS is a registered trademark of Mohawk Data Science.



8086 SOFTWARE TOOLBOX

- Collection of Tools That Speed Software Development
- MPL, a Standalone Macro Processor, is Ideal for Debugging Macros
- SCRIPT and SPELL Assist Text Preparation
- OMC286 and E80287 Aid 80286 and 80287 Software Development
- Many Other Valuable 16-Bit Software Tools Are Included
- Runs on Series III and Series IV Microcomputer Development Systems
- Runs under iRMX™ Operating System

The 8086 Software Toolbox is a collection of 16-bit software tools that can significantly improve programmer productivity. These tools are valuable for text formatting and preparation, software testing and performance analysis, 286/287 software development, and a multitude of other applications.

Text processing tools ease document formatting and preparation. SCRIPT is a text formatting program that uses commands embedded in text to do paging, centering, left and right margins, subscripts, etc. SPELL finds misspelled words in a text file and comes with a user expandable dictionary. COMP compares two text or source files and displays their differences.

Test and performance analysis tools aid software testing and performance evaluation. PERF, a performance analysis tool for 8086 software, is ideal for isolating code "hot spots." PASSIF is a general-purpose assertion checking and reporting tool perfect for running test suites.

Software development for 286/287 components is assisted by two software tools: OMC286, an 8086 to 80286 object module convertor, and E80287, an 80287 emulator that runs on the 80286.

Additional tools are included that aid 16-bit software development efforts. All tools run on Series III and Series IV Microcomputer Development Systems.

TEXT PROCESSING
SCRIPT MPL SPELL WSORT

PERFORMANCE MEASUREMENT & TESTING
PERF GRAFIT PASSIF

286/287 DEVELOPMENT
OMC286 E80287

MISCELLANEOUS TOOLS
COMP FUNC XREF DC HSORT ESORT

8086 SOFTWARE TOOLBOX TOOLS

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. Information Contained Herein Supercedes Previously Published Specifications On These Devices From Intel.

FUNCTIONAL DESCRIPTION

Text Processing

SCRIPT—text formatting program that does paging, centering, left and right margins, justification, page headers and footers, underlines, boldface type, subscripts and superscripts, upper and lower case, and much more. Formatting commands are embedded in text.

MPL—standalone macro processor that processes the macro language used in 8086, 80286, 8089, and 8051 assemblers. Can be used interactively which makes it ideal for debugging macros. MPL can be used to preprocess any text file.

SPELL—finds misspelled words in a text file. Dictionary of correctly spelled words is user expandable.

WSORT—utility for creating the SPELL dictionary.

COMP—performs line-oriented text file comparison (shows source changes). Also understands 8086 object module formats for comparing 8086 object files.

Performance Measurement and Testing

PASSIF—general-purpose assertion checking, testing, and reporting tool. Helps automate the software testing process.

PERF—performance analysis tool for 8086 software. Monitors references in the code segment; segment monitored is user defined. Works with small or compact bound loadable modules. Ideal for isolating code "hot spots." Will only run on the Series III.

GRAFIT—graphing utility for use with PERF.

Miscellaneous Tools

OMC286—object module converter that converts 8086 object modules into 80286 object modules.

E80287—an 80287 emulator that runs on the 80286.

FUNC—allows user to redefine the keys on a Series III keyboard and define function keys. Requires the iMDX 511 firmware.

XREF—produces cross-reference tables from translator list files. Cross-references all symbols—variables, labels, literals, and quoted strings.

DC—floating point desk calculator program; allows variable definitions.

HSORT—in memory heap sort utility.

ESORT—very flexible sort program.

SPECIFICATIONS

Operating Environment

ISIS Operating System with RUN or iNDX Operating System executing on Series III or Series IV Microcomputer Development Systems.

iRMX™86 Operating System executing in SYS X86/3XX environment.

Required Hardware

Series III or Series IV Microcomputer Development System

Required System Software

ISIS Operating System with RUN or iNDX Operating System

Documentation

"8086 Software Toolbox"
(122203)

Software Support

This product includes a 90-day initial support consisting of new software releases, updates, subscription services (software performance reports and technical reports), and telephone hotline support. Additional software support services are available separately.

ORDERING INFORMATION

Product Code	Description
iMDX-364	8086 Software Toolbox



AEDIT TEXT EDITOR

- **AEDIT-80 operates on any Intellec® Series II, Model 800 or iPDS™ Development System**
- **Full Screen Editing**
- **Menu-Driven, Easy to Use**
- **Easy Handling of Large Blocks of Text**
- **Dual File Editing**
- **AEDIT-86 Operates on any Intellec® Series III, Series IV, or iRMX™ system.**
- **Powerful Macro Facility**
- **Split-Screen Windowing**
- **Designed for the Programmer and Technical Writer**

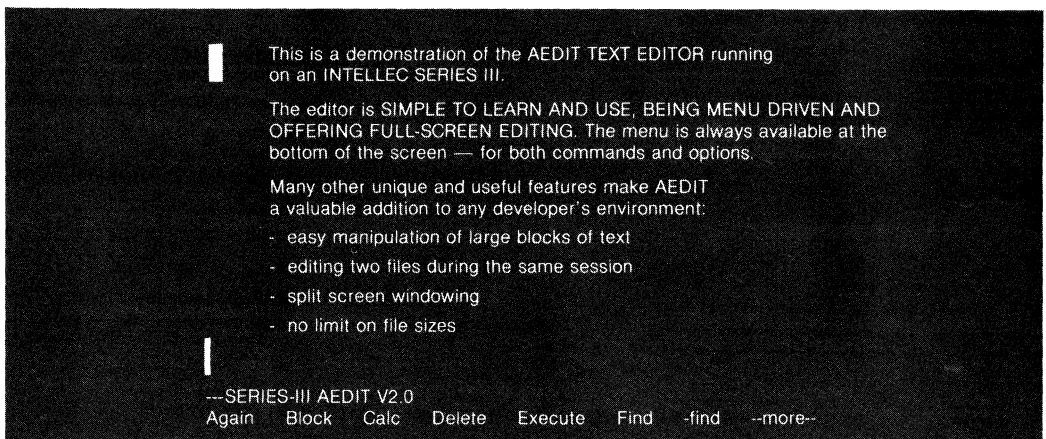
AEDIT is a full screen editor for use on any Intellec® Development or iRMX™ system. It is designed to be easy to learn and easy to use. At all times the user is guided by a menu which is used not only to select commands, but also to select options to commands. There is no need to constantly refer to or memorize detailed manuals.

AEDIT provides full screen editing capabilities and offers features to easily handle (move, copy, delete) large blocks of text. In addition to the basic editing abilities, AEDIT supports tagging positions in the text, string search and replace commands, and the option of automatic text indentation, spilling, and formatting. AEDIT is able to edit files of any length and optionally creates back-up copies of the file being edited

With AEDIT, two files can be edited during one session. The user can easily switch between the files for quick reference, editing, or to transfer text from one file to the other. Using the windowing capabilities available with AEDIT-86, both of these files may be displayed simultaneously in a split-screen format

AEDIT supports a powerful macro facility. AEDIT can create macros by simply keeping track of what a user is executing, "learning" the function the macro is to perform. The editor remembers the user's actions for later execution, and can store them in a file if requested. Alternately, a user may enter a macro using AEDIT's macro language, or modify any existing macro interactively.

These and many other features combine to make AEDIT the editor of choice



The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products. BXP, CREDIT, ICE, iCS, iM, iMsite, iMtel, INTEL, Intelelevision, Intellec, iMMX, iOSP, iPDS, iRMX, iSBC, iSBX, Library manager, MCS, MULTIMODULE, Megachassis, Micromainframe, Micromap, MULTIBUS, Multichannel, Plug-A-Bubble, PROMPT, Promware, RMX/80, System 2000, UPI, and the combination of iCS, iRMX, iSBC, iSBX, ICE, MCS, or UPI and a numerical suffix Intel Corporation Assumes No Responsibility for the use of Any Circuitry Other Than Circuitry Embodied in an Intel Product No Other Patent Licenses are implied
© INTEL CORPORATION, 1983

AEDIT TEXT EDITOR

MANUALS

AEDIT is supplied with a user manual documenting all the aspects of the editor, and a pocket reference card. The manual includes an introductory tutorial.

HOST SYSTEM

AEDIT-80 is an 8080/8085-based utility and can be run on any Intellec Development System, Series IIE, Series II, Model 800, or iPDS, as well as on ISIS Cluster workstations.

The higher-performance AEDIT-86 is an 8086-based utility that can be run on any Intellec Series IIIE, Series III, or Series IV Development system. Any Series IIE, Series II or Model 800 system can be upgraded to Series III functionality. AEDIT-86 is also available for the iRMX™ Operating Systems.

AEDIT can be configured to run with non-Intel terminals. Tested configurations are available for the following popular terminals:

ADDS Regent 200, Viewpoint 3A +
Beehive Mini-Bee
DEC VT52, VT100
Hazeltine 1510
Lear-Seigler ADM-3A
Zentec ZMS-35

*Regent 200 is a trademark of ADDS
Mini-Bee is a trademark of Beehive
DEC designated Digital Equipment Corporation
ADM-3A is a trademark of Lear-Siegler*

ORDERING INFORMATION

iMDX-335 AEDIT-80 Text Editor.
Includes 8" single and double density diskettes for Series IIE, Series II, or Model 800, and a 5¼" diskette for iPDS.

iMDX-334 AEDIT-86 Text Editor
Includes 8" single and double density diskettes for Series III.

COMMUNICATION SOFTWARE

Communications software is essential in an environment of multiple host systems. Intel recognizes the need to exchange information between Intel development systems, IBM mainframes, and DEC VAX minicomputers. Intel provides software to make this communication easy.

The asynchronous communication link (ACL) enables an Intel development system to transfer files to and from a VAX minicomputer. Two versions of this program are available, one for VAX/VMS and one for UNIX.

The mainframe link allows any ISIS-II user to transfer files to an IBM mainframe. This package uses the bisynch protocol to communicate with the IBM host by emulating a 2780/3780 batch terminal. Conversion of text files between ASCII and EBCDIC is supported.

iNA955 allows a user to connect an iRMX system to the NDS-II network for file transfer purposes. Network directories can be inspected and created remotely from the iRMX system. File transfer with this program occurs at Ethernet speeds.

iNA960 is an OEM product for use in iRMX systems. It is a general purpose Local Area Network software package that provides the user with guaranteed end to end message delivery. iNA960 conforms to ISO and Ethernet standards. It provides network management functions as well as the 82586 device drivers.

NDS-II Electronic Mail allows human to human communication without wasting time on missed telephone calls. Several types of mailboxes, such as private, group, or bulletin board, are supported. Typical mail uses are to send memos, collect project milestone data, and be a telephone message center.

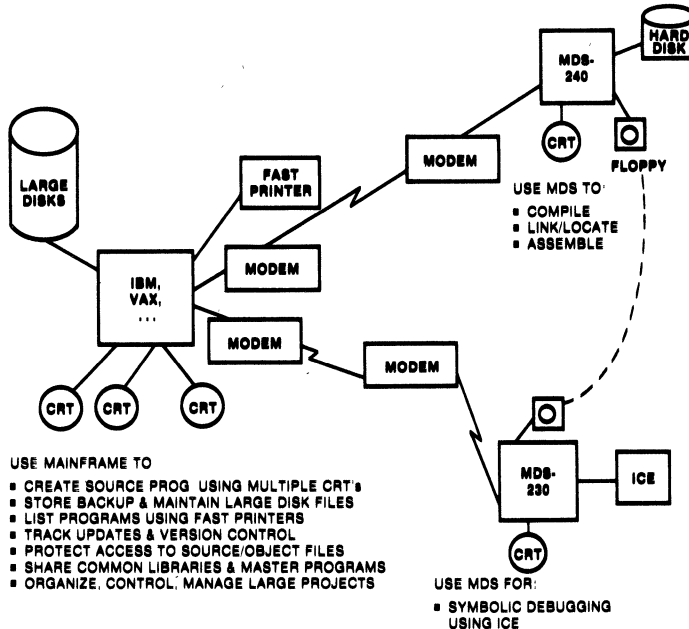


MAINFRAME LINK FOR DISTRIBUTED DEVELOPMENT

- Integrates user mainframe resources with Intellec® Development Systems.
- Uses IBM 2780/3780 standard BISYNC protocol supported by a majority of mainframes and minicomputers.
- Protocol supports full error detection with automatic retry.
- Software runs under ISIS-II on any Intellec® Development System.
- Communicates with remote systems on dedicated or switched (dial-up) telephone lines.
- Package also includes tests and a connector for loop-back self-test capability.

The Mainframe Link consists of software, modem cable to connect the development system to the modem and a loopback connector for diagnostic testing. The software runs under ISIS-II on Intellec Development Systems. It emulates the operation of an IBM 2780 or 3780 Remote Job Entry (RJE) terminal to (1) transmit ISIS-II files to a remote system or (2) receive files from a remote system using standard BISYNC 2780/3780 protocol. The remote system can be any mainframe or minicomputer which supports the IBM 2780 or 3780 communications interface standard. Files may contain ASCII or binary data so that either program source files (ASCII) or program object files (binary) may be transmitted.

The Mainframe Link allows the user to integrate in-house mainframe resources with Intellec Microcomputer Development resources. The mainframe can be used for storage, maintenance and management of program source and object files. The program source can be downloaded to a development system for compilation, assembly, linkage, and/or location. The linked modules can be transmitted and saved on the mainframe to be shared by all programmers. The linked program can then be downloaded to a development system for debugging using ICE emulation.



The following are trademarks of Intel Corporation and may be used only to identify Intel products: i, Intel, INTEL, INTELLEC, MCS, 'm, ICS, ICE, UPI, BXP, ISBC, ISBX, INSITE, IRMX, CREDIT, RMX/80, µScope, Multibus, PROMPT, Promware, Megachassis, Library Manager, MAIN MULTI MODULE, and the combination of MCS, ICE, SBC, RMX or ICS and a numerical suffix; e.g., ISBC-80.

FEATURES

- Runs under ISIS-II on any Inteltec® Microcomputer Development System.
- Communicates with a remote system using IBM 2780/3780 standard BISYNC protocol, which is supported by a majority of minicomputers and mainframes, on dedicated or switched (dial-up) telephone lines.
- The modem cable supplied with the package can be used to connect the Inteltec® Development System to the modem (or modem eliminator) using the standard RS232C port.
- Supports user selectable data transmission rates of up to 9600 baud.
- Package includes diagnostic tests used to verify the operation of the Inteltec® Development System using the loop-back connector supplied and data transmission up to the modem using the analog loop-back feature.
- System can be configured to match the requirements of the installation, i.e., using modem eliminators for connections up to fifty (50) feet, or by using modems and telephone lines.
- Software can be configured from several configuration options such as:
 - 2780, 3780 or Intel Mode
 - Transparent mode for binary data
 - Non-transparent mode for ASCII data

Automatic translation from ASCII to EBCDIC and vice versa

Receive chaining for receiving multiple files

- Intel mode is used mainly for file transfers between two Inteltec® Development Systems. The files are duplicated exactly.
- Console commands support all standard features including:
 - SEND data in Transparent or Non-transparent mode, with or without translation to EBCDIC
 - RECEIVE in Transparent or Non-transparent mode, with or without translation to EBCDIC.
 - Support for an IBM RJE console (such as HASP)
- Special utility programs are provided. STRZ strips extra binary zero's from the end of object files. CONSOL assigns system console input to an ISIS-II disk file.
- Can process commands interactively from the console or sequentially from an ISIS-II file under the SUBMIT facility for semi-automatic batch operation.
- Error detection in line transmission and error recovery by automatic retransmission.
- A special command such as DIAGNOSE, allows logging of all data activity on the line, during transmission and reception.
- When not used for communicating with the mainframe, the Inteltec® Development System is available as a complete, stand-alone system.

BENEFITS

- Allows the customer to use an in-house mainframe or minicomputer for program source-preparation, editing, back-up and maintenance using inexpensive CRT's and multi-terminal access. The common files may be shared and others protected.
- Many programmers can use and share the high-performance devices normally available on large computer systems, e.g., fast printers to reduce listing time, the large capacity disks with their fast access time to store large program files.
- The source files can be downloaded using the Mainframe Link to an Inteltec Development System (e.g., Model 240 or 245) for compilation, linking and locating.

- The compiled and/or linked object files may be transmitted back to the remote for storage. Updates and version numbers and dates can be tracked to ensure that the latest version is always used and back-up files are available. Binary object files can be later downloaded to an Inteltec Development System for debugging using an ICE emulator.
- In short, provides a powerful and flexible tool combining the best of both micro and mainframe worlds, i.e., powerful CPU with large disk capacity, file sharing, multi-terminal access, etc., from a mainframe or minicomputer with Intel's versatile and compatible software support systems (including PL/M, PASCAL, FORTRAN, Assembler, R & L) and sophisticated debugging tools such as ICE emulators.

SPECIFICATIONS**Operating Environment****Required Hardware:**

Intellec® Microcomputer Development System
Model 800
Models 220, 225, 230, 235, 240 or 245

64KB of Memory

One Diskette Drive
Single or Double Density

System Console
Intel CRT or non-Intel CRT

Recommended Hardware for Compilation:

Hard Disk (Models 240, 245, or Model 740 Upgrade)

Additional Hardware Required for Model 800
ISBC-955™, ISBC-534™

Required Software:

ISIS-II Diskette Operating System
Single or Double Density

Documentation Package

Mainframe Link User's Guide (121565-001)

Shipping Media

Flexible Diskettes
Single and Double Density

Remote System Requirements

- IBM 2780/3780 BISYNC protocol as supported by a majority of mainframes and minicomputers including: all IBM-360/370 Systems, PDP-11/70, VAX-11/780, Data General ECLIPSE.
- Users should purchase this standard software package from the remote system vendor and any additional required hardware such as a synchronous communications interface.
- The operating system at the remote must be configured (SYSGEN'ed) with correct options such as line address, 2780 or 3780, . . .

Communication Equipment Requirements

The Intellec Development System may be connected to the remote system using any one of the following methods:

- For short distances (up to 50 feet), use a synchronous modem eliminator (e.g., SPECTRON ME-81FS-2).
- For distances up to four miles, use short haul synchronous modems and telephone lines.
- For distances greater than four miles, use synchronous modems and telephone lines. The following BELL modems or their equivalents are recommended:
 - BELL 201C 2400 bits/second
(half duplex, switched line)
 - BELL 208A 4800 bits/second
(full duplex, leased line)
 - BELL 208B 4800 bits/second
(half duplex, switched line)
 - BELL 209A 9600 bits/second
(full duplex, leased line)
- Modems at either end must be compatible.

ORDERING INFORMATION

Part Number	Description
*MDS-384 Kit	Mainframe Link for Distributed Development

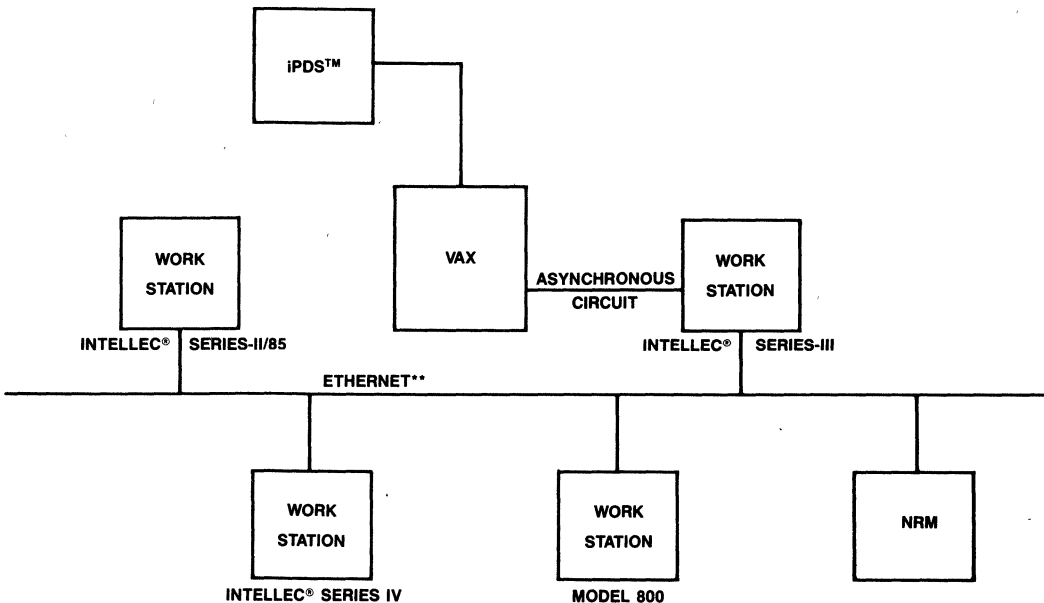
*MDS is an ordering code only and is not used as a product name or trademark.
MDS® is a registered trademark of Mohawk Data Sciences Corporation.



INTEL ASYNCHRONOUS COMMUNICATIONS LINK

- Communications software for VAX* host computer and Intel microcomputer development systems
- Compatible with VAX/VMS* and UNIX† operating systems
- Supports Intel's Model 800, Intellec® Series II, Series III, Series IV and iPDS™ microcomputer development systems
- Supports NDS-II workstations
- Allows development system console to function as a host terminal
- Operates through direct cable connection or over telephone lines
- Software selectable transmission rate from 300 to 9600 baud

Intel's Asynchronous Communications Link (ACL) enables one or more Intel microcomputer development systems to communicate with a Digital Equipment Corporation VAX family computer. The link supports Intel Model 800, Intellec Series II, Series III, Series IV or iPDS™ development systems and NDS-II workstations. Programmers can use the editing and file management tools of the host computer and then download to the Intel microcomputer development system for debugging and execution. Programmers can use their microcomputer development system as a host terminal and control the host directly without changing terminals.



NDS-II Example

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. Information Contained Herein Supercedes Previously Published Specifications On These Devices From Intel.

*VAX and VAX/VMS are trademarks of Digital Equipment Corporation.

†UNIX is a trademark of Bell Laboratories.

**Ethernet is a trademark of Xerox Corp.

FUNCTIONAL DESCRIPTION

The Asynchronous Communications Link (ACL) consists of cooperating programs: one that runs on the host computer, and others that run on each microcomputer development system. The development system programs execute under the ISIS-II or ISIS-III(N), ISIS-IV, ISIS-II(W) or ISIS-PDS operating system. They invoke the companion program on the VAX-11/7XX, which runs under either the VAX/VMS or UNIX operating system.

The link provides three modes of communication: on-line transmission, single-line transmission, and file transfer. In on-line mode, the development system functions as a host terminal, enabling the programmer to develop programs using the host computer's editing, compilation, and file-management tools directly from the development system's console. Later, switching to file transfer mode, text files and object code can be downloaded from the host to the development system for debugging and execution. Alternatively, files can be sent back to the host for editing or storage. In single line mode, the programmer can send single-line commands to the host computer while remaining in the ISIS environment.

The user can select transmission rates over the link from 300 to 9600 baud. The link transmits in encapsulated blocks. The receiver program validates the transmission by checking record-number and checksum information in each block's header. In the event of a transmission error, the receiving program recognizes a bad block and requests the sender to retransmit the correct block. The result is highly reliable data communications.

SOFTWARE PACKAGE

The Asynchronous Communications Link Package contains either a VAX/VMS or UNIX compatible magnetic tape, a single 8", double 8", Series-IV 5 1/4", and PDS 5 1/4" diskette compatible with the Intel development system, and the *Asynchronous Communications Link User's Guide* containing installation, configuration, and operation information.

HARDWARE CONNECTION

The Link sends data over an RS232C cable. The communication line from the host computer connects directly to a development system port.

TELECOMMUNICATIONS USING THE LINK

The ACL is ideal for cross-host program development using a commercial timesharing service. This configuration requires RS232C compatible modems and a telecommunications line. Depending on the anticipated level of usage, wide-area telephone service (WATS), a leased line, or a data communications network may be chosen to keep operating overhead low.

NDS-II ACCESS USING THE LINK

The ACL is ideal for interconnecting VAX host computers with NDS-II. This configuration requires that an NDS-II workstation be connected to the VAX host computer using the RS232C interface and to NDS-II using the Ethernet interface.

All three modes of communication operate identically on NDS-II. In the on-line mode, the development workstation operates as a host terminal, and concurrently, as an NDS-II workstation. It is an easy transition between the VAX and ISIS operating system environments as LOGON/LOGOFF sequences are not required to re-enter environments.

In file transfer mode, text and object files can be transferred from the VAX directly to the Winchester Disk at the NRM without first copying the files to the workstation local floppy disk. Similarly, files residing on the NDS-II Network File System (the Winchester Disk at the NRM) can be transferred directly to the VAX without using local workstation storage.

Using the EXPORT/IMPORT mechanisms of NDS-II, a network workstation which is not directly connected to the VAX can cause files to be transferred between the VAX and NRM. For example, any NDS-II workstation can "EXPORT" ACL commands to another "IM-



INTEL ASYNCHRONOUS COMMUNICATIONS LINK

PORT"ing NDS-II workstation which is physically connected to a VAX. The "IMPORT"ing workstation executes the ACL command file causing the desired action to occur.

VAX ACCESS USING THE LINK

Users who want multiple workstations concurrently

operating as VAX terminals (ONLINE mode) must physically connect each workstation to the VAX. However, users who want multiple workstations to be able to upload/download files, for example, must only physically connect one workstation to the VAX. By using the EXPORT/IMPORT mechanism of NDS-II as described above, the user can have multiple workstations accessing the VAX using only one connection.

SPECIFICATIONS

Software

Asynchronous Communications Link development system programs

VAX/VMS or UNIX companion program

Media

Single- or double-density ISIS 8" and Series-IV, PDS 5¼" compatible diskette

600-ft. 1600 bpi magnetic tape, VAX/VMS or UNIX compatible

Data Transfer Speeds

All systems up to 9600 bps

Online Terminal Mode Speeds

Series II, Series III, Series IV — 2400 bps max

PDS — 9600 bps max

Model 800 — equal to or less than the Terminal speed

Manual

Asynchronous Communications Link User's Guide, Order No. 172174-001

Required Host Configuration

VAX-11/7XX running VAX/VMS (Version 3.2) or fourth Berkeley distribution of UNIX 4.1

Required Intel Development System Configuration

Model 800, Series II, Series III, Series IV, or iPDS under ISIS

Required Connection

RS232C compatible — cable 3M-3349/25 or equivalent; 25-pin connector 3M-3482-1000 or equivalent

Recommended Modems for Telecommunications

300 baud — Bell* 103 modem; VADIC† 3455 modem or equivalent

1200 baud — Bell 202 modem; VADIC 3451 modem or equivalent

9600 baud — Bell 209A (full duplex, leased line) or equivalent

Note: Since one of the two Model 800 ports uses a current loop interface, Model 800 users need a terminal or modem that is current loop compatible, or a current loop/RS232C converter.

The Model 800 might require modification by a qualified hardware technician. Intel does not repair or maintain boards with these changes.

ORDERING INFORMATION

Product Name

Asynchronous Communications Link

Ordering Code‡

iMDX 394 for VAX/VMS systems

iMDX 395 for UNIX systems

*Bell is a trademark of American Telephone and Telegraph.

†VADIC is a trademark of Racal-Vadic Inc.

‡See price book for proper suffixes for options and media selection.



iNA 960 NETWORK SOFTWARE

- **ISO Transport (8073) Class 4 services**
 - Guaranteed message integrity
 - Data rate matching (flow control)
 - Multiple connection capability
 - Variable length messages
 - Expedited delivery
 - Negotiation of virtual circuit characteristics during opens
- **Additional functionality**
 - Connectionless transport (Datagram)
 - External Data Link
- **IEEE 802.3 Data Link protocol (CSMA/CD) supported**
- **Comprehensive Network Management services**
 - Collection of network usage statistics
 - Setting and inspecting of transport and data link parameters
 - Fault isolation and detection
 - Boot Server
- **Compatible with multiple system environments**
 - Runs as an iRMX™ 86 job
 - Supports host operating system independent designs based on 8086, 8088 or 80186 and 82586 components
- **Runs on iSBC® 186/51 COMMputer™ Board**
- **Size configurable to suit specific application requirements**

iNA 960 is a general purpose local area network software package implementing the class 4 services of the ISO transport specification and network management functions in system designs based on the 8086, 8088 and 80186 microprocessors and the 82586 communications co-processor. iNA 960 also supports Intel's board level LAN products, the iSBC® 550 KIT and the iSBC® 186/51. Combined with the iSBC 186/51 COMMputer™ board, iNA 960 offers a high performance, cost effective network solution for MULTIBUS® iRMX™ 86 users. See Figure 1 for iNA 960 functionality and operating environments.

iNA 960 is a ready-to-use software building block for OEM suppliers of networked systems for both technical and commercial applications. Examples for such applications include networked design stations, manufacturing process control, communicating word processors, and financial services workstations. Using the iNA 960 software the OEM can minimize development cost and time while achieving compatibility with a growing number of equipment suppliers adapting the IEEE and ISO standards.

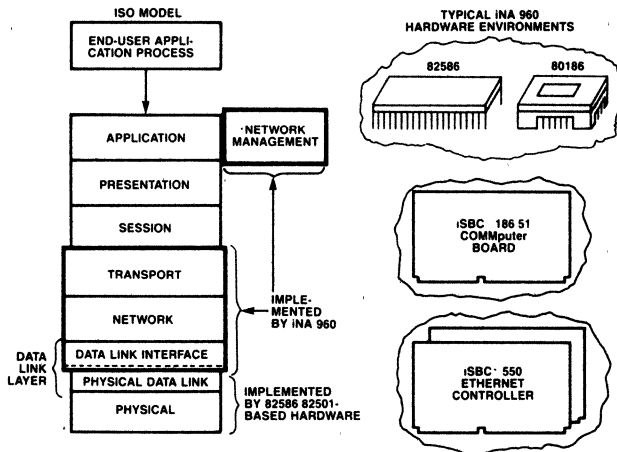


Figure 1.

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. Information Contained Herein Supersedes Previously Published Specifications On These Devices From Intel

FUNCTIONAL OVERVIEW

The iNA 960 design is a standard implementation of the Class 4 transport protocol defined by the ISO OSI model. The Transport Layer provides a reliable full-duplex message delivery service on top of the "best effort" IEEE 802.3 standard packet delivery service implemented by the 82586 (or equivalent) physical and data link functions.

Consisting of linkable modules, the software can be configured to implement a range of capabilities and interface protocols. In addition to reliable process-to-process message delivery, the capabilities include a datagram service, a boot server, a direct user access to the Data Link Layer, and a comprehensive network management facility.

iNA 960 can be configured to run under iRMX 86 along with the user software, or to run on top of a

dedicated 8086, 8088 or 80186 processor coupled with an 82586 to provide a communications front end processor.

The software also includes a Network Management service. This facility enables the user to monitor and adjust the network's operation in order to optimize its performance.

The current release of iNA 960 includes a "null" Network Layer supporting the Data Link and Transport Layers without providing internetwork routing service. This capability will be implemented in later releases of iNA 960.

For a conceptual block diagram of iNA 960, refer to Figure 2.

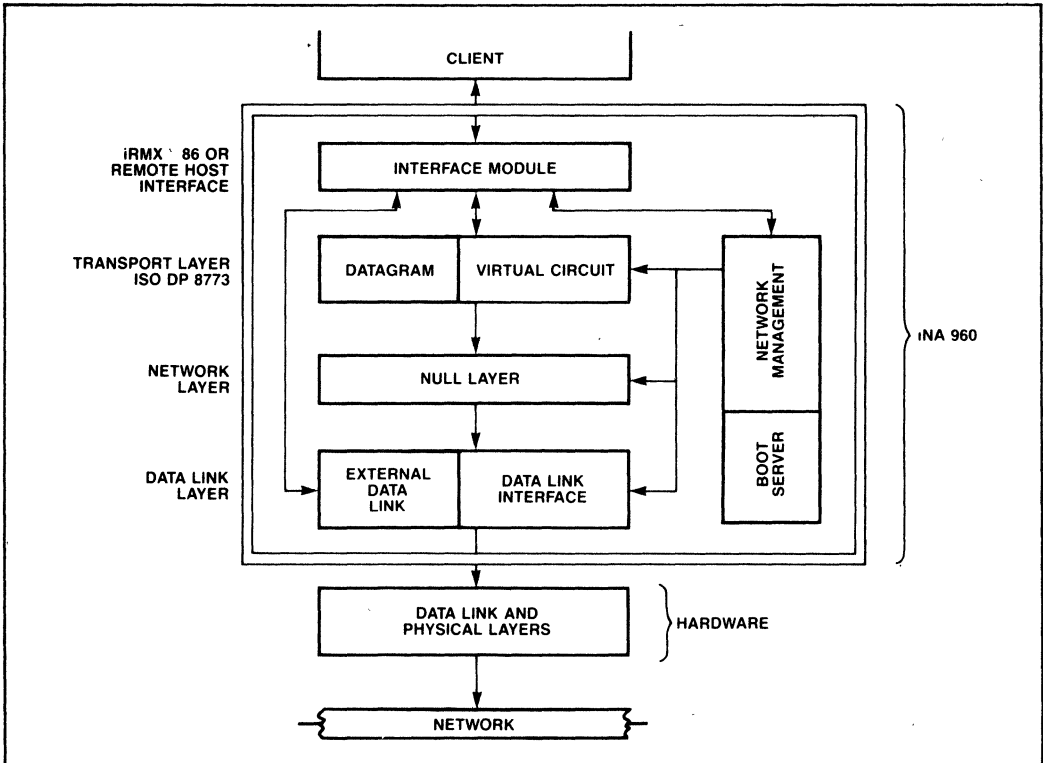


Figure 2. iNA 960 Conceptual Block Diagram

TRANSPORT LAYER

The Transport Layer provides message delivery services between client processes running on computers (network "hosts" or "nodes") anywhere in the network.

Client processes are identified by a combination of a network address defining the node and a transport service access point defining the interface point through which the client accesses the transport services. The combined parameters, called the transport address, are supplied by the user for both the local and the remote client processes to be connected:

The iNA 960 transport layer implements two kinds of message delivery services: virtual circuit and datagram. The virtual circuit provides a reliable point-to-point message delivery service ensuring maximum data integrity, and it is fully compatible with the ISO 8073 Class 4 protocol. The datagram service provides a best effort message delivery between client processes requiring less overhead and therefore allowing higher throughput than virtual circuits.

Both the datagram and the virtual circuit services are optional and can be included when configuring iNA 960.

Virtual Circuit Services

- Reliable Delivery: Data is delivered to the destination in the exact order it was sent by the source, with no errors, duplications or losses, regardless of the quality of service available from the underlying network service.
- Data Rate Matching (flow control): The Transport Layer attempts to maximize throughput while conserving communication subsystem resources by controlling the rate at which messages are sent. That rate is based on the availability of receive buffers at the destination and its own resources.
- Multiple Connection Capability (Process Multiplexing): Several processes can be simultaneously using the Transport Layer with no risk that progress or lack of progress by one process will interfere with others.
- Variable Length Messages: The client software can submit arbitrarily short or long messages for transmittal without regard for the minimum or maximum network service data unit (NSDU) lengths supported by the underlying network services.

- Expedited Delivery (optional). With this service the client can transmit up to 16 bytes of urgent data bypassing the normal flow control. The expedited data is guaranteed to arrive before any normal data submitted afterward.

Connectionless Transport (Datagram) Service

The datagram service transfers data between client processes without establishing a virtual circuit. The service is a "best effort" capability and data may be lost or misordered. Data can be transferred at one time to a single destination or to several destinations (multicast).

NETWORK MANAGEMENT FACILITY (NMF)

The network management facility provides the users of the network with planning, operation, maintenance and initialization services described below.

- Planning: This service captures network usage statistics on the various layers to help plan network expansion. Statistics are maintained by the layers themselves and are made available to users via an interface with the NMF.
- Operation: This service allows the user to monitor network functions and to inspect and adjust network parameters. The goal is to provide the tools for performance optimization on the network.
- Maintenance: This service deals with detecting isolating and correcting network faults. It also provides the capability to determine the presence of hosts and the viability of their connection to the network.
- Initialization: NMF provides initialization and remote loading facilities.

Network management provides distributed management of the network; the user can request any of the services to be performed on a remote as well as a local node. The NMF interfaces to every other network layer both to utilize their services and to access their internal data bases.

In support of the above services, the NMF capabilities include layer management, echo testing, limited debugging facilities, and the ability to down line load and dump a remote system.

Layer management deals with manipulating the internal database of a layer. The elements of these data bases are termed objects. Some examples for objects are the number of collisions, retransmission time-out limit, the number of packets sent, and the list of nodes to boot. NMF can examine and modify objects in a layer's data base.

An echo facility is provided. Using this facility the host can determine if a node is present on the network or not, test the communication path to that node and determine whether the remote node is functional.

NMF enables the user to read or write memory in any host present on the network. This feature is provided as an aid to debugging.

NMF can down line load any system present on the network. A simple Data Link protocol is used to ensure reliability. This facility can be used to load databases, to boot systems without local mass storage or to boot a set of nodes remotely, thus ensuring that they have the same version of software, etc.

Dumping is an operation equivalent to memory read from the user's standpoint; however, dumping uses the Data Link facilities while memory read uses the transport facilities.

EXTERNAL DATA LINK (EDL)

The External Data Link option allows the user to access the functionalities of the Data Link Layer directly instead of having to go through the network and transport layers. This flexibility is useful when the user needs custom higher layer software, or does not need the Network Layer and Transport Layer services (e.g., when sending "best effort" messages, or running customer diagnostics).

Through the EDL the capabilities supporting the lower layers in iNA 960 are made directly available to the user. EDL enables the user to establish and delete data link connections, transmit packets to individual and multiple receivers, and configure the data link software to meet the requirements of the given network environment.

USER ENVIRONMENT

iNA 960 is designed to run on hardware based on the 8086, 8088 or 80186 microprocessors and the 82586 LAN Coprocessor. The software can be configured to run under iRMX 86 or on a dedicated 8086, 8088 or 80186 processor separately from the host. The following section describes these two operating environments.

iRMX Environment

In this configuration, both the user program and iNA 960 are running under iRMX 86. The communications software is implemented as an iRMX 86 job requiring the nucleus only for most operations. The only exception is the boot server option which also needs the Basic I/O System iNA 960 will run in any iRMX environment including configurations based on the 80130. See Figure 3 for an illustration of iNA 960 running under iRMX 86.

Some of the typical hardware implementations include the iSBC 550 KIT combined with an 8086, 8088 or 80186 based host or the iSBC 186/51 COMMputer™ board integrating the host processor and the communications controller into a single, high performance MULTIBUS board. See Figure 4A and 4B for a conceptual block diagram of these configurations.

Operating System/Processor Independent Implementation

In those systems where iRMX 86 is not the primary operating system, where off-loading the host of the communications tasks is necessary for performance reasons, or where an existing communications front-end processor configuration is being upgraded, the user may wish to dedicate a processor for communications purposes. iNA 960 can be configured to support such implementations by providing network services on an 8086, 8088 or 80186 processor. Figure 5 depicts the conceptual block diagram of this configuration.

This approach provides the component and system designer with an ISO standard communications software building block that can be adapted to his system's needs with a minimum interfacing effort. For added flexibility, iNA 960 provides the user with the alternative of using the included interface module or writing his own module, if necessary.

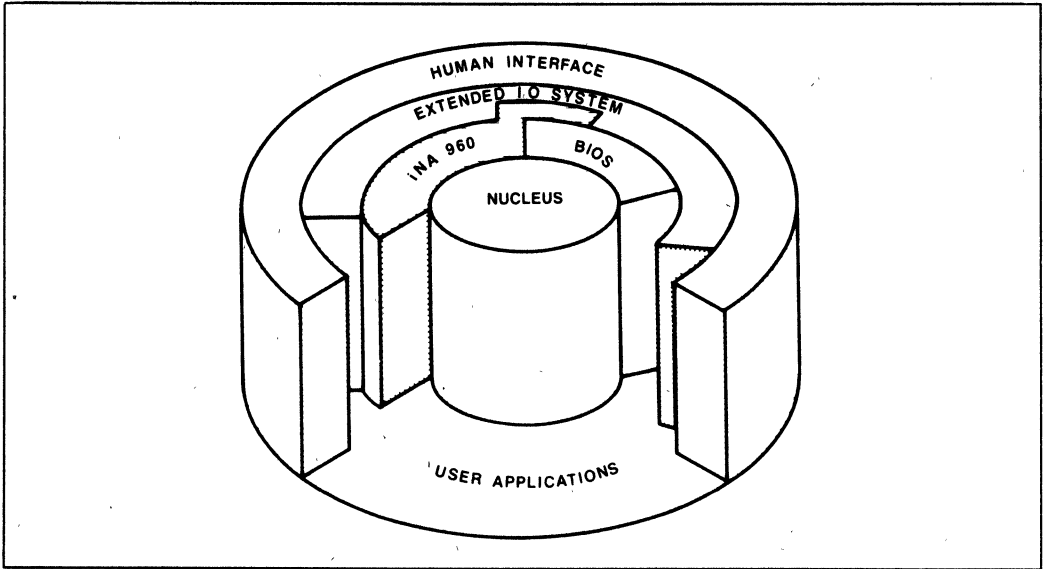


Figure 3. As an iRMX™ job, iNA 960 uses nucleus calls and, when the Boot Server is present, BIOS calls.

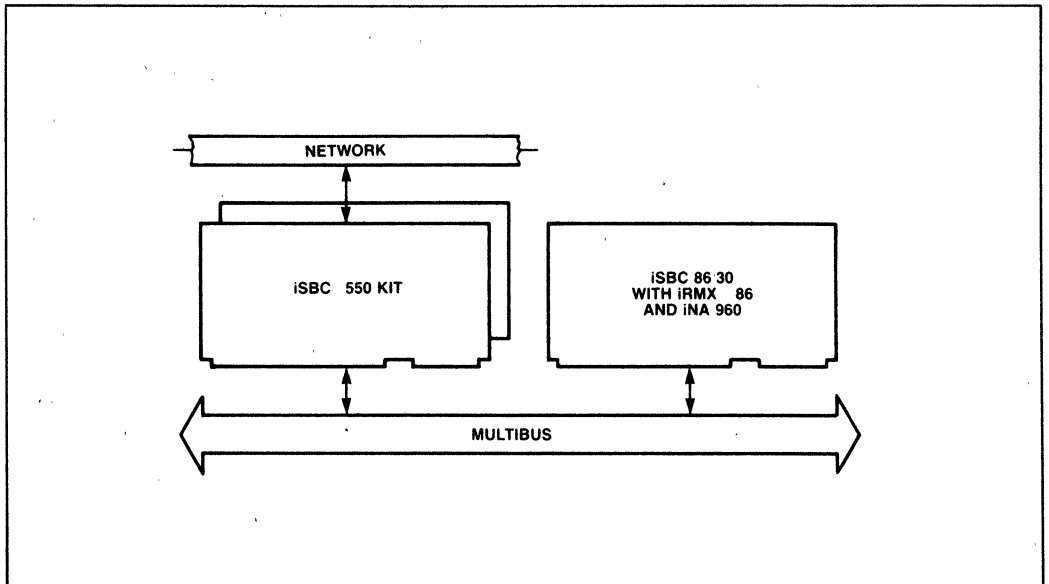


Figure 4A. Typical configuration using iSBC® 550 kit, iSBC® 86/30, iRMX 86™ and iNA 960.

