Using the iRMX 86™ Operating System

Steve Vestuye
OEM Microcomputer Systems Applications
RELATED INTEL PUBLICATIONS

Introduction to the iRMX 86™ Operating System 9803124
iRMX 86™ Nucleus, Terminal Handler, and Debugger Reference Manual 9803122
iRMX 86™ I/O System Reference Manual 9803123
iRMX 86™ System Programmer’s Reference Manual 142721
iRMX 86™ Installation Guide for ISIS-II Users 9803125
iRMX 86™ Configuration Guide for ISIS-II Users 9803126
The 8086 Family User’s Manual 9800722
iSBC 86/12™ Single Board Computer Hardware Reference Manual 9800645
iSBC 534™ Four Channel Communications Expansion Board Hardware Reference Manual 9800450
PL/M 86 Programming Manual 9800466
MSC-86™ Assembly Language Reference Manual 9800640
MCS-86™ Software Development Utilities Operating Instructions for ISIS-II Users 9800639

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 APR 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:

ICE    Prompt    Micromap    Intel    ICS
iAPX    iSBC      UPI        MCS     CREDIT
Intellec Insite    iSBX       MULTIBUS iRMX
Megachassis Library Manager MULTIMODULE Scope

and the combinations of iAPX, ICE, iSBC, MCS, iSBX or iRMX and a numerical suffix.

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

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

© INTEL CORPORATION, 1980
Using the iRMX 86™ Operating System

Contents

INTRODUCTION ............................................. 1

OVERVIEW .................................................. 1

INTRODUCTION TO THE iRMX 86 OPERATING SYSTEM .............. 4

Architecture .............................................. 5
Tasks ......................................................... 5
Job and Free Space Management ................................ 5
Segments ................................................... 6
Communication & Synchronization ............................... 7
Interrupt Management ....................................... 7
Error Management .......................................... 7
Asynchronous I/O ........................................... 7
Synchronous I/O ........................................... 8
Loaders ....................................................... 8
File Management ........................................... 9
Human Interface Subsystem .................................. 9
Debugging Subsystem ...................................... 9
Configuration and Initialization ................................ 9

DESIGN METHODOLOGY .................................... 10

Application Example 1 ..................................... 10
System Requirements ...................................... 10
Hardware Requirements ................................... 10
System Design ............................................. 11

Application Example 2 ..................................... 13
Overview of Device Driver Construction ....................... 13
Design of an iSBC 534™ Driver ............................. 15

CODE EXAMPLES .......................................... 16

APPENDIX A .............................................. 25
Code Listings

APPENDIX B .............................................. 51
Configuration Listings/Worksheets
INTRODUCTION

Companies seeking to develop microcomputer applications are faced with two significant problems. First, applications are growing more and more sophisticated. With competition always present, products are continually being enhanced with new features. This burdens the underlying computer system by increasing both the complexity of the software and the number of events and functions that must be handled by the system.

The second problem is a management problem. These newer and more sophisticated application systems must be developed quickly in order to hit shrinking market windows. Also, they must be developed with lower manpower costs to be feasible in an engineering community struck by insufficient technical personnel and skyrocketing software development costs.

These are the needs addressed by the iRMX 86™ Operating System. The two goals in the development of this product have been power/flexibility to meet the needs of increasingly complex application systems, and ease of understanding and use, to boost the productivity of available engineering resources. Users of Intel’s line of iSBC 86™ Single Board Computers or custom-designed 8086-based boards can now obtain the same benefits from Intel supplied system software as they can from Intel supplied system hardware.

The reader of this application note is provided with information in four subject areas.

- The requirements of operating systems are discussed along with traditional solutions.
- The iRMX 86 Operating System is introduced and its features are discussed in relation to the requirements studied earlier.
- System design using the iRMX 86 Operating System is studied using example solutions.
- Code for two example systems is examined to learn the details of system implementation.

Some of the topics in this note may not be of interest to all readers. For example, an experienced real-time programmer may not need to read the entire overview of real-time systems. For those who want to brush up on a few topics, the overview is organized to allow the reader to focus attention on areas of specific interest.

Throughout this application note, various terms and concepts are introduced and discussed. If further information on any of these topics is desired, the references listed in the front of this note should be used.

OVERVIEW

This overview is provided to investigate both the problems encountered in the design of applications software and also the classical solutions to these problems.

Multitasking

A real-time system is defined to be a system that reacts to events occurring external to the computer and which monitors or controls these events as they occur (or in “real-time”). The converse of a real-time system is known as a batch system where the outcome of a program does not depend on when it is run (for example, a payroll program).

Two other characteristics typically encountered in a real-time system are asynchronous event occurrences and concurrent activity. The first characteristic is caused by events occurring randomly rather than at scheduled intervals. The second characteristic, concurrent activity, takes place when two or more events occur nearly at the same time, requiring simultaneous activity.

One method of dealing with the requirements of a real-time system would be to write a program that knows what events could potentially occur (for example, an interrupt occurrence, a real-time clock counting down to zero, a byte in memory being modified by another program). This program could then execute a large loop checking for the occurrence of these events.

There are several problems with this approach. While processing one event which has occurred, the program is not responsive to other events. Also, the programmer has no way of prioritizing the importance of the various events. From a maintenance standpoint, this program is complex and difficult to enhance or modify.

The traditional solution to these problems is a technique called multitasking. Essentially, this involves writing many small routines instead of one large one. Each of these routines (tasks) can process events independent of the other tasks in the system. In addition, a priority can be assigned each task so that the operating system can decide as to which task is the most important when more than one task requests control of the CPU.

The support for multitasking involves a scheduler which is part of the service provided by the operating system. The scheduler allows each task to execute its program as if it has sole control of the CPU, ensuring that all tasks desiring CPU time are serviced according to the priority associated with each task.
From the standpoint of system design, multitasking has many desirable qualities. Large and potentially complex application programs can be decomposed into smaller more manageable units. This makes feasible the use of programmer teams to implement the application. Perhaps even more importantly, the potentially overwhelming problems surrounding concurrent execution and interrupt handling become transparent to the application programmer. Also, multitasking makes the modification of existing tasks and the addition of new ones become a manageable objective since the interaction between tasks is minimized.

Interrupt Handling
A common event in a real-time system is the occurrence of an interrupt. Because this event is so common, an important feature of a real-time operating system is its interrupt processing capabilities.

From the standpoint of application software, interrupt handling can be cumbersome. The currently running task must be preempted, various hardware devices must be manipulated and perhaps a hardware interrupt controller must be dealt with.

A real-time operating system can abstract the occurrence of an interrupt into something more consistent with the way other events are handled. A task can simply inform the scheduler that it does not require any CPU time until an interrupt occurs. The relative priority of different interrupts can also be handled in the same manner as the priority of multiple tasks are handled. Thus, the application programmer need only deal with the actual processing related to interrupt occurrence.