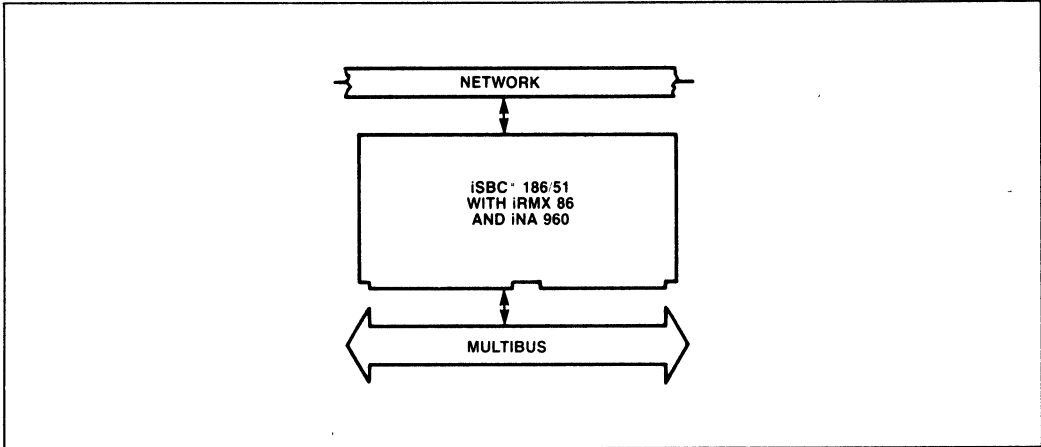


Figure 4B. Configuration using iSBC® 186/51, iRMX 86 and iNA 960.

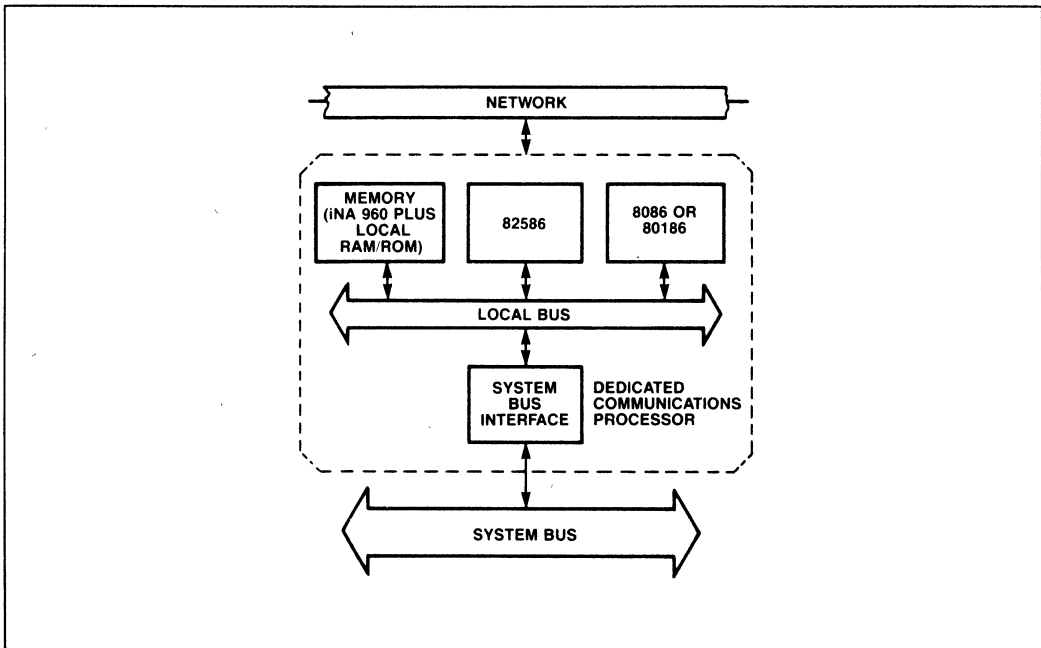


Figure 5. In the operating system/processor independent implementation iNA 960 is running on a dedicated 8086, 8088 or 80186 processor.

USER INTERFACE

iNA 960 is designed to run both under iRMX 86 and on a dedicated communications front end processor separately from the host. In both environments, the interface is based on exchanging memory segments called request blocks between iNA 960 and the client. The format and contents of the request blocks remain the same in both configurations; only the request block delivery mechanism changes. See Figure 6 for a simplified interface diagram.

Request blocks are memory segments containing the data to be passed from the user to iNA 960 (commands), or from iNA 960 to the user (responses). The iNA 960 request blocks consist of fixed format fields identical across all user commands and argument fields unique to the individual commands. Refer to Figure 7 for the standard request block format.

Issuing an iNA 960 command consists of filling in the request block fields and transferring the block to iNA 960 for execution. After processing the command, iNA 960 returns the request block with one of the pre-defined response codes placed in the response code field of the request block. The response code indicates whether the command was executed successfully or whether an error occurred. By examining the response code, the user can take appropriate action for that command.

For iRMX users, iNA 960 also provides a procedural interface option to simplify writing the application software interface. In this case, the allocation and formatting of request blocks are replaced by a procedure call with parameters that specify the user's command options. The procedure execution will create a request block and fill in the appropriate fields from the user's parameter list.

For component users the request block delivery mechanism is the means by which the host processor and the communications processor running iNA 960 software exchange the request blocks. iNA 960 provides three such mechanisms: the MIP (Multibus Inter-process Protocol), the BCB (Base Control Block) and a user-defined mechanism. The MIP interface is included for use in systems already supporting this protocol; the BCB is a simple interface for single host environments, and the user-defined interface accommodates unique application requirements.

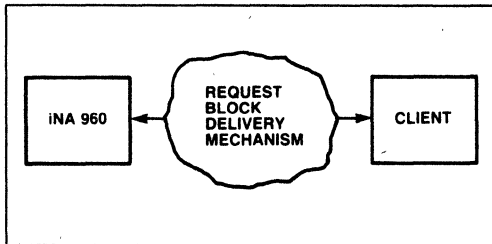


Figure 6.

<u>FIELDS</u>	<u>WORD/BYTE</u>	
Reserved (2)	WORD	} FIXED FORMAT FIELDS (same for all commands)
Length	BYTE	
User I.D.	WORD	
Response Port	BYTE	
Return Mailbox Token	WORD	
Segment Token	WORD	
Subsystem	BYTE	
Opcode	BYTE	
Response Code	WORD	
Arguments	BYTE	
.	.	
.	.	
.	.	

Figure 7. iNA 960 Request Block Format

Transport Layer User Interface

The following table summarizes the user commands and the corresponding transport layer responses

Command	Function
1. OPEN	Allocates memory for the connection data base of a virtual circuit (or connection) to be established. The connection database contains data concerning the connection
2. SEND CONNECT REQUEST	Requests connection to a fully specified remote transport address using specified ISO connection negotiation options
3. AWAIT CONNECT REQUEST TRAN	Indicates that the transport client is willing to consider incoming connection requests based on pre-established acceptance criteria
4. AWAIT CONNECT REQUEST USER	Indicates that the transport client is willing to consider incoming connection requests. If the request meets the address and negotiation option criteria, it is passed to the client for further consideration
5. ACCEPT CONNECT REQUEST	Indicates that the connection requested by a remote transport service is accepted by the client
6. SEND DATA or SEND EOM DATA	With this command, the client requests the transmission of the data in the buffers using the normal delivery service of the specified connection. The SEND EOM DATA command signals that the end of the data marks the end of the transport service data unit
7. RECEIVE DATA	Posts normal receive data buffers for a specific connection or for a buffer pool used by a class of connections
8. SEND EXPEDITED DATA	Transmits up to 16 bytes of data using the expedited delivery service. The expedited data is guaranteed to arrive at the destination before any normal data submitted afterward.
9. RECEIVE EXPEDITED DATA	Posts receive data buffers for expedited delivery for a specific connection or for a pool of buffers used by a class of connections
10. CLOSE	Terminates an existing connection or rejects an incoming connection request. Any normal or expedited data queued up to be sent will not be sent
11. AWAIT CLOSE	Requests notification of the client of the termination of a specified connection
12. SEND DATAGRAM	Requests transmission of the data in the buffers using the transport datagram service.
13. RECEIVE DATAGRAM	Posts a receive buffer for a specific receiver or a class of receivers to receive data from a transport datagram.

Network Management Layer User Interface

Command	Function
1. READ OBJECT	Returns the value of the specified object to the client.
2. SET OBJECT	Sets the value of an object as specified by the client.
3. READ AND CLEAR OBJECT	Returns the value of the specified object to the client then clears the object.
4. ECHO	This function is used to determine the presence of a node, to test the communication path to the node and to ascertain the viability and functionality of the remote host addressed.
5. UP LINE DUMP	Requests a remote node to dump a specified memory area.
6. READ MEMORY	Reads memory of the specified network node.
7. SET MEMORY	Sets memory of the specified network node.
8. FORCE LOAD	Causes a node to attempt a remote load from another node.

External Data Link Interface

Command	Function
1. CONNECT	With this command the client establishes a data link connection.
2. DISCONNECT	Eliminates a previously established connection.
3. TRANSMIT	Transmits data contained in buffers specified by the client.
4. POST RECEIVE PACKET DESCRIPTOR	Allocates memory for maintaining records on receive data buffers. Also may be used to allocate memory for buffering receive data.
5. POST RECEIVE BUFFER	Allocates memory for buffering receive data.
6. ADD MULTICAST ADDRESS	Adds an address to the list of data link multicast addresses.
7. REMOVE MULTICAST ADDRESS	Removes an address from the list of data link multicast addresses.
8. SET DATA LINK I.D.	Sets up a unique data link I.D. for the station.

CONFIGURING INA 960

In order to adapt iNA 960 to his specific application, the user must configure the software to define the desired functions, to select the appropriate interface, to set the layer parameters and to set up for the required hardware configuration.

There are a number of capability combinations the user may elect to implement in his application. At the transport layer level the options are: virtual circuit service with or without expedited delivery, or datagram service, or both. At the data link level, the user may include or exclude the External Data Link interface.

The Network Management Facility is also optional.

When it is configured in, the user may also include the boot server module. These capabilities can be made available simply by linking in the corresponding software modules. The interface options are also implemented in a modular fashion; the user links in the desired module to set up for the iRMX 86 or the operating system independent configurations.

Layer parameters and configuration options are first edited into layer configuration files, then assembled and linked into iNA 960. Layer parameters adjust the network's operation to match the usage pattern and the available resources. For example, within the Transport Layer, the flow control parameters, the retransmission timer parameters, the transport data base parameters, etc. can be set via this process.

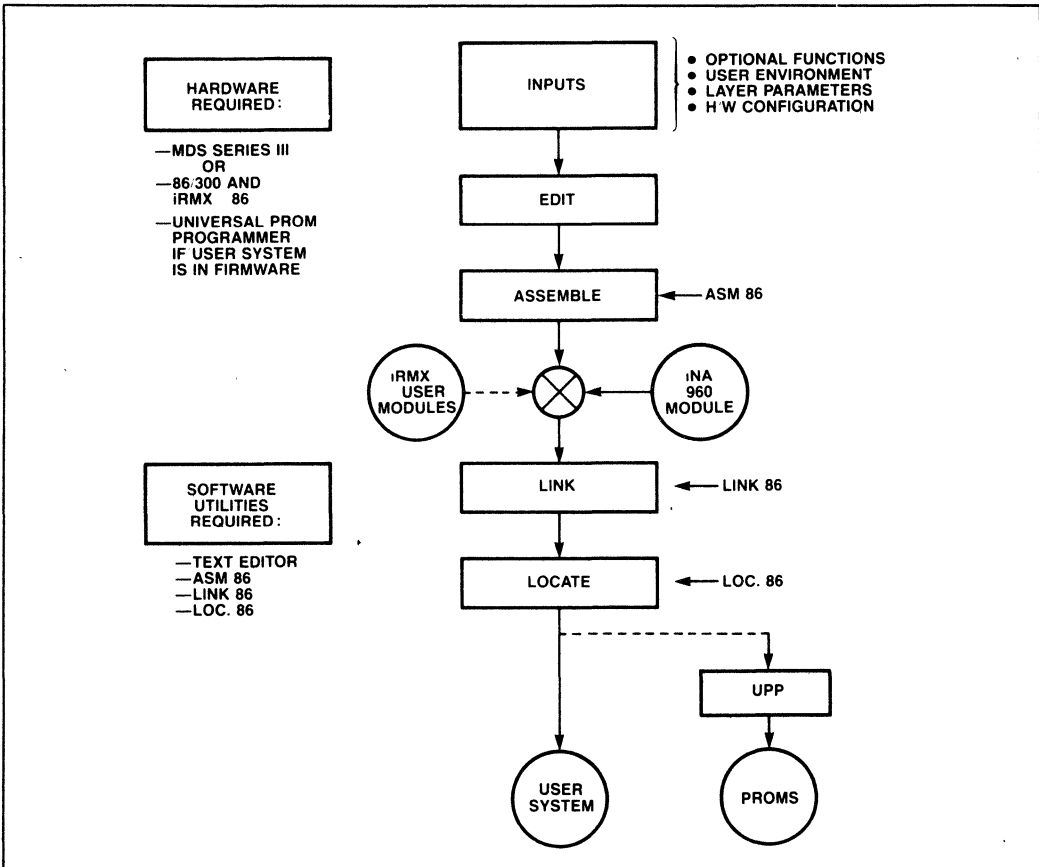


Figure 8. The Configuration Process for iNA 960

The user also sets up for the required hardware configuration, such as port addresses and interrupt levels, during this process. For the flow diagram of configuring iNA 960, refer to Figure 8.

SPECIFICATIONS

Hardware Supported:

- iSBC 186/51 Communicating Computer.
- iSBC 550 KIT Ethernet controller board(s) configured to run with iSBC 86/30 or iSBC 86/12B Multi-bus processor boards.
- Custom designs based on 8086, 8088 and 80186 microprocessors and the 82586 Local Communications Controller.

Typical Throughput at transport:

Environments:	
186/51 and iRMX 86	50K to 200K bytes/sec
Dedicated 80186/82586 COMMengine	100K to 300K bytes/sec

Memory Requirements: (in bytes)

Base System	12K plus configurable Buffer Memory
Normal Virtual Circuit Option	18K plus configurable Buffer Memory
Expedited Delivery Option	2K
Datagram Option	3K plus Data Base Memory
Net Management Option	1K to 5K
External Data Link Option	5K
Boot Server Option	5K

Available Literature/Reference Materials:

- iNA 960 Programmer's Reference Manual (11/83)
- iSBC 186/51 Data Sheet (Now)
- iSBC 186/51 Hardware Reference Manual (11/83)

Ordering Information

The following is a list of ordering options for the iNA 960 Network Software. All options include a full year of update service that provides a periodic NEWSLETTER, Software Problem Report Service, and copies of system updates that occur during this period. All of the object code options listed are available on either ISIS or RMX compatible double density diskettes.

As with all Intel software, purchase of any of these options requires the execution of a standard Intel Master Software License. The specific rights granted to users depend on the specific option and the License signed.

Order Code	Description
iNA 960 YRO	OEM object code license requiring the payment of incorporation fees for each derivative work based on iNA 960; ISIS and RMX formatted diskettes
iNA 960 YST	Object code license to use the product at a second site or facility; ISIS and RMX formatted diskettes
iNA 960 YBY	Object code buy-out license requiring no further payment of incorporation fees; ISIS and RMS formatted diskettes
iNA 960 YSU	Object code single use license only; ISIS and RMS formatted diskettes
iNA 960 ESR	License for machine readable source code of iNA 960. RMX formulated diskettes.
iNA 960 LST	Source listing of iNA 960 provided on microfiche under a special source code license agreement
iNA 960 RF	Order code for the payment of incorporation fees



NDS-II ELECTRONIC MAIL

- Improves Project Coordination and Communication
- Minimizes "Phone Tag" and Excess Paperwork
- Users Can Send and Receive Text or Object Files
- MAIL Operates Either Interactively or in Command-Tail Format
- User, Group, and "Bulletin Board" Mailboxes Can Be Created
- Operates on any Workstation in the NDS-II Development Environment

Electronic Mail enables users to send and receive messages and files between any nodes on the NDS-II network. In doing so, Electronic Mail improves the communication and coordination between members, reduces "phone tag" and paper generation, aids project configuration management by enabling simplified file transfers, and increases flexibility in workstation location.

The Mail system is governed by an Electronic Mail directory which contains user, group, and bulletin board mailboxes. Each NDS-II user has a mailbox which is only accessible to that user. Group mailboxes are accessible by a defined group of users, and bulletin board mailboxes are accessible by all users. Both group and bulletin board mailboxes can be easily created by any system users.

Users can send a message to any of the mailbox types listed above. Messages can consist of text generated when Mail is invoked, or a text or object file. Options available when sending mail include using a subject string to categorize a message, specifying a message expiration date and time, delaying message delivery until a specific date and time, marking the message URGENT, and maintaining a log of all messages sent.

Users can interactively read their mail and perform the following operations: print messages on their workstation console, delete messages from a mailbox, save messages in a file, forward messages to other users, and reply to message senders. In addition, users can request a mailbox summary which includes, for each message, the sender's name, date sent, subject, urgency, code type (text or object), and message number.

NDS-II Electronic Mail executes on all existing NDS-II workstations using either the iNDX or ISIS-III(N)/ISIS-III(C) operating systems.

TYPICAL MAIL USAGE
<ul style="list-style-type: none">• DISTRIBUTE SUPERUSER MESSAGES• CREATE AND SEND INTERNAL MEMOS• COLLECT PROJECT MILESTONE DATA• REPORT PROGRAM BUGS AND RECOMMEND SYSTEM CHANGES• SEND SOURCE AND OBJECT FILES• USE AS TELEPHONE MESSAGE CENTER



TYPICAL MAIL BENEFITS
<ul style="list-style-type: none">• IMPROVE TEAM COMMUNICATION AND COORDINATION• REDUCE PHONE TAG• MINIMIZE PAPER GENERATION• AID PROJECT CONFIGURATION MANAGEMENT• INCREASE WORKSTATION LOCATION FLEXIBILITY• OVERALL, BOOST DEVELOPMENT TEAM PRODUCTIVITY

NDS-II ELECTRONIC MAIL

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. Information Contained Herein Supercedes Previously Published Specifications On These Devices From Intel.

OPERATING ENVIRONMENT

Required Hardware

NDS-II Environment with any 8- or 16-bit Microcomputer Development System Workstation

Required Software

iNDX or ISIS-III(N)ISIS-III(C) System Software

DOCUMENTATION

"NDS-II Electronic Mail User's Guide"
(122146)

SOFTWARE SUPPORT

This product includes a 90-day initial support consisting of new software releases, updates, subscription services (software performance reports and technical reports), and telephone hotline support. Additional software support services are available separately.

ORDERING INFORMATION

Product Code	Description
iMDX-337	NDS-II Electronic Mail



iNA955 iRMX™ NDS-II LINK

- Transfers files between iRMX™86-based systems and the NDS-II NRM
- Supports fast and reliable download into iRMX™86 target system
- Supports Intel's 86/310, 86/330A, 86/380 systems
- Configurable at nucleus level with iRMX™86 operating system
- Operates through Ethernet communications controller and cable connected to NDS-II
- Utilizes Ethernet technology with data transmission speeds at 10M bits per second

The iNA955/iRMX™ NDS-II LINK is a software package that allows an iRMX based system 86/310, 86/330A, or 86/380 system to be connected to an Intel Network Development System (NDS-II) network via an Ethernet coaxial cable or Intellink™ module.

iRMX system developers can use the Series II, III, IV and Model 800 for editing, compilation and debugging to develop, store, and manage software programs at the Network Resource manager. Using iNA955 these developers can download programs at Ethernet speeds from the Network Resource Manager into their target iRMX hosts for execution and system integration.

System developers can also use the iNA955 programmatic interface to develop their own application programs which run in the iRMX environment and interface with the NRM. This is a way for OEM developers to customize the operating environment to suit their own application.

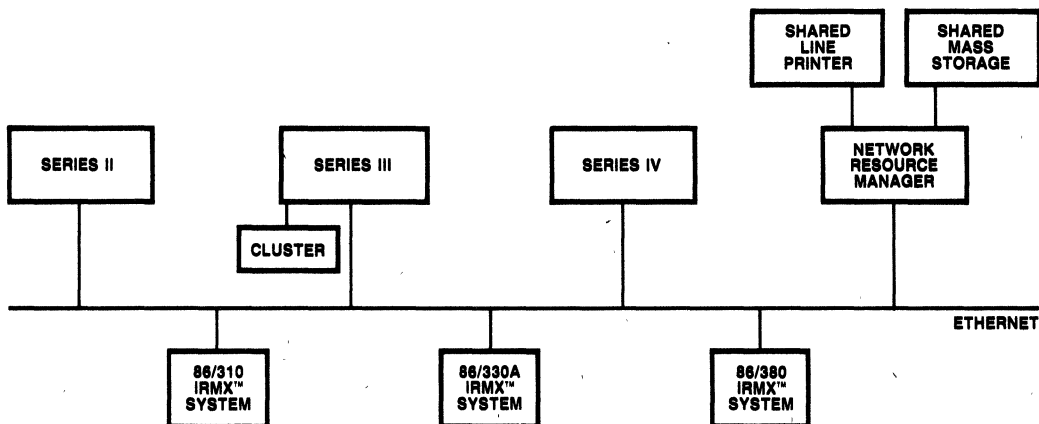


Figure 1. Example of NDS-II Configuration using the iRMX™ NDS-II LINK

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. Information Contained Herein Supersedes Previously Published Specifications On These Devices From Intel.

NDS-II OVERVIEW

The NDS-II is a distributed processing local area network optimized for development of microcomputer-based products. It addresses the needs of both software and hardware engineers by providing the base environment for shared development tools plus the capacity for expansion.

An NDS-II network consists of an NRM which serves as the file server for a variety of Intel's development systems. These development systems include Series II, Series III, Series IV, and Model 800. By configuring iNA955 into an iRMX 86 system, an iRMX system can also be served by the NRM.

NDS-II's Network Resource Manager (NRM) manages all workstation requests for network resources. NRM tasks include service of workstation file requests, printer spooling, management of the distributed Hierarchical File System, the Distributed Job Control System and network maintenance functions such as user-name creation, file archival and system generation.

iNA955 provides a basic upload/download file transfer capability between an iRMX 86 system and the NRM. When used with an iSBC® 550 Ethernet controller, iNA955 allows users at iRMX 86 systems to move files between iRMX systems and the NRM, list directories at the NRM, delete or rename files at the NRM and copy files between two directories on the same NRM.

Access to files is accomplished using two interfaces:

- A A CUSP interface which operates on the network file system in a manner similar to iRMX CUSPS which operate on local iRMX files under a full iRMX operating system.
- B A programmatic interface which allows user programs running with a iRMX nucleus to access files at the NRM. These interfaces are similar to those present in UDI and EIOS.

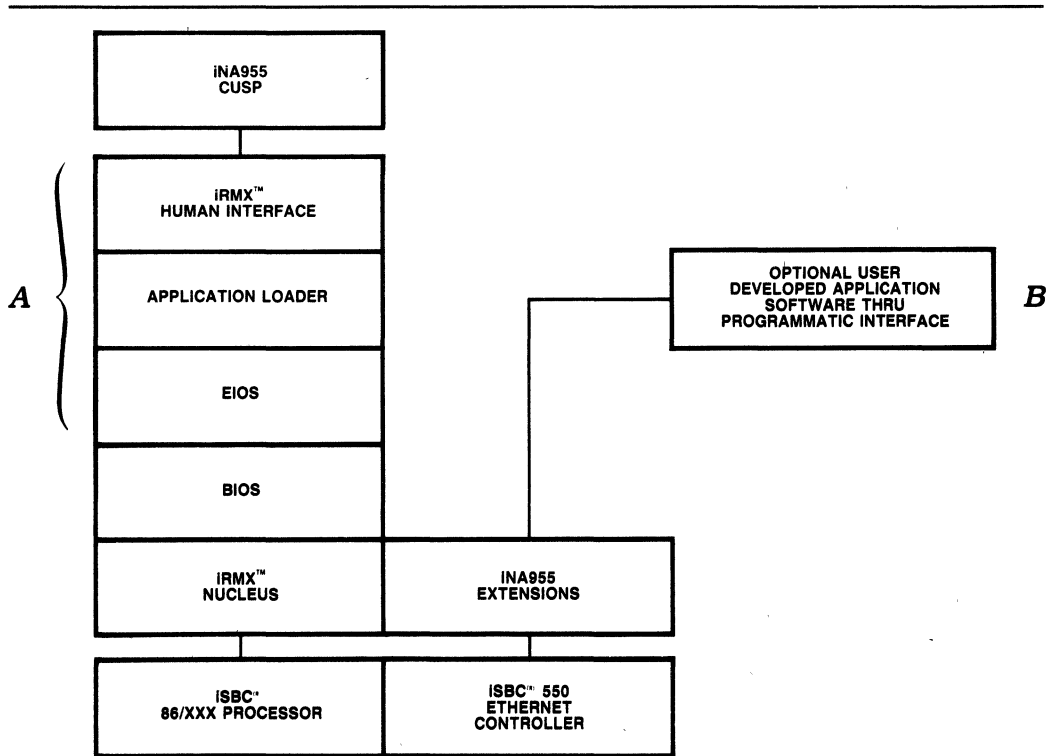


Figure 2. iNA955 Functional Diagram

FUNCTIONAL DESCRIPTION

iNA955/iRMX NDS-II LINK consists of a program which runs on the system 86/3XX family of host computers. iNA955 executes under the iRMX 86 operating system and uses the local iRMX file system.

The iRMX-based host computers communicate with the NRM via iNA (Intel Network Architecture) which

is based upon Ethernet communication protocols. These protocols are supplied by the iSBC550 board set which must be included in the iRMX host system since iNA955 uses the iSBC550 to communicate over Ethernet.

The following tables summarize the user commands and programmatic calls with their descriptions.

User Interface Commands	Function
NACCESS NCREATE NDELETE NDIR NLOGOFF NLOGON NRCOPY NNCOPY RNCOPY NRENAME	examines/changes NRM file access rights creates NRM directory deletes NRM file examines NRM directory logs off from NRM logs on to NRM copies file from NRM to iRMX station copies NRM file to NRM file on the same NRM copies files from iRMX station to NRM renames NRM file or directory

Programmatic Calls	Function
NQ\$CHANGE\$ACCESS NQ\$CREATE\$DIR NQ\$DELETE NQ\$FILE\$INFO NQ\$GET\$VIRTUAL\$ROOT NQ\$LOGOFF NQ\$LOGON NQ\$OPEN NQ\$READ NQ\$READ\$DIR\$ENTRY\$EXP NQ\$RENAME NQ\$WRITE	change access of file on the NRM create directory on the NRM delete file on the NRM get information of file on the NRM get names of volumes at NRM accessible to user logoff user from the NRM logon user to the NRM open file at the NRM read contents of file at the NRM read expanded directory entry at the NRM rename file at the NRM write file to the NRM

Configuring iNA955

Like other iRMX systems iNA955 must be configured according to the system environment. To assist you in configuring your system, iNA955 comes with a configuration template. The file containing this template is contained on the release diskette. This template is designed to be self-explanatory.

The user has the option of integrating into his applications the iNA955 CUSPS. iNA955 CUSPS require the iRMX Human Interface to execute.

Physical Connections

The physical Ethernet connections can be made either through an "Intellink"TM module or through transceivers and the Ethernet cable. The Intellink module serves as an Ethernet local station concentrator. It allows workstations to be located up to 50 meters from the Intellink module and has 9 ports for connecting the NRM and workstations, and one port for connecting an Ethernet cable or other Intellink modules.

SPECIFICATIONS
Operating Environment
HARDWARE SUPPORTED

- System 86/310
- System 86/330A
- System 86/380

HARDWARE REQUIRED

- iSBC[™] 550 Ethernet Communication Controller Set

SOFTWARE REQUIRED

- iRMX[™]86 Operating System version 5.0
- NDS-II System software Release 2.5 or greater

Software Supplied
MEDIA

One 8 inch, single sided, double density iRMX[™]86 format diskette

One 5¼ inch, single sided, double density iRMX[™]86 format diskette

PROGRAMS

- iRMX/NDS-II LINK software linked into iRMX system library
- Examples of iRMX Integration Configuration utilities
- iSBC550 Diagnostics

DOCUMENTATION

- iNA955/iRMX NDS-II LINK Installation and User's Guide, Order Number 12256-001
- Complete NRM and Network operating manuals are included with the NDS-II systems
- iSBC550 Ethernet communications controller Hardware Reference Manual 121746.

ORDERING INFORMATION
Part Number
Description

iNA 955	iRMX/NDS-II Link
iSBC 550	Ethernet Communication Controller Set
iMDX 457	10 meter transceiver cable
iMDX 458	50 meter transceiver cable
iDCM 911-1	Intellink Module
iMDX 3015	Ethernet transceiver kit
iDMX 3016-1	25 meter Ethernet coaxial assembly
iMDX 3016-2	100 meter Ethernet coaxial assembly

Installation

On-site installation is included with the NDS-II Network Resource Manager. iNA955 is customer installable.

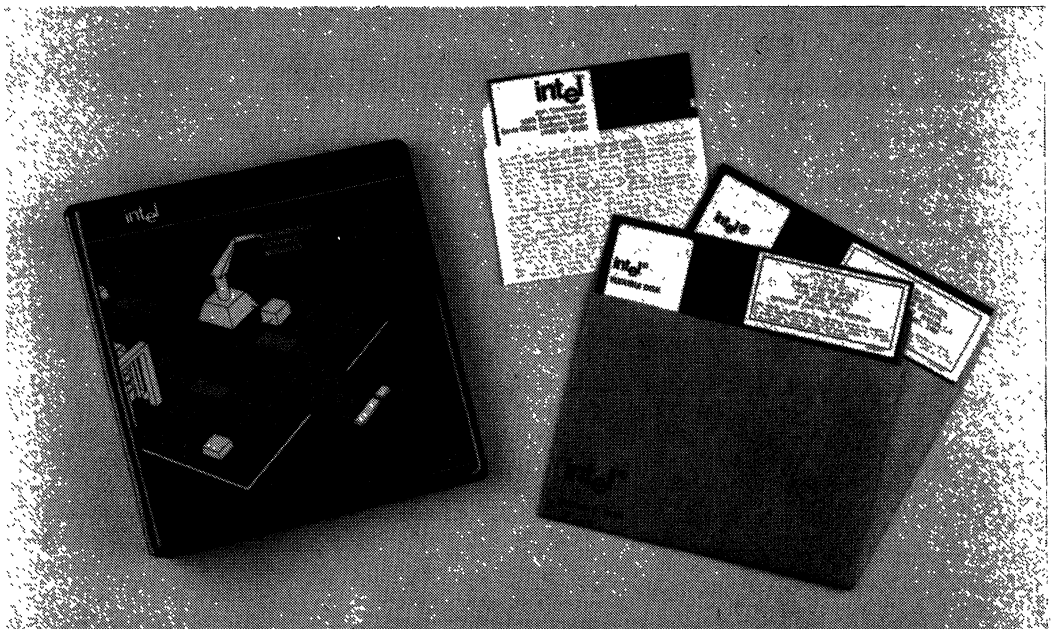


iRMX™ 510 iDCM SUPPORT PACKAGE

- Low cost remote communication/control expansion for MULTIBUS® based systems
- Extends functionality of BITBUS™/iDCM systems
- Software development support for BITBUS™/iDCM products: iSBX™ 344 and iRCB 44/10 boards
- Simple software interface for iRMX™ 86, 286, 88, and iPDS™ ISIS operating system compatibility

The iRMX™ 510 iDCM Support Package contains the necessary software tools to interface MULTIBUS®, and iPDS™ ISIS systems to BITBUS™ systems in both a development environment and during run-time. With other members of the Distributed Control Modules family, the iRMX 510 iDCM Support Package expands Intel's OEM Microcomputer Systems capabilities to include distributed real-time control.

The iRMX 510 Package software interface handlers and the iSBX™ 344 BITBUS Controller MULTI-MODULE™ board extend the capabilities of other microprocessors such as the 8086, 80186, or 80286 in iDCM, MULTIBUS, or iPDS systems. Support of iRMX 51 applications is provided via the iRMX 51 libraries incorporated in the iRMX 510 Support Package. Also, the Support Package completes the development environment for BITBUS/iDCM products: iSBX 344 and iRCB 44/10 boards. When used with an ICE-44 Emulator the iDCM controller is accurately simulated resulting in a highly effective product development effort.



MULTIBUS®, iPDS,™ and iDCM SYSTEM EXPANSION

The iRMX 510 Support Package provides the software interface between Intel's MULTIBUS and iPDS environment, and the BITBUS environment. With Intel's Distributed Control Modules hardware interface, the iSBX 344 MULTIMODULE board, this capability enables the user to expand the existing functionality of an iRMX-based SYSTEM 310, for example, to include control and monitoring of a material handling operation. Intel's Personal Development System (iPDS) can be used as a central supervisory station for data acquisition in a laboratory or for program development. The iRMX 510 iDCM Support Package provides a general purpose interface. For custom applications, users may wish to develop a custom interface.

OPERATING ENVIRONMENT

The iRMX 510 Support Package is supplied on diskettes formatted for iRMX, Intellec® Series II or III and iPDS ISIS development systems. Application programs or tasks residing on an extension in the iDCM environment may use the iRMX 510 interface. (Application programs or tasks are written in iRMX 88, 86 or ISIS compatible code.) Some examples of extensions in an iDCM system are the iSBC 86/05, 88/25, 186/03 boards and the iPDS system. Figure 2 shows how the iRMX 510 interface is integrated into an iDCM system.

For iRMX 86, 88, or 286R-based systems, configuration of the iRMX 510 interface requires two steps: configuring the interface to the hardware and then the supporting executive. Hardware configuration requires creating a file of configuration parameters, compiling it, and linking the result with the application program. When using the iRMX 510 Package with the iPDS ISIS system, hardware configuration is not required.

ARCHITECTURE

The major functional blocks of the iRMX 510 Support Package are: iRMX 86, 286R, 88 and iPDS ISIS parallel interface handlers, iDCM Controller firmware files, and iRMX 51 include files.

Simple Parallel Interface Handlers

The iRMX 510 Support Package includes parallel interface handlers for systems using the iRMX 86 or 286R Operating System, the iRMX 88 Executive, or Intel's Personal Development System ISIS Operating System. These software handlers pass iRMX 51 messages to and from the iSBX 344 parallel interface (Byte FIFO). In iRMX 86, 286R or 88 — based systems, the interface executes as two tasks: one to transmit, the other to receive the message. In iPDS systems the interface is a procedural call: DCM TRANSMIT, DCM RECIEVE, or DCM STATUS CHECK. In both cases the handlers are straightforward and easy to use. Figure 1 illustrates transmission of a message in an iRMX-based system.

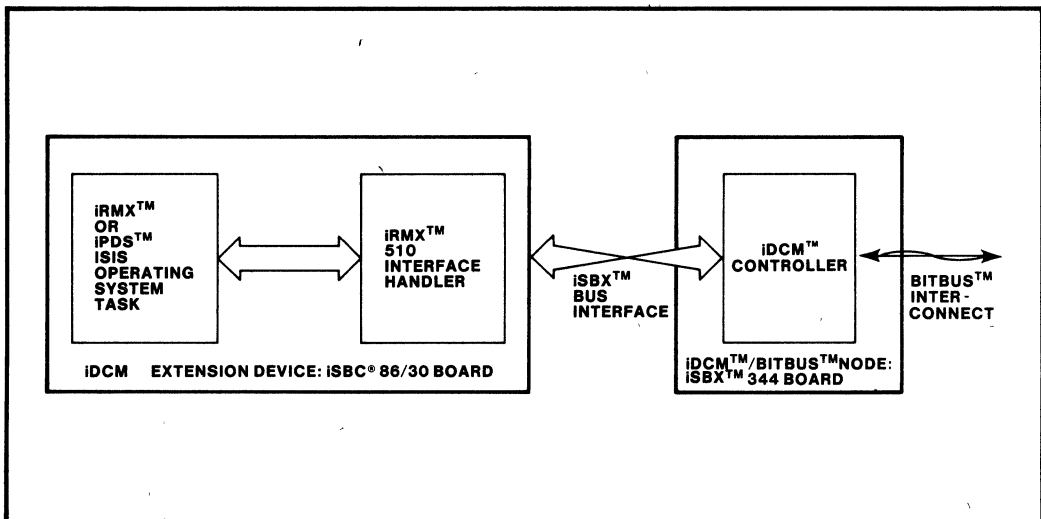


Figure 1. Message Transfer to an iDCM System

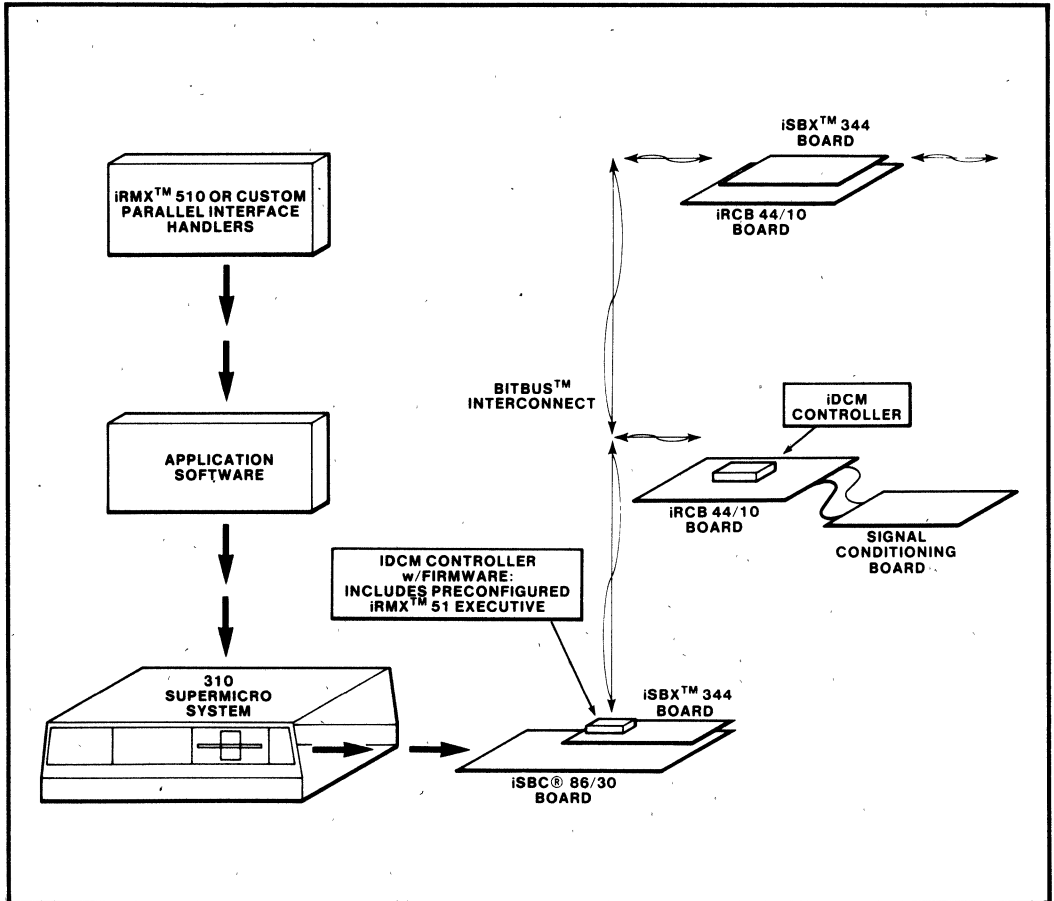


Figure 2. iDCM Operating Environment

The software handlers ease integration of other processors into an iDCM system and provide the tools to quickly expand a MULTIBUS system, or an iPDS ISIS system. Significant reduction in application system software development time results, with more effort concentrated on the overall application.

iDCM Controller Firmware

Also included in the iRMX 510 Support Package is the iDCM Controller firmware in loadable object files, iRMX 51 libraries, and iDCM Controller Include files. An Intellec Development System and ICE-44 Emulator can be used with the loadable object files to accurately simulate the iDCM Controller. This capability significantly decreases development effort by reducing trial and error

production of application system software. The iRMX 51 Interface Library and iDCM Controller Include files allow development of user code for iDCM systems.

DEVELOPMENT ENVIRONMENT

The iRMX 510 Support Package completes the development environment for iDCM application system development when used with an Intellec Series II or III Development System and In-Circuit Emulator (ICE-44), or an iPDS system EMV-440 and the 8051 Software Development Package. As part of Intel's complete development environment for the 8051 family of microcontrollers, the iRMX 510 Support Package may also be used with an iPDS system and EMV-51 or an Intellec Series II or III Development System and an ICE-51 Emulator.

SPECIFICATIONS
Supported Hardware/Software for iDCM Systems

Operating System	Supported Extension*
iRMX 86 Release 5.0	iSBC 86/05, 86/14, 86/30, 186/03, 186/51, 188/48, 88/25, 88/45 boards
iRMX 88 Release 3.0	iSBC 86/05, 86/14, 86/30, 186/03, 186/51, 188/48, 88/25, 88/45 boards
iRMX 286R	iSBC 286/10 board
ISIS Release 1.0 (PDS)	iPDS System

*Each extension device uses an iSBX 344 BIT-BUS Controller MULTIMODULE Board

Supported Hardware — 8051 Microcontroller Family

8051	80C51
8052	8044
8751	8744
8031	80C31
8032	8344

Compatible Software

iRMX 86 Release 5.0
 iRMX 286R
 iRMX 88 Release 3.0
 iPDS ISIS Release 1.0
 iRMX 51 Release 1.0

Development Tools

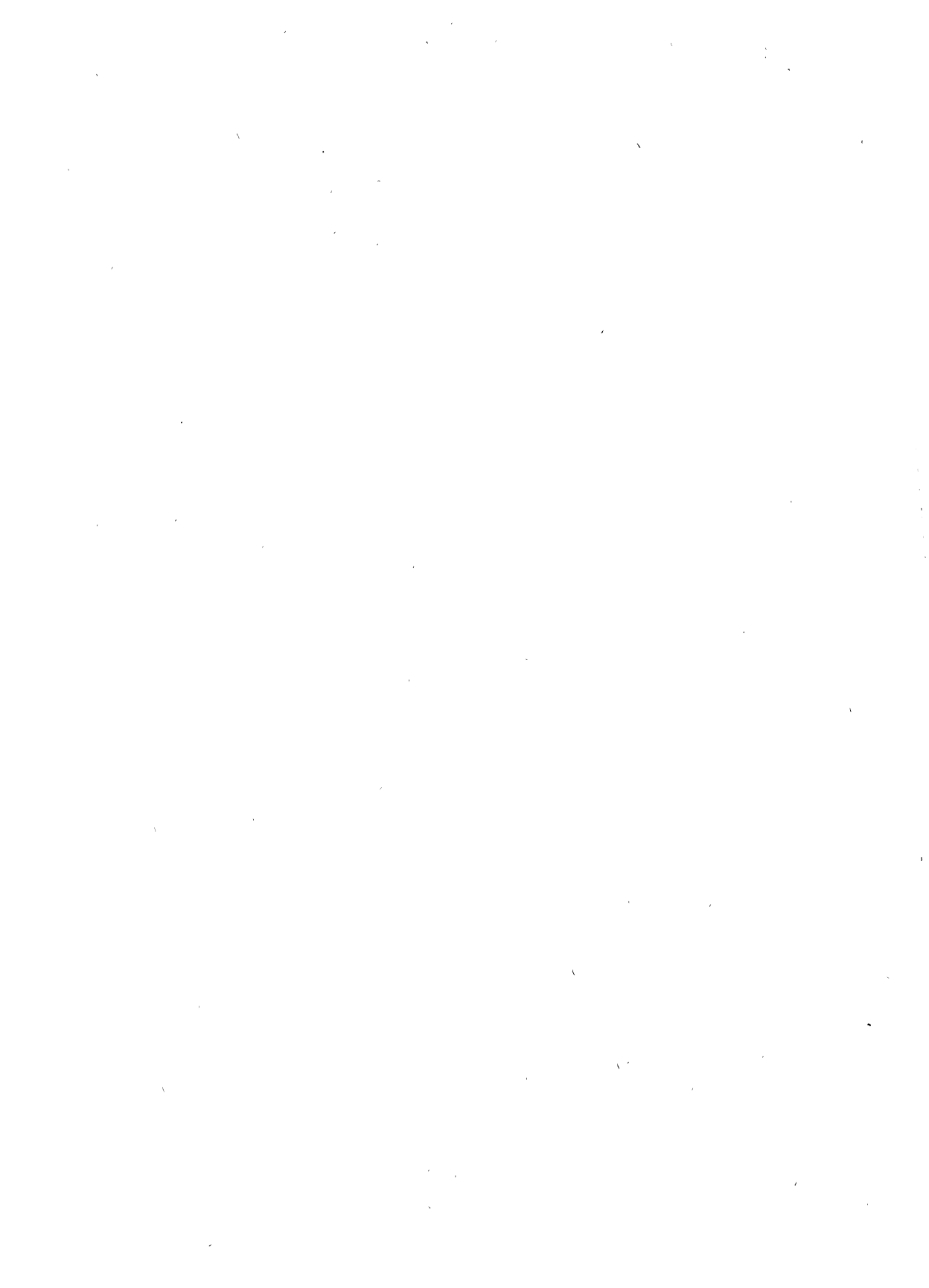
ICE-51 or ICE-44 Emulators
 iPDS System with EMV-51
 Inteltec Series II or III Development System
 8051 Software Development Package

Reference Manual

146312-001 — Guide to Using the Distributed Control Modules (Supplied)

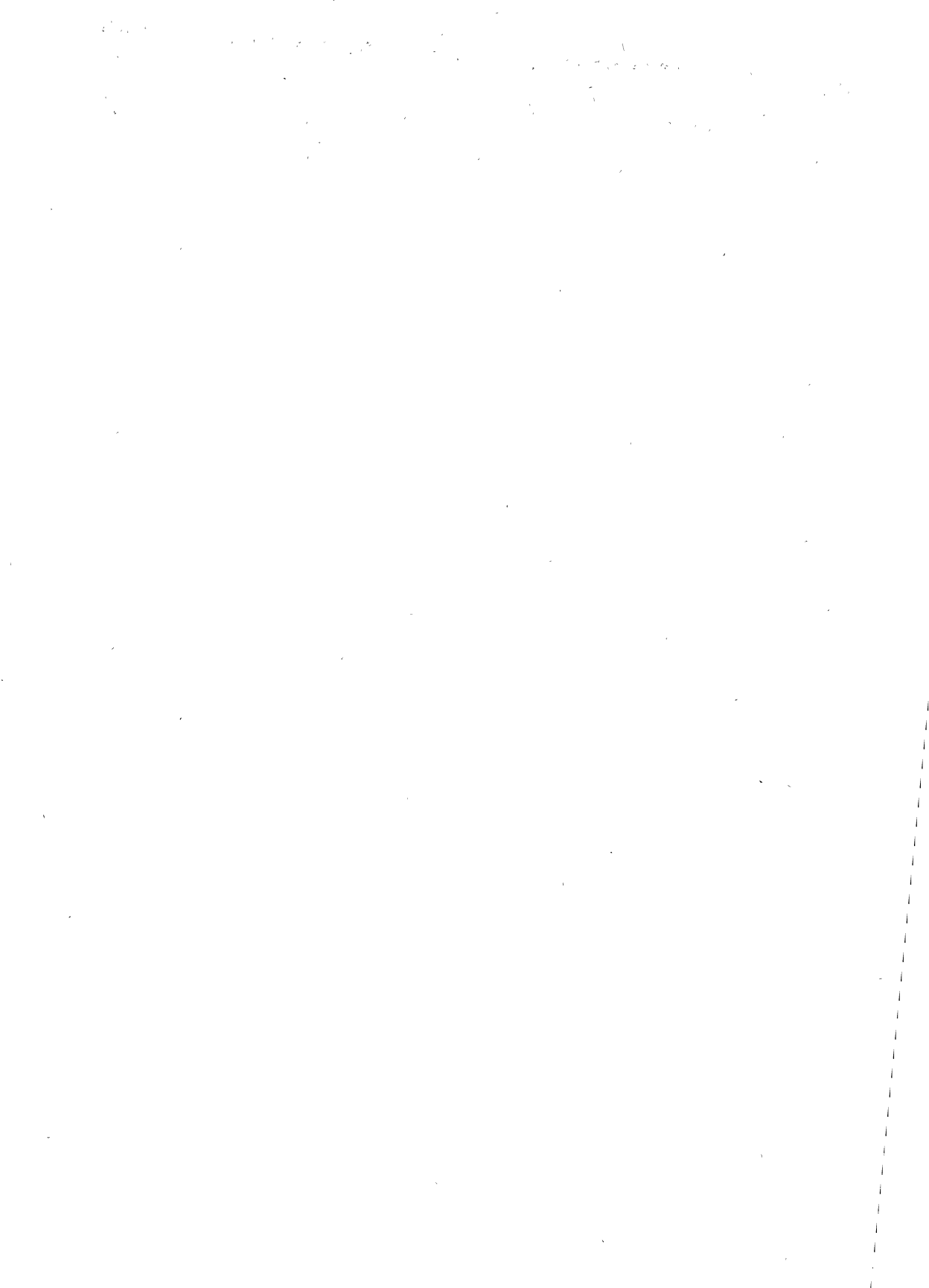
Ordering Information

Part Number	Description
iRMX 510BY	iDCM Support Package w/ Reference Manual A,B,E, and F Media Formats Supplied.



Systems and Applications Software

6



SYSTEM & APPLICATION SOFTWARE

Thus far in this Handbook you have read about a rich set of software available for Intel's hardware and offered by Intel. Almost all of this software is targeted at the programmer or the engineer, i.e. to highly technical and specialized audiences that are intimately familiar with computers. Intel also has software which can be used by the non-programmer to help him solve problems in his professional arena.

The professional end user (and the Value-Added Reseller who targets his systems at the professional end user) is rapidly evolving as the largest segment of the computer user base. With predictions that within a decade every professional will have a computer on his (or her) desk or at least readily available, it is easy to see the need for generalized tools to help improve the productivity of these masses.

The advent of the Personal Computer not only made such predictions plausible but it has already defined many of these tools as necessary for professional productivity: spreadsheets, electronic mail systems, word processors, and graphics displays.

The passage of time and shifting emphasis of our educational systems have also impressed upon these professionals the importance of the data residing in their corporate data processing computers — and their work environment has shown them explicitly what is meant by the data processing application backlog. Database management system (DBMS) is a term now familiar to most professionals.

A demand for combining the benefits of the corporate mainframe with those of the personal computer is now emerging as users see needs for sharing their data and the work performed by other users, for accessing subsets of the mainframe DBMS but not being constrained by the DP backlog, and for doing these things without the constant assistance of a "compute guru". Multi-user small systems with data extract facilities, host communications, PC links, and remote file transfer capabilities are beginning to address this need.

Intel is responding well to this emerging demand. Our multi-user XENIX-based systems are ideally suited for this environment. Our mainframe DBMS SYSTEM 2000 product family, which provides sophisticated data manipulation services, also offers user-friendly non-programmer facilities for today's professional.

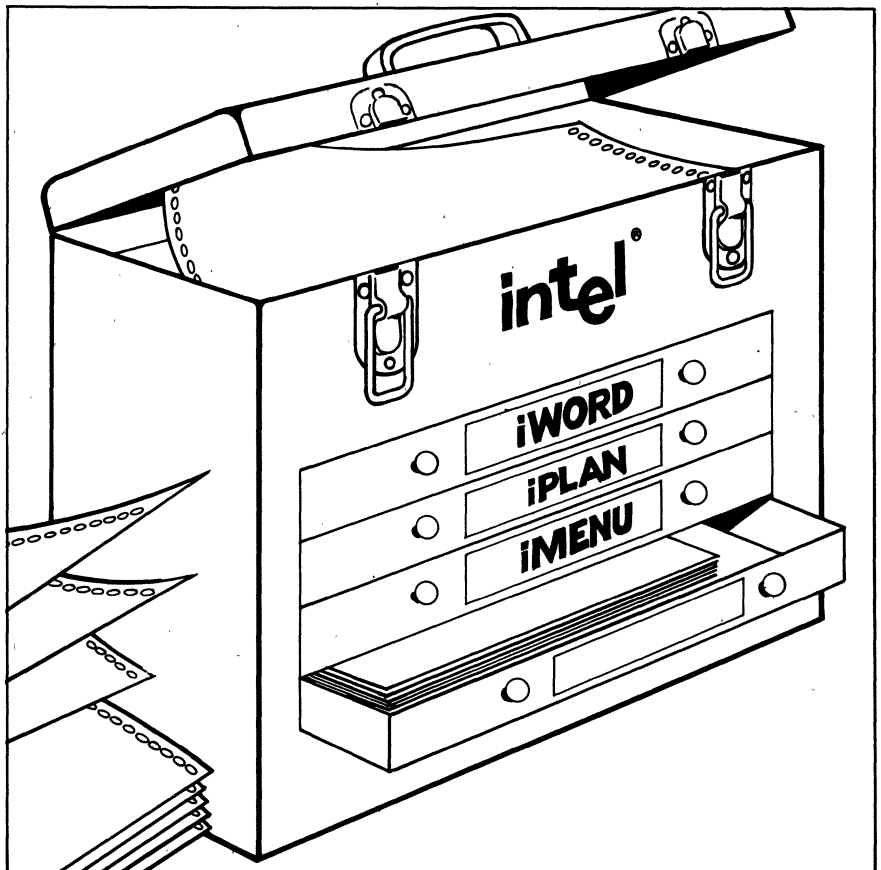
In addition, Intel has in place an active and successful program for attracting, qualifying, and referencing third-party software to execute on our systems.

The Intel Database Information System (iDIS 715) is a multiuser XENIX 286-based microcomputer system that includes a complete set of end-user productivity and application development tools. The iDIS system can be purchased as an integrated hardware/software microcomputer system configured for specific departmental computing applications. In addition, the various iDIS productivity tools, application development tools, relational database management system, and communication software options can be purchased separately to run on XENIX-based Intel microcomputers.



**XENIX*
Productivity
Software
Tools**

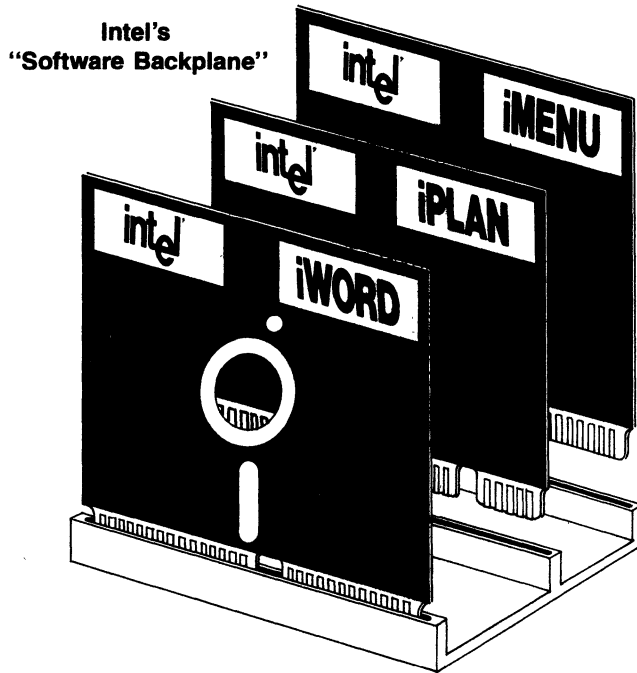
- **iWORD Processing**
- **iPLAN (Multiplan*) Spreadsheet**
- **iMENU Development System**



INTRODUCTION

Software tools for the XENIX environment

Intel's productivity software tools are designed to meet the basic information processing needs of the office environment. Tailored specifically for the XENIX* operating system, the software tools are available as individual packages which can be applied to specific end-user tasks. Intel's application packages are also offered as a Seamless™ set of software tools, integrated with a hardware/software system such as Intel's Database Information System (iDIS™ 86/735). Seamless software tools support the transparent sharing of data files among various application packages with complete data integrity. With Seamless software, results from one application package are readily accessible and compatible as input for another form of processing.



iWORD*

- Standard text editing/formatting commands
- Designed especially for the XENIX operating system
- Easy-to-use for beginners, powerful for experts
- Full-screen text editor
- Access to XENIX typesetter and printer drivers
- Embedded commands for global formatting
- On-screen display of formatted text
- On-line Help facility, spelling/dictionary module, and mail/merge facility
- Worldwide service and support

The powerful, versatile word processing tool

Intel's iWORD package is a sophisticated, yet friendly word processing tool for preparing business documents, such as reports, letters, memoranda, technical papers, and more. Written in the "C" language and tailored to the XENIX operating system, the iWORD package can run in both multi-user and single-user environments. Menu-driven and screen-oriented, the iWORD package supports all standard text editing, storage, and formatting development functions.

An efficient, easy-to-use text processor

Inexperienced users will find the iWORD software concepts intuitively easy. For example, the user accesses a document file by opening a "drawer," and editing commands follow familiar "cut and paste" procedures.

All commands are in plain English. No memorization is necessary, and many operations are executed by a

single keystroke. Concise command menus and an on-line Help facility are continuously available so that novice users can quickly advance in their word processing abilities. The iWORD system is sufficiently powerful to meet the needs of more experienced users as well.

Designed around office needs

Intel's iWORD software is based on the simple concept of an office file cabinet, defined by a collection of drawers, each of which contains files (documents).

The user may:

- Open an existing drawer or make a new drawer
- Create a new file or select an existing file
- Rename a drawer or file
- Add to, change, copy, move or delete the selected file.

There is no limit to the number of user-created drawers other than the availability of disk storage space.

*iWORD is a version of Horizon Word Processing, a trademark of Horizon Software Systems, Inc.

On-screen display of formatted text

The word processor allows users to visually format documents and print them as they are displayed on the terminal, or to format them with the powerful text processing facilities inherent to the XENIX operating system. This on-screen display capability is particularly helpful in preparing documents for typesetting.

The results of text formatting commands appear immediately on the screen. Examples of these commands include:

- Right justification
- Underlining
- Indentation
- Centering
- Alignment.

Advanced word processing features

Inexperienced users may execute commands from a simple menu (and related Help screens), while more proficient users may opt to use up to 64 function keys without accessing the menu.

The iWORD system allows simultaneous support for multiple character and/or line printers at the local or system level; printer selection is an operator option at print time.

The iWORD package includes a spelling checker and correction facility with an extensive on-line dictionary.

A Mail/Merge facility is available to combine mailing lists and document files (e.g., form letters) for printer output. Mail/Merge also provides the capability of incorporating paragraphs from a third file.

The iWORD processing allows on-screen sorting of numeric or alphabetic text.

Special editing commands

- Find commands ("string search") to locate characters or words in text for possible changes or additions
- Deletion commands for removing words, sentences, lines, paragraphs and entire files
- Fill commands to fit as many words as possible in a finite space
- Form command to type over existing text
- Command to mark location of the cursor within text
- Paste-in command to copy a section of text
- Replace command that replaces one text area with another
- Tab setting commands

Embedded "dot" commands

When formatting or printing needs are complex, the user has easy access to more powerful embedded "dot" commands. "Dot" commands are most useful for medium-sized and long documents requiring sophisticated formatting functions like subscripts, superscripts, and footnotes. "Dot" commands are fully compatible with NROFF and TROFF, the XENIX-supplied printing and typesetting utilities. The results of "dot" commands are displayed on-screen before the document is printed.

Embedded commands for global formatting include:

- Page layout
- Justification
- Automatic hyphenation
- Running headers and footers
- Footnotes
- Superscripts and subscripts
- Automatic page numbering
- XENIX typesetting commands (TROFF)
- XENIX printing commands (NROFF).

Editing two files simultaneously

The iWORD package provides a moveable "window" into a file for full-screen text editing. The user may simultaneously display two areas of the same document or two different documents in two screen "windows." Employing this "split screen" capability, the user may review two different files at the same time, as well as move text between files.

Designed for experienced and novice users

The iWORD software provides the experienced word processing user the full strength of XENIX text preparation commands, such as NROFF and TROFF. The iWORD package also offers a comprehensive menu shell which makes the word processing software easy to use for even the most inexperienced computer user. In conclusion, the iWORD system is a powerful, "user friendly," and flexible word processing package intended for all levels of computer proficiency.

iPLAN*

- Industry standard advanced electronic spreadsheet functions
- Sophisticated formatting options
- Easy-to-use English commands
- Extensive, on-line Help facilities
- Scrolling features and multi-window/multi-table display
- Links and updates multiple inter-related spreadsheets
- Automatically updates calculations
- Worldwide service and support

The most advanced electronic spreadsheet

The iPLAN Multiplan Spreadsheet software is one of the most powerful, easy-to-use "electronic worksheet" programs available. Developed by Microsoft Corporation, Multiplan has been enhanced for Intel hardware environments operating under XENIX.

The iPLAN package is a multi-purpose tool capable of a wide variety of business and scientific applications: financial modeling, planning, forecasting, tabulations, calculation of engineering formulas, and much more. It supports "what-if" decision-modeling with a versatile two-dimensional matrix that can be custom-tailored for specific use.

Unlike other spreadsheets, the iPLAN system is designed to meet the needs of both inexperienced and sophisticated computer users. It also offers versatile presentation and reporting capabilities.

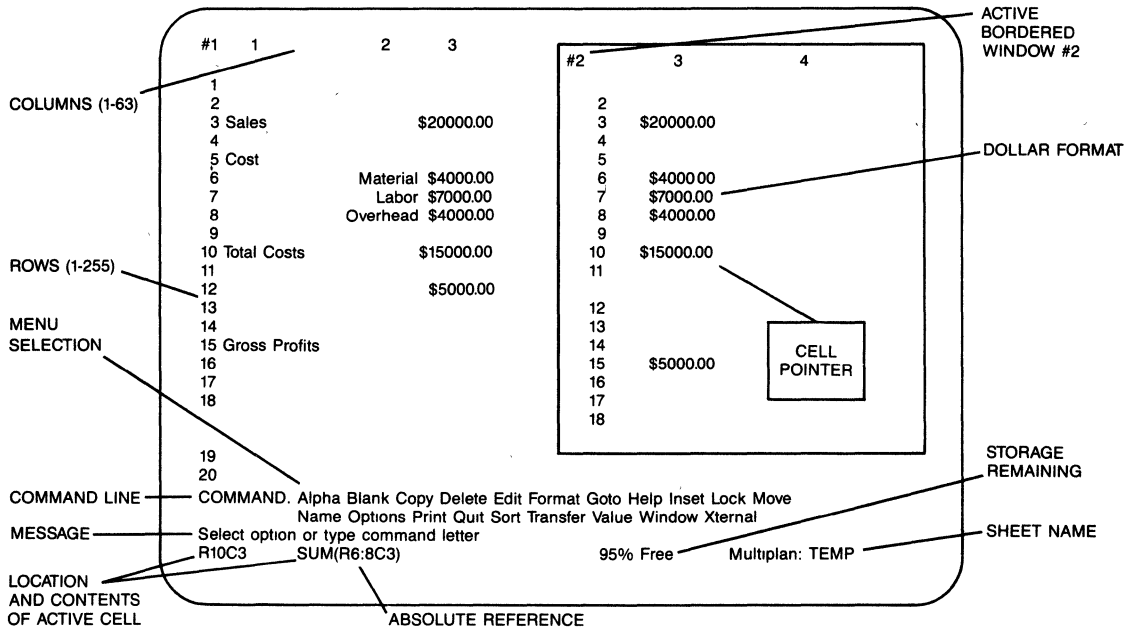
The iPLAN matrix format

The iPLAN software displays numerical data, text, or formulas in matrix (row/column) format. The spreadsheet screen is divided into 'cells' which are referenced by row and column numbers. Cells may contain numeric data, formulas, text, or labels. Commands are listed at the bottom of the screen along with the current addressed cell, the amount of unused spreadsheet storage space, and the name of the file in use.

Designed for ease-of-use

Beginning iPLAN users can start building worksheets after a couple hours of initial use. While simple to operate, the iPLAN system functionality is enhanced by the skill of the user.

* iPLAN is a version of Microsoft Multiplan, a trademark of Microsoft Corporation.



Typical Multiplan Screen Display

The iPLAN package does not use cryptic, abbreviated commands or reference codes (e.g. "AZ23"). Instead, it uses plain English commands (e.g. COPY) and reference names (e.g. COSTS or SALES). Completely menu-driven, the iPLAN software prompts the user with simple commands that can be executed with a single keystroke. To help in command selection, the user can access a reference guide.

Notable iPLAN features include:

- Ability to build formulas by high-lighting cells
- Menu-driven functions and command prompting
- Plain English command words and formulas
- Comprehensive on-line reference guide
- Eight-window display option
- Full-screen display of worksheet formulas.

A dynamic, versatile workspace

The iPLAN package offers an effective workspace that is 63 columns wide by 255 rows long. Worksheets are easily designed to fit project requirements. Moreover, worksheets can be linked to automatically receive or transmit data into other related iPLAN worksheets. Column width can be varied to accept long (or short) words and numerals; lines of text can be typed across several columns.

Up to eight windows are available with vertical and horizontal scrolling, such that different areas of a very large worksheet can be viewed simultaneously. The windows can be aligned, scrolled together, opened, or closed at the user's choice.

Built-in data security

The iPLAN software features cell locking to protect worksheet data. When data and formulas have been entered, the specified information can be "locked" in place so that vital data cannot be accidentally erased or altered.

Flexible presentation features

The iPLAN system enables users to produce printed reports of professional caliber. The program includes special formatting, alignment, and printing functions that support the printing of presentation-quality reports. The iPLAN software can automatically break a spreadsheet into multiple pages, and the user can specify the appropriate margins.

Powerful modeling capabilities

Highlights of iPLAN modeling features include:

- Alphabetical or numerical sorting capabilities
- Links and automatically updates up to eight interrelated worksheets
- Automatically updates subtotals, totals, percentages, growth curves and other calculations
- Performs multiple iterations to solve closed-loop problems
- Automatically revises formulas when reordering rows and columns in displays
- Cells and areas can be named for clarity
- Continuous formatting allows entries across cell boundaries
- Formulas moved to various worksheet locations without retyping
- Includes special editing area for quick additions or deletions
- Sheet display may be redesigned or formatted in various ways without altering the stored data
- Formulas, words, or numerals can be entered into any location so that printed sheets have titles and descriptions
- Offers a rich repertoire of advanced math functions and operators.

iMENU*

- Hierarchical control of menu screens to organize application program use
- On-line application development/maintenance system
- A menu screen design and menu development system for non-programmers
- On-line Help facility
- Written in the "C" language, specifically for XENIX
- Supports turnkey application development
- Worldwide service and support

Simplifies use, development and maintenance of XENIX-based applications

The iMENU software package is a hierarchical user interface and application development tool that ties together XENIX-based applications to achieve a high level of software integration. The iMENU package allows applications developers to create integrated, logical, and friendly interfaces to XENIX applications.

In effect, the iMENU system allows the XENIX operating system to appear transparent to the non-technical user, while it offers all the power and functionality inherent to XENIX to the more experienced user. The iMENU package interfaces with virtually all character-oriented terminals.

Aids application development and software packaging

Programmers and experienced users can apply the iMENU system in maintaining or creating menus, forms, or Help screens for existing or new applications.

* iMENU is a version of Schmidt's /menus, a trademark of Schmidt Associates.

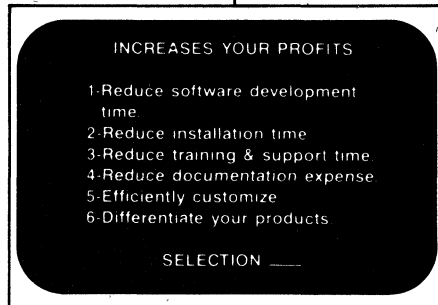
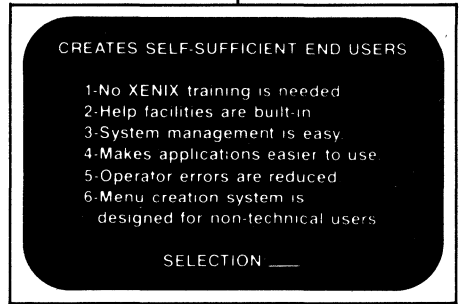
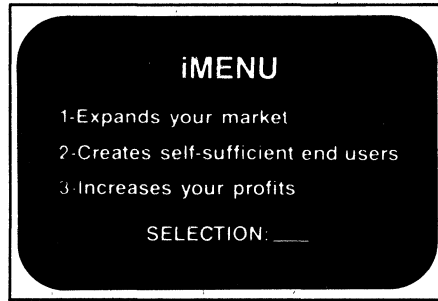
Full XENIX functionality is retained and simplified with the iMENU software. Experienced users have the option of skipping step-by-step menu selection via the fast menu selection mode. Advanced users can also modify the menu system to reflect changes in existing applications or to incorporate new applications.

Standard iMENU package functions include:

- Definition of login IDs
- Add, delete, and list login IDs
- Authorization control
- Define, update, delete and list menu items and attributes
- Screen and forms building
- Interaction with XENIX shell commands
- Powerful macros
- Fast menu selection mode for experienced users
- On-line Help facility.

The iMENU software includes a Menu Development Subsystem, which is a menu-driven set of maintenance functions allowing:

- Menu screen maintenance
- Form screen maintenance
- Help screen maintenance
- Menu selection maintenance
- Macro maintenance
- Shellscrip maintenance
- Login ID maintenance
- Deauthorization maintenance
- Backup/restore.



Comprehensive on-line Help system

The Help system is an interactive user-assistance facility. The Help system is completely integrated with the iMENU package so the user need not rely on bulky reference manuals to operate a particular application. In the event the user encounters problems or has questions, explanatory solutions can be made available at all times. Using the iMENU system itself, programmers and experienced users can extend or modify the Help system to include new applications.

Expands markets and increases profits

Systems integrators will find the iMENU system to be an indispensable tool for packaging XENIX-based applications software and integrated hardware/software systems.

Using the iMENU package, application developers can:

- Extend the user-interface to wrap around new and existing applications software
- Customize existing applications to meet varying customer needs
- Develop application demos
- Create applications that are easily used by non-programmers
- Package separate programs into integrated applications.

The iMENU software enhances the cost-effectiveness of application development by:

- Improving time to market for new software products
- Reducing software development time
- Reducing the need for training and support
- Decreasing software installation time
- Cutting documentation expenses
- Unifying a family of software products for consistent screen appearance and operation.

Integrated software for the iDIS system

Intel's Database Information System (iDIS 86/735) provides end users and systems builders with a vehicle for incorporating a Seamless set of software productivity tools. Seamless software supports the transparent sharing of data files among application packages with

SOFTWARE PRODUCTIVITY TOOLS

- Technology tailored to Intel's XENIX® system
- Experienced, world-wide Intel service and support

iWORD

- All-purpose office word processing

iPLAN

- Comprehensive what-if decision modeling

iMENU

- Menu-driven system management, application development, application operation

complete data integrity. The iDIS system is a multi-user, multi-tasking XENIX-based microcomputer available with the iWORD processor, the iPLAN spreadsheet, the iMENU development system, iXTRACT communication facilities for downloading mainframe databases, the iDB DBMS for local relational database management, and software that supports networking of personal computers. Application development tools and high-level programming languages are also offered with the iDIS system. Data can be transferred among the Seamless software packages, and the iDIS menu system and iHELP facility provide a friendly, common user interface. The iDIS system is an example of how Intel provides hardware/software components at all levels of integration to meet individual system needs.

Worldwide service and support

All Intel software included under an active software maintenance agreement is fully supported by Intel's staff of trained software engineers. Depending on the system configuration, several levels of support are available. Each package is offered with complete documentation, including a comprehensive user manual and installation guide.

SPECIFICATIONS Required Hardware:

- Any 8086 or 80286-based system including Intel's SYSTEM 86/300, 286/300 family and iDIS systems
- Minimum of 128 KB memory
- At least two floppy disks or one hard disk
- One 8 in. or 5.25 in. double-density floppy disk drive for distribution media.

Required Software:

- Intel's XENIX 86/286 Operating System

Warranty:

90 days for: Software Updates and application support. Continuing support services available with subscription to a Software maintenance agreement.



The following are trademarks of Intel Corporation and may be used only to describe Intel products: BXP, CREDIT, i, ICE, i²ICE, ICS, iDBP, iDIS, iLBX, i_m, iMMX, Insite, INTEL, , Intelevison, Inteltec, Intelligent Identifier™, Int₂BOS, intelligent Programming™, Intellink, iOSP, iPDS, iRMS, iSBC, iSBX, iSDM, iSXM, Library Manager, MCS, Megachassis, Micromainframe, MULTIBUS, Multichannel™ Plug-A-Bubble, Seamless, MULTIMODULE, PROMPT, Ripplemode, RMX/80, RUPI, SYSTEM 2000, Data Pipeline, iDIS, iDBP, and UPI, and the combination of ICE, ICS, iRMX, iSBX, MCS, or UPI and a numerical suffix. Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other patent licenses are implied. Specifications are subject to change without notice.

iWORD is a version of Horizon Word Processing, a trademark of Horizon Software Systems, Inc. iPLAN is a version of Microsoft's multiplan, a trademark of Microsoft Corporation. iMENU is a version of Schmidt's /menus, a trademark of Schmidt Associates.

Information contained herein supercedes previously published specifications on these devices from Intel.



THIRD PARTY SOFTWARE FOR INTEL SYSTEMS

- Over 125 Intel qualified software packages to meet your software needs
- Select from a choice of packages in most applications areas
- Support from the experts—the software manufacturers themselves
- Tested by Intel to ensure quality and reliability on Intel Systems

SOFTWARE EXPRESS

RealWorld™ Business Software for Microcomputers Corporation

QUADRATRON

Thoroughbred™ SOFTWARE

OPEN SYSTEMS INC.

BURR BROWN

MICRO FOCUS

DATA LANGUAGES

UNIFY

Key Systems, Inc.

RM/RYAN-McFARLAND

UX-Basic®

Access Technology, Inc.

TOM SOFTWARE CONSULTANT

CYMA

DATA RETRIEVAL CORPORATION

GDS

TTN, Inc.

Xicom

RELATIONAL DATABASE SYSTEMS, INC.

MCBA

MICRO-INTEGRATION

SMI CORPORATION

UNIFY CORPORATION

Qmtool

RHODNIUS

CONETIC SYSTEMS INC.

CLINICAL DATA DESIGN

HORIZON™ software systems

NMI

Norton/Murphy International, Inc. Businessware

SOFTEST Inc.

Micro Data Base Systems, Inc.

AMERICAN BUSINESS SYSTEMS INC.

PRODUCT NAME	VENDOR	SYSTEM				AVAILABILITY
		86/310	286/310	86/380	286/380	
Accounting						
Thoroughbred Accounting	SMC		X			Immediate
Open Systems Accounting	Open Systems		X			Immediate
MCBA Accounting	MCBA		X			Immediate
APPGEN Accounting	Software Express		X			Immediate
BACs	ABS		X			Immediate
Real World Accounting	Real World		X			Q4/84
CYMA Accounting	CYMA		X			Q4/84
Complete Accounting	NMI		X			Q4/84
Manufacturing						
MCBA Manufacturing	MCBA		X			Q4/84
ProfitKey	Key Systems		X			Q1/85
A&M Manufacturing	TOM Software		X			Q4/84
Specialty Manufacturers	NMI		X			Q4/84
Medical						
MDX	Clinical Data		X			Immediate
Vertical						
Contractor Management	TOM Software		X			Q4/84
Distributor Management	TOM Software		X			Q4/84
Not-for-Profit	TOM Software		X			Q4/84
Project Management	TOM Software		X			Q4/84
Property Management	TOM Software		X			Q4/84
Public Accountant	TOM Software		X			Q4/84
Restaurant Management	TOM Software		X			Q4/84
Personnel Searcher	NMI		X			Q4/84
Magazine Circulation	NMI		X			Q4/84
Customer Profile	NMI		X			Q4/84
Trucking Dispatcher	NMI		X			Q4/84
Phototypesetting	NMI		X			Q4/84
Client Accounting	CYMA		X			Q4/84
Construction	CYMA		X			Q4/84
Chiropractic	CYMA		X			Q4/84
Orthodontic	CYMA		X			Q4/84
Dental	CYMA		X			Q4/84
Medical	CYMA		X			Q4/84
THIRD PARTY SOFTWARE FOR INTEL iRMX™ SYSTEMS						
Languages						
Microsoft BASIC	Intel	X	X	X	X	Immediate
Mark Williams C	Intel	X	X	X	X	Immediate
Graphics						
PBG 100	Pacific Basin Graphics	X	X			Immediate
Communications						
3270 SNA	Xicom	X	X	X	X	Immediate
X.25	T.I.T.N.	X	X	X	X	Immediate
3270 Bisync	Data Retrieval	X	X	X	X	Immediate
3780 Bisync	Micro Integration	X	X	X	X	Immediate
Database						
DxSystem	GDS	X	X			Q4/84
Manufacturing						
Ladder 86	Engineering Tools	X	X			Q4/84
Other						
IEEE-488	Ziatech	X	X			Immediate
Driver to Data I/O subsystem	Burr Brown	X	X			Q4/84

THIRD PARTY SOFTWARE FOR INTEL XENIX* SYSTEMS						
PRODUCT NAME	VENDOR	SYSTEM				AVAILABILITY
		86/310	286/310	86/380	286/380	
Languages						
Microsoft BASIC	Intel	X	X	X	X	Immediate
Micro Focus COBOL	Intel	X	X	X	X	Immediate
Microsoft FORTAN	Intel	X	X	X	X	Immediate
RM COBOL	Ryan McFarland		X			Immediate
SMC BASIC	SMC		X			Immediate
Softbol	Omtool		X	X		Immediate
UX BASIC	UX Software		X			Immediate
S-TRAN	SMI		X			Immediate
TOM BASIC	TOM Software		X			Q4/84
Spreadsheet						
iPlan	Intel	X	X	X	X	Immediate
20/20	Access Technology		X			Q4/84
Office Automation						
iWord	Intel	X	X	X	X	Immediate
iMenu	Intel	X	X	X	X	Immediate
Q-One	Quadratron		X			Immediate
Q-Menu	Quadratron		X			Immediate
Q-Calc	Quadratron		X			Immediate
Q-Mail	Quadratron		X			Immediate
Q-Date	Quadratron		X			Immediate
Q-Call	Quadratron		X			Immediate
Q-Note	Quadratron		X			Immediate
Q-Form	Quadratron		X			Q4/84
DATA 3500	Tom Software		X			Q4/84
LEX	Softest		X			Immediate
Application Generator						
APPGEN	Software Express		X			Immediate
C/Tools	Conetic Systems		X			Q4/84
Graphics						
iGraph	Intel	X	X	X	X	Q4/84
PBG 200	Pacific Basin Graphics		X			Q4/84
SMC Color Graphics	SMC		X			Q4/84
Communications						
PC Link	Intel	X	X	X	X	Q4/84
3270 Bisync	Intel	X	X	X	X	Q4/84
3270 SNA	Xicom		X		X	Q4/84
X.25	T.I.T.N.		X		X	Q4/84
3780 Bisync	Micro Integration		X		X	Q4/84
HASP	Intel	X	X	X	X	Immediate
Database						
iDB	Intel	X	X	X	X	Q4/84
Informix	RDS		X			Immediate
File-it!	RDS		X			Immediate
C-ISAM	RDS		X			Immediate
Unify	Unify		X			Q4/84
MDBS III	MDBS		X			Q4/84
IDOL	SMC		X			Immediate
Progress	Data Languages		X			Q4/84

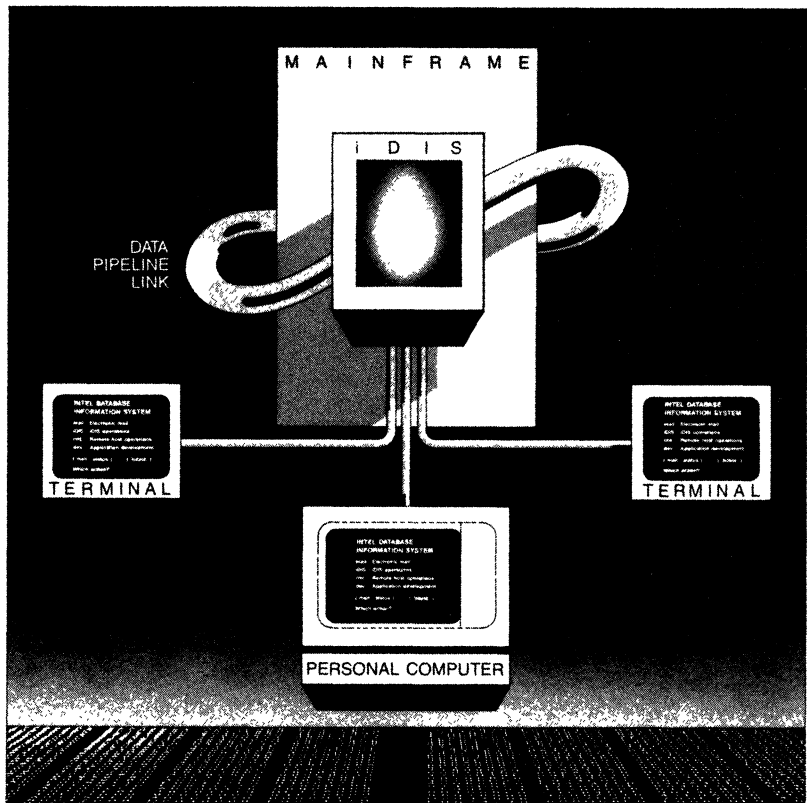
*XENIX is a trademark of Microsoft Inc



Database Information System iDIS™ 715

- Building block for departmental-level applications
- Data Pipeline™ system to distribute databases
- Direct mainframe database extract and file transfer facilities
- Gateway for personal computer and terminal access
- Multiuser XENIX® 3.0 operating system
- Local relational database management and report writer
- Integrated software with on-line help facility
- Word processing, spreadsheet, graphics, menu development, and communication options
- C programming language
- Desk-top integrated microsystem
- Worldwide vendor service and support

*XENIX is a registered trademark of Microsoft Corporation.



Building Vertical Applications with the iDIS™ System

The Intel Database Information System (iDIS™) is a fully-integrated multiuser hardware/software microcomputer system. It serves as a building block for end-user applications and a powerful access tool in the Data Pipeline connection between a mainframe and the end-user. Data can be maintained by central data processing departments and distributed to departmental users through a network of terminals and PCs. The system can be configured as a gateway in the micro-to-mainframe flow of data or as a stand-alone processor with shared local database capabilities. The iDIS system includes an SQL-compatible, multiuser relational DBMS for shared access to disk storage and features a full range of information processing functions for multiple concurrent users at all levels of technical skill.

The iXTRACT remote database extract facilities

The iDIS system offers two interactive, menu-driven modes of database extract. With the Remote File Transfer (RFT) iXTRACT facility, a "flat file" (sequential) data structure can be downloaded from the mainframe and converted into a local relational database. Using host computer utilities to generate the flat file, the RFT facility can download data from virtually any DBMS or file management system. The facility is bidirectional, such that flat files can be transmitted between a mainframe host and an iDIS system with its network of terminals and personal computers.

A second facility, the Direct iXTRACT facility, is a menu-driven data extract facility which directly downloads Intel SYSTEM 2000® databases (from IBM, CDC, and Sperry environments) into an iDIS database. Both RFT and direct modes allow non-technical users to access remote corporate databases and extract information while central data processing controls data security at every terminal.

*UNIX is a trademark of AT&T Bell Laboratories. Multiplan is a registered trademark of Microsoft Corporation.

Microsoft XENIX

The iDIS operating system is provided by XENIX 3.0, an enhanced industry-standard version of UNIX*. XENIX is a general-purpose, multi-user, interactive operating system designed to make the computing environment simple, efficient and productive for a wide range of users. While the system developer has access to all XENIX functionality, the operating system appears to be transparent to the user who interacts with the iDIS software through its menu system.

The XENIX system supplies:

- A flexible and logical hierarchical file system, with cross-directory file linking and multiple protection and security modes
- The XENIX shell command language, with conditional, recursive, and iterative constructs (for development of user/application procedures)
- Sequential, asynchronous, and background process execution
- Sophisticated editing and text-processing facilities supporting printers and typesetters
- Device-independent input and output.