Reliability
Reliability is a keyword in all real-time systems. In this type of system, reliability does not refer to mean time between failure. In fact, the software in a real-time application typically cannot be allowed to fail. The difficulty imposed on the software by the environment comes from the near infinite number of permutations that can occur. A system that appears to be fully debugged can fail in the field because of a combination of simultaneous events that never occurred before.

The only means to avoid failure in these instances is through the use of a consistent, well-thought-out model for handling events. Any special-cased solution is subject to failure when the special cases that were designed for are violated in the real world.

Error handling can also add reliability to an application system. When the application software is unable to anticipate the outcome of certain conditions, or the software has undiscovered bugs, it is vital for the operating system to gracefully handle the situation and allow for further processing to continue as best as possible.

I/O Handling
Many applications for 16-bit microcomputers require a variety of I/O devices. The support for I/O operations on these devices is typically provided by the operating system. Both sequential access and random access devices are typically encountered and, in addition, flexibility in handling I/O requests and acknowledgements is important.

The flexibility necessary typically involves the scheduling of a task's execution after an I/O request has been made. The greatest flexibility can be obtained by an asynchronous I/O system. In this system, a task makes an I/O request by calling the operating system. Once the processing of the request has begun, control is returned to the calling task.

In this manner the task can continue executing its program while the I/O operation is progressing. When the results of the operation are desired, the task can call the operating system again to wait for the completion of the previous I/O request.

The second type of I/O support is less flexible but also easier to use. An operating system that supports synchronous I/O allows a task to make a single operating system call to make an I/O request. Once control is returned to the calling task, the I/O operation is complete and the results are immediately available. This type of I/O support sometimes takes advantage of a technique known as autobuffering to regain some of the performance advantage of the overlapped I/O found in the asynchronous system.

Debug Support
The inherent characteristics of the real-time environment sometimes make it difficult to debug new software. If the simultaneous occurrence of two events causes a bug in the software, detection may be difficult because the next time the system is run the error is not reproduced. Also, because of the fact that the software is broken down into many independent tasks, the interaction may be difficult to track using standard debugging techniques.

The solution to these problems is a piece of software called the system debugger. The debugger typically has three characteristics.
1) It is designed to interact with the operating system and therefore has intimate knowledge of code, data structures and system objects.

2) Since the debugger is just another task in the system, it does not affect the operation of the other tasks that are running.

3) Through the use of sophisticated breakpointing facilities, the debugger allows the designer to track the tasks in the system, investigate their interaction with other tasks and selectively stop one or more tasks without stopping the entire system.

**Multiprogramming**

In some application systems, there arises the requirement to run several "applications" on the computer at the same time. This may be due to the desire to squeeze more use out of the hardware or it may be due to some system design consideration. These separate "applications" (often termed jobs) share many system resources (especially the CPU) but at the same time they need to be protected as much as possible from other jobs. In essence, it should be possible to develop two jobs independently and then run them both on the same hardware without any interaction. If interaction is desired, the operating system should support some well-defined protocol for jobs to use to communicate.

**Free Space Management**

One of the most important resources in the computer system is the memory. In some applications, the amount of memory needed can be determined when the system is designed. In the more general case, the amount of memory needed by the system fluctuates. One solution to this management problem is to have available the amount needed in the worst possible case. A more flexible and economical solution is to dynamically allocate memory from a central pool upon demand and return it when possible. This service provides two tangible advantages. First, total memory needs are reduced. Second, this service allows for ease of use by the application programmer because there is no need to set aside blocks of memory and implement code to maintain information about current usage.

**File Management**

The ability to easily store and retrieve data stored on mass storage devices is a requirement in many application systems. Devices such as disks, tapes and bubble memories are used to store program code, data files and parameter tables. The operating system is called upon to store and retrieve the data and organize it such that application programs can easily find and manipulate the data when necessary.

![Figure 1. Hierarchical File System](image)
Device Independence

One of the unfortunate characteristics of I/O devices is that they all tend to present different interfaces to the system software. When this is the case, the application programmer must become familiar with the unique characteristics of each device in order to communicate with it. One solution is to create an I/O driver which does the actual I/O. This driver can then be called by the application program whenever communication with the device is desired.

The problem with this solution is that the programmer must still know what type of device is being talked to since the I/O driver is specialized. If the system configuration changes, all of the software must be rewritten to call new device drivers. The best solution is to design a standard interface to device drivers and postpone until run-time the decision about which devices to use. With this type of system, an application program can be written assuming that at run-time the human or program that invokes it will provide a specification of which devices should be used.

High-Level Man-Machine Interface

In addition to the services provided for application programs by the operating system, a set of services typically is offered to the human user sitting at the system console. System utilities are needed for file copying, disk formatting, and directory maintenance. Programs need to be loaded off disk to run and the programs themselves must be able to retrieve parameters passed to them by the operator. All of these functions are usually provided by the man-machine interface software in the operating system.

Make Versus Buy

The previous sections dealt with operating system requirements. These requirements are encountered in the application development process. Whether the solution to meet the needs comes from the individual application designer or from a computer system vendor, the requirements do not change.

There usually exists a rather simple tradeoff between designing a custom operating system or buying a generalized system and tailoring it to the individual needs of the application. There are advantages to the custom solution. The system can often be made smaller since the requirements are known in great detail. Also, some small performance improvements can sometimes be made by taking advantage of the special cases to speed things up.

Buying an operating system from a computer system vendor offers five advantages.

1) Engineering resources are becoming scarce. The use of an operating system from a vendor allows attention to be focused on the application software.
2) The time taken to bring the product to market can be shortened, thereby gaining a competitive edge and generating early revenue.
3) Long-term maintenance costs can be reduced because the vendor supports the operating system software.
4) Personnel in all branches of the company can become familiar with one software architecture and apply this knowledge to a range of products. This applies not only to the design engineers, but also to quality assurance, customer engineers and system analysts.
5) The computer system vendor has knowledge of future technological advances coming in the product lines. For this reason, the operating system can be constructed so that applications software can be transported to future hardware without the need for expensive redesign.

In summary, the trade-offs are clear. An operating system from a computer system vendor is not the answer for every application. But in most cases, the most economical and safest bet is to take advantage of the expertise of the vendor for the system software and use engineering resources to more quickly solve the application problem.

INTRODUCTION TO THE iRMX 86™ OPERATING SYSTEM

The iRMX 86 Operating System meets the needs of real-time applications while simultaneously providing the full set of services normally found in a general-purpose operating system.