The iDB-local relational database management (DBMS)

The iDIS system offers the iDB DBMS, a full-function relational DBMS that supports an interactive query/update language similar to that of IBM's SQL. Included with iDB is a Report Writer package. This allows users to prepare custom reports quickly from information in iDB without programming knowledge. The iDB DBMS offers all the power of a mainframe DBMS at the microsystem level. Multiple iDB users can concurrently access common local databases with confidence in system integrity.

Other features include:

- A user-prompting data entry and update subsystem
- A bulk loading and unloading utility for rapid transfer of data among files and databases
- Extensive on-line help facilities
- Descriptive error and diagnostic messages
- Programmatic interface to the C language and XENIX shell.

Seamless™ software interface

The iDIS software family is integrated into a Seamless set of productivity tools. Data can be easily transferred among the various iDIS application packages, such that the iWORD processor, iPLAN (Multiplan*) spreadsheet, and iDB DBMS can interchange data and reports. All iDIS decision-support tools can be easily brought to bear on a particular data-analysis problem.

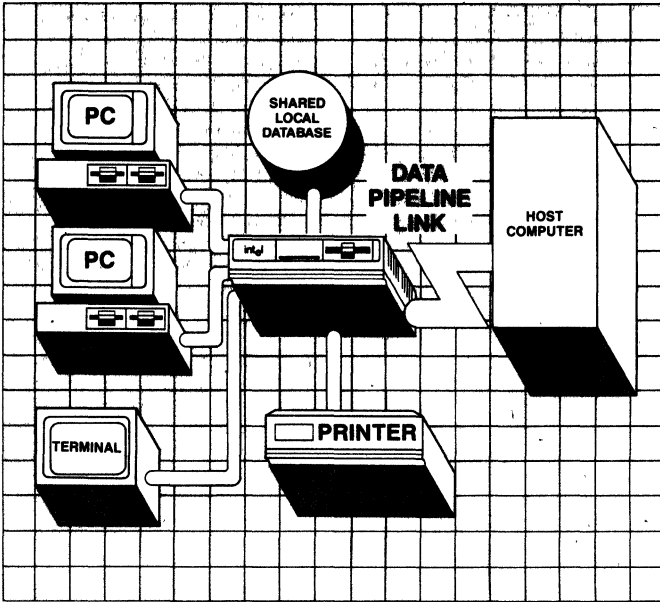
Individually, each package is accessible through a common user interface—a hierarchical menu system serving as a superstructure for the complete iDIS system. A common help facility binds all iDIS software.

The iWORD processor

The Intel iWORD facility is a sophisticated word processing tool that supports a complete office-wide range of document preparation functions. The iWORD user can develop, edit, store, format and print a variety of presentation-quality business documents, including reports, memoranda, technical documents, specifications and manuals. All iWORD commands are in plain English and many can be executed by a single keystroke. The iWORD processor is also sufficiently powerful for the experienced user, offering access to XENIX text processing capabilities including the printer- and typesetter-drivers nroff and troff. An on-line help facility is continuously available.

Major iWORD editing and formatting features include:

- Full-screen editor with on-line display of formatted text
- Embedded commands for global formatting
- Spelling/dictionary module and mail/merge facility
- Right justification, underlining, indentation, centering, footnotes, superscripts and subscripts.



The iPLAN (Multiplan) spreadsheet

The iDIS system supports 'what if' decision-modeling with iPLAN (Multiplan) Spreadsheet, a multi-purpose tool capable of a wide variety of business and scientific tabulations. The iPLAN user can custom-tailor a versatile two-dimensional matrix for specific analyses, including financial modeling, planning and forecasting. Like the other functions in the iDIS software environment, the iPLAN spreadsheet accepts data four ways: from the keyboard, from the iDB DBMS, from mainframe databases (via the data extract facility), and from formatted XENIX files.

Important iPLAN features include:

- Easy-to-use English commands
- Vertical and horizontal scrolling, multi-window and multi-table display
- Presentation of extra large tables
- Linking and updating multiple interrelated spreadsheets
- Automatic updating of calculations
- Alphanumeric sorting capabilities
- Extensive, on-line help facility.

The personal computer (iPC) connection

To complete the Data Pipeline connection, the iDIS system offers a menu-driven file conversion and transfer facility that allows single-user PC files to be accessed in the multiuser XENIX environment. The PC user can use the iDIS system to convert database and spreadsheet files from popular PC file formats (such as dBASE II* Lotus 1-2-3* and Multiplan formats) to iDB file formats. As a result, mainframe files can be downloaded to relational structures within iDB databases and further converted and downloaded to PC-based files for local applications analysis. The PC user can operate in three modes: bidirectional iDIS-to-PC file transfers, iDIS terminal emulation, and local PC-DOS control.

Office automation features

The XENIX operating system provides an electronic mail service in which business messages are shared and relayed with ease. XENIX also includes handy "desk calculator" functions and an electronic calendar that provides an automatic reminder (via electronic mail) of any user appointments.

Application development tools

The iDIS application development subsystem includes such software tools as the iMENU development system, C programming language, the XENIX Shell program, and the full-screen 'vi' editor (visual editor). These tools support efficient development and maintenance of program and text files by technical users.

The iDIS system offers a complete program development and execution environment for C, the versatile general-purpose language in which the operating system and all iDIS application packages are implemented. C maximizes development productivity by its structured programming methodologies and standard flow-control constructions—if, while, for, do, and switch (case). It provides pointers, the ability to perform address arithmetic, and recursive functions. Many existing C-based applications can be efficiently ported to the iDIS system.

The iMENU development facility

The iMENU development facility provides the iDIS system-level user interface, tying together the XENIX operating system, iDIS applications software, and help system. The iMENU facility retains and yet simplifies full XENIX functionality. Programmers and non-programmers alike can use the iMENU facility in creating or modifying menus, forms, and help screens for existing or custom-developed applications.

The on-line help facility

The help facility, a comprehensive on-line documentation feature, is integrated with the menu system so the user need not refer to hard copy reference manuals when using iDIS applications. Experienced users can employ the iMENU facility to extend or modify the help facility to specify help procedures for custom applications.

The iGRAPH facility

The iDIS system offers a presentation graphics package, iGRAPH, that provides high quality output to most

*dBASE II is a trademark of Ashton-Tate. Lotus and 1-2-3 are trademarks of Lotus Development Corporation

standard graphics peripherals. Through iDIS integration, data can easily be moved from iPLAN and iDB to iGRAPH, from iGRAPH to iWORD for printed output, and to and from iPC. Graphic peripherals supported are Tektronix 4105, color terminal, Tele-video 950 monochrome terminal with Retrographics board, Hewlett-Packard HP7475 plotter, the Epson MX80 printer, and the IBM PC with graphics board. The Intel terminal can be used to generate graphics hard copy.

Communications

The iDIS communications subsystem provides remote job entry (RJE) to mainframe hosts through its emulation of a HASP multileaving workstation or 2780/3780 protocol. TTY passthrough facilities also provide direct access to remote interactive applications, including other iDIS systems and personal computers. Support for 3270 BCS emulation is also available, and SNA support is planned.

BASE SYSTEM HARDWARE Processor

The iDIS 715 uses the MULTIBUS®-based iSBC® 286/10 board with the 80286 processor. An Intel 80287 coprocessor is standard to provide significant performance boost for numeric operations. Instructions are 8, 16, or 32 bits in length; data are 8 or 16 bits long; numeric processing, with the 80287, is carried out in 80-bit words. Memory management and protection are also included.

One megabyte of high-performance RAM with ECC is standard. To provide faster access, Intel memory boards are connected directly to the 286/10 processor board via iLBX™ (Local Bus Exchange).

Communications support

Asynchronous communication and synchronous mainframe communications support is handled by an iSBC 188/48 Advanced Communications Processor Board. Eight connections can be configured for terminals, PCs and/or mainframe communications.

MASS STORAGE

Winchester disk drive—The iDIS 715 contains a 40 MB 5¼" Winchester technology disk drive for program and data storage. The drive has an average access time about 40 milliseconds and a transfer rate of 6.44 Mbits/sec.

Intelligent controller—The iDIS 715 includes an intelligent, 8089-based iSBC 215 Winchester controller. This high performance interface contains firmware which is executed directly on the iSBC 215 controller to offload a significant portion of disk I/O overhead from the host 80286 processor. In addition, the iSBC 215 board supports an iSBX™ 218 controller to manage the floppy disk.

Floppy disk drive—A 5¼" 320 KB floppy disk drive is included in the base system. This floppy drive has an average access time of 91 milliseconds and a transfer rate of 250 KB/sec.

Optional peripherals

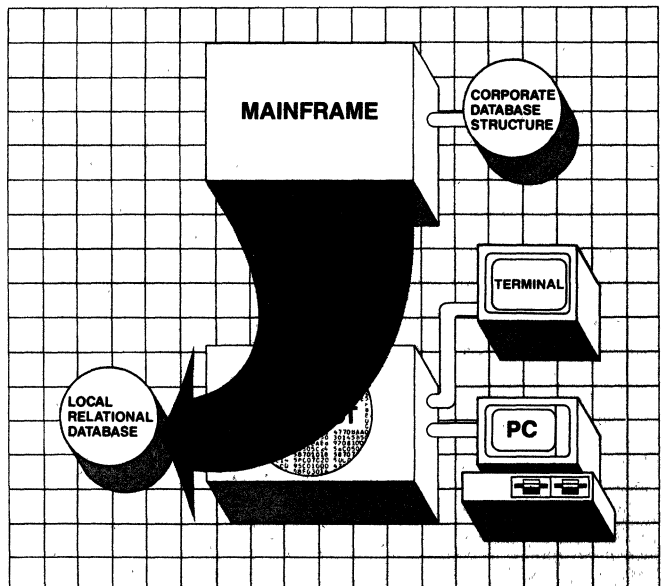
Display terminal—The standard iDIS system can connect up to 10 terminals. The terminal connected to the

console port can be used for system control and administration, as well as a regular workstation. An iDIS terminal is also available, with 24-80 character lines and a 25th status line. The 96 upper and lower ASCII characters are formed in a 7x9 matrix in an 8x10 cell. The screen is non-reflective and uses green P31 phosphor.

Printer—The iDIS system supports a Centronics-compatible printer. A dot matrix character printer capable of 200 character-per-second bidirectional printing is also available. It includes a Centronics-compatible parallel interface and a 218-character buffer. The printer features a 7x9 dot matrix to form all 96 ASCII characters. Maximum standard print line length is 132 characters or 218 with condensed print. The Intel printer prints 10 characters per inch; spacing of 5, 8.25, and 16.5 characters per inch can also be accommodated.

System support

The iDIS system is fully supported by Intel's worldwide service staff, including a group of information system



professionals with over 15 years experience in commercial database technologies. Support for iDIS applications software is included with the system price for 90 days and is optional thereafter. All iDIS software with a current software maintenance agreement is supported by the Intel Austin Systems Support Hotline.

In addition to the hotline, system support includes software updates, customer problem reporting, and a product newsletter. Intel provides comprehensive training classes on all iDIS applications, the XENIX operating system, programming languages, and hardware operation.

The iDIS hardware includes a warranty for 90 days mechanical, 45 days labor, and 90 days electrical components. After the warranty period, subscription maintenance is available from the Intel field service organization. Hardware service is also available for users on a per-call basis.

Extensive system documentation

The iDIS 715 is shipped with multiple hardware and software manuals that address all aspects of system operations. Software documentation includes manuals on the XENIX Release 3.0 operating system, iDB and Report Writer software, and each optional application package that is ordered. General overviews and detailed tutorials are an integral part of this documentation. A system installation and maintenance manual, a system overview manual, and a site preparation manual are also provided.

SYSTEM CONFIGURATION

Base Hardware System:

- 1 MB of RAM memory
- 320 KB floppy drive
- 40 MB Winchester disk
- Support for up to 10 terminals and/or PCs
- Printer support for Centronics-compatible printer
- Disk controller board
- Communications processor

Optional Hardware:

- Additional communications processor
- Terminals and dot-matrix printers

Base Software System:

- XENIX 3.0 operating system
- C programming language
- 'vi' editor
- XENIX utilities (including nroff and troff text processors)
- Electronic mail and calendar
- iDB and Report Writer
- iMENU (runtime) system
- Help facility
- Complete systems diagnostics

Optional Software:

- iWORD word processor
- iPLAN (Multiplan) spreadsheet
- iMENU menu development system
- Direct iXTRACT facility
- Remote File Transfer facility
- iPC (personal computer link)
- iGRAPH presentation graphics
- RJE communication support (2780/3780 and/or HASP protocol)
- 3270 BSC emulation

SPECIFICATIONS

Instruction cycle time

250 nanoseconds for fastest executable instructions.

Disk

Standard 40 MB Winchester disk; Second 40 MB disk is planned.

External/PC interface

Serial—8 asynchronous ports, configurable from 110 to 9600 baud. EIA Standard RS232C signal support is provided.

Parallel—one Centronics-compatible parallel I/O port for printer connections.

Regulatory Agency Specifications


Meets UL114-Safety; CSA 22.2-Safety; FCC Docket 20780-RFI/EMI. Designed to meet IEC 435-Safety; VDE 0871-RFI/EMI.

ENVIRONMENTAL OPERATING REQUIREMENTS

Altitude—Sea level to 8000 feet.

Temperature—15 degrees C to 35 degrees C.

Relative humidity—20% to 80% non-condensing over the operating temperature range. The environmental combination of humidity and temperature together cannot exceed 26 degrees C wet bulb.

The following are trademarks of Intel Corporation and may be used only to describe Intel products: BXP, CREDIT, i, ICE, iICE, ICS, iDBP, iDIS, iLBX, iM, iMMX, Insite, INTEL, , Intelevison, Inteltec, Intelligent Identifier™, InteIBOS, intelligent Programming™, Intellink, iOSP, iPDS, iRMS, iSBC, iSBX, iSDM, iSXM, Library Manager, MCS, Megachassis, Micromainframe, MULTIBUS, Multichannel™, Plug-A-Bubble, Seamless, MULTIMODULE, PROMPT, Ripplemode, RMX/80, RUPI, SYSTEM 2000, Data Pipeline, iDIS, iDBP, and UPI, and the combination of ICE, iCS, iRMX, iSBX, MCS, or UPI and a numerical suffix. Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other patent licenses are implied. Specifications are subject to change without notice.

iWORD is a version of Horizon Word Processing, a trademark of Horizon Software Systems, Inc. iPLAN is a version of Microsoft's Multiplan, a trademark of Microsoft Corporation. iMENU is a version of Schmidt's /menus, a trademark of Schmidt Associates.

Information contained herein supersedes previously published specifications on these devices from Intel.

80130/80130-2 iAPX 86/30, 88/30, 186/30, 188/30 iRMX 86 OPERATING SYSTEM PROCESSORS

- High-Performance 2-Chip Data Processors Containing Operating System Primitives
- Standard iAPX 86/10, 88/10 Instruction Set Plus Task Management, Interrupt Management, Message Passing, Synchronization and Memory Allocation Primitives
- Fully Extendable To and Compatible With iRMX® 86
- Supports Five Operating System Data Types: Jobs, Tasks, Segments, Mailboxes, Regions
- 35 Operating System Primitives
- Built-In Operating System Timers and Interrupt Control Logic Expandable From 8 to 57 Interrupts
- 8086/80150/80150-2/8088/80186/80188 Compatible At Up To 8 MHz Without Wait States
- MULTIBUS® System Compatible Interface

The Intel iAPX 86/30 and iAPX 88/30 are two-chip microprocessors offering general-purpose CPU (8086) instructions combined with real-time operating system support. They provide a foundation for multiprogramming and multitasking applications. The iAPX 86/30 consists of an iAPX 86/10 (16-bit 8086 CPU) and an Operating System Firmware (OSF) component (80130). The 88/30 consists of the OSF and an iAPX 88/10 (8-bit 8088 CPU). (80186 or 80188 CPUs may be used in place of the 8086 or 8088.)

Both components of the 86/30 and 88/30 are implemented in N-channel, depletion-load, silicon-gate technology (HMOS), and are housed in 40-pin packages. The 86/30 and 88/30 provide all the functions of the iAPX 86/10, 88/10 processors plus 35 operating system primitives, hardware support for eight interrupts, a system timer, a delay timer and a baud rate generator.

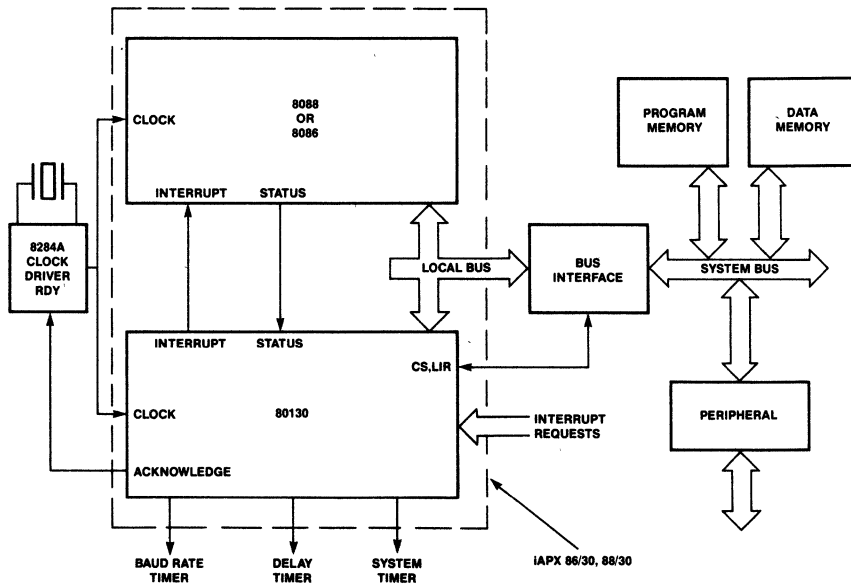


Figure 1. iAPX 86/30, 88/30 Block Diagram

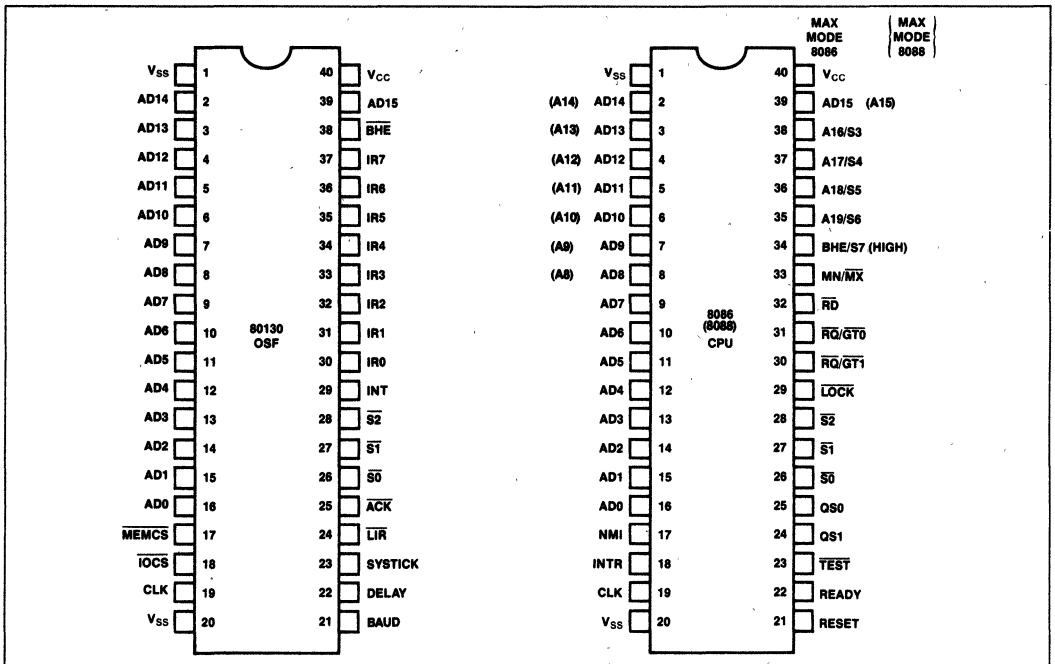


Figure 2. IAPX 86/30, 88/30 Pin Configuration

Table 1. 80130 Pin Description

Symbol	Type	Name and Function																																
AD ₁₅ -AD ₀	I/O	Address Data: These pins constitute the time multiplexed memory address (T ₁) and data (T ₂ , T ₃ , T _W , T ₄) bus. These lines are active HIGH. The address presented during T ₁ of a bus cycle will be latched internally and interpreted as an 80130 internal address if MEMCS or IOCS is active for the invoked primitives. The 80130 pins float whenever it is not chip selected, and drive these pins only during T ₂ -T ₄ of a read cycle and T ₁ of an INTA cycle.																																
BHE/S ₇		Bus High Enable: The 80130 uses the BHE signal from the processor to determine whether to respond with data on the upper or lower data pins, or both. The signal is active LOW. BHE is latched by the 80130 on the trailing edge of ALE. It controls the 80130 output data as shown. <table style="margin-left: 40px;"> <tr> <td>BHE</td> <td>A₀</td> <td></td> </tr> <tr> <td>0</td> <td>0</td> <td>Word on AD₁₅-AD₀</td> </tr> <tr> <td>0</td> <td>1</td> <td>Upper byte on AD₁₅-AD₈</td> </tr> <tr> <td>1</td> <td>0</td> <td>Lower byte on AD₇-AD₀</td> </tr> <tr> <td>1</td> <td>1</td> <td>Upper byte on AD₇-AD₀</td> </tr> </table>	BHE	A ₀		0	0	Word on AD ₁₅ -AD ₀	0	1	Upper byte on AD ₁₅ -AD ₈	1	0	Lower byte on AD ₇ -AD ₀	1	1	Upper byte on AD ₇ -AD ₀																	
BHE	A ₀																																	
0	0	Word on AD ₁₅ -AD ₀																																
0	1	Upper byte on AD ₁₅ -AD ₈																																
1	0	Lower byte on AD ₇ -AD ₀																																
1	1	Upper byte on AD ₇ -AD ₀																																
S ₂ , S ₁ , S ₀	I	Status: For the 80130, the status pins are used as inputs only. 80130 encoding follows: <table style="margin-left: 40px;"> <tr> <td>S₂</td> <td>S₁</td> <td>S₀</td> <td></td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>INTA</td> </tr> <tr> <td>0</td> <td>0</td> <td>1</td> <td>IORD</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> <td>IOWR</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> <td>Passive</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> <td>Instruction fetch</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> <td>MEMRD</td> </tr> <tr> <td>1</td> <td>1</td> <td>X</td> <td>Passive</td> </tr> </table>	S ₂	S ₁	S ₀		0	0	0	INTA	0	0	1	IORD	0	1	0	IOWR	0	1	1	Passive	1	0	0	Instruction fetch	1	0	1	MEMRD	1	1	X	Passive
S ₂	S ₁	S ₀																																
0	0	0	INTA																															
0	0	1	IORD																															
0	1	0	IOWR																															
0	1	1	Passive																															
1	0	0	Instruction fetch																															
1	0	1	MEMRD																															
1	1	X	Passive																															

Table 1. 80130 Pin Description (Continued)

Symbol	Type	Name and Function																																																						
CLK	I	Clock: The system clock provides the basic timing for the processor and bus controller. It is asymmetric with a 33% duty cycle to provide optimized internal timing. The 80130 uses the system clock as an input to the SYSTICK and BAUD timers and to synchronize operation with the host CPU.																																																						
INT	O	Interrupt: INT is HIGH whenever a valid interrupt request is asserted. It is normally used to interrupt the CPU by connecting it to INTR.																																																						
IR ₇ -IR ₀	I	Interrupt Requests: An interrupt request can be generated by raising an IR input (LOW to HIGH) and holding it HIGH until it is acknowledged (Edge-Triggered Mode), or just by a HIGH level on an IR input (Level-Triggered Mode).																																																						
ACK	O	Acknowledge: This line is LOW whenever an 80130 resource is being accessed. It is also LOW during the first INTA cycle and second INTA cycle if the 80130 is supplying the interrupt vector information. This signal can be used as a bus ready acknowledgement and/or bus transceiver control.																																																						
MEMCS	I	Memory Chip Select: This input must be driven LOW when a kernel primitive is being fetched by the CPU. AD ₁₃ -AD ₀ are used to select the instruction.																																																						
IOCS	I	<p>Input/Output Chip Select: When this input is low, during an IORD or IOWR cycle, the 80130's kernel primitives are accessing the appropriate peripheral function as specified by the following table:</p> <table border="1"> <thead> <tr> <th>BHE</th> <th>A₃</th> <th>A₂</th> <th>A₁</th> <th>A₀</th> <th></th> </tr> </thead> <tbody> <tr> <td>0</td> <td>X</td> <td>X</td> <td>X</td> <td>X</td> <td>Passive</td> </tr> <tr> <td>X</td> <td>X</td> <td>X</td> <td>X</td> <td>1</td> <td>Passive</td> </tr> <tr> <td>X</td> <td>0</td> <td>1</td> <td>X</td> <td>X</td> <td>Passive</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> <td>X</td> <td>0</td> <td>Interrupt Controller</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>Systick Timer</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> <td>1</td> <td>0</td> <td>Delay Counter</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>Baud Rate Timer</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> <td>1</td> <td>0</td> <td>Timer Control</td> </tr> </tbody> </table>	BHE	A ₃	A ₂	A ₁	A ₀		0	X	X	X	X	Passive	X	X	X	X	1	Passive	X	0	1	X	X	Passive	1	0	0	X	0	Interrupt Controller	1	1	0	0	0	Systick Timer	1	1	0	1	0	Delay Counter	1	1	1	0	0	Baud Rate Timer	1	1	1	1	0	Timer Control
BHE	A ₃	A ₂	A ₁	A ₀																																																				
0	X	X	X	X	Passive																																																			
X	X	X	X	1	Passive																																																			
X	0	1	X	X	Passive																																																			
1	0	0	X	0	Interrupt Controller																																																			
1	1	0	0	0	Systick Timer																																																			
1	1	0	1	0	Delay Counter																																																			
1	1	1	0	0	Baud Rate Timer																																																			
1	1	1	1	0	Timer Control																																																			
LIR	O	Local Bus Interrupt Request: This signal is LOW when the interrupt request is for a non-slave input or slave input programmed as being a local slave.																																																						
V _{CC}		Power: V _{CC} is the +5V supply pin.																																																						
V _{SS}		Ground: V _{SS} is the ground pin.																																																						
SYSTICK	O	System Clock Tick: Timer 0 Output. Operating System Clock Reference. SYSTICK is normally wired to IR2 to implement operating system timing interrupt.																																																						
DELAY	O	DELAY Timer: Output of timer 1. Reserved by Intel Corporation for future use.																																																						
BAUD	O	Baud Rate Generator: 8254 Mode 3 compatible output. Output of 80130 Timer 2.																																																						

FUNCTIONAL DESCRIPTION

The increased performance and memory space of iAPX 86/10 and 88/10 microprocessors have proven sufficient to handle most of today's single-task or single-device control applications with performance to spare, and have led to the increased use of these microprocessors to control *multiple* tasks or devices in real-time. This trend has created a new challenge to designers—development of real-time, multitasking application systems and software. Examples of such systems include control systems that monitor and react to external events in real-time, multifunction desktop and personal computers, PABX equip-

ment which constantly controls the telephone traffic in a multiphone office, file servers/disk subsystems controlling and coordinating multiple disks and multiple disk users, and transaction processing systems such as electronics funds transfer.

The iAPX 86/30, 88/30 Operating System Processors

The Intel iAPX 86/30, 88/30 Operating System Processors (OSPs) were developed to help solve this

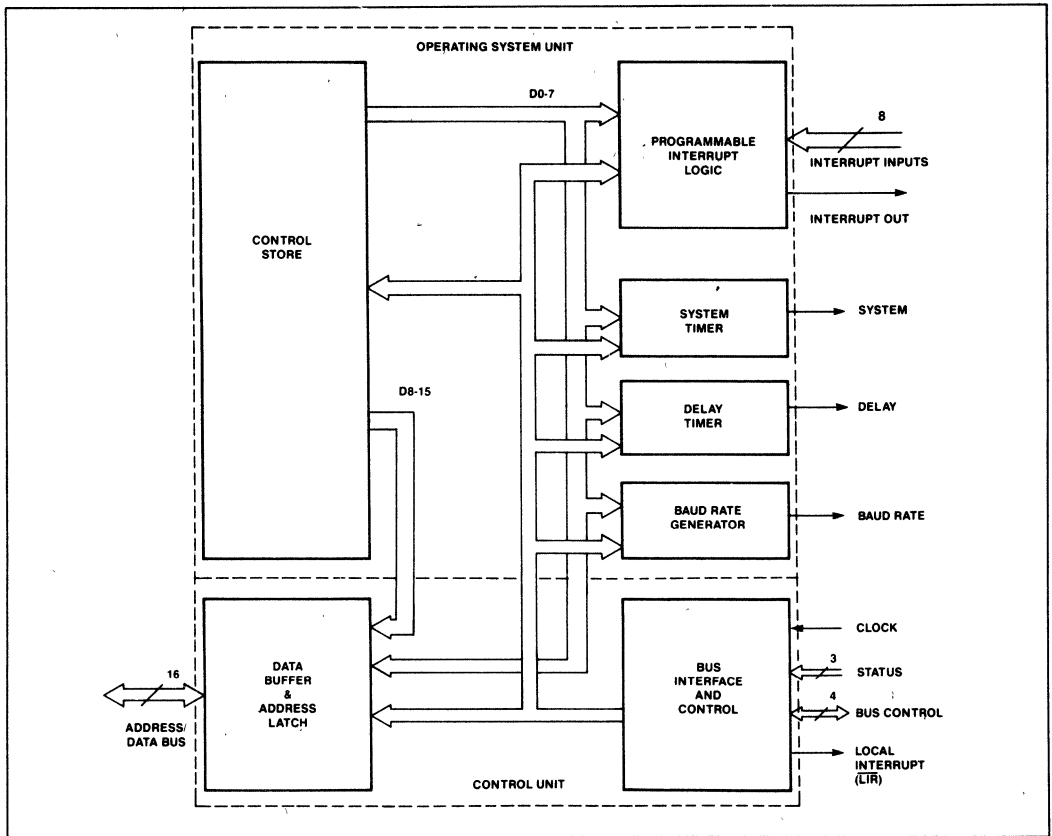


Figure 3. OSF Internal Block Diagram

problem. Their goal is to simplify the design of multi-tasking application systems by providing a well-defined, fully debugged set of operating system primitives implemented directly in the hardware, thereby removing the burden of designing multitasking operating system primitives from the application programmer.

Both the 86/30 and the 88/30 OSPs are two-chip sets consisting of a main processor, an 8086 or 8088 CPU, and the Intel 80130, Operating System Firmware component (OSF) (see Figure 1). The 80130 provides a set of multitasking kernel primitives, kernel control storage, and the additional support hardware, including system timers and interrupt control, required by these primitives. From the application programmer's viewpoint, the OSF extends the base iAPX 86, 88 architecture by providing 35 operating system primitive instructions, and supporting five new system data types, making the OSF a logical and

easy-to-use architectural extension to iAPX 86, 88 system designs.

The OSP Approach

The OSP system data types (SDTs) and primitive instructions allocate, manage and share low-level processor resources in an efficient manner. For example, the OSP implements task context management (managing a task state image consisting of both hardware register set and software control information) for either the basic 86/10 context or the extended 86/20 (8086+8087) numerics context. The OSP manages the entire task state image both while the task is actively executing and while it is inactive. Tasks can be created, put to sleep for specified periods, suspended, executed to perform their functions, and dynamically deleted when their functions are complete.

The Operating System Processors support event-oriented systems designs. Each event may be processed by an individual responding task or along with other closely related events in a common task. External events and interrupts are processed by the OSP interrupt handler primitives using its built-in interrupt controller subsystem as they occur in real-time. The multiple tasks and the multiple events are coordinated by the OSP integral scheduler whose preemptive, priority-based scheduling algorithm and system timers organize and monitor the processing of every task to guarantee that events are processed as they occur in order of relative importance. The 86/30 also provides primitives for intertask communication (by mailboxes) and for mutual exclusion (by regions), essential functions for multitasking applications.

Programming Language Support

Programs for the OSP can be written in ASM 86/88 or PL/M 86/88, Intel's standard system languages for iAPX 86,88 systems.

The Operating System Processor Support Package (iOSP 86) provides an interface library for application programs written in any model of PL/M-86. This library also provides 80130 configuration and initialization support as well as complete user documentation.

OSF PROGRAMMING INTERFACE

The OSF provides 35 operating system kernel primitives which implement multitasking, interrupt management, free memory management, intertask communication and synchronization. Table 4 shows each primitive, and Table 5 gives the execution performance of typical primitives.

OSP primitives are executed by a combination of CPU and OSF (80130) activity. When an OSP primitive is called by an application program task, the iAPX CPU registers and stacks are used to perform the appropriate functions and relay the results to the application programs.

OSP Primitive Calling Sequences

A standard, stack-based, calling sequence is used to invoke the OSF primitives. Before a primitive is called, its operand parameters must be pushed on the task stack. The SI register is loaded with the offset of the last parameter on the stack. The entry code for the primitive is loaded into AX. The primitive invocation call is made with a CPU software interrupt

(Table 4). A representative ASM86 sequence for calling a primitive is shown in Figure 4. In PL/M the OSP programmer uses a call to invoke the primitive.

SAMPLE ASSEMBLY LANGUAGE PRIMITIVE CALL	
PUSH P ₁	:PUSH PARAMETER 1
PUSH P ₂	:PUSH PARAMETER 2
.	: .
.	: .
.	: .
PUSH P _N	:PUSH PARAMETER N
PUSH BP	:STACK CALLING CONVENTION
MOV BP,SP	
LEA SI,SS:NUM_BYTES_PARAM + 2(BP)	:SS:SI POINTS TO FIRST :PARAMETER ON STACK
MOV AX, ENTRY CODE	:AX SETS PRIMITIVE ENTRY CODE
INT 184	:OSF INTERRUPT
OSP PRIMITIVE INVOKED	
POP BP	:POP PARAMETERS
RET NUM_BYTES_PARAM	:CX CONTAINS EXCEPTION CODES :DL CONTAINS PARAMETER NUMBER : THAT CAUSED EXCEPTION (IF : CX IS NON ZERO) :AX CONTAINS WORD RETURN VALUE :ES:BX CONTAINS POINTER : RETURN VALUE

Figure 4. ASM/86 OSP Calling Convention

OSP Functional Description

Each major function of the OSP is described below. These are:

- Job and Task Management
- Interrupt Management
- Free Memory Management
- Intertask Communication
- Intertask Synchronization
- Environmental Control

The system data types (or SDTs) supported by the OSP are capitalized in the description. A short description of each SDT appears in Table 2.

JOB and TASK Management

Each OSP JOB is a controlled environment in which the applications program executes and the OSF system data types reside. Each individual application program is normally a separate OSP JOB, whether it has one initial task (the minimum) or multiple tasks. JOBS partition the system memory into pools. Each memory pool provides the storage areas in which the OSP will allocate TASK state images and other system data types created by the executing TASKs, and free memory for TASK working space. The OSP supports multiple executing TASKs within a JOB by managing the resources used by each, including the CPU registers, NPX registers, stacks, the system data types, and the available free memory space pool.

When a TASK is created, the OSP allocates memory (from the free memory of its JOB environment) for the TASK's stack and data area and initializes the additional TASK attributes such as the TASK priority level and its error handler location. (As an option, the caller of CREATE TASK may assign previously defined stack and data areas to the TASK.) Task priorities are integers between 0 and 255 (the lower the priority number the higher the scheduling priority of the TASK). Generally, priorities up to 128 will be assigned to TASKs which are to process interrupts. Priorities above 128 do not cause interrupts to be disabled, these priorities (129 to 255) are appropriate for non-interrupt TASKs. If an 8087 Numerics Processor Extension is used, the error recovery interrupt level assigned to it will have a higher priority than a TASK executing on it, so that error handling is performed correctly.

EXECUTION STATUS

A TASK has an execution status or execution state. The OSP provides five execution states: RUNNING, READY, ASLEEP, SUSPENDED, and ASLEEP-SUSPENDED.

- A TASK is RUNNING if it has control of the processor.
- A TASK is READY if it is not asleep, suspended, or asleep-suspended. For a TASK to become the running (executing) TASK, it must be the highest priority TASK in the ready state.
- A TASK is ASLEEP if it is waiting for a request to be granted or a timer event to occur. A TASK may put itself into the ASLEEP state.
- A TASK is SUSPENDED if it is placed there by another TASK or if it suspends itself. A TASK may have multiple suspensions, the count of suspensions is managed by the OSP as the TASK suspension depth.
- A TASK is ASLEEP-SUSPENDED if it is both waiting and suspended.

TASK attributes, the CPU register values, and the 8087 register values (if the 8087 is configured into the application) are maintained by the OSP in the TASK state image. Each TASK will have a unique TASK state image.

SCHEDULING

The OSP schedules the processor time among the various TASKs on the basis of priority. A TASK has an execution priority relative to all other TASKs in the system, which the OSP maintains for each TASK in its TASK state image. When a TASK of higher priority than the executing TASK becomes ready to execute,

the OSP switches the control of the processor to the higher priority TASK. First, the OSP saves the outgoing (lower priority) TASK's state including CPU register values in its TASK state image. Then, it restores the CPU registers from the TASK state image of the incoming (higher priority) TASK. Finally, it causes the CPU to start or resume executing the higher priority TASK.

TASK scheduling is performed by the OSP. The OSP's priority-oriented preemptive scheduler determines which TASK executes by comparing their relative priorities. The scheduler insures that the highest priority TASK with a status of READY will execute. A TASK will continue to execute until an interrupt with a higher priority occurs, or until it requests unavailable resources, for which it is willing to wait, or until it makes specific resources available to a higher priority TASK waiting for those resources.

TASKs can become READY by receiving a message, receiving control, receiving an interrupt, or by timing out. The OSP always monitors the status of all the TASKs (and interrupts) in the system. Preemptive scheduling allows the system to be responsive to the external environment while only devoting CPU resources to TASKs with work to be performed.

TIMED WAIT

The OSP timer hardware facilities support timed waits and timeouts. Thus, in many primitives, a TASK can specify the length of time it is prepared to wait for an event to occur, for the desired resources to become available or for a message to be received at a MAILBOX. The timing interval (or System Tick) can be adjusted, with a lower limit of 1 millisecond.

APPLICATION CONTROL OF TASK EXECUTION

Programs may alter TASK execution status and priority dynamically. One TASK may suspend its own execution or the execution of another TASK for a period of time, then resume its execution later. Multiple suspensions are provided. A suspended TASK may be suspended again.

The eight OSP Job and TASK management primitives are:

- | | |
|-------------|--|
| CREATE JOB | Partitions system resources and creates a TASK execution environment. |
| CREATE TASK | Creates a TASK state image. Specifies the location of the TASK code instruction stream, its execution priority, and the other TASK attributes. |

DELETE TASK	Deletes the TASK state image, removes the instruction stream from execution and deallocates stack resources. Does not delete INTERRUPT TASKS.
SUSPEND TASK	Suspends the specified TASK or, if already suspended, increments its suspension depth by one. Execute state is SUSPEND.
RESUME TASK	Decrements the TASK suspension depth by one. If the suspension depth is then zero, the primitive changes the task execution status to READY, or ASLEEP (if ASLEEP/SUSPENDED).
SLEEP	Places the requesting TASK in the ASLEEP state for a specified number of System Ticks. (The TICK interval can be configured down to 1 millisecond.)
SET PRIORITY	Alters the priority of a TASK.

Interrupt Management

The OSP supports up to 256 interrupt levels organized in an interrupt vector, and up to 57 external interrupt sources of which one is the NMI (Non-Maskable Interrupt). The OSP manages each interrupt level independently. The OSF INTERRUPT SUBSYSTEM provides two mechanisms for interrupt management: INTERRUPT HANDLERS and INTERRUPT TASKS. INTERRUPT HANDLERS disable all maskable interrupts and should be used only for servicing interrupts that require little processing time. Within an INTERRUPT HANDLER only certain OSF Interrupt Management primitives (DISABLE, ENTER INTERRUPT, EXIT INTERRUPT, GET LEVEL, SIGNAL INTERRUPT) and basic CPU instructions can be used, other OSP primitives cannot be. The INTERRUPT TASK approach permits all OSP primitives to be issued and masks only lower priority interrupts.

Work flow between an INTERRUPT HANDLER and an INTERRUPT TASK assigned to the same level is regulated with the SIGNAL INTERRUPT and WAIT INTERRUPT primitives. The flow is asynchronous. When an INTERRUPT HANDLER signals an INTERRUPT TASK, the INTERRUPT HANDLER becomes immediately available to process another interrupt. The number of interrupts (specified for the level) the

INTERRUPT HANDLER can queue for the INTERRUPT TASK can be limited to the value specified in the SET INTERRUPT primitive. When the INTERRUPT TASK is finished processing, it issues a WAIT INTERRUPT primitive, and is immediately ready to process the queue of interrupts that the INTERRUPT HANDLER has built with repeated SIGNAL INTERRUPT primitives while the INTERRUPT TASK was processing. If there were no interrupts at the level, the queue is empty and the INTERRUPT TASK is SUSPENDED. See the Example (Figure 5) and Figures 6 and 7.

OSP external INTERRUPT LEVELS are directly related to internal TASK scheduling priorities. The OSP maintains a single list of priorities including both tasks and INTERRUPT LEVELS. The priority of the executing TASK automatically determines which interrupts are masked. Interrupts are managed by INTERRUPT LEVEL number. The OSP supports eight levels directly and may be extended by means of slave 8259As to a total of 57.

The nine Interrupt Management OSP primitives are:

DISABLE	Disables an external INTERRUPT LEVEL.
ENABLE	Enables an external INTERRUPT LEVEL.
ENTER INTERRUPT	Gives an Interrupt Handler its own data segment, separate from the data segment of the interrupted task.
EXIT INTERRUPT	Performs an "END of INTERRUPT" operation. Used by an INTERRUPT HANDLER which does not invoke an INTERRUPT TASK. Reenables interrupts, when the INTERRUPT HANDLER gives up control.
GET LEVEL	Returns the interrupt level number of the executing INTERRUPT HANDLER.
RESET INTERRUPT	Cancels the previous assignment made to an interrupt level by SET INTERRUPT primitive request. If an INTERRUPT TASK has been assigned, it is also deleted. The interrupt level is disabled.
SET INTERRUPT	Assigns an INTERRUPT HANDLER to an interrupt level and, optionally, an INTERRUPT TASK.

```

/* CODE EXAMPLE A INTERRUPT TASK TO KEEP TRACK OF TIME-OF-DAY
DECLARE SECONDSCOUNT BYTE,
MINUTESCOUNT BYTE,
HOURSACCOUNT BYTE;
TIMESTASK: PROCEDURE;
DECLARE TIMESEXCPTSCODE WORD;
AC$CYCLE$COUNT=0;
CALL RQ$SET$INTERRUPT(AC$INTERRUPT$LEVEL, 01H),
@AC$HANDLER,@TIMESEXCPTSCODE);
CALL RQ$RESUME$TASK(INIT$TASK$TOKEN,@TIMESEXCPTSCODE);
DO HOURSACCOUNT=0 TO 23;
DO MINUTESCOUNT=0 TO 59;
DO SECONDSCOUNT=0 TO 59;
CALL RQ$WAIT$INTERRUPT(AC$INTERRUPT$LEVEL,
@TIMESEXCPTSCODE);
IF SECONDSCOUNT MOD 5=0
THEN CALL PROTECTED$CRT$OUT(BEL);
END; /* SECOND LOOP */
END; /* MINUTE LOOP */
END; /* HOUR LOOP */
CALL RQ$SET$INTERRUPT(AC$INTERRUPT$LEVEL, @TIMESEXCPTSCODE);
END TIMESTASK;
/* CODE EXAMPLE B INTERRUPT HANDLER TO SUBDIVIDE A.C. SIGNAL BY 60. */
DECLARE AC$CYCLE$COUNT BYTE;
AC$HANDLER: PROCEDURE INTERRUPT 59;
DECLARE AC$EXCPTSCODE WORD;
AC$CYCLE$COUNT=AC$CYCLE$COUNT +1;
IF AC$CYCLE$COUNT>=60 THEN DO;
AC$CYCLE$COUNT=0;
CALL RQ$SIGNAL$INTERRUPT(AC$INTERRUPT$LEVEL,@AC$EXCPTSCODE);
END;
END AC$HANDLER;

```

Figure 5. OSP Examples

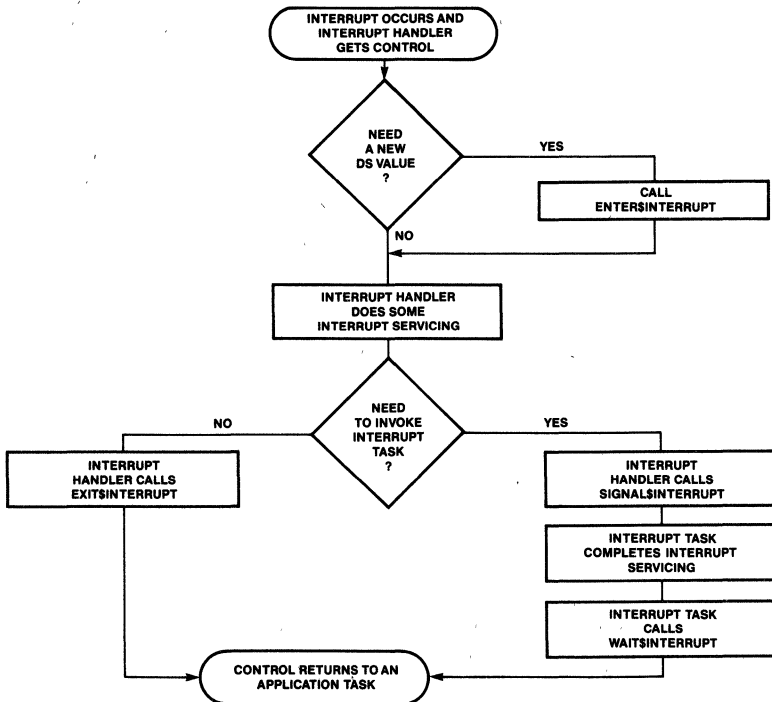


Figure 6. Interrupt Handling Flowchart

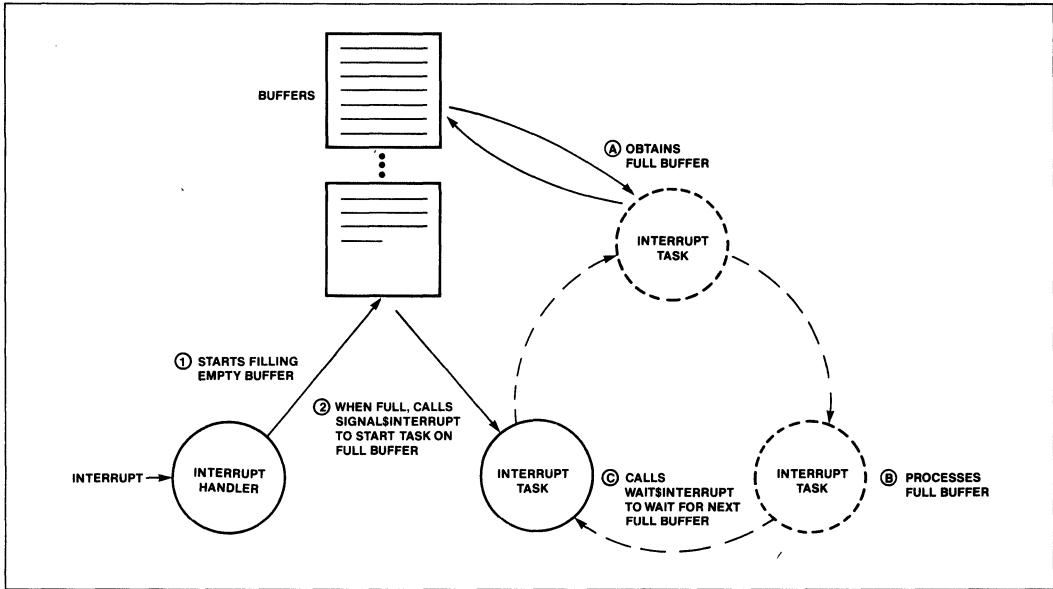


Figure 7. Multiple Buffer Example

SIGNAL INTERRUPT Used by an INTERRUPT HANDLER to activate an Interrupt Task.

WAIT INTERRUPT Suspends the calling Interrupt Task until the INTERRUPT HANDLER performs a SIGNAL INTERRUPT to invoke it. If a SIGNAL INTERRUPT for the task has occurred, it is processed.

FREE MEMORY MANAGEMENT

The OSP Free Memory Manager manages the memory pool which is allocated to each JOB for its execution needs. (The CREATE JOB primitive allocates the new JOB's memory pool from the memory pool of the parent JOB.) The memory pool is part of the JOB resources but is not yet allocated between the tasks of the JOB. When a TASK, MAILBOX, or REGION system data type structure is created within that JOB, the OSP implicitly allocates memory for it from the JOB's memory pool, so that a separate call to allocate memory is not required. OSP primitives that use free memory management implicitly include CREATE JOB, CREATE TASK, DELETE TASK, CREATE MAILBOX, DELETE MAILBOX, CREATE REGION, and DELETE REGION. The

CREATE SEGMENT primitive explicitly allocates a memory area when one is needed by the TASK. For example, a TASK may explicitly allocate a SEGMENT for use as a memory buffer. The SEGMENT length can be any multiple of 16 bytes between 16 bytes and 64K bytes in length. The programmer may specify any number of bytes from 1 byte to 64 KB, the OSP will transparently round the value up to the appropriate segment size.

The two explicit memory allocation/deallocation primitives are:

CREATE SEGMENT Allocates a SEGMENT of specified length (in 16-byte-long paragraphs) from the JOB Memory Pool.

DELETE SEGMENT Deallocates the SEGMENT's memory area, and returns it to the JOB memory pool.

Intertask Communication

The OSP has built-in intertask synchronization and communication, permitting TASKs to pass and share information with each other. OSP MAILBOXes contain controlled handshaking facilities which guarantee that a *complete* message will always be sent from a sending TASK to a receiving TASK. Each MAILBOX consists of two interlocked queues, one of TASKs

and the other of Messages. Four OSP primitives for intertask synchronization and communication are provided:

CREATE MAILBOX	Creates intertask message exchange.
DELETE MAILBOX	Deletes an intertask message exchange.
RECEIVE MESSAGE	Calling TASK receives a message from the MAILBOX.
SEND MESSAGE	Calling TASK sends a message to the MAILBOX.

The CREATE MAILBOX primitive allocates a MAILBOX for use as an information exchange between TASKs. The OSP will post information at the MAILBOX in a FIFO (First-In First-Out) manner when a SEND MESSAGE instruction is issued. Similarly, a message is retrieved by the OSP if a TASK issues a RECEIVE MESSAGE primitive. The TASK which creates the MAILBOX may make it available to other TASKs to use.

If no message is available, the TASK attempting to receive a message may choose to wait for one or continue executing.

The queue management method for the task queue (FIFO or PRIORITY) determines which TASK in the MAILBOX TASK queue will receive a message from the MAILBOX. The method is specified in the CREATE MAILBOX primitive.

Intertask Synchronization and Mutual Exclusion

Mutual exclusion is essential to multiprogramming and multiprocessing systems. The REGION system data type implements mutual exclusion. A REGION is represented by a queue of TASKs waiting to use a resource which must be accessed by only one TASK at a time. The OSP provides primitives to use REGIONs to manage mutually exclusive data and resources. Both critical code sections and shared data structures can be protected by these primitives from simultaneous use by more than one task. REGIONs support both FIFO (First-In First-Out) or Priority queueing disciplines for the TASKs seeking to enter the REGION. The REGION SDT can also be used to implement software locks.

Multiple REGIONs are allowed, and are automatically exited in the reverse order of entry. While in a REGION, a TASK cannot be suspended by itself or any other TASK, and thereby avoids deadlock.

There are five OSP primitives for mutual exclusion:

CREATE REGION	Create a REGION (lock).
SEND CONTROL	Give up the REGION.
ACCEPT CONTROL	Request the REGION, but do not wait if it is not available.
RECEIVE CONTROL	Request a REGION, wait if not immediately available.
DELETE REGION	Delete a REGION.

The OSP also provides dynamic priority adjustment for TASKs within priority REGIONs: If a higher-priority TASK issues a RECEIVE CONTROL primitive, while a (lower-priority) TASK has the use of the same REGION, the lower-priority TASK will be transparently, and temporarily, elevated to the waiting TASK's priority until it relinquishes the REGION via SEND CONTROL. At that point, since it is no longer using the critical resource, the TASK will have its normal priority restored.

OSP Control Facilities

The OSP also includes system primitives that provide both control and customization capabilities to a multitasking system. These primitives are used to control the deletion of SDTs and the recovery of free memory in a system, to allow interrogation of operating system status, and to provide uniform means of adding user SDTs and type managers.

DELETION CONTROL

Deletion of each OSP system data type is explicitly controlled by the applications programmer by setting a deletion attribute for that structure. For example, if a SEGMENT is to be kept in memory until DMA activity is completed, its deletion attribute should be disabled. Each TASK, MAILBOX, REGION, and SEGMENT SDT is created with its deletion attribute enabled (i.e., they may be deleted). Two OSP primitives control the deletion attribute: ENABLE DELETION and DISABLE DELETION.

ENVIRONMENTAL CONTROL

The OSP provides inquiry and control operations which help the user interrogate the application environment and implement flexible exception handling. These features aid in run-time decision making and in application error processing and recovery. There are five OSP environmental control primitives.

OS EXTENSIONS

The OSP architecture is defined to allow new user-defined System Data Types and the primitives to manipulate them to be added to OSP capabilities

provided by the built-in System Data Types. The type managers created for the user-defined SDTs are called user OS extensions and are installed in the system by the SET OS EXTENSION primitive. Once installed, the functions of the type manager may be invoked with user primitives conforming to the OSP interface. For well-structured extended architectures, each OS extension should support a separate user-defined system data type, and every OS extension should provide the same calling sequence and program interface for the user as is provided for a built-in SDT. The type manager for the extension would be written to suit the needs of the application. OSP interrupt vector entries (224–255) are reserved for user OS extensions and are not used by the OSP. After assigning an interrupt number to the extension, the extension user may then call it with the standard OSP call sequence (Figure 4), and the unique software interrupt number assigned to the extension.

ENABLE DELETION	Allows a specific SEGMENT, TASK, MAILBOX, or REGION SDT to be deleted.
DISABLE DELETION	Prevents a specific SEGMENT, TASK, MAILBOX, or REGION SDT from being deleted.
GET TYPE	Given a token for an instance of a system data type, returns the type code.
GET TASK TOKENS	Returns to the caller information about the current task environment.
GET EXCEPTION HANDLER	Returns information about the calling TASK's current information handler: its address, and when it is used.
SET EXCEPTION HANDLER	Provides the address and usage of an exception handler for a TASK.
SET OS EXTENSION	Modifies one of the interrupt vector entries reserved for OS extensions (224–255) to point to a user OS extension procedure.
SIGNAL EXCEPTION	For use in OS extension error processing.

EXCEPTION HANDLING

The OSP supports exception handlers. These are similar to CPU exception handlers such as OVERFLOW and ILLEGAL OPERATION. Their purpose is to

allow the OSP primitives to report parameter errors in primitive calls, and errors in primitive usage. Exception handling procedures are flexible and can be individually programmed by the application. In general, an exception handler if called will perform one or more of the following functions:

- Log the Error.
- Delete/Suspend the Task that caused the exception.
- Ignore the error, presumably because it is not serious.

An EXCEPTION HANDLER is written as a procedure. If PLM/86 is used, the "compact," "medium" or "large" model of computation should be specified for the compilation of the program. The mode in which the EXCEPTION HANDLER operates may be specified in the SET EXCEPTION HANDLER primitive. The return information from a primitive call is shown in Figure 4. CX is used to return standard system error conditions. Table 7 shows a list of these conditions, using the *default* EXCEPTION HANDLER of the OSP.

HARDWARE DESCRIPTION

The 80130 operates in a closely coupled mode with the iAPX 86/10 or 88/10 CPU. The 80130 resides on the CPU local multiplexed bus (Figure 8). The main processor is always configured for maximum mode operation. The 80130 automatically selects between its 88/30 and 86/30 operating modes.

The 80130 used in the 86/30 configuration, as shown in Figure 8 (or a similar 88/30 configuration), operates at both 5 and 8 MHz without requiring processor wait states. Wait state memories are fully supported, however. The 80130 may be configured with both an 8087 NPX and an 8089 IOP, and provides full context control over the 8087.

The 80130 (shown in Figure 3) is internally divided into a control unit (CU) and operating system unit (OSU). The OSU contains facilities for OSP kernel support including the system timers for scheduling and timing waits, and the interrupt controller for interrupt management support.

iAPX 86/30, iAPX 88/30 System Configuration

The 80130 is both I/O and memory mapped to the local CPU bus. The CPU's status S0/S2/ is decoded along with IOCS/ (with BHE and AD₃–AD₀) or MEMCS/ (with AD₁₃–AD₀). The pins are internally latched. See Table 1 for the decoding of these lines.

Memory Mapping

Address lines A_{19} – A_{14} can be used to form MEMCS/ since the 80130's memory-mapped portion is aligned along a 16K-byte boundary. The 80130 can reside on any 16K-byte boundary excluding the highest (FC000H–FFFFH) and lowest (00000H–003FFH). The 80130 control store code is position-independent except as limited above, in order to make it compatible with many decoding logic designs. AD_{13} – AD_0 are decoded by the 80130's kernel control store.

I/O Mapping

The I/O-mapped portion of the 80130 must be aligned along a 16-byte boundary. Address lines A_{15} – A_4 should be used to form IOCS/.

System Performance

The approximate performance of representative OSP primitives is given in Table 5. These times are shown for a typical iAPX 86/30 implementation with an 8 MHz clock. These execution times are very comparable to the execution times of similar functions in minicomputers (where available) and are an order of magnitude faster than previous generation microprocessors.

Initialization

Both application system initialization and OSP-specific initialization/configuration are required to use the OSP. Configuration is based on a "database" provided by the user to the iOSP 86 support package. The OSP-specific initialization and configuration information area is assigned to a user memory address adjacent to the 80130's memory-mapped location. (See Application Note 130 for further details.) The configuration data defines whether 8087 support is configured in the system, specifies if slave 8259A interrupt controllers are used in addition to the 80130, and sets the operating system time base (Tick Interval). Also located in the configuration area are the exception handler control parameters, the address location of the (separate) application system configuration area and the OSP extensions in use. The OSP application system configuration area may be located anywhere in the user memory and must include the starting address of the application instruction code to be executed, plus the locations of the RAM memory blocks to be managed by the OSP free memory manager. Complete application system support and the required 80130 configuration support are provided by the iAPX 86/30 and iAPX 88/30 OPERATING SYSTEM PROCESSOR SUPPORT PACKAGE (iOSP 86).

RAM Requirements

The OSP manages its own interrupt vector, which is assigned to low RAM memory. Working RAM storage is required as stack space and data area. The memory space must be allocated in user RAM.

OSP interrupt vector memory locations 0H–3FFH must be RAM based. The OSP requires 2 bytes of allocated RAM. The processor working storage is dynamically allocated from free memory. Approximately 300 bytes of stack should be allocated for each OSP task.

TYPICAL SYSTEM CONFIGURATION

Figure 8 shows the processing cluster of a "typical" iAPX 86/30 or iAPX 88/30 OSP system. Not shown are subsystems likely to vary with the application. The configuration includes an 8086 (or 8088) operating in maximum mode, an 8284A clock generator and an 8288 system controller. Note that the 80130 is located on the CPU side of any latches or transceivers. See Intel Application Note 130 for further details on configuration.

OSP Timers

The OSP Timers are connected to the lower half of the data bus and are addressed at even addresses. The timers are read as two successive bytes, always LSB followed by MSB. The MSB is always latched on a read operation and remains latched until read. Timers are not gatable.

Baud Rate Generator

The baud rate generator is 8254 compatible (square wave mode 3). Its output, BAUD, is initially high and remains high until the Count Register is loaded. The first falling edge of the clock after the Count Register is loaded causes the transfer of the internal counter to the Count Register. The output stays high for $N/2$ [$(N+1)/2$ if N is odd] and then goes low for $N/2$ [$(N-1)/2$ if N is odd]. On the falling edge of the clock which signifies the final count for the output in low state, the output returns to high state and the Count Register is transferred to the internal counter. The whole process is then repeated. Baud Rates are shown in Table 6.

The baud rate generator is located at OCH (12), relative to the 16-byte boundary in the I/O space in which the 80130 component is located ("OSF" in the following example), the timer control word is located at

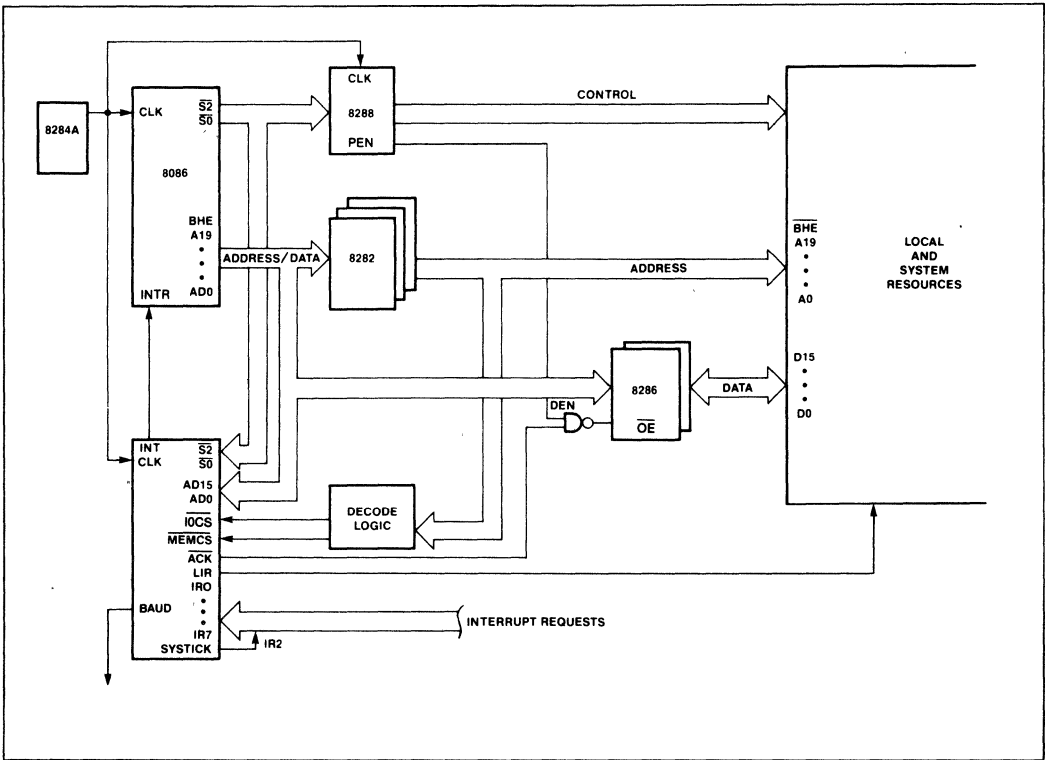


Figure 8. Typical OSP Configuration

relative address, 0EH(14). Timers are addressed with IOCS=0. Timers 0 and 1 are assigned to the use by the OSP, and should not be altered by the user.

For most baud-rate generator applications, the command byte

0B6H Read/Write Baud-Rate Delay Value

will be used. A typical sequence to set a baud rate of 9600 using a count value of 52 follows (see Table 6):

```
MOV AX,0B6H ;Prepare to Write Delay to
              Timer 3.
OUT OSF+14,AX ;Control Word.
MOV AX, 52
OUT OSF+12,AL ;LSB written first
XCHG AL,AH
OUT OSF+12,AL ;MSB written after.
```

The 80130 timers are subset compatible with 8254 timers.

Interrupt Controller

The Programmable Interrupt Controller (PIC), is also an integral unit of the 80130. Its eight input pins handle eight vectored priority interrupts. One of these pins must be used for the SYSTICK time function in timing waits, using an external connection as shown. During the 80130 initialization and configuration sequence, each 80130 interrupt pin is individually programmed as either level or edge sensitive. External slave 8259A interrupt controllers can be used to expand the total number of OSP external interrupts to 57.

In addition to standard PIC functions, 80130 PIC unit has an $\overline{\text{LIR}}$ output signal, which when low indicates an interrupt acknowledge cycle. $\overline{\text{LIR}}=0$ is provided to control the 8289 Bus Arbiter SYSB/ $\overline{\text{RESB}}$ pin. This will avoid the need of requesting the system bus to acknowledge local bus non-slave interrupts. The user defines the interrupt system as part of the configuration.

INTERRUPT SEQUENCE

The OSP interrupt sequence is as follows:

1. One or more of the interrupts is set by a low-to-high transition on edge-sensitive IR inputs or by a high input on level-sensitive IR inputs.
2. The 80130 evaluates these requests, and sends an INT to the CPU, if appropriate.
3. The CPU acknowledges the INT and responds with an interrupt acknowledge cycle which is encoded in S_2-S_0 .
4. Upon receiving the first interrupt acknowledge from the CPU, the highest-priority interrupt is set by the 80130 and the corresponding edge detect latch is reset. The 80130 does not drive the address/data bus during this bus cycle but does acknowledge the cycle by making $\overline{ACK}=0$ and sending the LIR value for the IR input being acknowledged.
5. The CPU will then initiate a second interrupt acknowledge cycle. During this cycle, the 80130 will supply the cascade address of the interrupting input at T_1 on the bus and also release an 8-bit pointer onto the bus if appropriate, where it is read by the CPU. If the 80130 does supply the pointer, then \overline{ACK} will be low for the cycle. This cycle also has the value LIR for the IR input being acknowledged.
6. This completes the interrupt cycle. The ISR bit remains set until an appropriate EXIT INTERRUPT primitive (EOI command) is called at the end of the Interrupt Handler.

OSP APPLICATION EXAMPLE

Figure 5 shows an application of the OSP primitives to keep track of time of day in a simplified example. The system design uses a 60 Hz A.C. signal as a time base. The power supply provides a TTL-compatible

signal which drives one of 80130 edge-triggered interrupt request pins once each A.C. cycle. The Interrupt Handler responds to the interrupts, keeping track of one second's A.C. cycles. The Interrupt Task counts the seconds and after a day deletes itself. In typical systems it might perform a data logging operation once each day. The Interrupt Handler and Interrupt Task are written as separate modular programs.

The Interrupt Handler will actually service interrupt 59 when it occurs. It simply counts each interrupt, and at a count of 60 performs a SIGNAL INTERRUPT to notify the Interrupt Task that a second has elapsed. The Interrupt Handler (ACS HANDLER) was assigned to this level by the SET INTERRUPT primitive. After doing this, the Interrupt Task performed the Primitive RESUME TASK to resume the application task (INITS TASKS TOKEN).

The main body of the task is the counting loop. The Interrupt Task is signaled by the SIGNAL INTERRUPT primitive in the Interrupt Handler (at interrupt level ACS INTERRUPTS LEVEL). When the task is signaled by the Interrupt Handler it will execute the loop exactly one time, increasing the time count variables. Then it will execute the WAIT INTERRUPT primitive, and wait until awakened by the Interrupt Handler. Normally, the task will now wait some period of time for the next signal. However, since the interface between the Handler and the Task is asynchronous, the handler may have already queued the interrupt for servicing, the writer of the task does not have to worry about this possibility.

At the end of the day, the task will exit the loop and execute RESET INTERRUPT, which disables the interrupt level, and deletes the interrupt task. The OSP now reclaims the memory used by the Task and schedules another task. If an exception occurs, the coded value for the exception is available in TIMES EXCEPTS CODE after the execution of the primitive.

A typical PL/M-86 calling sequence is illustrated by the call to RESET INTERRUPT shown in Figure 5.

Table 2. OSP System Data Type Summary

Job	Jobs are the means of organizing the program environment and resources. An application consists of one or more jobs. Each iAPX 86/30 system data type is contained in some job. Jobs are independent of each other, but they may share access to resources. Each job has one or more tasks, one of which is an initial task. Jobs are given pools of memory, and they may create subordinate offspring jobs, which may borrow memory from their parents.
Task	<p>Tasks are the means by which computations are accomplished. A task is an instruction stream with its own execution stack and private data. Each task is part of a job and is restricted to the resources provided by its job. Tasks may perform general interrupt handling as well as other computational functions. Each task has a set of attributes, which is maintained for it by the iAPX 86/30, which characterize its status. These attributes are:</p> <ul style="list-style-type: none"> its containing job its register context its priority (0-255) its execution state (asleep, suspended, ready, running, asleep/suspended). its suspension depth its user-selected exception handler its optional 8087 extended task state
Segment	Segments are the units of memory allocation. A segment is a physically contiguous sequence of 16-byte, 8086 paragraph-length, units. Segments are created dynamically from the free memory space of a Job as one of its Tasks requests memory for its use. A segment is deleted when it is no longer needed. The iAPX 86/30 maintains and manages free memory in an orderly fashion, it obtains memory space from the pool assigned to the containing job of the requesting task and returns the space to the job memory pool (or the parent job pool) when it is no longer needed. It does not allocate memory to create a segment if sufficient free memory is not available to it, in that case it returns an error exception code.
Mailbox	Mailboxes are the means of intertask communication. Mailboxes are used by tasks to send and receive message segments. The iAPX 86/30 creates and manages two queues for each mailbox. One of these queues contains message segments sent to the mailbox but not yet received by any task. The other mailbox queue consists of tasks that are waiting to receive messages. The iAPX 86/30 operation assures that waiting tasks receive messages as soon as messages are available. Thus at any moment one or possibly both of two mailbox queues will be empty.
Region	Regions are the means of serialization and mutual exclusion. Regions are familiar as "critical code regions." The iAPX 86/30 region data type consists of a queue of tasks. Each task waits to execute in mutually exclusive code or to access a shared data region, for example to update a file record.
Tokens	The OSP interface makes use of a 16-bit TOKEN data type to identify individual OSF data structures. Each of these (each instance) has its own unique TOKEN. When a primitive is called, it is passed the TOKENs of the data structures on which it will operate.

Table 3. System Data Type Codes and Attributes

S.D.T.	Code	Attributes
Jobs	1	Tasks Memory Pool S.D.T. Directory
Tasks	2	Priority Stack Code State Exception Handler
Mailboxes	3	Queue of S.D.T.s (generally segments) Queue of Tasks waiting for S.D.T.s
Region	5	Queue of Tasks waiting for mutually exclusive code or data
Segments	6	Buffer Length

Table 4. OSP Primitives

Class	OSP Primitive	Interrupt Number	Entry Code in AX	Parameters On Caller's Stack
J O B	CREATE JOB	184	0100H	*See 80130 User Manual
T A S K	CREATE TASK	184	0200H	Priority, IP Ptr, Data Segment, Stack Seg, Stack Size Task Information, ExcptPtr
	DELETE TASK	184	0201H	TASK, ExcptPtr
	SUSPEND TASK	184	0202H	TASK, ExcptPtr
	RESUME TASK	184	0203H	TASK, ExcptPtr
	SET PRIORITY	184	0209H	TASK, Priority, ExcptPtr
	SLEEP	184	0204H	Time Limit, ExcptPtr
I N T E R R U P T	DISABLE	190	0705H	Level, ExcptPtr
	ENABLE	184	0704H	Level #, ExcptPtr
	ENTER INTERRUPT	184	0703H	Level #, ExcptPtr
	EXIT INTERRUPT	186	NONE	Level #, ExcptPtr
	GET LEVEL	188	0702H	Level #, ExcptPtr
	RESET INTERRUPT	184	0706H	Level #, ExcptPtr
	SET INTERRUPT	184	0701H	Level, Interrupt Task Flag Interrupt Handler Ptr, Interrupt Handler DataSeg ExcptPtr
	SIGNAL INTERRUPT WAIT INTERRUPT	185 187	NONE NONE	Level, ExcptPtr Level, ExcptPtr
S E G M E N T	CREATE SEGMENT	184	0600H	Size, ExcptPtr
	DELETE SEGMENT	184	0603H	SEGMENT, ExcptPtr

Table 4. OSP Primitives (Continued)

Class	OSP Primitive	Interrupt Number	Entry Code in AX	Parameters On Caller's Stack
M A I L B O X	CREATE MAILBOX	184	0300H	Mailbox flags, ExcptPtr MAILBOX, ExcptPtr MAILBOX, Time Limit ResponsePtr, ExcptPtr MAILBOX,Message Response, ExcptPtr
	DELETE MAILBOX	184	0301H	
	RECEIVE MESSAGE	184	0303H	
	SEND MESSAGE	184	0302H	
R E G I O N	ACCEPT CONTROL	184	0504H	REGION, ExcptPtr Region Flags, ExcptPtr REGION, ExcptPtr REGION, ExcptPtr ExcptPtr
	CREATE REGION	184	0500H	
	DELETE REGION	184	0501H	
	RECEIVE CONTROL	184	0503H	
	SEND CONTROL	184	0502H	
E N V I R O N M E N T A L	DISABLE DELETION	184	0001H	TOKEN, ExcptPtr TOKEN, ExcptPtr Ptr, ExcptPtr TOKEN, ExcptPtr Request, ExcptPtr Ptr, ExcptPtr Code, InstPtr, ExcptPtr Exception Code, Parameter Number, StackPtr, 0, 0, ExcptPtr
	ENABLE DELETION	184	0002H	
	GET EXCEPTION HANDLER	184	0800H	
	GET TYPE	184	0000H	
	GET TASK TOKENS	184	0206H	
	SET EXCEPTION HANDLER	184	0801H	
	SET OS EXTENSION SIGNAL	184	0700H	
	EXCEPTION	184	0802H	

NOTES:

All parameters are pushed onto the OSP stack. Each parameter is one word. See Figure 3 for Call Sequence.

Explanation of the Symbols

JOB	OSP JOB SDT Token
TASK	OSP TASK SDT Token
REGION	OSP REGION SDT Token
MAILBOX	OSP MAILBOX SDT Token
SEGMENT	OSP SEGMENT SDT Token
TOKEN	Any SDT Token
Level	Interrupt Level Number
ExcptPtr	Pointer to Exception Code
Message	Message Token
Ptr	Pointer to Code, Stack etc. Address
Seg	Value Loaded into appropriate Segment Register
---	Value Parameter.

Table 5. OSP Primitive Performance Examples

Datatype Class	Primitive Execution Speed* (microseconds)	
JOB	CREATE JOB	2950
TASK	CREATE TASK (no preemption)	1360
SEGMENT	CREATE SEGMENT	700
MAILBOX	SEND MESSAGE (with task switch)	475
	SEND MESSAGE (no task switch)	265
	RECEIVE MESSAGE (task waiting)	540
	RECEIVE MESSAGE (message waiting)	260
REGION	SEND CONTROL	170
	RECEIVE CONTROL	205

*8 MHz iAPX 86/30 OSP Configuration.

Table 6. Baud Rate Count Values (16X)

Baud Rate	8 MHz Count Value	5 MHz Count Value
300	1667	1042
600	833	521
1200	417	260
2400	208	130
4800	104	65
9600	52	33

Table 7a. Mnemonic Codes for Unavoidable Exceptions

E\$OK	Exception Code Value = 0 the operation was successful
E\$TIME	Exception Code Value = 1 the specified time limit expired before completion of the operations was possible
E\$MEM	Exception Code Value = 2 insufficient nucleus memory is available to satisfy the request
E\$BUSY	Exception Code Value = 3 specified region is currently busy
E\$LIMIT	Exception Code Value = 4 attempted violation of a job, semaphore, or system limit
E\$CONTEXT	Exception Code Value = 5 the primitive was called in an illegal context (e.g., call to enable for an already enabled interrupt)
E\$EXIST	Exception Code Value = 6 a token argument does not currently refer to any object; note that the object could have been deleted at any time by its owner
E\$STATE	Exception Code Value = 7 attempted illegal state transition by a task
E\$NOT\$CONFIGURED	Exception Code Value = 8 the primitive called is not configured in this system
E\$INTERRUPT\$SATURATION	Exception Code Value = 9 The interrupt task on the requested level has reached its user specified saturation point for interrupt service requests. No further interrupts will be allowed on the level until the interrupt task executes a WAIT\$INTERRUPT. (This error is only returned, in line, to interrupt handlers.)
E\$INTERRUPT\$OVERFLOW	Exception Code Value = 10 The interrupt task on the requested level previously reached its saturation point and caused an E\$INTERRUPT\$SATURATION condition. It subsequently executed an ENABLE allowing further interrupts to come in and has received another SIGNAL\$INTERRUPTcall, bringing it over its specified saturation point for interrupt service requests. (This error is only returned, in line, to interrupt handlers).

Table 7b. Mnemonic Codes for Avoidable Exceptions

E\$ZERO\$DIVIDE	Exception Code Value = 8000H divide by zero interrupt occurred
E\$OVERFLOW	Exception Code Value = 8001H overflow interrupt occurred
E\$TYPE	Exception Code Value = 8002H a token argument referred to an object tha was not of required type
E\$BOUNDS	Exception Code Value = 8003H an offset argument is out of segment bounds
E\$PARAM	Exception Code Value = 8004H a (non-token,non-offset) argument has an illegal value
E\$BAD\$CALL	Exception Code Value = 8005H an entry code for which there is no corresponding primitive was passed
E\$ARRAY\$BOUNDS = 8006H	Hardware or Language has detected an array overflow
E\$NDP\$ERROR	Exception Code Value = 8007H an 8087 (Numeric data Processor) error has been detected; (the 8087 status information is contained in a parameter to the exception handler)

ABSOLUTE MAXIMUM RATINGS*

Ambient Temperature Under Bins 0°C to 70°C
 Storage Temperature -65°C to 150°C
 Voltage on Any Pin With
 Respect to Ground -1.0V to +7V
 Power Dissipation 1.0 Watts

**NOTICE: Stresses above those listed under Absolute Maximum Ratings may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended period may affect device reliability.*

D.C. CHARACTERISTICS ($T_A = 0^\circ\text{C to } 70^\circ\text{C}$, $V_{CC} = 4.5 \text{ to } 5.5\text{V}$)

Symbol	Parameter	Min.	Max.	Units	Test Conditions
V_{IL}	Input Low Voltage	- 0.5	0.8	V	
V_{IH}	Input High Voltage	2.0	$V_{CC} + .5$	V	
V_{OL}	Output Low Voltage		0.45	V	$I_{OL} = 2\text{mA}$
V_{OH}	Output High Voltage	2.4		V	$I_{OH} = -400\mu\text{A}$
I_{CC}	Power Supply Current		200	mA	$T_A = 25^\circ\text{C}$
I_{LI}	Input Leakage Current		10	μA	$0 < V_{IN} < V_{CC}$
I_{LR}	IR Input Load Current		10 -300	μA μA	$V_{IN} = V_{CC}$ $V_{IN} = 0$
I_{LO}	Output Leakage Current		10	μA	$.45 = V_{IN} = V_{CC}$
V_{CLI}	Clock Input Low		0.6	V	
V_{CHI}	Clock Input High	3.9		V	
C_{IN}	Input Capacitance		10	pF	
C_{IO}	I/O Capacitance		15	pF	
I_{CI}	Clock Input Leakage Current		10 150 10	μA μA μA	$V_{IN} = V_{CC}$ $V_{IN} = 2.5\text{V}$ $V_{IN} = 0\text{V}$

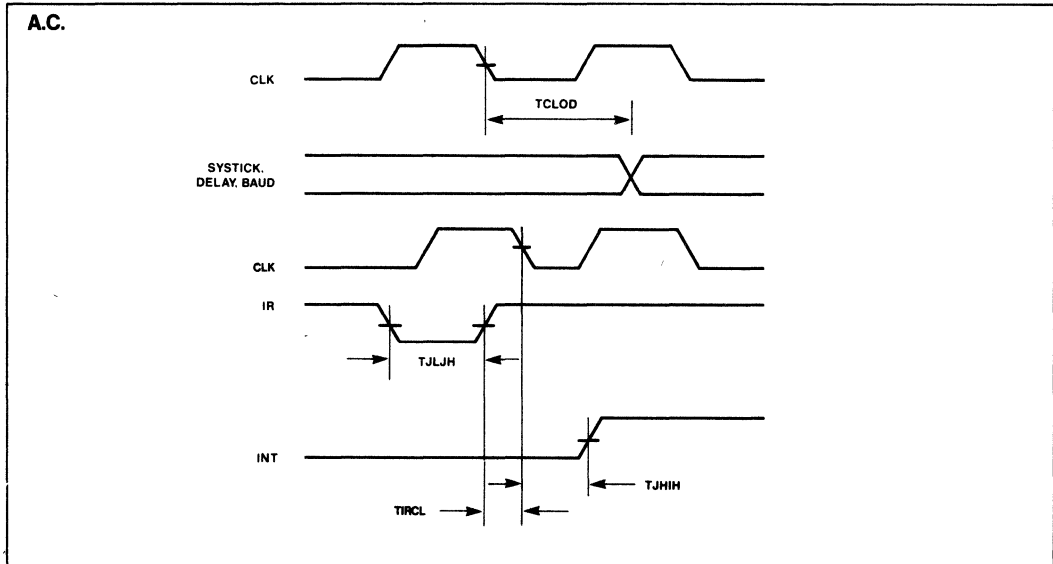
A.C. CHARACTERISTICS ($T_A = 0\text{-}70^\circ\text{C}$, $V_{CC} = 4.5\text{-}5.5 \text{ Volt}$, $V_{SS} = \text{Ground}$)

Symbol	Parameter	80130		80130-2		Units	Test Conditions
		Min.	Max.	Min.	Max.		
T_{CLCL}	CLK Cycle Period	200	-	125	-	ns	
T_{CLCH}	CLK Low Time	90	-	55	-	ns	
T_{CHCL}	CLK High Time	69	2000	44	2000	ns	
T_{SVCH}	Status Active Setup Time	80	-	65	-	ns	
T_{CHSV}	Status Inactive Hold Time	10	-	10	-	ns	
T_{SHCL}	Status Inactive Setup Time	55	-	55	-	ns	
T_{CLSH}	Status Active Hold Time	10	-	10	-	ns	
T_{ASCH}	Address Valid Setup Time	8	-	8	-	ns	
T_{CLAH}	Address Hold Time	10	-	10	-	ns	
T_{CSCL}	Chip Select Setup Time	20	-	20	-	ns	
T_{CHCS}	Chip Select Hold Time	0	-	0	-	ns	
T_{DSCL}	Write Data Setup Time	80	-	60	-	ns	
T_{CHDH}	Write Data Hold Time	10	-	10	-	ns	
T_{JLJH}	IR Low Time	100	-	100	-	ns	
T_{CLDV}	Read Data Valid Delay	-	140	-	105	ns	$C_L = 200 \text{ pE}$
T_{CLDH}	Read Data Hold Time	10	-	10	-	ns	
T_{CLDX}	Read Data to Floating	10	100	10	100	ns	
T_{CLCA}	Cascade Address Delay Time	-	85	-	65	ns	

A.C. CHARACTERISTICS (Continued)

Symbol	Parameter	80130		80130-2		Units	Notes
		Min.	Max.	Min.	Max.		
T_{CLCF}	Cascade Adresse Hold Time	10	—	10	—	ns	
T_{IAVE}	INTA Status t Acknowledge	—	80	—	80	ns	
T_{CHEH}	Acknowledge Hold Time	0	—	0	—	ns	
T_{CSAK}	Chip Select to ACK	—	110	—	110	ns	
T_{SACK}	Status to ACK	—	140	—	140	ns	
T_{AACK}	Address to ACK	—	90	—	90	ns	
T_{CLOD}	Timer Output Delay Time	—	200	—	200	ns	$C_L = 100 \text{ pF}$
T_{CLOD1}	Timer1 Output Delay Time	—	200	—	200	ns	$C_L = 100 \text{ pF}$
T_{JHIH}	INT Output Delay	—	200	—	200	ns	
T_{IRCL}	IR Input Set Up	20	—	20	—	ns	

WAVEFORMS





80150/80150-2 iAPX 86/50, 88/50, 186/50, 188/50 CP/M-86 OPERATING SYSTEM PROCESSORS

ADVANCE INFORMATION

- High-Performance Two-Chip Data Processors Containing the Complete CP/M-86 Operating System
- Standard On-Chip BIOS (Basic Input/Output System) Contains Drivers for 8272A, 8274, 8255A, 8251A, 7220 Bubble Memory Controller
- BIOS Extensible with User-Supplied Peripheral Drivers
- User Intervention Points Allow Addition of New System Commands
- Memory Disk Makes Possible Diskless CP/M-86 Systems
- No License or Serialization Required
- Built-in Operating System Timers and Interrupt Controller
- 8086/80150/80150-2/8088/80186/80188 Compatible At Up To 8 MHz Without Wait States

The Intel iAPX 86/50, 88/50, 186/50, and 188/50 are two-chip microprocessors offering general-purpose CPU instructions combined with the CP/M-86 operating system. Respectively, they consist of the 8- and 16-bit software compatible 8086, 8088, 80186, and 80188 CPU plus the 80150 CP/M-86 operating system extension.