The overall picture of the iRMX 86 Operating System is shown in Figure 2. The iRMX 86 Nucleus provides

![Figure 2. Layers of Support in the iRMX 86™ System](image-url)
support for multitasking, multiprogramming, inter-
task communication, interrupt handling and error
checking. The Basic I/O System provides support for
device independent and file format independent
manipulation of data on I/O devices. The Extended I/O
system provides synchronous I/O calls, automatic
buffering, logical file name support and high-level job
management. The application loader provides the
ability to load code and data from mass storage devices
into RAM memory. The Human Interface provides for
a high-level man-machine interface as well as file
utilities and parsing support for application programs.

The following sections deal in more detail with each of
these iRMX 86 pieces. If more information is desired on
the features discussed, please refer to the documents
listed in the front of this application note.

Architecture
The iRMX 86 architecture is an object-oriented archi-
tecture. This means that the operating system is
organized as a collection of building blocks that are
manipulated by operators. The building blocks of the
iRMX 86 system are called objects and are of several
types. Some of the object types are tasks, jobs, mail-
boxes, semaphores and segments. These types are
explained in subsequent sections of this application
note.

This type of architecture has two major advantages.
First, the system is easier to learn and use. The at-
tributes of the various objects and the operations that
can be performed on them are well defined and con-
sistent. Once an object type is understood, all objects
of that type are understood.

The second advantage to an object-oriented archi-
tecture is the ease with which the operating system
can be tailored to the application. If there is no need
for a given object in the application, all operators for
that object are not included in the final configured
system. On the other hand, if the application designer
needs a more complex building block that is not in the
basic system, he can define and use a new object type.

Table 1 lists all of the system calls in the iRMX 86
Nucleus. There are three groupings of system calls in
this table.

1) The general system calls apply to all objects uni-
momously.
2) The first two system calls for each object are the
create and delete calls. These calls simply create a
new object and initialize its attributes or delete an
existing object.
3) The remaining system calls are specific to the at-
tributes of a particular object. With this organiza-
tion in mind, the entire operation of the iRMX 86
nucleus can be glimpsed in a single table.

Tasks
Tasks are the active objects in the iRMX 86 archi-
tecture. Tasks execute program code and therefore are
the only objects that can manipulate other objects. The
attributes of a task include its program counter, stack,
priority and dispatcher state.

Tasks compete with each other for CPU time and the
iRMX 86 scheduler determines which task to run based
upon priorities. The dispatcher states for an iRMX 86
task are shown in Figure 3. At any given point in time,
the highest priority task that is ready to run has
control of the CPU. Control is transferred to another
task only when

![Figure 3. Task State Transition Diagram](image)

1) the running task makes a request that cannot im-
mediately be filled and is, therefore, moved to the
asleep state,
2) an interrupt occurs causing a higher-priority task to
become ready to run or
3) the running task causes a higher-priority asleep
task to become ready by releasing some resource.

The suspended and asleep-suspended states are
entered whenever the suspend system call is invoked
for a particular task.

Job and Free Space Management
Support for multiprogramming is provided by the job
object. A job provides the environment for tasks to
execute their programs. All other objects needed for a
particular application are contained within the job.
Table 1. Nucleus Object Management System Calls

<table>
<thead>
<tr>
<th>System Calls for All Objects</th>
<th>O.S. Objects</th>
<th>Attributes</th>
<th>Object-Specific System Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOBS</td>
<td>Tasks</td>
<td></td>
<td>CREATE$JOB DELETE$JOB SET$POOL$MIN GET$POOL$ATTRIB OFFSPRING</td>
</tr>
<tr>
<td></td>
<td>Memory pool</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Object directory</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Exception handler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TASKS</td>
<td>Priority</td>
<td></td>
<td>CREATE$TASK DELETE$TASK SUSPEND$TASK RESUME$TASK GET$EXCEPTION$HANDLER SET$EXCEPTION$HANDLER SLEEP</td>
</tr>
<tr>
<td></td>
<td>Stack</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Code</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>State</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Exception handler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEGMENTS</td>
<td>Buffer with length</td>
<td></td>
<td>CREATE$SEGMENT DELETE$SEGMENT GET$SIZE</td>
</tr>
<tr>
<td>MAILBOXES</td>
<td>List of objects</td>
<td></td>
<td>CREATE$MAILBOX DELETE$MAILBOX SEND$MESSAGE RECEIVE$MESSAGE</td>
</tr>
<tr>
<td></td>
<td>List of tasks waiting for objects</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEMAPHORES</td>
<td>Semaphore unit value</td>
<td></td>
<td>CREATE$SEMAPHORE DELETE$SEMAPHORE RECEIVE$UNITS SEND$UNITS</td>
</tr>
<tr>
<td></td>
<td>List of tasks waiting for units</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REGIONS</td>
<td>List of tasks waiting for critical section</td>
<td></td>
<td>CREATE$REGION DELETE$REGION RECEIVE$CONTROL ACCEPT$CONTROL SEND$CONTROL</td>
</tr>
<tr>
<td>USER OBJECTS</td>
<td>License rights to a given extension type</td>
<td></td>
<td>CREATE$EXTENSION DELETE$EXTENSION</td>
</tr>
<tr>
<td></td>
<td>New object template</td>
<td></td>
<td>CREATE$COMPOSITE DELETE$COMPOSITE INSPECT$COMPOSITE ALTER$COMPOSITE</td>
</tr>
</tbody>
</table>

A specific attribute of the job is a free memory pool from which blocks can be allocated only by tasks within the job. Also, the job contains an object directory which can be used by tasks to catalog objects under ASCII names so that other tasks, knowing the ASCII name, can look up the object and thereby gain addressability to it.

More than one job can co-exist in the computer system. Tasks within jobs can also create children jobs forming a hierarchical tree of jobs (see Figure 4). Each job in the system has its unique set of contained objects, its own memory pool and its own object directory.

**Segments**

A fundamental resource that tasks need is memory. Memory is allocated to tasks in the form of the segment object. The segment is a block of contiguous memory. The attributes of a segment are its base address and size. A task needing memory requests a segment of whatever size it requires. The Nucleus attempts to create a segment from the memory pool given to the task's job when the job was created.

Figure 4. iRMX 86™ Job Tree Example
If there is not enough memory available, the Nucleus will try to get the needed memory from ancestors of the job.

**Communication and Synchronization**

In many cases it is necessary for two tasks to communicate in order to exchange data and commands. This is supported through the use of an object known as a mailbox. As its name implies, a mailbox is a holding place for objects. One task can send an object to a mailbox, causing the object to be queued there. Another task can later receive an object from the mailbox and thereby gain access to it (see Figure 5). If a task tries to receive an object from a mailbox and there are no objects there, the task can optionally be made to sleep for a specified time for an object to appear.