CP/M-86 is a single-user operating system designed for computers based on the Intel iAPX 86, 88, 186, and 188 microprocessors. The system allows full utilization of the one megabyte of memory available for application programs. The 80150 stores CP/M-86 in its 16K bytes of on-chip memory. The 80150 will run third-party applications software written to run under standard Digital Research CP/M-86.

The 80150 is implemented in N-Channel, depletion-load, silicon-gate technology (HMOS), and is housed in a 40-pin package. Included on the 80150 are the CP/M-86 operating system, Version 1.1, plus hardware support for eight interrupts, a system timer, a delay timer, and a baud rate generator.

*CP/M-86 is a trademark of Digital Research, Inc.

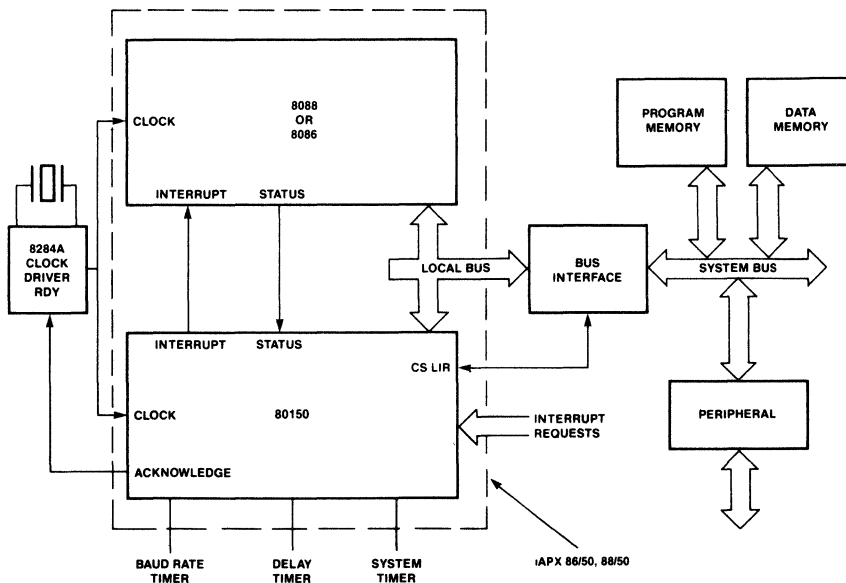


Figure 1. iAPX 86/50, 88/50 Block Diagram

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products: BXP CREDIT, i, ICE, iCS, im, Insite, Intel, INTEL, InteleVision, Intellink, Inteltec, iMMX, iOSP, iPDS, iRMX, iSBC, iSBX, Library Manager, MCS, MULTIMODULE, Megachassis, Micromainframe, MULTIBUS, Multichannel, Plug-A-Bubble, PROMPT, Promware, RUPi, RMX/80, System 2000, UPI, and the combination of iCS, iRMX, iSBC, iSBX, ICE, i²ICE, MCS, or UPI and a numerical suffix. Intel Corporation Assumes No Responsibility for the use of Any Circuitry Other Than Circuitry Embodied in an Intel Product. No Other Patent Licenses are implied. ©INTEL CORPORATION, 1982

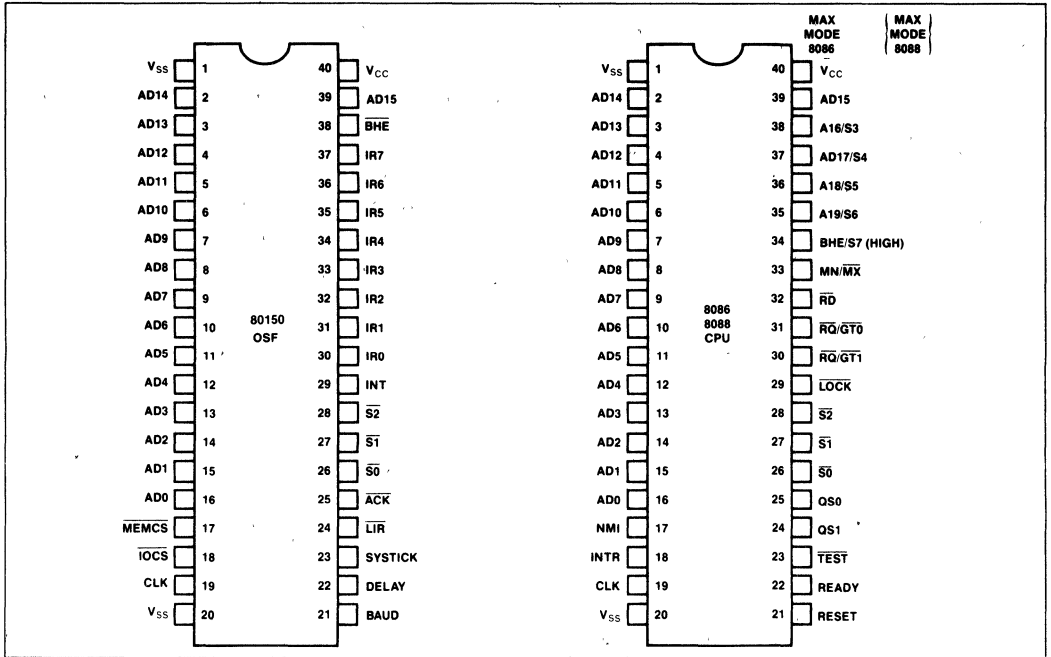


Figure 2. iAPX 86/50, 88/50 Pin Configuration

Table 1. 80150 Pin Description

Symbol	Type	Name and Function																																
AD ₁₅ -AD ₀	I/O	Address Data: These pins constitute the time multiplexed memory address (T ₁) and data (T ₂ , T ₃ , T _W , T ₄) bus. These lines are active HIGH. The address presented during T ₁ of a bus cycle will be latched internally and interpreted as an 80150 internal address if MEMCS or IOCS is active for the invoked primitives. The 80150 pins float whenever it is not chip selected, and drive these pins only during T ₂ - T ₄ of a read cycle and T ₁ of an INTA cycle.																																
BHE/S ₇	I	Bus High Enable: The 80150 uses the BHE signal from the processor to determine whether to respond with data on the upper or lower data pins, or both. The signal is active LOW. BHE is latched by the 80150 on the trailing edge of ALE. It controls the 80150 output data as shown. <table style="margin-left: 40px;"> <tr> <td>BHE</td> <td>A₀</td> <td></td> </tr> <tr> <td>0</td> <td>0</td> <td>Word on AD₁₅-AD₀</td> </tr> <tr> <td>0</td> <td>1</td> <td>Upper byte on AD₁₅-AD₈</td> </tr> <tr> <td>1</td> <td>0</td> <td>Lower byte on AD₇-AD₀</td> </tr> <tr> <td>1</td> <td>1</td> <td>Upper byte on AD₇-AD₀</td> </tr> </table>	BHE	A ₀		0	0	Word on AD ₁₅ -AD ₀	0	1	Upper byte on AD ₁₅ -AD ₈	1	0	Lower byte on AD ₇ -AD ₀	1	1	Upper byte on AD ₇ -AD ₀																	
BHE	A ₀																																	
0	0	Word on AD ₁₅ -AD ₀																																
0	1	Upper byte on AD ₁₅ -AD ₈																																
1	0	Lower byte on AD ₇ -AD ₀																																
1	1	Upper byte on AD ₇ -AD ₀																																
S ₂ , S ₁ , S ₀	I	Status: For the 80150, the status pins are used as inputs only. 80150 encoding follows: <table style="margin-left: 40px;"> <tr> <td>S₂</td> <td>S₁</td> <td>S₀</td> <td></td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>INTA</td> </tr> <tr> <td>0</td> <td>0</td> <td>1</td> <td>IORD</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> <td>IOWR</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> <td>Passive</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> <td>Instruction fetch</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> <td>MEMRD</td> </tr> <tr> <td>1</td> <td>1</td> <td>X</td> <td>Passive</td> </tr> </table>	S ₂	S ₁	S ₀		0	0	0	INTA	0	0	1	IORD	0	1	0	IOWR	0	1	1	Passive	1	0	0	Instruction fetch	1	0	1	MEMRD	1	1	X	Passive
S ₂	S ₁	S ₀																																
0	0	0	INTA																															
0	0	1	IORD																															
0	1	0	IOWR																															
0	1	1	Passive																															
1	0	0	Instruction fetch																															
1	0	1	MEMRD																															
1	1	X	Passive																															

Table 1. 80150 Pin Description (Continued)

Symbol	Type	Name and Function																																																						
CLK	I	Clock: The system clock provides the basic timing for the processor and bus controller. It is asymmetric with a 33% duty cycle to provide optimized internal timing. The 80150 uses the system clock as an input to the SYSTICK and BAUD timers and to synchronize operation with the host CPU.																																																						
INT	O	Interrupt: INT is HIGH whenever a valid interrupt request is asserted. It is normally used to interrupt the CPU by connecting it to INTR.																																																						
IR ₇ -IR ₀	I	Interrupt Requests: An interrupt request can be generated by raising an IR input (LOW to HIGH) and holding it HIGH until it is acknowledged (Edge-Triggered Mode), or just by a HIGH level on an IR input (Level-Triggered Mode).																																																						
$\overline{\text{ACK}}$	O	Acknowledge: This line is LOW whenever an 80150 resource is being accessed. It is also LOW during the first INTA cycle and second INTA cycle if the 80150 is supplying the interrupt vector information. This signal can be used as a bus ready acknowledgement and/or bus transceiver control.																																																						
$\overline{\text{MEMCS}}$	I	Memory Chip Select: This input must be driven LOW when a kernel primitive is being fetched by the CPU. AD ₁₃ -AD ₀ are used to select the instruction.																																																						
$\overline{\text{IOCS}}$	I	<p>Input/Output Chip Select: When this input is low, during an IORD or IOWR cycle, the 80150's kernel primitives are accessing the appropriate peripheral function as specified by the following table:</p> <table border="1"> <thead> <tr> <th>$\overline{\text{BHE}}$</th> <th>A₃</th> <th>A₂</th> <th>A₁</th> <th>A₀</th> <th></th> </tr> </thead> <tbody> <tr> <td>0</td> <td>X</td> <td>X</td> <td>X</td> <td>X</td> <td>Passive</td> </tr> <tr> <td>X</td> <td>X</td> <td>X</td> <td>X</td> <td>1</td> <td>Passive</td> </tr> <tr> <td>X</td> <td>0</td> <td>1</td> <td>X</td> <td>X</td> <td>Passive</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> <td>X</td> <td>0</td> <td>Interrupt Controller</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>Systick Timer</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> <td>1</td> <td>0</td> <td>Delay Counter</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>Baud Rate Timer</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> <td>1</td> <td>0</td> <td>Timer Control</td> </tr> </tbody> </table>	$\overline{\text{BHE}}$	A ₃	A ₂	A ₁	A ₀		0	X	X	X	X	Passive	X	X	X	X	1	Passive	X	0	1	X	X	Passive	1	0	0	X	0	Interrupt Controller	1	1	0	0	0	Systick Timer	1	1	0	1	0	Delay Counter	1	1	1	0	0	Baud Rate Timer	1	1	1	1	0	Timer Control
$\overline{\text{BHE}}$	A ₃	A ₂	A ₁	A ₀																																																				
0	X	X	X	X	Passive																																																			
X	X	X	X	1	Passive																																																			
X	0	1	X	X	Passive																																																			
1	0	0	X	0	Interrupt Controller																																																			
1	1	0	0	0	Systick Timer																																																			
1	1	0	1	0	Delay Counter																																																			
1	1	1	0	0	Baud Rate Timer																																																			
1	1	1	1	0	Timer Control																																																			
$\overline{\text{LIR}}$	O	Local Bus Interrupt Request: This signal is LOW when the interrupt request is for a non-slave input or slave input programmed as being a local slave.																																																						
V _{CC}		Power: V _{CC} is the +5V supply pin.																																																						
V _{SS}		Ground: V _{SS} is the ground pin.																																																						
SYSTICK	O	System Clock Tick: Timer 0 Output.																																																						
DELAY	O	DELAY Timer: Output of timer 1.																																																						
BAUD	O	Baud Rate Generator: 8254 Mode 3 compatible output. Output of 80150 Timer 2.																																																						

The 80150 breaks new ground in operating system software-on-silicon components. It is unique because it is the first time that an industry-standard personal/small business computer operating system is being put in silicon. The 80150 contains Digital Research's CP/M-86 operating system, which is designed for Intel's line of software- and interface-compatible iAPX 86, 88, 186, and 188 microprocessors. Since the entire CP/M-86 operating system is contained on the chip, it is now possible to design a diskless computer that runs proven and commonly available applications software. The 80150 is a

true operating system extension to the host microprocessor, since it also integrates key operating system-related peripheral functions onto the chip.

MODULAR DESIGN

Based on a proven, modular design, the system includes the:

- CCP: Console Command Processor

The CCP is the human interface to the operating system and performs decoding and

execution of user commands.

- **BDOS: Basic Disk Operating System**

The BDOS is the logical, invariant portion of the operating system; it supports a named file system with a maximum of 16 logical drives, containing up to 8 megabytes each for a potential of 128 megabytes of on-line storage.

- **BIOS: Basic Input/Output System**

The physical, variant portion of the operating system, the BIOS contains the system-dependent input/output device handlers.

CP/M* COMPATIBILITY

CP/M-86 files are completely compatible with CP/M for 8080- and 8085-based microcomputer systems. This simplifies the conversion of software developed under CP/M to take full advantage of iAPX 86, 88, 186, 188-based systems.

The user will notice no significant difference between CP/M and CP/M-86. Commands such as DIR, TYPE, REN, and ERA respond the same way in both systems.

CP/M-86 uses the iAPX 86, 88, 186, 188 registers corresponding to 8080 registers for system call and return parameters to further simplify software transport. The 80150 allows application code and data segments to overlap, making the mixture of code and data that often appears in CP/M applications acceptable to the iAPX 86, 88, 186, 188.

Unique Capabilities of CP/M-86 in Silicon

1. CP/M-86 on-a-chip reduces software development required by the system designer. It can change the implementation of the operating system into the simple inclusion of the 80150 on the CPU board.

As described later, the designer can either simply incorporate the Intel chip without the need for writing even a single line of additional code, or he can add additional device drivers by writing only the small amount of additional code required.

2. The 80150 is the most cost-effective way to implement CP/M-86 in a microcomputer. The integration of CP/M-86 with the 16K bytes of system memory it requires, the two boot ROMs required in a diskette-based CP/M-86, and the on-chip peripherals (interrupt controller and timers) lead to savings in software, parts cost, board space, and interconnect wiring.
3. The reliability of the microcomputer is in-

creased significantly. Since CP/M-86 is now always in the system as a standard hardware operating system, a properly functioning system diskette is not required. CP/M-86 in hardware can no longer be overwritten accidentally by a runaway program. System reliability is enhanced by the decreased dependence on floppy disks and fewer chips and interconnections required by the highly integrated 80150.

4. The microcomputer system boots up CP/M-86 on power-on, rather than requiring the user to go through a complicated boot sequence, thus lowering the user expertise required.
5. Diskless CP/M-based systems are now easy to design. Since CP/M is already in the microcomputer hardware, there is no need for a disk drive in the system if it is not desired. Without a disk drive, a system is more portable, simpler to use, less costly, and more reliable.
6. The administrative costs associated with distributing CP/M-86 are eliminated. Since CP/M-86 is now resident on the 80150 in the microcomputer system, there is no end-user licensing required nor is there any serialization requirement for the 80150 (because no CP/M diskette is used).
7. End-users will value having their CP/M operating system resident in their computer rather than on a diskette. They will no longer have to back up the operating system or have a diskette working properly to bring the system up in CP/M, increasing their confidence in the integrity, reliability, and usability of the system.

80150 FUNCTIONAL DESCRIPTION

The 80150 is a processor extension that is fully compatible with the 8086, 8088, 80186, and 80188 microprocessors. When the 80150 is combined with the microprocessor, the two-chip set is called an Operating System Processor and is denoted as the iAPX 86/50, 88/50, 186/50, or 188/50. The basic system configuration is shown in Figure 1. The 80150 connects directly to the multiplexed address/data bus and runs up to 8 MHz without wait states.

- A. **Hardware.** Figure 3 is a functional diagram of the 80150 itself. CP/M-86 is stored in the 16K-bytes of control store. The timers are compatible with the standard 8254 timer. The interrupt controller, with its eight programmable interrupt inputs and one interrupt output, is compatible with the 8259A Programmable Interrupt Controller. External slave 8259A inter-

*CP/M is a registered trademark of Digital Research, Inc

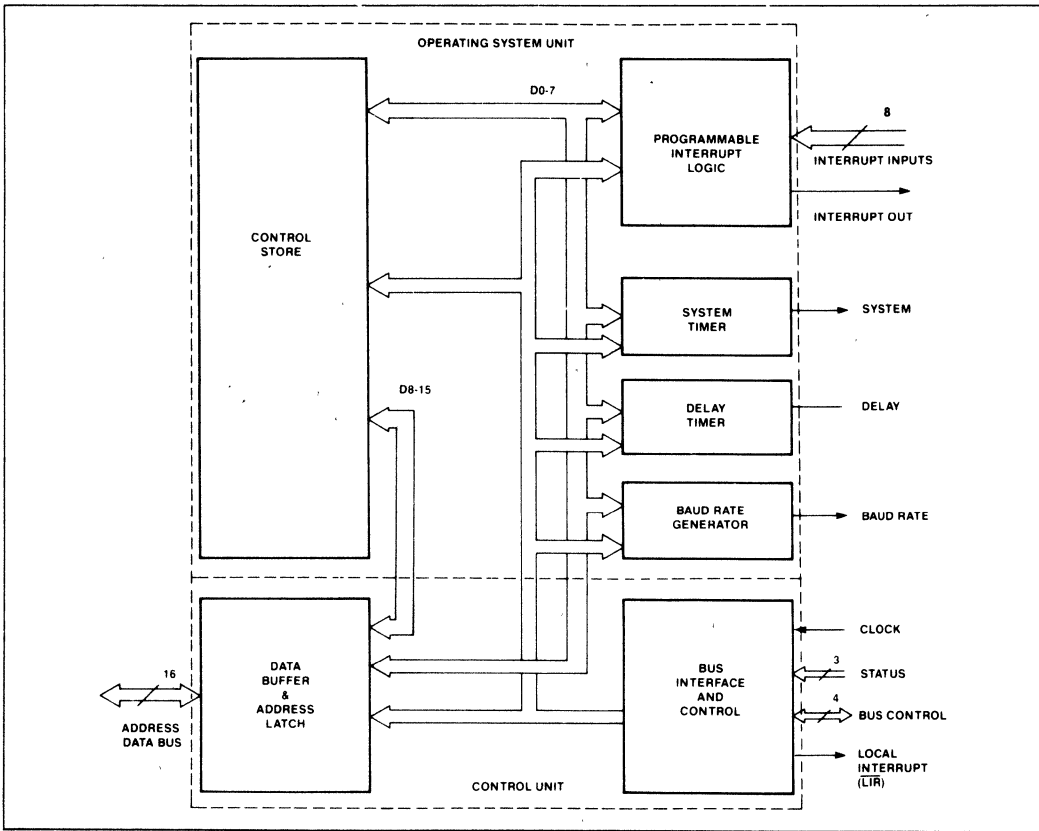


Figure 3. 80150 Internal Block Diagram

rupt controllers can be cascaded with the 80150 to expand the total number of interrupts to 57.

- B. Software. Digital Research's version 1.1 of CP/M-86 forms the basis of the 80150. CP/M consists of three major parts: the Console Command Processor (CCP), the Basic Disk Operating System (BDOS), and the Basic Input/Output System (BIOS). Details on CP/M-86 are provided in Digital Research's *CP/M-86 Operating System User's Guide* and *CP/M-86 Operating System System Guide*.

CCP - Console Command Processor

The CCP provides all of the capabilities provided by Digital Research's CCP. Built-in commands have been expanded to include capabilities normally included as transient utilities on the Digital Research CP/M-86 diskette. Commands are pro-

vided to format diskettes, transfer files between devices (based on Digital Research's Peripheral Interchange Program PIP), and alter and display I/O device and file status (based on Digital Research's STAT).

Through User Intervention Points, the standard CP/M-86 CCP is enhanced to allow the user to add new built-in commands to further customize a CP/M-86 system.

BDOS - Basic Disk Operating System

Once the CCP has parsed a command, it sends it to the BDOS, which performs system services such as managing disk directories and files. Some of the standard BDOS functions provide:

- Console Status
- Console Input and Output
- List Output
- Select Drive
- Set Track and Sector

Read/Write Sector
Load Program

The BDOS in the 80150 provides the same functions as the standard Digital Research CP/M-86 BDOS.

BIOS - Basic Input/Output System

The BIOS contains the system-dependent I/O drivers. The 80150 BIOS offers two fundamental configuration options:

1. A **predefined configuration** which supports minimum cost CP/M-86 microcomputer systems and which requires no operating system development by the system designer.
2. An **OEM-configurable mode**, where the designer can choose among several drivers of-

ferred on the 80150 or substitute or add any additional device drivers of his choice.

These two options negate the potential software-on-silicon pitfall of inflexibility in system design. The OEM can customize the end system as desired.

The **predefined configuration** offers a choice among several peripheral chip drivers included on the 80150. Drivers for the following chips are included in the 80150 BIOS:

8251A	Universal Synchronous/ Asynchronous Receiver/Transmitter (USART)
8274	Multi-Protocol Serial Controller (MPSC)
8255A	Programmable Parallel Interface (PPI)
8272A	Floppy Disk Controller
7220	Bubble Memory Controller

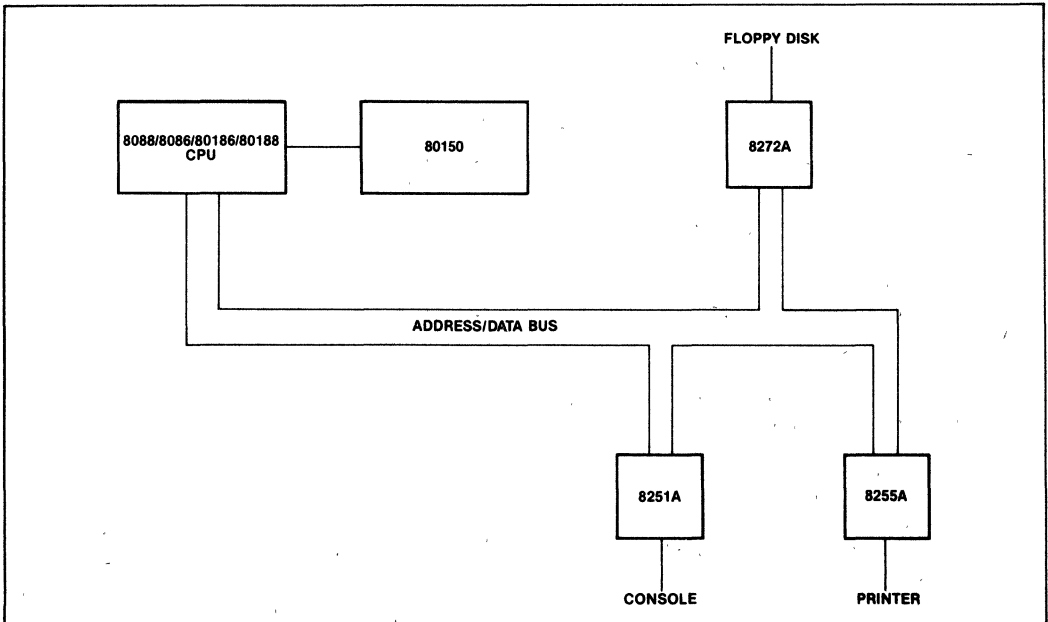


Figure 4. Predefined Configuration

Even in the predefined configuration, the system designer (or end user, if the system designer desires) may select parameters such as the baud rates for the console and printer, and the floppy disk size (standard 8" or 5¼" mini-floppy) and format (FM single density or MFM double density, single-sided or double-sided).

Drivers for the 80150 on-chip timers and interrupt controller are also included in the BIOS.

The 80150 takes advantage of the 80186 and 80188 on-chip peripherals in an iAPX 186/50 or 188/50 system. For example, the integrated DMA controller is used. Also fully utilized are the integrated memory chip selects and I/O chip selects.

Since all microcomputer configurations cannot be anticipated, the **OEM-configurable mode** allows the system designer to use any set of peripheral chips desired. This configuration is shown in Figure 5.

By simply changing the jump addresses in a configuration table, the designer can also gain the flexibility of adding custom BIOS drivers for other

peripheral chips, such as bubble memories or more complex CRT controllers. These drivers would be stored in memory external to the 80150 itself. By providing the configurability option, the 80150 is applicable to a far broader range of designs that it would be with an inflexible BIOS.

MEMORY ORGANIZATION

When using the **predefined configuration** of the 80150 BIOS, the 80150 must be placed in the top 16K of the address space of the microprocessor (starting at location FC000H) so that the 80150 gains control when the microprocessor is reset. Upon receipt of control, the 80150 writes a **configuration block** into the bottom of the microprocessor's address space, which must be in RAM. The 80150 uses the area after the interrupt vectors for system configuration information and scratch-pad storage.

When using the **OEM-configurable mode** of the 80150 BIOS, the 80150 is placed on any 16K bound-

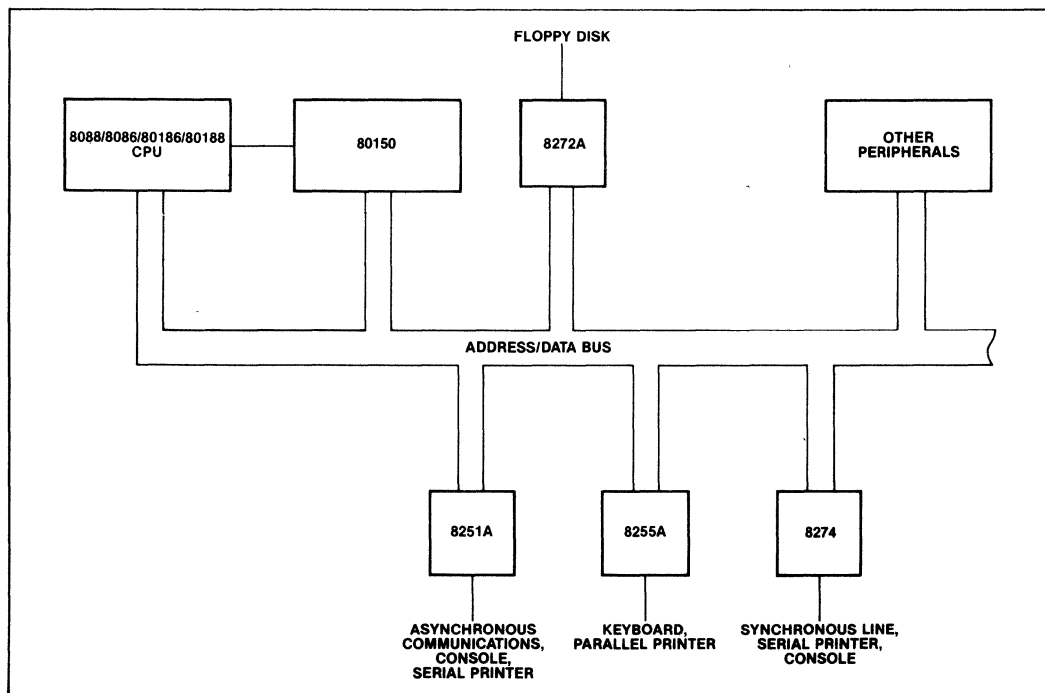


Figure 5. OEM Configurable System

dary of memory **except** the highest (FC000H) or lowest (00000H). The user writes interface code (in the form of a simple boot ROM) to incorporate and link additional features and changes into the standard 80150 environment. The configuration block may be located as desired in the address space, and its size may vary widely depending on the application.

Memory Disk and Bubble Memories

A unique capability offered by the 80150 is the Memory Disk. The Memory Disk consists of a block of RAM whose size can be selected by the designer. The Memory Disk is treated by the BDOS as any standard floppy disk, and is one of the 16 disks that CPM can address. Thus files can be opened and closed, programs stored, and statistics gathered on the amount of Memory Disk space left.

The 80150 also contains software drivers for 7220 bubble memory controller. Use of a bubble memory board as a substitute for one floppy disk drive is directly supported.

The Memory Disk opens the possibility of a portable low-cost diskless microcomputer or network station. Applications software can be provided in a number of ways:

- a. telephone lines via a modem.
- b. ROM-based software.
- c. a network.
- d. bubble memory based software.
- e. low-cost cassettes.

TYPICAL SYSTEM CONFIGURATION

Figure 6 shows the processing cluster of a "typical" iAPX 86/50 or iAPX 88/50 OSP system. Not shown are subsystems likely to vary with the application. The configuration includes an 8086 (or 8088) operating in maximum mode, an 8284A clock generator and an 8288 system controller. Note that the 80150 is located on the CPU side of any latches or transceivers.

Timers

The Timers are connected to the lower half of the data bus and are addressed at even addresses. The timers are read as two successive bytes.

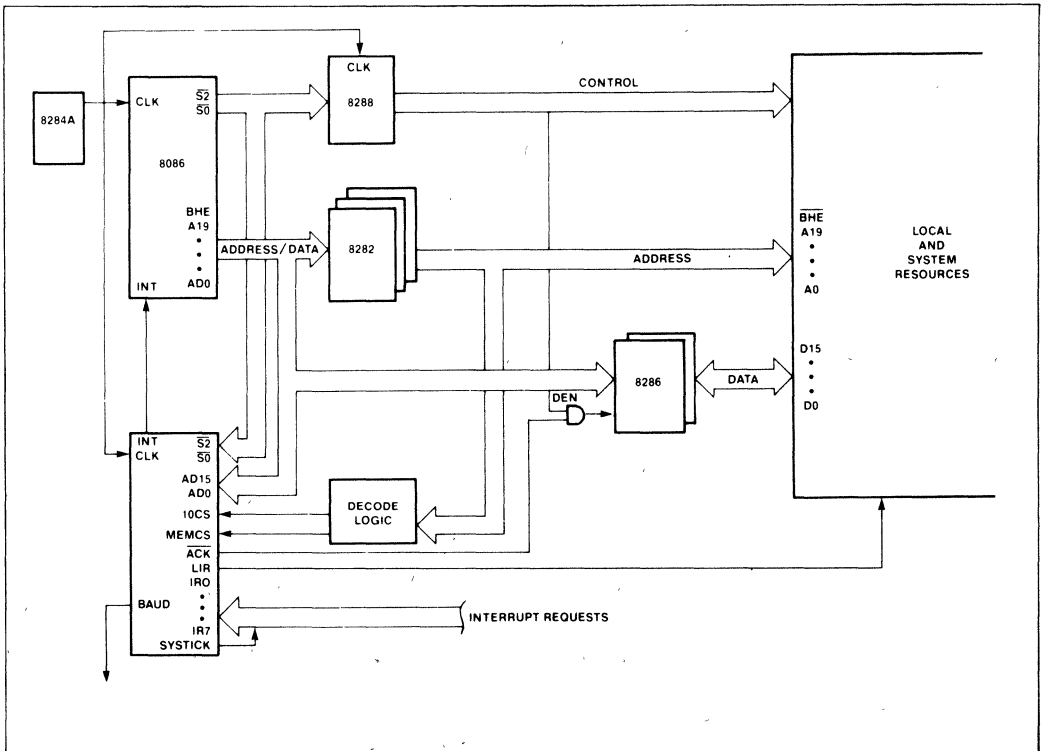


Figure 6. Typical OSP Configuration

always LSB followed by MSB. The MSB is always latched on a read operation and remains latched until read. Timers are not gatable. An external 8254 Programmable Interval Timer may be added to the system.

Baud Rate Generator

The baud rate generator operates like an 8254 (square wave mode 3). Its output, BAUD, is initially high and remains high until the Count Register is loaded. The first falling edge of the clock after the Count Register is loaded causes the transfer of the internal counter to the Count Register. The output stays high for $N/2$ [$(N + 1)/2$ if N is odd] and then goes low for $N/2$ [$(N - 1)/2$ if N is odd]. On the falling edge of the clock which signifies the final count for the output in low state, the output returns to high state and the Count Register is transferred to the internal counter. The baud rates can vary from 300 to 9600 baud.

The baud rate generator is located at 0CH (12), relative to the 16-byte boundary in the I/O space in which the 80150 component is located. The timer control word is located at relative address, 0EH(14). Timers are addressed with IOCS=0. Timers 0 and 1 are assigned to use by the OSP, and should not be altered by the user.

The 80150 timers are subset compatible with 8254 timers.

Interrupt Controller

The Programmable Interrupt Controller (PIC), is also an integral unit of the 80150. Its eight input pins handle eight vectored priority interrupts. One of these pins must be used for the SYSTICK time function in timing waits, using an external connection as shown. During the 80150 initialization and configuration sequence, each 80150 interrupt pin is individually programmed as either level or edge sensitive. External slave 8259A interrupt controllers can be used to expand the total number of interrupts to 57.

In addition to standard PIC functions, the 80150 PIC unit has an $\overline{\text{LIR}}$ output signal, which when low indicates an interrupt acknowledge cycle. $\overline{\text{LIR}} = 0$ is provided to control the 8289 Bus Arbiter SYSB/RESB pin. This will avoid the need of requesting the system bus to acknowledge local bus non-slave interrupts. The user defines the interrupt system as part of the configuration.

INTERRUPT SEQUENCE

The interrupt sequence is as follows:

1. One or more of the interrupts is set by a low-to-high transition on edge-sensitive IR inputs or by a high input on level-sensitive IR inputs.
2. The 80150 evaluates these requests, and sends an INT to the CPU, if appropriate.
3. The CPU acknowledges the INT and responds with an interrupt acknowledge cycle which is encoded in $\overline{\text{S}}_2 - \overline{\text{S}}_0$.
4. Upon receiving the first interrupt acknowledge from the CPU, the highest-priority interrupt is set by the 80150 and the corresponding edge detect latch is reset. The 80150 does not drive the address/data bus during this bus cycle but does acknowledge the cycle by making $\overline{\text{ACK}} = 0$ and sending the $\overline{\text{LIR}}$ value for the IR input being acknowledged.
5. The CPU will then initiate a second interrupt acknowledge cycle. During this cycle, the 80150 will supply the cascade address of the interrupting input at T_1 on the bus and also release an 8-bit pointer onto the bus if appropriate, where it is read by the CPU. If the 80150 does supply the pointer, then $\overline{\text{ACK}}$ will be low for the cycle. This cycle also has the value $\overline{\text{LIR}}$ for the IR input being acknowledged.
6. This completes the interrupt cycle. The ISR bit remains set until an appropriate EXIT INTERRUPT primitive (EOI command) is called at the end of the Interrupt Handler.

ABSOLUTE MAXIMUM RATINGS*

Ambient Temperature Under Bias 0°C to 70°C
 Storage Temperature -65°C to 150°C
 Voltage on Any Pin With
 Respect to Ground -1.0V to +7V
 Power Dissipation 1.0 Watts

**NOTICE: Stresses above those listed under Absolute Maximum Ratings may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended period may affect device reliability.*

D.C. CHARACTERISTICS ($T_A = 0^\circ\text{C to } 70^\circ\text{C}$, $V_{CC} = 4.5 \text{ to } 5.5\text{V}$)

Symbol	Parameter	Min.	Max.	Units	Test Conditions
V_{IL}	Input Low Voltage	-0.5	0.8	V	
V_{IH}	Input High Voltage	2.0	$V_{CC} + .5$	V	
V_{OL}	Output Low Voltage		0.45	V	$I_{OL} = 2\text{mA}$
V_{OH}	Output High Voltage	2.4		V	$I_{OH} = -400\mu\text{A}$
I_{CC}	Power Supply Current		200	mA	$T_A = 25^\circ\text{C}$
I_{LI}	Input Leakage Current		10	μA	$0 < V_{IN} < V_{CC}$
I_{LR}	IR Input Load Current		10 -300	μA	$V_{IN} = V_{CC}$ $V_{IN} = 0$
I_{LO}	Output Leakage Current		10	μA	$45 \leq V_{IN} \leq V_{CC}$
V_{CLI}	Clock Input Low		0.6	V	
V_{CHI}	Clock Input High	3.9		V	
C_{IN}	Input Capacitance		10	pF	
C_{IO}	I/O Capacitance		15	pF	
I_{CLI}	Clock Input Leakage Current		10 150 10	μA	$V_{IN} = V_{CC}$ $V_{IN} = 2.5\text{V}$ $V_{IN} = 0\text{V}$

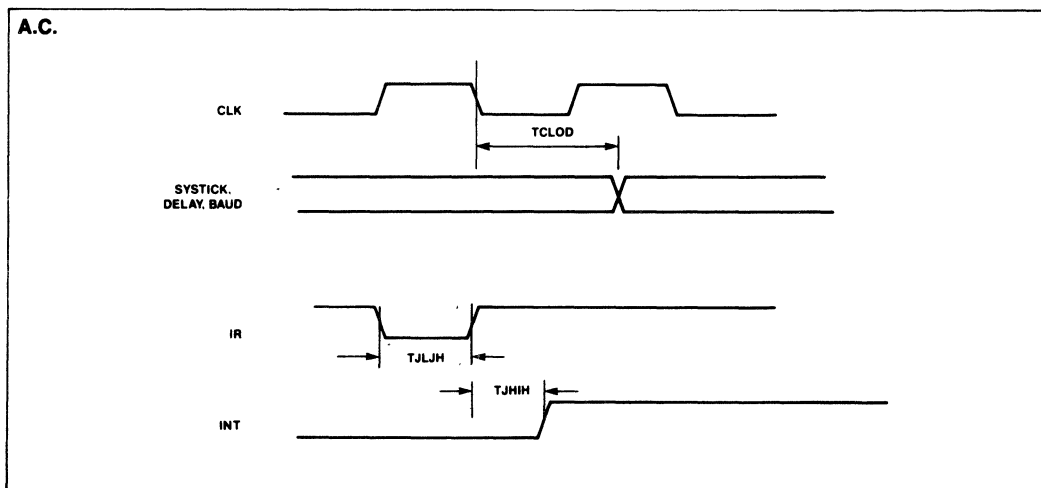
A.C. CHARACTERISTICS ($T_A = 0\text{-}70^\circ\text{C}$, $V_{CC} = 4.5\text{-}5.5\text{ Volt}$, $V_{SS} = \text{Ground}$)

Symbol	Parameter	80150		80150-2		Units	Test Conditions
		Min.	Max.	Min.	Max.		
T_{CLCL}	CLK Cycle Period	200	-	125	-	ns	
T_{CLCH}	CLK Low Time	90	-	55	-	ns	
T_{CHCL}	CLK High Time	69	2000	44	2000	ns	
T_{SVCH}	Status Active Setup Time	80	-	65	-	ns	
T_{CHSV}	Status Inactive Hold Time	10	-	10	-	ns	
T_{SHCL}	Status Inactive Setup Time	55	-	55	-	ns	
T_{CLSH}	Status Active Hold Time	10	-	10	-	ns	
T_{ASCH}	Address Valid Setup Time	8	-	8	-	ns	
T_{CLAH}	Address Hold Time	10	-	10	-	ns	
T_{CSCL}	Chip Select Setup Time	20	-	20	-	ns	
T_{CHCS}	Chip Select Hold Time	0	-	0	-	ns	
T_{DSCL}	Write Data Setup Time	80	-	60	-	ns	
T_{CHDH}	Write Data Hold Time	10	-	10	-	ns	
T_{JLJH}	IR Low Time	100	-	100	-	ns	
T_{CLDV}	Read Data Valid Delay	-	140	-	105	ns	$C_L = 200\text{ pF}$
T_{CLDH}	Read Data Hold Time	10	-	10	-	ns	
T_{CLDX}	Read Data to Floating	10	100	10	100	ns	
T_{CLCA}	Cascade Address Delay Time	-	85	-	65	ns	

A.C. CHARACTERISTIC (Continued)

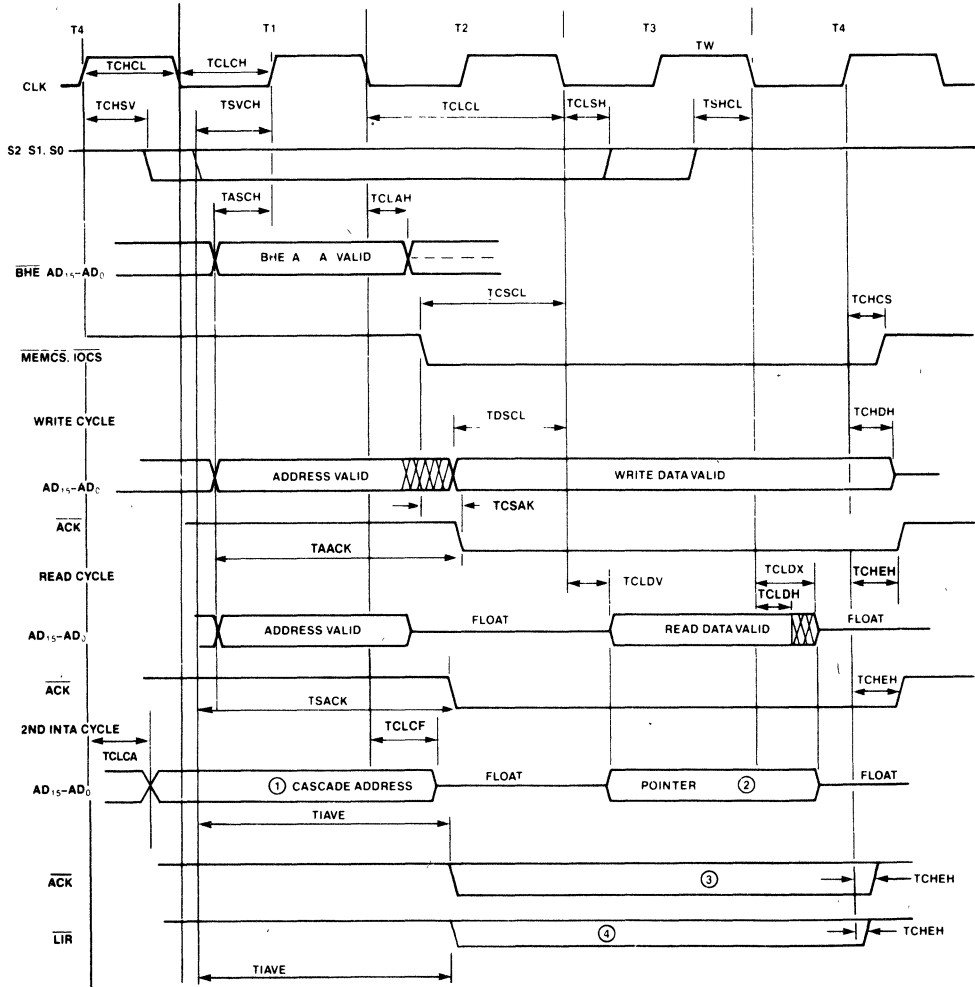
Symbol	Parameter	80150		80150-2		Units	Notes
		Min.	Max.	Min.	Max.		
T_{CLCF}	Cascade Address Hold Time	10	—	10	—	ns	
T_{IAVE}	INTA Status t Acknowledge	—	80	—	80	ns	
T_{CHEH}	Acknowledge Hold Time	0	—	0	—	ns	
T_{CSAK}	Chip Select to ACK	—	110	—	110	ns	
T_{SACK}	Status to ACK	—	140	—	140	ns	
T_{AACK}	Address to ACK	—	90	—	90	ns	
T_{CLOD}	Timer Output Delay Time	—	200	—	200	ns	$C_L = 100$ pF
T_{CLOD1}	Timer1 Output Delay Time	—	200	—	200	ns	$C_L = 100$ pF
T_{JHJH}	INT Output Delay	—	200	—	200	ns	
T_{IRCL}	IR Input Set Up	20	—	20	—	ns	

WAVEFORMS



WAVEFORMS

A.C.



- NOTES
- 1 CASCADE ADDRESS PRESENTED ON AD8, AD9 AND AD10 CORRESPONDING TO CAS0, CAS1 AND CAS2 RESPECTIVELY. AD11-AD15 LINES ARE ACTIVE AND HAVE UNKNOWN VALUES. AD0-AD7 ARE TRISTATE.
 - 2 POINTER VALUE IS ACTIVE ONLY IF POINTER IS GENERATED FROM THE 80150 AND NOT FROM EXTERNAL SLAVE UNIT.
 - 3 ACTIVE LOW ONLY WHEN POINTER DATA IS BEING SUPPLIED BY THE 80150.
 - 4 LOW ONLY FOR LOCAL INTERRUPT.

USER LIBRARY

The Insite User's Program Library is an Intel-sponsored software library supporting Intel microcomputer products. There are currently over 325 programs in the Library collection.

Insite offices are located in the U.S., Brussels, Paris, Germany, the U.K., and Japan, serving about 1,500 members worldwide.

As the Library collection is built on programs submitted by Intel employees as well as customers, we encourage and welcome all program contributions. These contributions are essential to the growth and success of Insite.

In the following pages you will be introduced to more in-depth information about Insite. Membership and program submittal forms, including a complete program index listing, are also included for your convenience.



INSITE™ USER'S PROGRAM LIBRARY

- Programs for 8048, 8051, 8080/8085, and 8086/8087/8088 Processors
- Accepted Program Submittals Entitle You to a Free Membership or Free Program Package
- Worldwide Offices to Serve You
- Diskettes, Paper Tapes, and Listings Available for Library Programs
- Program Library Catalog Offering Hundreds of Programs
- Updates of New Programs Sent During Subscription Period

Insite, Intel's Software Index and Technology Exchange Library, is a varied collection of programs and routines that have been written by users of Intel microcomputers, single-board computers, and development systems. This expanding library of programs covers a broad range of software tools that includes monitors, conversion routines, peripheral drivers, translators, math packages, and even games. As a library member, you can acquire a copy of any program within the library on any of its available types of media. By taking advantage of the availability of existing library programs, numerous hours of coding and debugging time can be saved and routine or redundant programming operations can be eliminated. The Insite Program Library also serves as a learning tool for individuals unfamiliar with assembly or high-level languages associated with Intel's family of microcomputers.

Membership. Membership in Insite is available on an annual basis. Intel customers may become members through an accepted program contribution or paid membership fee.

Program Submittals. The Insite Library is built on program submittals contributed by users. Customers are encouraged to submit their programs. (Details and forms are available through the Insite Library.) For each accepted program, submitters will receive a choice of up to three free programs (for a maximum value of \$300), or free membership with Insite for one year.

Program Library Service. DISKETTES, SOURCE LISTINGS or PAPER TAPES are available for every program in Insite. Diskettes are available on single or double density, 8" or iPDS 5¼". Membership is required to purchase programs.

Insite™ Program Library Catalog. Each member will be sent the Program Library Catalog consisting of an abstract for each program indicating the function of the routine, required hardware and software, and memory requirements.

Insite members will be updated with abstracts of new programs submitted to the Library during the subscription period. For catalog and yearly subscription fee please refer to the Intel OEM Price List or contact the nearest Insite or Intel Sales Office.

INSITE OFFICES ARE WORLDWIDE, WITH FIVE LOCATIONS TO SERVE YOU:

NORTH AMERICA

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051
ATTN: Insite User's Program Library
Telephone: 408-987-8080

THE ORIENT

Intel Japan K.K.
5-6 Tohkohdai, Toyosato-cho,
Tsukuba-gun, Ibaraki, 300-26, Japan
ATTN: Insite User's Program Library
Telephone: 029747-8511

EUROPE

Intel Corporation S.A.R.L.
5 Place de la Balance
Silic 223
94528 Rungis Cedex, France
ATTN: Insite User's Program Library
Telephone: 0687-22-21

Intel Semiconductor GmbH
Seidlstrasse 27
8000 Muenchen 2
West Germany
ATTN: Insite User's Program Library
Telephone: 089-5389-1

Intel Corporation (U.K.) Ltd.
Pipers Way
Swindon SN3 LRJ
Wiltshire, England
ATTN: Insite User's Program Library
Telephone: 0793-488-388

SUBMITTAL REQUIREMENTS

Programs submitted for Insite review must follow the guidelines listed below:

Programs must be written in a language capable of compilation and assembly by the currently-supported version of an Intel standard compiler/assembler. Accepted languages are documented in the following manuals available through Intel's Literature Department.

- BASIC-80 Reference Manual, Order No. 980758
- FORTRAN-80 Programming Manual, Order No. 980481
- FORTRAN-86 User's Guide, Order No. 121570
- Pascal-80 User's Guide, Order No. 981015
- Pascal-86 User's Guide, Order No. 121539
- PL/M-80 Programming Manual, Order No. 980268
- PL/M-86 Programming Manual, Order No. 980466
- MCS-48 and UPI-41A Assembly Language Manual, Order No. 980255
- MCS-86 Macro Assembly Language Reference Manual, Order No. 121703
- 8080/8085 Assembly Language Programming Manual, Order No. 980940
- 8086/8087/8088 Macro Assembly Language Reference Manual for 8085 Based Development System, Order No. 121623
- 8086/8087/8088 Macro Assembly Language Reference Manual for 8086 Based Development System, Order No. 121703
- 8089 Assembler User's Guide, Order No. 980938
- Microsoft BASIC Compiler Reference Manual, Order No. 121805
- Microsoft BASIC-80 Reference Manual, Order No. 121806
- Microsoft BASIC Reference Book, Order No. 121857
- Microsoft Cobol User's Guide, Order No. 121802
- Microsoft FORTRAN-80 Reference Manual, Order No. 121798
- Microsoft FORTRAN-80 User's Manual, Order No. 121799
- Microsoft M/Sort Reference Manual, Order No. 121809
- Microsoft Utility Software Manual, Order No. 121797
- C-86 Compiler Language User's Guide, Order No. 122085

A well-documented source code furnished on an ISIS-formatted 8" diskette, CP/M-formatted 8" diskette, RMX-formatted 8" diskette, or PDS 5¼" ISIS CP/M diskette.

A source listing of the program must be included. This must be the output listing of a compilation or an assembly. No consideration will be given to incomplete programs or duplications of programs already in the Library.

A link and locate listing (whenever applicable).

A demonstration program which assures the validity of the contributed program must be included. This must show the accurate operation of the program.

A complete submittal form.

Licensed software or copyrighted material must be accompanied by a written release from the appropriate, authorized person.



INSITE™ USER'S PROGRAM LIBRARY SUBMITTAL FORM

Processor

8048 8051 8080/8085 8086/8087/8088 Other _____

Indicate the MDS series model the program was created on by checking the appropriate box, and identify other MDS series models the program may be compatible with.

**Program
Title**

Function

**Required
Hardware**

**Required
Software**

**Input
Parameters**

**Output
Results**

Registers Modified:	Programmer:
RAM Required:	Company:
ROM Required:	Address:
Maximum Subroutine Nesting Level:	City:
Assembler/Compiler Used:	State:
Programming Language:	Telephone:

ACKNOWLEDGEMENT AND AGREEMENT

To the best of my knowledge, I have the right to contribute this program material without breaching any obligation concerning nondisclosure of proprietary or confidential information of other persons or organizations. I am contributing this program material on a nonconfidential nonobligatory basis to the Insite User's Library for inclusion in its program library, and I agree that the Library may use, duplicate, modify, publish, and sell the program material without obligation or liability of any kind. The Insite User's Library may publish my name and address, as the contributor, to facilitate user inquiries pertaining to this program material.

Signature _____ Date _____



**LIST OF PROGRAMS
ALPHABETICAL, BY APPLICATION**

Program Title	Order No.
ADD AND SUBTRACT: BCD Numbers	CB11
AEDIT: Tutorial	E7
ASSEMBLER: 8080 MACRO, V4.1	BF4
ASSEMBLER, CROSS: 8008 Code	BC5
ASSEMBLER, CROSS: 8048 On DG Nova	BC6
ASSEMBLER, CROSS: DEC PDP-8 or PDP-11	BC2
ASSEMBLER, CROSS: DEC PDP-11	PC3
ASSEMBLER, CROSS: DEC PDP-11	BC4
ASSEMBLER, CROSS: MCS-48	BC1
ASSEMBLER: MCS-48	BF11
ASSEMBLER, ON-LINE	BF5
BAUD RATE: Detection and Setting Routine for MCS-51	BG50
BAUD RATE: Modify	BG25
BAUD RATE: Modify Under CP/M	BG26
BIT HANDLING: 8048	BG35
BRANCH: MCS-48 Branch Table Routine	BG37
BREAKPOINT: 8089	BD15
CALCULATE: CHECKSUM	BD16
CALCULATE: Sine or Cosine Routine	CB13
CALCULATE: Square Root	CB5
CALCULATION: Least Squares Quadratic Fitting	CB3
CALCULATION: Natural Logarithm	CB4
CAPITALIZE: PL/M-86 Keywords	BG55
CHANGE: Load Addresses, iAPX-86/88 Object File	BG42
CHECKBOOK	BA6
CLOCK: 8748 Clock and LCD Tachometer	BG30
CLOCK: MICRO/SYS MC1460 Real Time Clock Board Utilities	BG31
CLOCK: Real Time	BG29
COMMANDS: Meta-Programs	BG38
COMMUNICATION: DEC PDP-11 to Intel Development System	BB16
COMMUNICATION: HP Calculator with Intel Development System-800	AD1
COMMUNICATION: Intel Development System 220/230 with SDK-85, V1.0	AD4
COMMUNICATION: Intel Model 220/230 to Timesharing Computer	AD6
COMMUNICATION: Intel Model 800 to/from DEC PDP-10	AD8
COMMUNICATION: Intel Development System to/from DEC	AD10
COMMUNICATION: Intel Development System to/from Tektronix 8001	AD11
COMMUNICATION: Intel Development System Series-II with Minicomputer	AD9
COMMUNICATION: Intel Development System Series-II with PROMPT-48	AD2
COMMUNICATION: Intel Development System PROMPT-48 or -80	AD3
COMMUNICATION: Intel System to Serial Output Device	AD14
COMMUNICATION: Intel Development System to/from Hewlett-Packard Computer	AD15
COMMUNICATION: Intel Development System to/from VAX 11	AD13
COMMUNICATION: Intel MDS-Data I/O Programmer Interface	BE8
COMMUNICATION: iPDS to/from MDS-800 under CP/M-80	AD19
COMMUNICATION: NDS-II to/from iPDS Running CP/M-80	AD17
COMMUNICATION: Series-III to/from IBM PC or PC-Compatible	AD23
COMMUNICATION: Tektronix DAS 9100 Digital Analysis System to Intel Development System	AD12
COMMUNICATION: Two Intel Series-II Development Systems	AD7
COMMUNICATION: Xerox File Transfer Facility	AD16

Program Title	Order No.
COMPARE: 8048 or 8049 ROMS	AE11
COMPARE: Files	BD11
COMPILER: Pascal	BF1
CONSOLE ACCESS: Input and Output for Series III	BD36
CONTROLLER: 8278 Keyboard/Display	AC3
CONTROLLER: 8292 on 8741A	AC4
CONTROLLER: Dual Floppy Disk Drive	AB11
CONTROLLER: Firmware for iSBC-589	AC7
CONTROLLER: PID Control Loops	AB20
CONTROLLER: PROMPT-48 Interactive	AB2
CONTROLLER: UP1-41 8-Digit LED Display	AC1
CONTROLLER: UP1-41A/42 Digital Cassette, V2.5	AC5
CONVERSION: ASCII-Decimal to/from FPAL Number	BB13
CONVERSION: ASCII Floating Point Numbers to AM9711 and Intel 8231 4 Byte FP Format	BB5
CONVERSION: ASCII to Floating Point	BB14
CONVERSION: ASCII to/from EBCDIC	BB1
CONVERSION: ASCII to/from Floating Point	BB11
CONVERSION: ASCII Code to/from Intel Floating Point	BB12
CONVERSION: Binary to BCD	BB6
CONVERSION: Binary to BCD	BB7
CONVERSION: Convert/Format/Print	BB8
CONVERSION: Decimal to/from Floating Point	BB9
CONVERSION: FORTRAN or FPAL Floating Point to/from Decimal	BB10
CONVERSION: Hex to ASCII	BB2
CONVERSION: ISIS-II to/from CP/M	BB18
CONVERSION: MCON-6800 Source Code to 8086/88 Source Code	BB3
CONVERSION: ZCON-Z80 to 8086/88 Source Converter	BB4
CONVERT: 8051 Binary to/from BCD	BB24
CONVERT: 8086 HEX File to 8080 HEX File	BB26
CONVERT: ASCII Octal/Decimal/Hexadecimal to ASCII Octal/Decimal/Hexadecimal/Internal Binary	BB27
CONVERT: Double word to ASCII String	BB22
CONVERT: Fixed Point to Floating Point	BB21
CONVERT: FPAL Numbers to/from IBM 32-Bit Floating Point	BB28
CONVERT: Intel HEX Code to BASIC DATA Statements	BB25
CONVERT: ISIS Object Module to CP/M Object Module	BB23
COPY: Disk	BG28
COPY: Diskette	BG27
COPY: Diskette	BG43
COPY: iPDS CP/M-80 Diskette	BG45
COPY: PDP-11 Disk File to Intel ISIS-II Disk File	BB15
COUNT: ICE-80 Machine Cycles	BD10
COUNT: Program Usage	BG40
CREDIT: Tutorial	E6
CREDIT: Used on Modified Hazeltine 1500	BG33
DEBUG: CAT.88 (iRMX88 Task Debugger)	BD34
DEMO: 208	AE7
DEMO: iAPX-88	AE13
DEMO: iRMX 86 Multitasking Spectrum Analysis	AE8
DEMO SOFTWARE: 8275	AE6
DEVICE, I/O: UPI-41A Combination	AC2
DIAGNOSTIC: 8080 I/O	AE2
DIAGNOSTIC: Microcomputer Development System 230	AE9
DISASM	BD6
DISASSEMBLER: 8048 Object Code	BD8



Program Title	Order No.
DISASSEMBLER: 8080 Code	BD1
DISASSEMBLER: 8080 Code	BD4
DISASSEMBLER: 8080 Object Code	BD2
DISASSEMBLER: ICE-80 Ver 2.1	BD3
DISASSEMBLER: ISIS-II Object Files	BD5
DISPLAY: ISIS Directory under CP/M	BG46
DIVISION: 32-Bit by 16-Bit	CB12
DOWNLOAD: iPDS to Serial Port	AD18
DRIVER: 8048 Seven-Segment Display	AB5
DRIVER: 8085 Serial I/O	AB1
DRIVER: Audio Cassette Recorder	AB6
DRIVER: Bios and Boot Program for CP/M-80	AB22
DRIVER: Cassette Operating System	AB7
DRIVER: Dumb Terminal Simulator	AB10
DRIVER: Intellec Development System Series-II as Dumb Terminal	AB9
DRIVER: iPDS Dumb Terminal	AB23
DRIVER: iSBC 86/12 Real Time Clock Driver	AB19
DRIVER: Okidata Microline 84 Line Printer	AB25
DRIVER: PROM Programmer	BE7
DRIVER: RMX-80, for the iSBC 254 Bubble Memory with 80/10 Board	AB14
DRIVER: RMX-80, for the iSBC 254 Bubble Memory with 80/20/30 Board	AB15
DRIVER: RMX-86, for the iSBC 254 Bubble Memory Board	AB16
DRIVER: RMX-80 for iSBC 534	AB12
DRIVER: RMX-80 for SBC 215 Controller Board	AB13
DRIVER: RMX-86, for the iPAB-128, iPAB-256, iSBX-251 Bubble Memory Products	AB17
DRIVER: RMX-86, High Performance Driver for iSBC-550 Ethernet Communications Controller	AB18
DRIVER: SYCOR 135 Cassette Operating System	AB8
DRIVER: Tektronix 4010 Graphic Screen	AB3
DRIVER: T.I. Omni 810 Lineprinter	AB4
DRIVER: USART for iSBC-86/XX	AB21
DUMP: Diskette	BD27
DUMP: Diskette File	BD28
DUMP: Diskette File	BD26
DUMP: iAPX-86/88 Absolute Object File	BD30
DUMP: iSBC 86/12 Memory	BD29
DUMP: Screen	BG54
DUMP: Symbol Table	BD21
EDIT: Disk	BD33
EDIT: Hex File	BD31
EDIT: Inspect and Change File	BD32
EDIT: Text	BA4
EDITOR: Text, Intel X111	BA3
EXECUTIVE: Real Time	AA8
EXERCISE: Data Translation MULTIBUS Analog I/O Boards	BE6
FIFO	BG13
FIFO	BG12
GAME: Bandit	D3
GAME: Black Box	D15
GAME: Breakout	D13
GAME: Craps	D5
GAME: Darts	D6



Program Title	Order No.
GAME: Fruit Machine	D4
GAME: Hangman	D7
GAME: Mastermind	D9
GAME: Maze	D2
GAME: Maze	D1
GAME: Othello	D10
GAME: Poker	D14
GAME: Slalom, V1.4	D8
GAME: Tiny Chess 86	D12
GENERATE: 16-Bit Random Number	CB2
GENERATE: Calendar	BA8
GENERATE: CCITT Cyclic Redundancy Check	BD37
GENERATE: Disk Directory Library	BA15
GENERATE: Fast Generation of IBM Bi-Sync CRC16	BD20
GENERATE: Graph	CB7
GENERATE: High and Low Bytes from 8086 Hex File	BD35
GENERATE: Histogram	CB8
GENERATE: IBM Bi-Sync CRC16	BD19
GENERATE: Music for SDK-85	D11
GENERATE: Output Signal	BG5
GENERATE: PL/M Cross Reference	BD25
GENERATE: PROM Checksum Calculation	BD18
GENERATE: Public Symbol Cross Reference	BD38
GENERATE: Public Symbol Cross Reference (Update)	BD24
GENERATE: Random Number	CB6
GENERATE: Software Documentation	BA14
GENERATE: Stochastic Variates and Histograms	CA23
GENERATE: Symbol List	BD24
GENERATE: Symbol Table for BASIC-80	BD23
GENERATE: Tabs	BA16
GENERATE: X-Y Graphs	CB9
HANDLER: Enhanced RMX-80 Terminal Handler	BE10
HANDLER: RMX/80 Minimal Terminal	BE2
INCREMENT: Program Counter	BG39
INFO: NDS-II File Information Utility	BG56
INITIALIZE: Baud Rate	BG24
INITIALIZE: Baud Rate	BG23
INTERPRETER: 8086 Tiny BASIC	BF9
INTERPRETER: Interactive 8087 Instruction Interpreter	AA12
INTERPRETER: LISP	BF3
INTERPRETER: LLL BASIC-II	BF7
INTERPRETER: LLL/Chernack BASIC	BF8
INTERPRETER: MCS-51 Tiny BASIC, V2.2	BF10
INTERPRETER: PILOT-80	BF2
INTERPRETER: RMX/80 Command Line	BG4
INTERPRETER: Single-Step	BD7
LINKAGE: Series III i8087 Linkage Modules	BG36
LIST: 8086 Public and External Symbols	BD41
LIST: Directory, ISIS Diskette/NDS Disk	BG18
LIST: Diskette Directory	BG17
LIST: File	BG15
LIST: File	BG16
LIST: File Errors	BD12
LIST: PL/M Compiler Errors	BD13



Program Title	Order No.
LIST/PRINT/TYPE	BG14
LIST: Save Error	BD14
LOAD/SAVE: RAM	BG1
MACROS: Block Structures	BG10
MACROS: Block Structures	BG11
MACROS: Enhancements for Credit Text Editor	BA22
MAIL LIST	BA9
MAIL LIST	BA11
MAIL LISTS FOR BASIC 80	BA12
MATH PACKAGE: 8231	CA17
MATH PACKAGE: 8051	CA18
MATH PACKAGE: 8080/8085 Fundamental Support Package	CA20
MATH PACKAGE: 8231 Arithmetic Processing Unit	CA16
MATH PACKAGE: Arithmetic Functions	CA11
MATH PACKAGE: Arithmetic Functions for MCS-48	CA22
MATH PACKAGE: Double Precision Floating Point	CA12
MATH PACKAGE: Double Precision Integer	CA4
MATH PACKAGE: Fixed and Floating Point	CA5
MATH PACKAGE: Floating Point	CA2
MATH PACKAGE: Floating Point	CA1
MATH PACKAGE: Floating Point	CA7
MATH PACKAGE: Floating Point	CA6
MATH PACKAGE: Floating Point Library/8086	CA13
MATH PACKAGE: Floating Point Utilities for FPAL.LIB	CA8
MATH PACKAGE: High Speed Binary Math Package for 8031/8051	CA21
MATH PACKAGE: Multiple Precision Arithmetic/8086	CA14
MATH PACKAGE: Multiplication, Division, and BCD-Binary Binary-BCD Conversion for 8051	CA24
MATH PACKAGE: Multiply/Divide	CA15
MATH PACKAGE: Optimized Floating Point	CA9
MATH PACKAGE: Optimized Floating Point	CA10
MATH PACKAGE: PL/M Multiple Precision	CA3
MATH PACKAGE: Recursive Computation of Mean and Standard Deviation	CA19
MERGE: Mailing List	BA10
MONITOR: Intellec 8/MOD80	AA1
MONITOR: Bubble Memory Development Software for Intel BPK-72	AA10
MONITOR: HSE-49 Expansion Monitor	AA13
MONITOR: Intellec Development System, V2.0	AA6
MONITOR: iSBC 250 1-Megabit Bubble Memory	AA9
MONITOR: iSBC 254 Bubble Memory Board Monitor	AA11
MONITOR: iSBC 544	AA7
MONITOR: iSBC 80/05 or 80/04	AA14
MONITOR: iSBC 80/10	AA15
MONITOR: iSBC 80/10 or 80/10A	AA16
MONITOR: iSBC 80/20 or 80/20-4	AA17
MONITOR: iSBC 80/24	AA18
MONITOR: iSBC 80/30	AA19
MONITOR: iSBC 86/12 Numeric Processor Extension (NPX) Monitor	AA2
MONITOR: SDK-85, V2.0	AA3
MONITOR: SDK-86 Keypad	AA5
MONITOR: SDK-86 Serial, V1.1	AA4
MONITOR: Super Monitor 80	AA20
MONITOR: Super Monitor 86	AA21
MONITOR: Super Monitor 86 for the iSBC 88/45	AA22
MORSE CODE TUTOR V2.0	E3



Program Title	Order No.
MULTIPLICATION: 8748 BCD	CB10
MULTIPLICATION: 40-Bit	CB14
PRINT .86	BG58
PRINT: Cover Page	BA1
PRINT: Discounted Cash Flow	BA7
PRINT: File	BA17
PRINT: Files	BA18
PRINT: Files	BA19
PRINT: High Speed Utility	BG32
PROCEDURE: Pascal 86 Screen/Cursor Control	BG34
PROCEDURE: PL/M DOCASE	BG9
PROCEDURES: PL/M-86 General Purpose Library	BG49
PROCEDURES: PL/M Output	BG8
PROCEDURES: PL/M Utilities	BG7
PROCESSOR: Macro	BF6
PROCESSOR: Text	BA5
PROGRAM: 8741A as iSBC 941	AC6
PROGRAMMER: EPROM-8755A	BE5
PROGRAMMER: EPROMS 2708/16/32	BE4
PURGE: Symbol Tables	BD42
READ/PUNCH: Paper Tape to/from SDK-85 RAM	BE3
RECEIVE	AD5
RECOVER: Diskette	BG2
RECOVER: Lost AEDIT Files	BG51
RECOVERY: Diskette File	BA2
RELOCATE	BG41
REMOTE: NDS-II Communication with IBM PC running MS/DOS	AD22
REMOTE: NDS-II Communication with iPDS/Series-II/III/IV (UPDATE)	AD20/ AD21
REMOTE: Terminal Control on Series-II under CP/M-80	AB24
REMOTE: iRMX-86 Communication Program	AD24
REPORT: Status of Exported Job	BG44
RUNOFF: ASCII Text File to Epson Printer	BG48
SEND: CP/M-80 PDS Files to Printer Via Modems	BG53
SEND: Intel HEX Code to PROM Programmer	BE9
SEND: ISIS-PDS Files to Printer Via Modems	BG52
SIMULATE: iACX-96	BD40
SIMULATE: Light Box	E8
SIMULATOR: 8048/49 Code, V1.3	BB19
SIMULATOR: 8048/49 Simulator	BB20
SORT: Bubble Sort and Binary Search Routines	BG22
SORT: Disk Directory	BG19
SORT: Disk Directory	BG20
SORT: Diskette File	BG21
SORT: General	BA13
SORT: Public Symbols	BD39
SORT: Symbol Table from an Absolute File	BD22
SOURCE FILES: iAPX-86/88 System Workshop Summary and Review	E1
SOURCE FILES: iAPX-80/85 System Workshop Summary and Review	E2
SPELL	BA21
SUBMIT: ISIS Command String	BG6
SWEEP: ISIS-II General Disk File Utility	BG47



Program Title	Order No.
TEST: 8080 CPU	AE1
TEST: iSBC 80/10 I/O Ports	AE3
TEST: Error Correcting Code	AE12
TEST: MCS-48 Family CPU	AE10
TEST: Memory	AE5
TEST: Memory	AE4
TEST: PROM/ROM Checksum Self-Test	BD17
TEST: RAM	AE14
THERMOMETER: Thermistor Controlled	BE1
TRACE: ICE-80	BD9
TRANSFORM: DISCRETE Fourier	CB1
TREE: Utilities for Series-IV or NRM	BG57
UTILITIES: Circular Lists	BG3
UTILITIES: Menu	E5
UTILITIES: RT11 Diskette Utility for Intellec 800	BB17
UTILITIES: Talk	E4
WORD PROCESSOR	BA20

Appendix

A

INTEL SOFTWARE STANDARDS

Intel's software is built on standards which facilitate software portability and provide an open system for software.

Intel's software utilizes the emerging standards in graphics, networking, database and portable operating system interfaces. This base system software provides a mapping from architectural and operating system dependencies to a standard interface. High-level applications are built from the standard interfaces and remain portable across multiple configurations and operating systems. Figure 1 illustrates the open software model relationship.

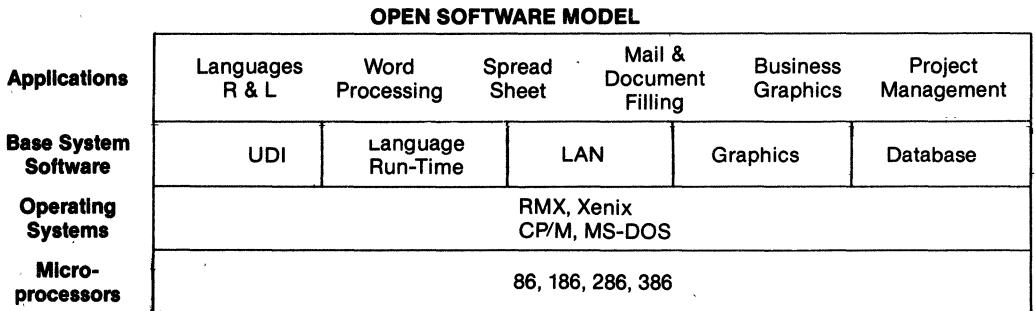


Figure 1

Intel has supported its fundamental software across multiple operating systems through the UDI operating system interface. By writing software to use the UDI interface which provides memory management, I/O routines, and exception handling—Intel is able to port high level languages, language run-time, and fundamental software to a new release or new operating system in minimal time. Thus Intel's software is operating system independent.

Intel's local area networking products use the IEEE 802.3 CSMA/CD Access Method and physical layer. The work for this standard was done jointly by Intel, DEC, and Xerox and is commonly known as the Ethernet protocol. The transport layer uses the proposed ISO transport protocol specification.



Intel supports the ANSI graphics standardization effort and will offer products which utilize these standards. The Virtual Device Interface (VDI) standard developed by the ANSI X3H3 committee is intended to provide a single standard which supports multiple graphics devices with the same set of graphics functions. A companion standard Virtual Device Metafile (VDM) will give a means of storing or transmitting pictures as streams of VDI functions. A third graphics standard being supported by Intel is the North American Presentation Level Protocol Syntax (NAPLPS). NAPLPS is suited for raster-scan display (both CRT and hardcopy) and is currently in final approval stage by ANSI.

Intel's Pascal and FORTRAN adhere to the ANSI standard and support optional extensions. All Intel languages (ASM-86, PL/M, FORTRAN, and Pascal) use common data types and parameter passing conventions to allow inter-language calls. The real number data types in these languages utilize the IEEE real math standard and use the numerics coprocessor or an emulator to support the real math operations and functions. The object module formats (OMF) are commonly used by all of the 86 language products as well as many language products supplied by independent software vendors.

INTEL SOFTWARE STANDARDS DOCUMENTS

Standard	Document
UDI	Run-Time Support Manual For iAPX 86, 88 Applications, Appendix A (Intel 121776-002)
NAPLPS	Intel's Guide To Understanding The ANSI Videotex Presentation Level Protocol (Intel 145412-001)
VDM, VDI	Draft of proposed American Standard for the Virtual Device Metafile, ANSI X3H33 Virtual Device Interface Task Group
Local Area Network <ul style="list-style-type: none">— Data link and physical layer (Ethernet)— Transport layer	Draft IEEE Standard 802.3 CSMA/CD Access Method and Physical Layer Specifications, IEEE Computer Society ISO draft proposal 8073 Information Processing Systems, Open Systems Interconnection-Connection Oriented Transport Protocol Specification
FORTRAN 77	Intel's extension to ANSI FORTRAN 77 specified in FORTRAN-86 User's Guide (Intel 121570-002)
Pascal	Intel's extension to ANSI Pascal specified in Pascal-86 User's Guide (Intel 121539-001)
Real Math	Draft 10.0 of IEEE Task P754, December 1982. 8087 Support Library Reference Manual (Intel 121725-001).
Language Data Types and Parameter Passing <ul style="list-style-type: none">— Pascal— FORTRAN	Pascal-86 User's Guide, Appendix J (Intel 121539-003) FORTRAN-86 User's Guide, Appendix H (Intel 121570-002)
Object Module Formats <ul style="list-style-type: none">— 86— 286	8086 Relocatable Object Module Formats (Intel 121748-001) The Concrete Representation of 80286 Object Modules, Intel internal document iAPX 286 Compilers Writer's Guide, (Intel-in preparation)



SOFTWARE SUPPORT SERVICES

A FULL SERVICE SUPPORT PROGRAM

Intel's Software Support Services is a comprehensive range of post-sales support programs for software and systems purchased from Intel. Its objectives are to maximize the system's performance and minimize unnecessary downtime for greater productivity. These services are provided for all Intel developed and most Intel marketed third-party software.

DESCRIPTION OF SERVICES

SOFTWARE SUPPORT CONTRACTS

1. **Subscription Service** Technical Reports

A technical report will be published quarterly for active products and semi-annually for mature products. This will contain a Configuration and Compatibility Guide, a product performance exceptions list providing solutions to known problems, a review of important current Software Problem Reports (SPRs) submitted by customers, and articles of general interest such as programming hints. A listing of product manuals available from Intel is also provided; comments newsletter is also provided monthly.

Software Problem Reporting Service (SPR)

Intel will respond to written questions (submitted on a standard SPR form) on product-specific software, system, or documentation issues. Intel will verify receipt of the SPR promptly and normally will respond within approximately 3 weeks. Intel does not guarantee a resolution will always be available to specific problems.

2. Software updates, associated manuals and documentation are provided at no additional charge for all software products purchased by the customer and covered under a software support contract. Each contract provides the customer with updates, manuals, and documentation for the covered software product being updated. Copying of software updates is per the terms of the master software license agreement.
3. Intel's Technical Information Phone Service (TIPS) provides the Customer with direct communication with a member of Intel's Software Support staff. The Customer may call a single service number (U.S.) between 7:00 A.M. and 6:00 P.M., (Mountain Time) for product-specific inquiries.

This service enables the Customer to:

Obtain assistance in using the product.

- documentation clarification.
- operational understanding.

Obtain product specific information.

- problem identification.
- work-around, patch, or other solution when available.
- information on existing SPRs.

If the reported condition is not an already documented SPR, obtain assistance in problem isolation techniques.

As part of the TIPS, Software Support Services maintains a list of reported problems and problem resolutions. TIPS does not include user application assistance or engineering time to derive a resolution to a problem if none is currently available. (See Phone Consulting under Consulting Services.) However, the Software Support Engineer will submit a problem report into the SPR system under the Customer name when appropriate.

TIPS is offered as a supplemental tool for obtaining maximum utilization of Intel software products. It is expected that the Customer will avail himself of training classes as appropriate, and will make reasonable efforts to utilize all product documentation.

The Customer must designate one System Manager and one alternate who are authorized to call the TIPS. This service is offered on a one-year period, and continues thereafter on a month to month basis until cancelled by either party with 30 days written notice.



ORIENTATION/INSTALLATION SUPPORT PACKAGES

We have structured very specific support packages to assist our customers with the installation and reconfiguration of such systems as NDS-II, 86/330, RMX and XENIX. See Intel's current price book for a listing of support packages available today.

CONSULTING SERVICES

Consulting Services provides customized support for system, board, and component level customers. Consulting services provide a wide range of support - from system designs to solving difficult development problems to complete project management and project implementation.

1. **Field Consulting** — The Customer may contract for an Intel Software Support Engineer to come on-site to assist and advise the customer in utilizing Intel software products. This service is available on a Time and Material basis. Minimum period: 1 day (8 hours). Travel time and expenses are billed separately as specified in the price list.
2. **Phone Consulting** — The Customer may contract for an Intel Software Support Engineer in the Customer Support Group to provide customer or application-specific research, effort, or consultation. Blocks of time may be purchased and utilized in minimum fifteen (15) minute increments.

INSITE USER'S PROGRAM LIBRARY

Intel's Software Index and Technology Library is a library of programs that have been submitted by users of Intel microcomputers, single-board computers, and development systems. Membership in INSITE enables the Customer to order programs at a nominal charge. Members are provided a program catalog and catalog updates.

LIMITATIONS

- A. Software Support Services are limited to standard Intel system configurations supported by software products, as defined in the applicable software product data sheet. Services will be performed within a 12-month period from effective date of the purchased services.
- B. Software support services do not include hardware maintenance.
- C. Any change in the equipment site of the system within the U.S. may affect Intel's ability to deliver the support services ordered and may result in increased charges. If the system is moved outside the continental U.S., it shall not be eligible for continued service as ordered, but may be eligible for continued service under Intel's local terms and conditions then in effect for a like system in the country or territory of reinstallation.

OTHER INFORMATION

- A. Software Support Services is Intel's commitment to providing the customer with consistent, high-quality, post-sales software support. It is our way of delivering guaranteed support which the customer can rely on. To tailor a full service software support program that addresses specific needs, contact the local Intel sales or service office for more information.
- B. Term: Service will be provided for the period specified in the price list.
- C. Charges: There are three kinds of billings utilized with the service offerings: front-end billing, monthly billing (not less than \$100 per month), and post-service billing. The customer will be billed on one of the referenced types of billings, depending on the type of service. Prices will be those specified in the current Intel price list.
- D. In order to obtain maximum service from Software Support Services, it is advised that the customer maintain the system to the latest revision level, and assign a System Manager who will be the key contact for Software Support Services.
- E. Guidelines:
 1. The customer must have signed an Intel Master Software License agreement. All services and materials made available to the customer through Software Support Services, including documentation and program materials, are subject to the terms and conditions of the license/sale.
 2. The License Fee or List Price for covered software products includes a period of Initial Support as defined in individual product descriptions. Additional support services may be obtained as listed in the price list.



iRUG DESCRIPTION

iRUG is the Intel iRMX™ 86 User's Group. It is a non-profit group chartered to establish a forum for users of the iRMX 86 Operating System and to promote and encourage development of iRMX 86 based software.

iRUG membership is free to licensed iRMX 86 Operating System users and to their employees. Benefits of membership include: access to the user's library of iRMX software tools and utilities; membership in local and national chapters; access to the group bulletin board; receipt of quarterly national newsletters; synopsis of software problem reports (SPRs) submitted by members; opportunity to present papers and conduct workshops; invitations to seminars devoted to the use of Intel products.

The user's library, maintained by iRUG, contains software programs written and submitted by members and Intel employees. Programs available range from file or directory manipulation commands and terminal attribute selection utilities to dynamic logon, background job facilities and basic communication utilities.

Programs in the library are available through a telephone dial-up service.

Local and national iRUG chapters provide a forum for members to meet other iRMX Operating System users in an informal setting. At local meetings and the annual international seminar, members can discuss their ideas, share their experiences and techniques, and give feedback to Intel for future improvements and features of the iRMX 86 Operating System. The meetings also showcase new products offered by Intel and other developments in iRMX based software supplied by other companies.

iRUG sponsors a Special Interest Group (SIG) on the CompuServe Information Service. The SIG offers two features, message facilities and an online conference facility. The message facility (bulletin board) allows members to leave and receive messages from other members. These might include problems and solutions regarding the iRMX 86 Operating System or new techniques to be shared. The online conference facility allows users to hold scheduled meetings on any topic. Whatever information a member types at his/her terminal will be displayed at all terminals logged into the conference facility.

"Human Interface" in iRUG's quarterly national newsletter in the United States. It serves as a supplement to chapter meetings by providing: library listings, information on the latest releases of products running on the iRMX 86 Operating System; officer messages; member SPRs; release and update plans for the iRMX Operating System; and member articles.

If you are interested in becoming a member of iRUG or desire further information contact the Intel iRUG liaison.