![Figure 5. Intertask Communication via Mailboxes](image)

Note that any object can be sent to a mailbox to be received by another task. Typically, the object sent is a segment which is a block of memory and can contain any commands or data. The term message is often used to describe the object during the time it is being sent through a mailbox.

In those cases where there is a requirement for synchronization between tasks but no data need be sent, a simpler more efficient mechanism exists. The semaphore object provides for the allocation of abstract entities called units. The primary attribute of the semaphore is an integer number. Tasks may send units to a semaphore thereby increasing the integer number or they can request units, thereby decreasing the number. If a task makes a request for more units than are available, it can optionally be made to sleep for a specified amount of time. This mechanism can be used for synchronization, resource allocation and mutual exclusion.

**Interrupt Management**

When an interrupt is sensed by the 8086 hardware, a user interrupt handler is executed. The interrupt handler can either perform all interrupt processing itself without making any iRMX 86 system calls, or it can signal an interrupt task allowing more general interrupt processing including calls to the operating system.

The operating system maps hardware interrupt priorities into the software priority scheme allowing the designer to specify what software functions are important enough to have some interrupt levels masked off during their execution. Although this mapping should always be kept in mind during design, the mechanics of dealing with interrupt control are handled by the operating system.

**Error Management**

One of the central themes in the design of the iRMX 86 operating system has been reliability. The results of these efforts are evident in two particular features of the architecture. Beyond the ease of understanding brought about by the symmetry of the system, the reliability of applications using the iRMX 86 software is increased.

In the event that errors do occur, the operating system is set to detect them. Virtually all parameters in calls to the operating system are checked for validity. Any inconsistency causes a jump to an error routine to handle the problem. Two types of errors can potentially occur and there are two ways of handling errors.

The first error type is the programmer error condition which comes about due to some mistake in the coding of a system call. The second type is an environmental condition which arises due to factors out of the control of the engineer (e.g. insufficient memory). Each of these error types can be handled in-line by checking a status code upon return from the call or can cause an error handling subroutine to be called by the system. The system designer can choose the desired method for the system, for a specific job, and even for individual tasks within a job.

**Asynchronous I/O**

Asynchronous I/O system calls are provided to support device independent I/O to any device in the
system. The type of I/O and the type of device are interrelated as shown in Figure 6. Every device driver in the I/O system is required to support a standard interface. In this manner, all devices look the same to higher level software. In the same manner, the individual file drivers, which provide the different types of file systems, all have a standard interface and call upon the various device drivers to perform I/O. These interface standards

1) provide for the device independence in the higher layers of the I/O system
2) make it easier for Intel to add future device drivers as new devices become available and
3) make it possible for iRMX 86 users to add their own drivers for custom I/O devices.

**Figure 6. I/O System Structure**

The iRMX 86 I/O system provides both asynchronous and synchronous system calls. The asynchronous I/O calls are faster, provide more flexibility in the selection of options and allow the program making the call to perform other functions while waiting for the I/O operation to complete.

The method by which the I/O system responds to the requestor is through the use of a mailbox. When any call is made to the asynchronous I/O system, one of the parameters indicates a mailbox where the caller expects to receive a segment containing the results of the operation (see Figure 7).

**Synchronous I/O**

The alternative to using the asynchronous I/O system is to use synchronous I/O system calls. As shown in Figure 8, the number of options available are fewer and the caller cannot continue execution until the entire I/O operation is completed but from an ease-of-use standpoint, the situation is much simplified.

**Figure 7. Asynchronous I/O Call**

Response$mailbox$token = RQ$create$mailbox(0, @status);
CALL RQ$A$read(connection$token, buf$ptr, count, response$mailbox$token, @status);
IORS$token = RQ$receive$message(response$mailbox$token,OFFFFH, @resp$t, @status);
{check status}
Call RQ$delete$segment(IORS$token, @status);

**Figure 8. Synchronous I/O Call**

Call RQ$S$read(connection$token,buf$ptr, count, @status);
{check status}

Two other features provided by the Extended I/O System are logical name support and autobuffering. Logical names allow the application designer to postpone the decision concerning which files to use until run-time. Essentially, all programs can be written and compiled using logical file names and then these logical names can be mapped into real file names at run-time.

The use of autobuffering regains much of performance advantage offered by overlapped I/O. When a user task opens a file for input, one or more buffers are automatically created and filled with data from the file. Thus, when the user task makes an I/O request, the data may already be available in memory. A similar case exists for write requests in that the I/O system will buffer data to be written to a device, allowing the user task to continue on.

**Loaders**

The iRMX 86 application loader and bootstrap loader perform a variety of services for the user software. The following is a brief summary of the available features.

1) Systems can be boot loaded from mass storage devices at system reset. This saves not only ROM or EPROM memory, but also reduces field maintenance costs by allowing easy field updates.
2) Users can design their own SYSGEN procedure allowing tailoring of an application system to the individual installation.
3) Infrequently used programs can be brought in from mass storage when needed instead of using system memory unnecessarily.
File Management

There are three types of files supported by the iRMX 86 I/O system, named files, physical files and stream files. Named files are supported on devices possessing mass storage capability. Files in this system have ASCII pathnames and are cataloged in directories. Each device in the system contains a directory tree as shown previously in Figure 1. Access protection is provided through the use of access lists for each file. Each user or group of users in the system can be given different types of access to the file or can be denied access to it.

For devices that cannot support a named file structure (e.g. printers and terminals) the physical file driver is used. Devices in this category are treated strictly as data going into and/or out of the device. If it is desirable to treat a mass storage device strictly as a large mass of data, it can also be addressed through the physical file driver.

The third type of file is the stream file. This file type has no correlation with any physical device but rather uses system memory for temporary storage of data. An example of the usage of a stream file is a job that gets its input stream of data from a file. Depending on which time the job is run, this file might be a named file on disk, a terminal, or a stream file being written to by another job (see Figure 9).

The Human Interface also contains a set of system utility routines which are used to copy files and disks, format disks, dynamically alter the system configuration and others.

Debugging Subsystem

The iRMX 86 Debugging Subsystem allows the designer to interact with the prototype system and isolate and correct program errors. Since the debugger is an object-oriented debugger and is aware of the internal structure of the operating system, it can provide detailed information concerning objects and can monitor mailboxes and semaphores providing a breakpoint facility as well as error detection.

Specifically, the iRMX 86 Debugging Subsystem provides six sets of functions:

1) Wake-up upon operator invocation. The operator types a control-D key to cause the debugger to wake up.
2) View system lists. The debugger can view lists of objects either globally or specifically for a given job. Also, lists of objects and tasks queued at mailboxes and semaphores can be seen.
3) Inspect objects. A detailed report on any object can be requested showing the current state of all relevant attributes.
4) Inspect and modify memory.
5) Breakpoint control. Any number of breakpoints can be set causing a single task to break on either execution of particular instructions or sends and receives of messages or units.
6) Error handling. The debugger can be set up to be the system default error handler thus catching system exceptions.

Configuration and Initialization

Once the application is designed and coded, the engineer needs a mechanism to inform the operating system of the software and hardware configuration. Essentially, this involves building tables of information using tools provided with the iRMX 86 product.

As shown earlier in Figure 4, the jobs in an iRMX 86 system form a hierarchical tree. The root in every job tree is known as the root job and is supplied as part of the iRMX 86 system. There are three important features of this job.

1) The root job has an object directory for cataloging and looking up objects. The special feature of this directory is that is is accessible by all tasks in the system since everyone can address the root job. For this reason the root object directory is useful for setting up inter-job communication paths.
2) The root job initially contains all free space in the system. Part of the system initialization code performs a memory scan to automatically determine the amount of free RAM in the system. This memory is put into the free space pool of the root job and parceled out as user jobs are created.

3) The root job contains only one task, the root task. This task scans the configuration tables generated by the user and creates the user-specified jobs.

Examples of configuration, initialization and the LINK 86 and LOC 86 operations needed to generate a system will be presented in the Code Examples section.

**DESIGN METHODOLOGY**

This section describes the design process involved in using the iRMX 86 system to solve application problems and presents two example solutions.

System design with the iRMX 86 Operating System should be viewed as a process starting with the highest level definition of system requirements and successively adding more detail until the end product is program code. This description sounds very much like the description of top-down design and, of course, it offers not only quicker designs, fewer design flaws and easier implementation, but also easier maintenance and enhancement.

In general, every iRMX 86 design progresses through the following steps:

1) Define system requirements.
2) Breakdown into highest level sub-functions (jobs).
3) Define job functions.
4) Determine inter-job command and data flow.
5) Break down each job into sub-functions.
6) Based upon requirements, assign tasks to perform job functions.
7) Determine inter-task command and data flow.
8) Write program code for each task.

Step 8 becomes the design process associated with the application programs themselves. The code for each task is essentially a sequential program that performs one of the functions of the computer system. Standard techniques for top-down design can therefore be used here to specify each module and its inputs and outputs as well as global and local data structures etc. The end product of this procedure is a modularized application system that should be easy to debug.

**APPLICATION EXAMPLE 1**

The first example presented here is based on the distributed local network diagrammed in Figure 10. Each workstation shown is an intelligent terminal having local data and program storage. The stations all use the File Sharing Node (FSN) for storage and retrieval of records in much the same way as the secretaries in an office would make use of a filing cabinet. The FSN maintains the files on a fixed disk device and responds to requests from the workstations for access to the data. The design to follow concentrates on the File Sharing Node.

**System Requirements**

Each intelligent terminal in the network has command processing software. When a file reference is made that cannot be satisfied by the local file system, a request is made to the File Sharing Node. This request consists of a log-on request followed by a string of I/O requests and ultimately a log-off request.

The number of intelligent terminals (workstations) hooked up to the FSN varies from installation to installation. Therefore, the FSN must be capable of handling many simultaneous requests and no assumptions can be made about the maximum number of workstations or requests that may need to be handled.

Each node in the network has a unique address. A packet is sent onto the network by one node and the address field is examined by all other nodes. If this field does not match the node's address the packet is ignored. If a match is found the packet is retrieved from the network.

**Hardware Requirements**

The three main hardware building blocks needed by this application are shown in Figure 11. The iSBC 86/12A Single Board Computer will communicate with the iSBC 544 Intelligent Communications Controller to establish and maintain communications with the network. The Intel 8085A on the iSBC 544 board will perform all of the address recognition, acknowledgements, packet retrieval and packet transmittal. The iSBC 206 Hard Disk Controller will be used to...
create, maintain and access the data files which are at the heart of this application.

**System Design**

The first step in the system design process is the breakdown of the system functions into one or several jobs. The reasons for doing this are system modularity and protection. With this type of design, each job can be designed separately, perhaps even by a different engineer or engineering team. The input and output requirements will be specified very tightly and the job will take on the appearance of a black box to other jobs in the system. If the job is enhanced or modified at a later date, the rest of the system can be left undisturbed providing that the input and output response remains the same.

The job object in the iRMX 86 operating system also affords a degree of software protection for the tasks and other objects contained within the job. Each job has a separate memory pool, a separate object directory and a separate identification to the I/O system.

The two primary groupings of functions in this application are those related to the network communications and those related to processing the file transaction request. A list of a possible split-up of system functions is shown in Figure 12.

**Figure 12. Function Split-up**

The communication between the file transaction job and the communication job must fulfill two basic needs. The communication job will receive interrupts when packets addressed to the FSN are received. In order to remain attentive to new requests coming in, the communications job should have the capability to "spool" the requests off to the file transaction job. This buffering can be provided by using the mailbox object. Segments can be created to contain the packet request data and can then be sent to a mailbox where the file transaction job can receive and process them.

When the file transaction job must send a packet to a workstation, the requirement is seen for another queue of requests. Since the communications board can only put one packet at a time on the network, a mailbox should be provided to allow tasks in the file transaction job to send output request segments into the queue and then continue on (see Figure 13).

**Figure 13. Output Mailbox Queue**

Since tasks in both the file transaction job and the communications job must have access to these input and output mailboxes, some means must be set up to "broadcast" the identifier for these objects.

In the iRMX 86 system, each object has associated with it a 16-bit number called a token. Whenever an object is referenced in an operating system call, the
token for the object is used. For example, assume that a segment must be sent to a mailbox. The segment and mailbox each have a token and these tokens are passed to the operating system as parameters in the send$message system call.

There are three major ways to get the token for an object. The first way is to create an object. Whenever the operating system is called to create a new object, the value returned from the procedure call is the token for the new object. The second way to receive a token is through the receive message system call where an object is received from the queue at a mailbox where it was sent by another task.

The third major mechanism for the receipt of a token is provided by the object directory concept. As mentioned previously, each job in the system has an object directory.

If a task in a job has the token for an object and wishes to let other tasks in other jobs have access to the object, the task can "catalog" the object in the object directory. The catalog$object system call takes the token for an object and an ASCII name as parameters and creates an entry in the object directory. If another task knows the ASCII name for an object, it can obtain the token by performing a lookup$object call.

The object directory mechanism will be used in this example to allow the communications job to "broadcast" the tokens for the input and output mailboxes. The jobs for this application are shown in Figure 14.

![Figure 14. Job Structure](image1)

The next step of the design methodology calls for each job to be further divided into sub-functions. In this application note, only the file transaction job is studied.