Terri Huggett
5200 N.E. Elam Young Parkway
Hillsboro, OR 97123
Mailstop HF2-2-352
(503) 640-7123



DOMESTIC SALES OFFICES

ALABAMA

Intel Corp
5215 Bradford Drive
Suite 2
Huntsville 35805
Tel (205) 630-4010

ARIZONA

Intel Corp
11225 N 28th Drive
Suite 214D
Phoenix 85029
Tel (602) 869-4980

Intel Corp
1161 N El Dorado Place
Suite 301
Tucson 85715
Tel (602) 299-6815

CALIFORNIA

Intel Corp
21515 Vanowen Street
Suite 116
Canoga Park 91303
Tel (818) 704-8500

Intel Corp
2250 E Imperial Highway
Suite 218
El Segundo 90245
Tel (310) 640-6040

Intel Corp
1510 Arden Way, Suite 101
Sacramento 95815
Tel (916) 920-8098

Intel Corp
450 Executive Drive
Suite 150
San Diego 92111
(919) 453-5880

Intel Corp *
2000 East 4th Street
Suite 100
Santa Ana 92705
Tel (714) 835-9642
TWX 910-595-1114

Intel Corp *
1350 Shonard Way
Mt. View 94043
Tel (415) 968-8086
TWX 910-339-9279
910-338-0255

COLORADO

Intel Corp
4445 Northpark Drive
Suite 100
Colorado Springs 80907
Tel (303) 594-6622

Intel Corp *
650 S Cherry Street
Suite 720
Denver 80222
Tel (303) 321-8086
TWX 910-931-2289

CONNECTICUT

Intel Corp
26 Mill Plain Road
Danbury 06810
Tel (203) 748-9130
TWX 710-456-1199

EMC Corp
222 Summer Street
Stamford 06901
Tel (203) 327-2934

FLORIDA

Intel Corp
242 N Westmonte Drive
Suite 105
Altamonte Springs 32714
Tel (305) 869-5888

Intel Corp
1500 N.W. 62nd Street
Suite 104
Ft. Lauderdale 33309
Tel (305) 771-0600
TWX 510-956-9407

FLORIDA (Cont'd)

Intel Corp
11300 4th Street South
Suite 170
St. Petersburg 33702
Tel (815) 577-2413

GEORGIA

Intel Corp
9280 Pointe Parkway
Suite 200
Norcross 30092
Tel (404) 449-0541

ILLINOIS

Intel Corp *
2550 Gulf Road
Suite 815
Rolling Meadows 60008
Tel (312) 981-7200
TWX 910-651-5881

INDIANA

Intel Corp
8777 Purdue Road
Suite 125
Indianapolis 46268
Tel (317) 875-0623

IOWA

Intel Corp
St. Andrews Building
1850 St. Andrews Drive N.E.
Cedar Rapids 52402
Tel (319) 393-5510

KANSAS

Intel Corp
8400 W. 110th Street
Suite 170
Overland Park 66210
Tel (913) 642-8080

LOUISIANA

Intel Corp
Industrial Digital Systems Corp
Tel (504) 899-1654

MARYLAND

Intel Corp *
7321 Parkway Drive South
Suite C
Hanover 21076
Tel (301) 796-7500
TWX 710-862-1944

Intel Corp
7833 Walker Drive
Greenbelt 20770
Tel (301) 441-1020

MASSACHUSETTS

Intel Corp *
27 Industrial Avenue
Chelmsford 01824
Tel (617) 256-1800
TWX 710-343-6333

MICHIGAN

Intel Corp
7071 Orchard Lake Road
Suite 100
West Bloomfield 48033
Tel (313) 851-8096

MINNESOTA

Intel Corp
3500 W. 80th Street
Suite 360
Bloomington 55431
Tel (612) 836-8722
TWX 910-576-2867

MISSOURI

Intel Corp
4203 Earth City Expressway
Suite 131
Earth City 63045
Tel (314) 291-1990

NEW JERSEY

Intel Corp *
Raritan Plaza III
Raritan Center
Edison 08837
Tel (201) 225-3000
TWX 710-480-6238

NEW MEXICO

Intel Corp
8500 Manual Boulevard N.E.
Suite B 295
Albuquerque 87112
Tel (505) 292-8086

NEW YORK

Intel Corp *
300 Vanderbilt Motor Parkway
Hauppauge 11788
Tel (516) 231-3300
TWX 510-227-6236

Intel Corp
Suite 28 Hollowbrook Park
15 Myers Corners Road
Wappinger Falls 12590
Tel (914) 297-6161
TWX 510-248-0060

Intel Corp *
211 White Spruce Boulevard
Rochester 14623
Tel (716) 424-1050
TWX 510-253-7391

T-Squared
6443 Redings Road
Syracuse 13206
Tel (315) 463-8592
TWX 710-541-0554

T-Squared
7353 Pfister-Victor Road
Victor 14564
Tel (716) 924-9101
TWX 510-254-8542

NORTH CAROLINA

Intel Corp
2700 Wyckoff Road
Suite 102
Raleigh 27607
Tel (919) 781-8022

OHIO

Intel Corp *
6500 Poe Avenue
Dayton 45414
Tel (513) 890-5350
TWX 810-450-2528

Intel Corp
Chagrin-Branard Bldg., No 300
28001 Chagrin Boulevard
Cleveland 44122
Tel (216) 464-2736
TWX 810-427-9298

OKLAHOMA

Intel Corp
4157 S. Harvard Avenue
Suite 123
Tulsa 74135
Tel (918) 749-8688

OREGON

Intel Corp
10700 S.W. Beaverton
Hillsdale Highway
Suite 22
Beaverton 97005
Tel (503) 641-8086
TWX 910-467-8741

PENNSYLVANIA

Intel Corp *
455 Pennsylvania Avenue
Fort Washington 19034
Tel (215) 641-1000
TWX 510-661-2077

Intel Corp *
400 Penn Center Boulevard
Suite 610
Pittsburgh 15235
Tel (412) 823-4970

PENNSYLVANIA (Cont'd)

Q.E.D. Electronics
139 Tarwood Road
Willow Grove 19090
Tel (215) 657-5600

TEXAS

Intel Corp *
12300 Ford Road
Suite 390
Dallas 75234
Tel (214) 241-8087
TWX 910-860-5617

Intel Corp *
7322 S.W. Freeway
Suite 1490
Houston 77074
Tel (713) 988-8086
TWX 910-881-2490

Industrial Digital Systems Corp
5925 Sovereign
Suite 101
Houston 77036
Tel (713) 988-9421

Intel Corp
313 E. Anderson Lane
Suite 314
Austin 78752
Tel (512) 454-3628

UTAH

Intel Corp
5201 Green Street
Suite 290
Salt Lake City 84123
Tel (801) 263-8051

VIRGINIA

Intel Corp
1603 Santa Rosa Road
Suite 109
Richmond 23288
Tel (804) 282-5668

WASHINGTON

Intel Corp
110 110th Avenue N.E.
Suite 510
Bellevue 98004
Tel (206) 453-8086
TWX 910-443-3002

Intel Corp
408 N. Mullan Road
Suite 102
Spokane 99206
Tel (509) 928-8086

WISCONSIN

Intel Corp
450 N. Sunnyslope Road
Suite 130
Chancellor Park I
Brookfield 53005
Tel (414) 784-8087

CANADA

ONTARIO

Intel Semiconductor of Canada, Ltd
Suite 202, Bell Mews
39 Highway 7
Nepesin, K2H 8R2
Tel (613) 829-9714
TELEX 053-4115

Intel Semiconductor of Canada, Ltd
190 Athwell Drive
Suite 500
Rexdale, M9W 6H8
Tel (416) 875-2105
TELEX 0986574

QUEBEC

Intel Semiconductor of Canada, Ltd
3860 Cote Vertu Rd
Suite 210
St. Laurent H4R 1V4
Tel (514) 334-0560
TELEX 05-824172

*Field Application Location



DOMESTIC DISTRIBUTORS

ALABAMA

Arrow Electronics, Inc
3611 Memorial Parkway So
Huntsville 35801
Tel (205) 862-2730
TWX 810-726-2192

Hamilton/Avnet Electronics
4940 Research Drive
Huntsville 35805
Tel (205) 837-7210
TWX 810-726-2192

Pioneer Electronics
1207 Putnam Drive N W
Huntsville 35805
Tel (205) 837-9300
TWX 810-726-2197

ARIZONA

Hamilton/Avnet Electronics
505 S Madison Drive
Tempe 85281
Tel (602) 231-5140
TWX 810-950-0077

Wyle Distribution Group
8155 N 24th Avenue
Phoenix 85021
Tel (602) 249-2232
TWX 910-951-4282

CALIFORNIA

Arrow Electronics, Inc
521 Weddell Drive
Sunnyvale 94086
Tel (408) 745-6600
TWX 910-339-9371

Arrow Electronics, Inc
19748 Dearborn Street
Chatsworth 91311
Tel (213) 701-7500
TWX 910-493-2086

Arrow Electronics, Inc
2961 Dow Avenue
Tustin 92680
Tel (714) 838-5422
TWX 910-595-2860

Avnet Electronics
350 McCormick Avenue
Costa Mesa 92626
Tel (714) 754-6051
TWX 910-595-1928

Hamilton/Avnet Electronics
1175 Bordeaux Drive
Sunnyvale 94086
Tel (408) 743-3300
TWX 910-339-9332

Hamilton/Avnet Electronics
4545 Viewridge Avenue
San Diego 92123
Tel (619) 571-7500
TWX 910-595-2638

Hamilton/Avnet Electronics
20501 Plummer Street
Chatsworth 91311
Tel (213) 700-6271
TWX 910-494-2207

Hamilton/Avnet Electronics
4103 Northgate Boulevard
Sacramento 95834
Tel (916) 920-3150

Hamilton/Avnet Electronics
3002 G Street
Ontario 91311
Tel (714) 989-9411

Hamilton/Avnet Electronics
19515 So Vermont Avenue
Torrance 90502
Tel (213) 615-3913
TWX 910-349-6263

Hamilton Electro Sales
10912 W Washington Boulevard
Culver City 20238
Tel (213) 558-2458
TWX 910-340-6364

Hamilton Electro Sales
3170 Pullman Street
Costa Mesa 92626
Tel (714) 641-4150
TWX 910-595-2638

Hamilton Electro Sales
9650 De Soto Avenue
Chatsworth 91311
Tel (818) 700-6500

Kierulff Electronics, Inc
1180 Murphy Avenue
San Jose 95131
Tel (408) 947-3471
TWX 910-379-6430

CALIFORNIA (Cont'd)

Kierulff Electronics, Inc
14101 Franklin Avenue
Tustin 92680
Tel (714) 731-5711
TWX 910-595-2599

Kierulff/Avnet Electronics, Inc
5650 Jillson Avenue
Commerce 90040
Tel (213) 725-0325
TWX 910-580-3106

Wyle Distribution Group
124 Maryland Street
El Segundo 90245
Tel (213) 322-8100
TWX 910-348-7140 or 7111

Wyle Distribution Group
17872 Cowan Avenue
Irvine 92714
Tel (714) 843-9953
TWX 910-595-1572

Wyle Distribution Group
1151 Sun Center Drive
Rancho Cordova 95670
Tel (916) 638-8282

Wyle Distribution Group
952 Chesapeake Drive
San Diego 92123
Tel (619) 565-9171
TWX 910-335-1590

Wyle Distribution Group
3000 Sowers Avenue
Santa Clara 95051
Tel (408) 727-2500
TWX 910-535-0236

Wyle Military
17610 Teller Avenue
Irvine 92750
Tel (714) 851-9958
TWX 910-371-9127

Wyle Systems
15292 Bolsa Chica
Huntington Beach 92649
Tel (714) 851-9953
TWX 910-595-2642

Wyle Distribution Group
451 E 124th Avenue
Thornton 80241
Tel (303) 457-9953
TWX 910-936-0770

Hamilton/Avnet Electronics
8765 E Orchard Road
Suite 708
Englewood 80111
Tel (303) 740-1017
TWX 910-935-0787

CONNECTICUT
Arrow Electronics, Inc
12 Besaumont Road
Wallingford 06492
Tel (203) 265-7741
TWX 710-476-0162

Hamilton/Avnet Electronics
Commerce Industrial Park
Commerce Drive
Danbury 06810
Tel (203) 797-2800
TWX 710-456-9974

Pioneer Northeast Electronics
112 Main Street
Norwalk 06851
Tel (203) 853-1515
TWX 710-468-3373

FLORIDA
Arrow Electronics, Inc
1001 N W 82nd Street
Suite 108
Ft Lauderdale 33309
Tel (305) 776-7790
TWX 510-955-9456

Arrow Electronics, Inc
1530 Bottelbush Drive N E
Palm Bay 32905
Tel (305) 725-1480
TWX 510-959-6337

Hamilton/Avnet Electronics
6801 N W 15th Way
Ft Lauderdale 33309
Tel (305) 971-2900
TWX 510-955-3097

Hamilton/Avnet Electronics
9100 Gaither Road
St Petersburg 33702
Tel (813) 576-3920
TWX 810-863-0374

FLORIDA (Cont'd)

Hamilton/Avnet Electronics
6947 University Boulevard
Winterpark 32792
Tel (305) 629-3888
TWX 810-853-0322

Pioneer Electronics
221 N Lake Boulevard
Suite 412
Alta Monte Springs 32701
Tel (305) 834-9090
TWX 810-853-0284

Pioneer Electronics
1500 62nd Street N W
Suite 506
Ft Lauderdale 33309
Tel (305) 771-7620
TWX 510-955-9653

GEORGIA

Arrow Electronics, Inc
2979 Pacific Drive
Norcross 30071
Tel (404) 449-8252
TWX 810-766-0439

Hamilton/Avnet Electronics
5825 D Peachtree Corners
Norcross 30092
Tel (404) 447-7500
TWX 810-766-0432

Pioneer Electronics
5835B Peachtree Corners E
Norcross 30092
Norcross 30092
Tel (404) 448-1711
TWX 810-766-4515

ILLINOIS
Arrow Electronics, Inc
2000 E Algonquin Street
Schaumburg 60195
Tel (312) 397-3440
TWX 910-291-5544

Hamilton/Avnet Electronics
1150 Thorndale Avenue
Bensenville 60106
Tel (312) 860-7780
TWX 910-227-0060

Pioneer Electronics
1551 Carmen Drive
Elk Grove Village 60007
Tel (312) 457-9660
TWX 910-222-1834

INDIANA
Arrow Electronics, Inc
2718 Rand Road
Indianapolis 46241
(317) 243-8353
TWX 810-341-3119

Hamilton/Avnet Electronics
465 Grady Drive
Carmel 46032
Tel (317) 844-9333
TWX 810-260-3966

Pioneer Electronics
6408 Castleplace Drive
Indianapolis 46250
Tel (317) 848-7200
TWX 810-260-1794

KANSAS
Hamilton/Avnet Electronics
9219 Quivera Road
Overland Park 66215
Tel (913) 889-8900
TWX 910-743-0005

MARYLAND
Arrow Electronics, Inc
8300 Gifford Road #H
Rivers Center
Columbia 21046
Tel (301) 965-0003
TWX 710-236-9005

Hamilton/Avnet Electronics
8822 Oak Hill Lane
Columbia 21045
Tel (301) 995-3500
TWX 710-862-1861

Mesa Technology Corporation
16021 Industrial Drive
Gaithersburg 20877
Tel (301) 548-4350
TWX 710-828-9702

Pioneer Electronics
9100 Gaither Road
Gaithersburg 20877
Tel (301) 848-0710
TWX 710-828-0545

MASSACHUSETTS

Arrow Electronics, Inc
1 Arrow Drive
Woburn 01801
Tel (617) 933-8130
TWX 710-393-6770

Hamilton/Avnet Electronics
50 Tower Office Park
Woburn 01801
Tel (617) 933-9700
TWX 710-393-0382

Pioneer Northeast Electronics
44 Hartwell Avenue
Lexington 02173
Tel (617) 863-1200
TWX 710-326-8617

MICHIGAN

Arrow Electronics, Inc
3810 Varsity Drive
Ann Arbor 48104
Tel (313) 971-8220
TWX 810-223-6020

Pioneer Electronics
13485 Stamford
Livonia 48150
Tel (313) 525-1800
TWX 810-242-3271

Hamilton/Avnet Electronics
32467 Schoolcraft Road
Livonia 48150
Tel (313) 522-4700
TWX 810-242-8775

Hamilton/Avnet Electronics
2215 29th Street S E
Space A5
Grand Rapids 49508
Tel (616) 243-8805
TWX 810-273-6921

MINNESOTA
Arrow Electronics, Inc
5230 W 73rd Street
Edina 55435
Tel (612) 830-1800
TWX 910-576-3125

Hamilton/Avnet Electronics
10300 Bren Road East
Minnetonka 55343
Tel (612) 832-9600
TWX (910) 576-2720

Pioneer Electronics
10203 Bren Road East
Minnetonka 55343
Tel (612) 935-5444
TWX 910-576-2738

MISSOURI
Arrow Electronics, Inc
2380 Schuetz
St Louis 63141
Tel (314) 567-8888
TWX 910-764-0882

Hamilton/Avnet Electronics
13743 Shoreline Court
Earth City 63045
Tel (314) 344-1200
TWX 910-762-0684

NEW HAMPSHIRE
Arrow Electronics, Inc
1 Penmeter Road
Manchester 03103
Tel (603) 668-9988
TWX 710-220-1684

NEW JERSEY
Arrow Electronics, Inc
6000 Lincoln East
Marlton 08053
Tel (609) 928-1800
TWX 710-897-0829

Arrow Electronics, Inc
2 Industrial Road
Fairfield 07026
Tel (201) 575-5300
TWX 710-998-2206

Hamilton/Avnet Electronics
810 Keystone Avenue
Bldg 36
Cherry Hill 08003
Tel (609) 424-0110
TWX 710-940-0262

Hamilton/Avnet Electronics
10 Industrial
Fairfield 07026
Tel (201) 575-3390
TWX 710-734-4388



DOMESTIC DISTRIBUTORS

NEW JERSEY (Cont'd)

†Pioneer Northeast Electronics
45 Route 46
Pinebrook 07058
Tel (201) 575-3510
TWX 710-734-3392

†MTI Systems Sales
383 Route 46 W
Fairfield 07006
Tel (201) 227-5552

NEW MEXICO

†Alliance Electronics Inc
11030 Cochiti S E
Albuquerque 87133
Tel (505) 292-3360
TWX 910-989-1151

†Hamilton/Avnet Electronics
2524 Bavor Drive S E
Albuquerque 87106
Tel (505) 765-1500
TWX 910-989-0614

NEW YORK

†Arrow Electronics, Inc
25 Hub Drive
Melville 11751
Tel (516) 694-6800
TWX 510-224-6126

†Arrow Electronics, Inc
3000 South Winton Road
Rochester 14623
Tel (716) 275-0300
TWX 510-253-4766

†Arrow Electronics, Inc
705 Wallage Drive
Liverpool 13088
Tel (315) 652-1000
TWX 710-545-0230

†Arrow Electronics, Inc
20 Oser Avenue
Hauppauge 11788
Tel (516) 231-1000
TWX 510-227-6623

†Hamilton/Avnet Electronics
333 Metro Park
Rochester 14623
Tel (716) 475-9130
TWX 510-253-5470

†Hamilton/Avnet Electronics
16 Corporate Circle
E Syracuse 13257
Tel (315) 437-2641
TWX 710-541-1560

†Hamilton/Avnet Electronics
5 Hub Drive
Melville, Long Island 11747
Tel (516) 454-6000
TWX 510-224-6166

†Pioneer Northeast Electronics
1906 Vestal Parkway East
Vestal 13850
Tel (607) 748-8211
TWX 510-252-9893

†Pioneer Northeast Electronics
60 Crossway Park West
Woodbury, Long Island 11797
Tel (516) 921-8700
TWX 510-221-2184

†Pioneer Northeast Electronics
840 Fairport Park 14450
Tel (716) 381-7070
TWX 510-253-7001

†MTI Systems Sales
38 Harbor Park Drive
P O Box 271
Port Washington 11050
Tel (516) 621-6200
TWX 510-223-0846

NORTH CAROLINA

†Arrow Electronics, Inc
5240 Greenday Road
Raleigh 27604
Tel (919) 876-3132
TWX 510-929-1856

†Hamilton/Avnet Electronics
3510 Spring Forest Drive
Raleigh 27604
Tel (919) 878-0819
TWX 510-929-1836

†Pioneer Electronics
9801 A-Southern Pine Boulevard
Charlotte 28210
Tel (704) 524-8188
TWX 810-621-0366

OHIO

†Arrow Electronics, Inc
7620 McEwen Road
Centerville 45459
Tel (513) 435-5563
TWX 810-459-1611

†Arrow Electronics, Inc
8238 Cochran Road
Solon 44139
Tel (216) 248-3990
TWX 810-427-9409

†Hamilton/Avnet Electronics
954 Senate Drive
Dayton 45459
Tel (513) 433-0610
TWX 810-450-2531

†Hamilton/Avnet Electronics
4588 Emery Industrial Parkway
Warransville Heights 44128
Tel (216) 831-3500
TWX 810-427-9452

†Pioneer Electronics
4433 Interport Boulevard
Dayton 45424
Tel (513) 236-9900
TWX 810-459-1622

†Pioneer Electronics
4900 E 131st Street
Cleveland 44105
Tel (216) 587-3600
TWX 810-422-2211

OKLAHOMA

†Arrow Electronics, Inc
4719 S Memorial Drive
Tulsa 74145
Tel (918) 665-7700

OREGON

†Almac Electronics Corporation
8022 S W Nimbus, Bldg 7
Beaverton 97005
Tel (503) 641-9070
TWX 910-467-8743

†Hamilton/Avnet Electronics
8024 S W Jean Road
Bldg C, Suite 10
Lake Oswego 97034
Tel (503) 835-7048
TWX 910-455-8179

PENNSYLVANIA

†Arrow Electronics, Inc
650 Saco Road
Monroeville 15146
Tel (412) 856-7000

†Pioneer Electronics
259 Kappa Drive
Pittsburgh 15238
Tel (412) 782-2300
TWX 710-795-3122

PENNSYLVANIA (Cont'd)

†Pioneer Electronics
251 Cassatt Road
Horsham 19044
Tel (215) 674-4000
TWX 510-665-6778

TEXAS

†Arrow Electronics, Inc
3220 Commander Drive
Carrollton 75006
Tel (214) 380-6464
TWX 910-860-5377

†Arrow Electronics, Inc
19899 Kinghurst
Suite 100
Houston 77099
Tel (713) 530-4700
TWX 910-880-4439

†Arrow Electronics, Inc
2227 W Braker Lane
Austin 78758
Tel (512) 835-4180
TWX 910-874-1348

†Hamilton/Avnet Electronics
2401 Rutland
Austin 78757
Tel (512) 837-8911
TWX 910-874-1319

†Hamilton/Avnet Electronics
2111 W Walnut Hill Lane
Irving 75062
Tel (214) 659-4100
TWX 910-860-5029

†Hamilton/Avnet Electronics
8750 West Park
Houston 77063
Tel (713) 780-1771
TWX 910-881-5523

†Pioneer Electronics
9901 Burnet Road
Austin 78758
Tel (512) 835-4000
TWX 910-874-1323

†Pioneer Electronics
13710 Omega Road
Dallas 75234
Tel (214) 386-7300
TWX 910-850-5563

†Pioneer Electronics
5853 Point West Drive
Houston 77036
Tel (713) 968-5555
TWX 910-881-1606

UTAH

†Hamilton/Avnet Electronics
1585 West 2100 South
Salt Lake City 84119
Tel (801) 972-2600
TWX 910-925-4018

Wyle Distribution Group
1959 South 4130 West, Unit B
Salt Lake City 84104
Tel (801) 974-9953

WASHINGTON

†Almac Electronics Corporation
14360 S E Eastgate Way
Bellevue 98007
Tel (206) 643-9992
TWX 910-444-2067

†Arrow Electronics, Inc
14320 N E 21st Street
Bellevue 98007
Tel (206) 643-4800
TWX 910-444-2017

†Hamilton/Avnet Electronics
14212 N E 21st Street
Bellevue 98005
Tel (206) 453-5874
TWX 910-443-2469

WISCONSIN

†Arrow Electronics, Inc
430 W Reasson Avenue
Oakcreek 53154
Tel (414) 764-6600
TWX 910-262-1193

†Hamilton/Avnet Electronics
2975 Moorland Road
New Berlin 53151
Tel (414) 764-4510
TWX 910-262-1182

CANADA

ALBERTA

†Hamilton/Avnet Electronics
2816 21st Street, N.E.
Calgary T2E 6Z2
Tel (403) 230-3596
TWX 03-927-642

Zentronics
Bay No 1
3300 14th Avenue N.E.
Calgary T2A 6J4
Tel (403) 272-1021

BRITISH COLUMBIA

Zentronics
108-11400 Bridgeport Road
Richmond V6X 1T2
Tel (604) 273-5575
TWX 04-5077-89

MANITOBA

Zentronics
590 Berry Street
Winnipeg R3H 0S1
Tel (204) 775-8661

ONTARIO

Hamilton/Avnet Electronics
6845 Rawwood Road
Units G & H
Mississauga L4V 1R2
Tel (416) 677-7432
TWX 610-492-8867

Hamilton/Avnet Electronics
210 Colonnade Road South
Nepean K2E 7L5
Tel (613) 226-1700
TWX 05-349-71

Zentronics

8 Tilbury Court
Brampton L6T 3T4
Tel (416) 451-9600
TWX 06-976-78

Zentronics

564/10 Weber Street North
Waterloo N2L 5C6
Tel (519) 884-5700

Zentronics

155 Colonnade Road
Unit 17
Nepean K2E 7K1
Tel (613) 225-8840
TWX 06-976-78

QUEBEC

Hamilton/Avnet Electronics
2670 Sabourin Street
St Laurent H4S 1H2
Tel (514) 331-6443
TWX 610-421-3731

Zentronics

505 Locke Street
St Laurent H4T 1X7
Tel (514) 735-5361
TWX 05-927-535



EUROPEAN SALES OFFICES

BELGIUM

Intel Corporation S A
Parc Sery
Rue du Moulin a Papier 51
Boite 1
B-1160 Brussels
Tel. (02)681 07 11
TELEX 24814

DENMARK

Intel Denmark A/S*
Glentevej 61 - 3rd Floor
DK-2400 Copenhagen
Tel (01) 19 80 33
TELEX 19567

FINLAND

Intel Finland OY
Hameentie 103
SF - 00550 Helsinki 55
Tel 07/16 955
TELEX 123 332

FRANCE

Intel Corporation, S A R L *
5 Place de la Balance
Slic 222
94529 Rungis Cedex
Tel (01) 687 22 21
TELEX 270475

FRANCE (Cont'd)

Intel Corporation, S A R L
Immeuble BSC
4 Quai des Etoiles
69005 Lyon
Tel (7) 842 40 89
TELEX 305153

WEST GERMANY

Intel Semiconductor GmbH*
Seidstrasse 27
D-8000 München 2
Tel (89) 53891
TELEX 05-23177 INTL D

Intel Semiconductor GmbH*

Mainzer Strasse 75
D-6200 Wiesbaden 1
Tel (6121) 70 06 74
TELEX 04186183 INTW D

Intel Semiconductor GmbH

Bruckstrasse 61
7012 Fellbach
Stuttgart

Tel (71) 58 00 82

TELEX 7254826 INTS D

Intel Semiconductor GmbH*

Hohenzollern Strasse 5*
3000 Hannover 1
Tel (511) 34 40 81
TELEX 923825 INTH D

ISRAEL

Intel Semiconductor Ltd *
P O Box 1659
Haifa
Tel 4/524 261
TELEX 46511

ITALY

Intel Corporation Italia Spa*
Mianofori, Palazzo E
20094 Assago (Milano)
Tel (02) 824 00 06
TELEX 315183 INTML

NETHERLANDS

Intel Semiconductor Nederland B V *
Alexandriepoort Building
Marten Meesweg 93
3068 Rotterdam
Tel (10) 21 23 77
TELEX 22283

NORWAY

Intel Norway A/S
P O Box 82
Hvamveien 4
N-2013
Skjetten
Tel (2) 742 420
TELEX 18018

SPAIN

Intel Ibena
Calle Zurbaran 28
Madrid 04
Tel (34) 1410 40 04
TELEX 46880

SWEDEN

Intel Sweden A B *
Deivagen 24
S-17136 Solna
Tel (08) 734 01 00
TELEX 12261

SWITZERLAND

Intel Semiconductor A G *
Talsackerstrasse 17
8152 Glattpfug postfach
CH-8065 Zurich
Tel (01) 829 23 77
TELEX 57989 IOH CH

UNITED KINGDOM

Intel Corporation (UK) Ltd *
Fifers Way
Swindon, Wiltshire SN3 1RJ
Tel (0793) 488 389
TELEX 444447 INT SWN

*Field Application Location

EUROPEAN DISTRIBUTORS/REPRESENTATIVES

AUSTRIA

Bacher Elektronische Gerate GmbH
Rietermuhlgasse 28
A 1120 Vienna
Tel (222) 83 56 46
TELEX 11532 BASAT A

BELGIUM

Inelco Belgium SA
Ave des Croix de Guerre 94
B1120 Brussels
Tel (02) 216 01 60
TELEX 25441

DENMARK

ITT MultiKomponent A/S
Naverland 29
DK-2600 Glostrup
Tel (02) 45 86 45
TX. 33355

FINLAND

Oy Fintronic AB
Melkonkatu 24 A
SF-00210
Helsinki 21
Tel (0) 692 60 22
TELEX 124 224 Firon SF

FRANCE

Generm
Z I de Courtaboeuf
Avenue de la Bataille
91943 Les Ulis Cedex-B P 88
Tel (1) 907 78 78
TELEX F691700

Jermyn S A
16, Avenue Jean-Jaures
94600 Choisy-Le-Roi
Tel (1) 853 12 00
TELEX 289967

Metrologie
La Tour d'Asnieres
4, Avenue Laurent Cely
92806-Asnieres
Tel (1) 790 62 40
TELEX 611-448

Tekelec Artronic
Cite des Bruyeres
Rue Carle Vernet B P 2
92310 Sevres
Tel (1) 534 75 35
TELEX 204552

WEST GERMANY

Electronic 2000 Vertriebs A G
Sonniguberring 12
D-8000 Munich 82
Tel (89) 42 00 10
TELEX 522561 BIEC D

Jermyn GmbH
Postfach 1180
Schulstrasse 84
D-6277 Bad Camberg
Tel (06434) 231
TELEX 484426 JERM D

CES Computer Electronics Systems
GmbH
Gulenbergstrasse 4
2359 Herstedt-Utzburg
Tel (04193) 4026
TELEX 2180260

Metrologie GmbH
Herselstrasse 15
8000 Munich 21
Tel (89) 57 30 84
TELEX D 523189

Proelectron Vertriebs GmbH
Max Planck Strasse 1-3
6072 Dreieich bei Frankfurt
Tel (6103) 35564
TELEX 417983

IRELAND

Micro Marketing
Glenageary Office Park
Glenageary
Co Dublin
Tel (1) 85 62 88
TELEX 31584

ISRAEL

Eastronics Ltd
11 Rozans Street
P O Box 39300
Tel Aviv 61590
Tel (3) 47 51 51
TELEX 33638

ITALY

Eledra 3S S P A
Viale Evezia, 18
I 20154 Milano
Tel (2) 34 97 51
TELEX 332332

Intes
Mianofori Pal E/5
20090 Assago
Milano
Tel (02) 82470
TELEX 311351

NETHERLANDS

Koning & Hartman
Koperwerf 30
P O Box 43220
2544 EN's Gravenhage
Tel (31) 701 210 101
TELEX 31526

NORWAY

Nordisk Elektronic (Norge) A/S
Postoffice box 122
Smedsvingen 4
1364 Hvalstad
Tel (2) 846 210
TELEX 17546

PORTUGAL

Ditram
Componentes E Electronica LDA
Av Miguel Bombarda, 133
P1000 Lisboa
Tel (19) 545 313
TELEX 14182 Bneks-P

SPAIN

Interface S A
Av Pompeu Fabra 12
08024 Barcelona
Tel (3) 219 80 11
TELEX 61508

ITT SESA
Miguel Angel 21, 6 Pao
Madrid 10
Tel (34) 14 1954 00
TELEX 27461

SWEDEN

AB Gosta Backstrom
Box 12009
Astromergatan 22
S-10221 Stockholm 12
Tel (8) 541 080
TELEX 10135

Nordisk Elektronic AB
Box 27301
Sandhamnsgatan 71
S-10254 Stockholm
Tel (8) 635 040
TELEX 10547

Telko AB
Gardsfogevagen 1
Box 188
S-161 26 Bromma
Tel (8) 98 08 20
TELEX 11941

SWITZERLAND

Industrade AG
Hertstrasse 31
CH-8304 Wallisellen
Tel (01) 830 50 40
TELEX 56788 INDEL CH

UNITED KINGDOM

Bytech Ltd
Unit 57
London Road
Earley, Reading
Berkshire
Tel (0734) 61031
TELEX 848215

Conway Microsystems Ltd
Market Street
UK-Bracknell, Berkshire
Tel 44 (944) 55333
TELEX 847201

Jermyn Industries
Vestry Estate
Sevenoaks, Kent
Tel (0732) 450144
TELEX 95142

M E D I
East Lane Road
North Wembley
Middlesex HA9 7PP
Tel (190) 49307
TELEX 28817

Rapid Recall Ltd
Rapid House/Denmark 2T
High Wycombe
Berk, England HP11 2ER
Tel (0494) 26 271
TELEX 837931

YUGOSLAVIA

H R Microelectronics Enterprises
P O Box 5604
San Jose, California 95150
Tel 408/978-9000
TELEX 278-559



INTERNATIONAL SALES OFFICES

AUSTRALIA

Intel Australia Pty Ltd *
 (Mailing Address)
 P O Box 571
 North Sydney NSW, 2065

(Shipping Address)
 Spectrum Building
 200 Pacific Highway
 Level 6
 Crows Nest, NSW, 2065
 Tel 011-61-2-957-2744
 TELEX 790-20097
 FAX 011-61-2-957-2744

HONG KONG

Intel Semiconductor Ltd *
 1701-3 Connaught Centre
 1 Connaught Road
 Tel 011-852-5-215-311
 TWX 60410 ITHLK

JAPAN

Intel Japan K K
 5-6 Tokodai, Toyosato-machi
 Tsukuba-gun, Ibaraki-ken 300-26
 Tel 029747-8511
 TELEX 03656-160

Intel Japan K K *
 2-1-15 Naka-machi
 Atsugi, Kanagawa 243
 Tel 0462-23-3511

Intel Japan K K *
 2-51-2 Kojima-cho
 Chofu, Tokyo 182
 Tel 0424-88-3151

Intel Japan K K *
 2-69 Hon-cho
 Kumegaya, Saitama 360
 Tel 0485-24-6871

Intel Japan K K *
 2-4-1 Terauchi
 Toyonaka, Osaka 560
 Tel 06-863-1091

JAPAN (Cont'd)

Intel Japan K K
 1-5-1 Marunouchi
 Chiyoda-ku, Tokyo 100
 Tel 03-201-3621

Intel Japan K K *
 1-23-9 Shimmachi
 Setagaya-ku, Tokyo 154
 Tel 03-426-2231

Intel Japan K K *
 Mitsui-Seimei Musashi-Kosugi Bldg
 915 Shimmeruko, Nakahara-ku
 Kawasaki-Shi, Kanagawa 211
 Tel 044-733-7011

Intel Japan K K
 1-1 Shibahon-cho
 Mehta-mshi
 Shizuoka-ken 411
 Tel 0559-72-4121

KOREA

Intel Semiconductor Asia Ltd
 Singapore Bldg 8th Floor #906
 25-4 Yodo-Dong, Youngdeungpo-Ku
 Seoul 150
 Tel 011-82-2-784-8198 or 8286
 TELEX K29312 INTELKO

SINGAPORE

Intel Semiconductor Ltd
 101 Thomson Road
 25-06 Goldhill Square
 Singapore 1130
 Tel 011-65-2507811
 TWX RS 39921
 CABLE INTELSPG

*Field Application Location

INTERNATIONAL DISTRIBUTORS/REPRESENTATIVES

ARGENTINA

VLC S R L
 Sarmiento 1630, 1 Paso
 1042 Buenos Aires
 Tel 011-54-1-35-1201/9242
 TELEX 17675 EDARG

Agent
 Somex International Corporation
 15 Park Row, Room #1730
 New York, New York 10038
 Tel (212) 406-3052
 Attn Gaston Briones

AUSTRALIA

Total Electronics
 (Mailing Address)
 Private Bag 250
 Burwood, Victoria 3125

(Shipping Address)
 9 Harker Street
 Burwood
 Victoria 3125
 Tel 011-61-3-288-4044
 TELEX AA 31261

Total Electronics
 P O Box 139
 Artarmon, NSW 2064
 Tel 011-61-02-438-1855
 TELEX 26297

BRAZIL

Icontron S A
 05110 Av Mutinga 3650-6 Andar
 Piratuba Sao Paulo
 Tel 011-55-11-833-2572
 TELEX 1122274 ICOTBR

CHILE

DIN
 (Mailing Address)
 Av VIC. Mackenna 204
 Casilla 6055
 Santiago
 Tel 011-56-2-277-564
 TELEX 352-0003

(Shipping Address)
 A102 Greenville Center
 3801 Kennett Pike
 Wilmington, Delaware 19807

HONG KONG

Novel Precision Machinery Co, Ltd
 Flat D 20 Kingsford Ind Bldg
 Phase 1 26 Kwai Hai Street NT
 Tel 011-852-5-0-232222
 TWX 39114 JINMI HK

Schmidt & Co Ltd
 18/F Great Eagle Centre
 Wancha
 Tel 011-852-5-833-0222
 TWX 74766 SCHMC HK

INDIA

Micronic Devices
 65 ARUN Complex
 D V G Road
 Basavan Gudi
 Bangalore 560004
 Tel 011-91-812-600-631
 TELEX 011-5947 MDEV

Micronic Devices
 104/105C Nirmal Industrial Estate
 Sion (E)
 Bombay 400022
 Tel 011-91-22-48-61-70
 TELEX 011-71447 MDEV IN

Micronic Devices
 R-694 New Rajinder Nager
 New Delhi 110060

Ramtek International, Inc (Agent)
 465 S Mathilda Avenue
 Suite 302
 Sunnyvale, CA 94096
 Tel (408) 733-8767

S & S Corporation
 (Mailing Address)
 P O Box 1185
 Maudlin, South Carolina 29657

(Shipping Address)
 308 Green Drive
 Liberty, South Carolina 29657

JAPAN

Asahi Electronics Co Ltd
 KMM Bldg Room 407
 2-14-1 Asano, Kokurakita-Ku
 Kitayushu City 802
 Tel (083) 511-6471
 TELEX AECKY 7126-16

JAPAN (Cont'd)

Hamilton-Avnet Electronics Japan Ltd
 YUI and YUJI Bldg 1-5-7 Hondome-
 Cho
 Nishinbashi Chuoh-Ku, Tokyo 103
 Tel (03) 682-9911
 TELEX 2523774

Ryoyo Electric Corporation
 Konwa Bldg
 1-12-22, Tsukiji
 Chuo-Ku, Tokyo 104
 Tel (03) 543-7711/541-7311

Tokyo Electron Ltd
 Shinjuku Nomura Bldg
 26-2 Nishi-Shinjuku 1-Chome
 Shinjuku-Ku, Tokyo 160
 Tel (03) 343-4411
 TELEX 232-2220 LABTEL J

KOREA

J-TEK Corporation
 2nd floor, Government Pension Bldg
 24-3, Yodo-Dong
 Youngdeungpo-Ku
 Seoul 150
 Tel 011-82-2-782-8039
 TELEX KODIGIT K25299

Koram Digital USA (Agent)
 14966 East Freestone Boulevard
 Santa Fe Springs, CA 90670
 Tel (714) 739-2204
 TWX 194715 KORAM DIGIT USA

NEW ZEALAND

McLean Information Technology Ltd
 459 Kyber Pass Road, Newmarket,
 P O Box 9484, Newmarket
 Auckland 1, New Zealand
 Tel 011-64-9-501-219, 501-801, 587-
 037
 TELEX NZ21570 THERMAL

PAKISTAN

Computer Applications Ltd
 7D Gazi Boulevard
 Defense
 Karachi-46
 Tel 011-92-21-530-306/7
 TELEX 24434 GAFAR PK

PAKISTAN (Cont'd)

Horizon Training Co, Inc (Agent)
 1 Lafayette Center
 1100 20th Street N W
 Suite 530
 Washington, D C 20036
 Tel (202) 887-1900
 TWX 248890 HORN

SINGAPORE

General Engineers Corporation Pty
 Ltd
 Units 1003-1008 Block 3
 10th Floor PSA Multi Storey Complex
 Telok Blangah Pasir
 Pan Jang
 Singapore 5
 Tel 011-65-271-3163
 TELEX RS23987 GENERCO
 CABLE GENRECORP

SOUTH AFRICA

Electronic Building Elements, Pty Ltd
 P O Box 4609
 Pretoria 0001
 Tel 011-27-12-46-9221
 TELEX 3-22786 SA
 TELEGRAM ELBLEM

TAIWAN

Milac Corporation
 3rd Floor #75, Section 4
 Nanjing East Road
 Taipei
 Tel 011-886-2-771-0940, 0941
 TELEX 11942 TAIAUTO

Mactel International, Inc (Agent)
 3385 Vaso Court
 Santa Clara, CA 95050
 Tel (408) 388-4513
 TWX 910-338-2201
 FAX 408-980-9742

YUGOSLAVIA

H R Microelectronics Enterprises
 P O Box 5604
 San Jose, California 95150
 Tel (408) 978-8000
 TELEX 278-558

*Field Application Location



DOMESTIC SERVICE OFFICES

CALIFORNIA

Intel Corp
1350 Shorebird Way
Mt View 94043
Tel (415) 968-8211
TWX 910-339-9279
910-338-0255

Intel Corp
2000 E 4th Street
Suite 110
Santa Ana 92705
Tel (714) 835-5577
TWX 910-595-2475

Intel Corp
4350 Executive Drive
Suite 150
San Diego 92121
Tel (619) 452-5880

Intel Corp
5550 N Corbin Avenue
Suite 120
Tarzana 91356
Tel (213) 708-0333

COLORADO

Intel Corp
650 South Cherry
Suite 720
Denver 80222
Tel (303) 321-8086
TWX 910-931-2289

CONNECTICUT

Intel Corp
26 Mill Plain Road
Danbury 06811
Tel (203) 748-3130

FLORIDA

Intel Corp
1500 N W 62nd Street
Suite 104
Ft Lauderdale 33309
Tel (305) 771-0600
TWX 510-956-9407

FLORIDA (Cont'd)

Intel Corp
500 N Westland Avenue
Suite 205
Maitland 32751
Tel (305) 628-2393
TWX 810-853-9219

GEORGIA

Intel Corp
3280 Pointe Parkway
Suite 200
Norcross 30092
Tel (404) 441-1171

ILLINOIS

Intel Corp
2550 Golf Road
Suite 815
Rolling Meadows 60008
Tel (312) 961-7295
TWX 910-253-1825

KANSAS

Intel Corp
8400 W 110th Street
Suite 170
Overland Park 66210
Tel (913) 642-8060

MARYLAND

Intel Corp
5th Floor Product Service
7833 Walker Drive
Greenbelt 20770
Tel (301) 441-1020

MASSACHUSETTS

Intel Corp
27 Industrial Avenue
Chelmsford 01824
Tel (617) 256-1800
TWX 710-343-6333

MICHIGAN

Intel Corp
7071 Orchard Lake Road
Suite 100
West Bloomfield 48033
Tel (313) 851-8905

MISSOURI

Intel Corp
4203 Earth City Expressway
Suite 143
Earth City 63045
Tel (314) 291-2015

NEW JERSEY

Intel Corp
365 Sylvan Avenue
Englewood Cliffs 07632
Tel (201) 567-0820
TWX 710-991-8593

Intel Corp
Rantan Plaza III
Rantan Center
Edison 08817
Tel (201) 225-3000

NORTH CAROLINA

Intel Corp
2306 W Meadowview Road
Suite 206
Greensboro 27407
Tel (919) 294-1541

OHIO

Intel Corp
Chagrin-Brainard Bldg
Suite 305
28001 Chagrin Boulevard
Cleveland 44122
Tel (216) 464-6915
TWX 810-427-9298

Intel Corp
6500 Poe
Dayton 45414
Tel (513) 890-5350

OREGON

Intel Corp
10700 S W Beaverton-Hillsdale
Highway
Suite 22
Beaverton 97005
Tel (503) 641-8086
TWX 910-467-8741

Intel Corp
5200 N E Elm Young Parkway
Hillsboro 97123
Tel (503) 681-8080

PENNSYLVANIA

Intel Corp
201 Penn Center Boulevard
Suite 301 W
Pittsburgh 15235
Tel (313) 354-1540

TEXAS

Intel Corp
313 E Anderson Lane
Suite 314
Austin 78752
Tel (512) 454-3628
TWX 910-874-1347

Intel Corp
12300 Ford Road
Suite 380
Dallas 75234
Tel (214) 241-8087
TWX 910-860-5617

WASHINGTON

Intel Corp
110 110th Avenue NE
Suite 510
Bellevue 98004
Tel 1-800-525-5560
TWX 910-443-3002

WISCONSIN

Intel Corp
450 N Sunnyslope Road
Suite 130
Brookfield 53005
Tel (414) 794-8087



UNITED STATES
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

JAPAN
Intel Japan K.K.
5-6 Tokodai Toyosato-machi
Tsukuba-gun, Ibaraki-ken 300-26
Japan

FRANCE
Intel
5 Place de la Balance
Silic 223
94528 Rungis Cedex
France

UNITED KINGDOM
Intel
Piper's Way
Swindon
Wiltshire, England SN3 1RJ

WEST GERMANY
Intel
Seidstrasse 27
D-8000 Munchen 2
West Germany