In time sequence, the file transaction job will:

1) Retrieve input requests from the mailbox set up by the communication job.
2) Determine state of specified workstation (for example, is it logged on?).
3) Perform I/O operation or log-on or log-off.
4) Build and send response to the workstation.

Recall from the discussion of system requirements that the number of nearly simultaneous requests that may be received by the FSN is not known. For this reason, some mechanism must be provided to allow parallel processing of many requests. This should prove feasible since the performance of step 3 will involve many delays while waiting for the operating system to perform I/O operations.

One straightforward way to provide for parallel processing is to create a task for each workstation that logs on. In this manner, each I/O request will be handled by a unique task. Through the use of the iRMX 86 scheduler, maximum CPU utilization will be gained by allowing each task to individually compete for CPU time. These "worker" tasks fulfill function 3 and 4 for the file transaction job.

Function 1 and 9 can be fulfilled by a single task. This task will wait at the input mailbox set up by the communications job. When a packet is received that requests a log-on operation, the "listener" task will create a new "worker" task to handle the request. Figure 15 shows a picture of the design.

![Figure 15. Diagram of Design of File Transaction Job](image2)

The string of transaction requests that follow will simply be demultiplexed by the listener task. The workstation ID will be searched for and, if found, the packet will be sent to the appropriate worker task. If a request comes in from a station that is not logged on, an error response is sent directly to the communications output mailbox for transmittal to the station that made the request.
If the request packet indicates that a station desires to log-off, the listener task will delete all local reference to the station and pass the packet along. The listener task cannot simply delete the worker since the worker may be in the process of servicing a previous I/O request. In general, it is never a good idea to arbitrarily delete another task. A better protocol is to pass along the message signaling the worker task to delete itself when convenient.

An investigation of the intertask communications needs highlights the requirement for passing data between tasks. The interjob communications protocol discussed earlier specified that the listener task will receive input request segments from the communications job via a mailbox.

Within these segments are fields containing the workstation ID and the command. Based upon these fields one of two things happens. If the command indicates that the station wishes to log on, a new worker task must be created to process the I/O requests that will follow.

The code executed by all worker tasks will be identical since they all perform identical functions. However, some unique pieces of information must be passed to a new worker task. This can be accomplished by having the worker task first wait at a "log on" mailbox. Here it will receive a segment from the listener task which contains the necessary information (see Figure 16).

The last communication path needed is predefined by the interjob communication protocol. When either the listener task or one of the worker tasks needs to transmit a packet to a workstation, a segment is sent to the output request mailbox of the communication job.

The final step in the design methodology is to write program code for the tasks in the system. This step is performed in the Code Examples section.

**APPLICATION EXAMPLE 2**

This example will deal with the design of a custom device driver for the iRMX 86 operating system. As shown in Figure 6, a device driver accepts high-level commands from the file drivers (such as read, write, seek, etc.) and transforms these commands into I/O port read and write commands in order to communicate with the device itself. By studying the construction of a driver for the isBC 534 Serial Communication Expansion Board, a better understanding of the iRMX 86 I/O system will be gained along with an example of the use of nucleus facilities to construct a higher-level software function.

**Overview of Device Driver Construction**

Each I/O device consists of a controller and one or more units. A device as a whole is identified by a device number. Units are identified by unit number and device-unit number. The unit number identifies the unit within the device and the device-unit number identifies the unit among all the units on all of the devices.

A device driver must be provided for every device in the hardware configuration. That device driver must handle the I/O requests for all of the units on the device. Different devices can use different device drivers; or if they are the same kind of device, they can use the same device driver code.

At its highest level, a device driver consists of four procedures which are called directly by the I/O System. These procedures can be identified according to purpose, as follows:

- Initialize I/O
- Finish I/O
- Queue I/O
- Cancel I/O

When a user makes an I/O System call to manipulate a device, the I/O System ultimately calls one or more of these procedures, which operate in conjunction with an interrupt handler to coordinate the actual I/O transfers. This section provides a general description of each of these procedures, and the interrupt handler.
**INITIALIZE I/O**

This procedure creates all of the iRMX 86 objects needed by the remainder of the routines in the device driver. It typically creates an interrupt task and a segment to store data local to the device. It also performs device initialization, if any such is necessary. The I/O System calls this routine just prior to the first attach of a unit on the device (the first R$A$PHYSICAL ATTACH$DEVICE system call). The time sequence of calls to these procedures will be described a little later.

**FINISH I/O**

The I/O System calls this procedure after all units of the device have been detached (the last R$A$PHYSICAL DETACH$DEVICE system call). The finish$I/O$ procedure performs any necessary final processing on the device and deletes all of the objects used by the device handler, including the interrupt task and the device-local data segment.

**QUEUE I/O**

This procedure places I/O requests on a queue, so that they can start when the appropriate unit becomes available. If the device is not busy, the queue$I/O$ procedure starts the request.

**CANCEL I/O**

This procedure cancels a previously queued I/O request. Unless the device is such that a request can take an indefinite amount of time to process (such as keyboard input from a terminal), this procedure can perform a null operation.

**INTERRUPT HANDLERS AND INTERRUPT TASKS**

After a device finishes processing an I/O request, it sends an interrupt to the iRMX 86 system. As a consequence, the interrupt handler for the device is called. This handler either processes the interrupt itself or signals an interrupt task to process the interrupt. Since an interrupt handler is limited in the types of system calls that it can make, an interrupt task usually services the interrupt. The interrupt task feeds the results of the interrupt back to the application software (data from a read operation, status from other types of operations). It then gets the next I/O request from the queue and starts the device processing this request. This cycle continues until the device is detached. The interrupt task is normally created by the initialize I/O procedure.

The I/O System calls each one of the four device driver procedures in response to specific conditions. Three of the procedures are called under the following conditions:

1) In order to start I/O processing, the user must make an I/O request. This can be done by making a variety of system calls. However, the first I/O request to each device-unit must be the R$A$PHYSICAL ATTACH$DEVICE system call.

2) The I/O System checks to see if the I/O request results from the first R$A$PHYSICAL ATTACH$DEVICE system call for the device (the first unit attached in a device). If it is, the I/O System realizes that the device has not been initialized and calls the initialize I/O procedure first, before queueing the request.

3) Whether or not the I/O System called the initialize I/O procedure, it calls the queue I/O procedure to queue the request for execution.

4) The I/O System checks to see if the request just queued resulted from the last R$A$PHYSICAL DETACH$DEVICE system call for the device (detaching the last unit of a device). If so, the I/O System calls the finish I/O procedure to do any final processing on the device and clean up objects used by the device driver routines.

The I/O System calls the fourth device driver procedure, the called I/O procedure, under the following conditions:

- If the user makes an R$A$PHYSICAL DETACH$DEVICE system call specifying the hard detach option, in order to forcibly detach the connection objects associated with a device-unit.
- If a job containing the task which made the request is deleted.

Each procedure will now be discussed in more detail. The initialize $I/O$ procedure takes three parameters:

\[
\text{init}\$i/o: \text{Procedure} (\text{duib}$p, \text{ret}$\text{data}$t$p, \text{status}$s$p)
\]

The duib$p parameter contains a pointer to a device unit information block (DUIB) which is the configuration table for the device in question. The structure of this table is shown in Figure 17. Note that this table contains pointers to device and unit information tables which can contain hardware specific information (such as I/O base addresses, interrupt levels etc.).

The second parameter is a pointer to a word which can be assigned the value of a token for an iRMX 86 object. Quite often this object would be a segment which could be created by the init$i/o$ procedure and filled with information needed by the other procedures in the driver. The token for this segment will be provided to the other procedures when they are called.
The final argument in the call is a pointer to a status word. This word should be assigned by the init$io procedure before a RETURN is executed. If a non-zero value is returned indicating an error condition, the I/O System assumes that init$io has deleted any objects created before the error was encountered.

The finish$io procedure is called by the I/O System just after the last detach$device call is made on the device. This procedure is expected to delete any objects created by the init$io procedure and shut down the connected device.

finish$io: Procedure (duib$p, ret$data$t);

Once again, the first parameter to the call is a pointer to a DUIB. The second parameter is the token returned by the init$io procedure.

The queue$io procedure is called to initiate an I/O request.

queue$io: Procedure (IORS$t, duib$p, ret$data$t)

The specifics of the request are indicated in an I/O request segment (IORS) which is provided by the first parameter. The format of this segment is shown in Figure 18. The most important fields here are the count, function, status and buffer pointer fields which tell the queue$io procedure what needs to be done. The second and third parameters are once again the pointer to the DUIB and the token for the object created by the init$io procedure.

The final device driver procedure is cancel$io. This procedure is called by the I/O System to cancel a previous I/O request. If the device is of such a nature that a request will complete in a bounded amount of time, this procedure can be a null procedure. The parameters to the call are identical to those for the queue$io call.

In addition to the elementary support discussed here, the I/O System provides extra support to the designer of a device driver if some simplifying assumptions about the device can be made. Also, if the device supports random access (such as disks, magnetic bubbles, etc.), support routines can be used to simplify the process of blocking and deblocking I/O requests. More detail on the process of writing I/O drivers can be found in the manual titled “A Guide to Writing Device Drivers for the iRMX 86 I/O System.”

**Design of an iSBC 534™ Device Driver**

The following section will discuss an example device driver for the iRMX 86 Operating System. The driver will be for the iSBC 534 board which contains four 8251 USART devices; therefore, there is one device and four units on the device.

The init$io procedure for this driver initializes the hardware, creates an interrupt task, creates other necessary objects and creates a segment to contain the relevant information.
The structure of the queue$io procedure is more complex. When calls are made to this procedure to perform data reading and writing, the actual operation could be somewhat lengthy (especially an input operation). Since the queue$io procedure is called by the I/O system, it is not efficient to perform the entire operation before control is returned to the I/O system.

A more efficient mechanism is to have an independent task take the request and fulfill it while the queue$io procedure returns to the I/O system allowing other operations to be started in parallel. This leads to the structure diagrammed in Figure 19. When a read or a write request is received, the I/O request segment is sent to the request mailbox where it is received by an I/O handler task. When the request is complete, the I/O task sends the segment to the response mailbox indicated in the segment.

![Figure 19. Queue$io Procedure Interface to I/O Tasks](image)

The remaining design of the device driver is concerned with interrupt handling. The iSBC 534 board contains four 8251 USART devices. Each device supplies two interrupts; one indicating that the receiver has a data character available and the other indicating that the transmitter is ready to accept a character. Each of these interrupts (8 in all) are connected to one of the 8259 Interrupt Controllers on the board. The software on the iSBC 86/12A board must read a register in the 8259 controller to determine which of the eight sources caused the current interrupt. This information must then be fed to the I/O task which may be waiting for the event.

One way to meet this requirement uses an interrupt task for the iSBC 534 board. The task receives the interrupt, determines which device caused it, and sends a unit to a semaphore to indicate the occurrence of the event. Thus, when an I/O task wishes to be informed of a receiver or transmitter interrupt, it simply tries to receive a unit from the appropriate semaphore. If a unit is available, the receiver has a character or the transmitter is ready. If the unit is not available, the UART is not ready and the task will be put in the asleep state until the interrupt occurs and the unit is sent.

**CODE EXAMPLES**

This chapter will present and analyze some sample code for the iRMX 86 applications presented in Chapter 4. The code listings are contained in Appendix A and the individual modules are numbered sequentially. When a specific line or sequence of lines of code must be pointed out in the text, a two part number is used where the first part is the module number and the second is the compiler-assigned line number. For example, 3.27 would be used to point out line 27 in module 3.

A standard set of suffixes to labels will be followed in the code to follow. A PL/M-86 WORD variable that will contain the token for an iRMX 86 object will have the suffix "$t." A POINTER variable will be followed by "$p" and a structure used to overlay a POINTER allowing access to the base and offset will be followed by "$p$o."
Whenever a workstation logs on, various actions are taken by the listener task. One of these actions involves adding a descriptor for the workstation to a list so that the state of the workstation can be maintained by the listener task. The list structure is shown in Figure 22. Statements 1.336-1.340 create the root of this list and initialize the list to an empty state.

Line 1.340 marks the beginning of an infinite loop. Most often a task executes a procedure which performs some initialization and then enters an endless loop performing the necessary processing. The literal "for­ever" translates into "while 1."

A packet is received from the input mailbox by the call in line 1.341. The command field of the message is checked in line 1.343. If the command indicates that a log on request is being made, lines 1.345-1.356 are executed. A log on information segment is created in line 1.345. A mailbox is created to handle further request packets and another is created to be used by the worker task as a response mailbox. The worker task that will handle I/O requests from this workstation is created in line 1.351. Note the use of the structure data$seg$p$o, which is declared at the same address as the POINTER data$seg$p. The POINTER is initialized to equal the beginning of the data segment of the worker task module (1.323) and then the base portion is used as a parameter in the create task call.

Once the worker task is created, it will wait at the log$on$info$mbox for a segment giving it its initialization information. The segment is sent in line 1.352 and received back as an acknowledgement in line 1.353. At this point, the segment is inserted on the list of active workstation descriptors by the call in line 1.354. Finally the request packet itself is sent to the worker task via the service mailbox for the new worker.

If a log off request is received, lines 1.358 to 1.366 are executed. First, the active workstation list is searched for the ID of the requesting station. If the station is not found to be logged on, the status field is set and the request segment is sent to the workstation through the communications job. If the station is logged on, the descriptor is deleted from the list, the packet is sent along to the worker task, and the descriptor is deleted.

If the command is anything but log on or log off, lines 1.368-1.376 are executed. Once again the station ID is checked to see if it is logged on. If not, an error message is returned. If the station is logged on, the request packet is sent along to the worker task.
WORKER TASK

The code for the worker task is shown in module 2. Upon creation of a new worker task, a segment is received at the $logon$informbox (2.242). The data in this segment is copied into local variables and the segment is returned (2.247).

The initialization task for this job has already created a user object for this job and has also set up a prefix which points to the root directory for the disk device. These tokens have been cataloged in the root object directory. The worker task obtains these tokens through the sequence of calls 2.248-2.250.

The worker task now enters an infinite loop servicing the workstation it is assigned to. The specific action to be taken by the worker is determined by inspecting the cmd field of the request message.

If the command is a log on, the code from 2.256-2.263 is executed. The file name specified in the request segment is attached and opened and thereby made ready for subsequent I/O requests. After this, an acknowledgement is sent back to the workstation via the output$req$mailbox (2.263).

If a log off command is received, the file is closed and detached, the service and response mailboxes are deleted, a response is sent to the workstation and the worker task is deleted.

If the command is either a read or write command, the operation is performed by calling the I/O system. When the response is received, an acknowledgement is sent to the workstation. Note that the task would normally perform more processing. In this example its duties have been kept simple.

POINTERIZE PROCEDURE

The ASM-86 code for the pointerize support routine is shown in Module 3. The token for a segment is the base portion of a 32-bit POINTER to the memory. In order to access the data in a segment, this 16-bit token must be loaded into the base part of a POINTER while the offset portion of the POINTER is set to zero. The base and offset values are returned in the ES and BX registers as specified by the PL/M-86 calling conventions. This is the operation performed by the pointerize routine.

LIST MANIPULATION ROUTINES

Lines 4.1-4.47 provide three subroutines used by the tasks in this system to manipulate the list of workstation descriptors. Insert$on$list (4.15-4.26) inserts the indicated node at the head of the list whose root is given as the first parameter. Delete$from$list (4.27-4.35) unlinks the indicated node from the list it belongs to. Search$list (4.36-4.46) searches a list for the workstation ID given. If the ID is not found, a zero is returned. If the ID is found, the token for that node is returned.

At this point an overview of the configuration process is needed. A more detailed coverage of the process of configuring an iRMX 86 system is provided in the manual entitled "iRMX 86 Configuration Guide for ISIS-II Users."

For each iRMX 86 application, the following steps must be performed.

1) Program code for each task in the system must be written and compiled or assembled.
2) A memory map for the software must be drawn up.
3) The system software must be linked and located.
4) The application jobs must be linked and located.
5) Tables of configuration data must be drawn up.
6) The tabular data from step 5 must be formatted into a memory data block through the use of a set of ASM-86 macros provided with the iRMX 86 product.
7) The root job must be linked and located.

The code executed by the root task is part of the iRMX 86 system code. This task is initially the only task in the system. The root task will access the data block constructed by the ASM-86 macros and will create the user jobs specified by the macros. The data for the configuration process for example 1 is shown in Appendix B.

The first page diagrams the memory map for the example. The iterative link and locate process to put these pieces together begins on the second page. The LINK86 and LOC86 commands shown place the iRMX/86 nucleus into memory. The LOCATE map indicates that the last memory location used by the nucleus was 077DFH. Therefore, the next contiguous piece, the I/O system, is located at 077EOH.

This process is repeated for the remainder of the jobs in the system.

When the link and locate process is complete, the information for the ASM-86 macros must be brought together. Worksheets are provided in the iRMX 86 configuration guide to simplify this process.

The filled-out worksheets for the macros are shown in the appendix. A configuration file is constructed using
the editor and the worksheet information is entered into this file. When the file is complete, the configuration table is created by assembling the file CTABLE. A86. This file accesses the configuration file built earlier.

The configuration tables are then linked and located together with the code for the root task and the system generation process is complete.

**EXAMPLE 2**

**INIT$IO AND FINISH$IO**

The *start$and*finish module (5.1-5.371) contains the code for the *init$534$io and finish$534$io procedures. The *init$534$io procedure creates a segment, shown in Figure 23, which is used to hold the various pieces of information needed by the other driver procedures (5.323). The discussion of this procedure in Chapter 4 pointed out that any errors encountered in the initialization are indicated by the non-zero status and that the assumption is made that any partial creations must be cleaned up by the *init$io procedure. This assumption is carried out by the check at line 5.324 (and the others at 5.331, 5.335, 5.339 and 5.342).

**Figure 23. init$534$io Segment Format**

The device information contained in the device unit information block for this device is retrieved in line 5.328-5.329. A mailbox to be used for sending I/O request segments to the I/O handler tasks is created in line 5.330. The interrupt task for this job is created by the call in line 5.337. The do loop starting at line 5.340 is executed to create eight semaphores to be used by the interrupt task to indicate the occurrence of an interrupt. Note that the initial value of the semaphore is zero (no interrupt pending) and the maximum value is one. Since the nature of the 8251 USART device does not support buffering, when a new character overruns the previous character before the interrupt can be serviced, the data is lost. Therefore, there is no need to indicate the occurrence of multiple interrupts pending on the same device.

The call at line 5.345 initializes the programmable devices on the iSBC 534 board. If execution has proceeded to line 5.346, the initialization is complete and a zero status is returned. If an error occurred at any point, the code in lines 5.348-5.356 will clean up the partial initialization.

The *finish$534$io procedure (5.358-5.370) undoes the work of the *init$534$io procedure. The segment, mailbox, interrupt task and semaphores are all deleted.

The *queue$534$io procedure is shown in lines 6.1-6.382. In line 6.322 the function field of the I/O request segment is checked to see if it is within bounds. If it is not, a bad status code is returned. If the function is valid, a *do case* block is executed using the function code as the index.

If a read request is encountered, the auxiliary pointer is set to point to the *ret$data structure (initialized earlier by the *init$534$io procedure). In line 6.327 the segment is then sent to the request mailbox to be received and processed by an I/O processor task. In lines 6.330-6.334 the same action is taken with write requests.

Since this driver does not support seeking and special functions, the code for these two cases simply returns an error condition.

In the case of an *attach$device call, the code in lines 6.341-6.361 is executed. First, two I/O processing tasks are created. All of these tasks execute identical code and each task is capable of servicing a read or a write request on any 8251. Two tasks are created for each 8251 device so that the peak load can always be handled (that is, all receivers and transmitters going simultaneously). Lines 6.346-6.357 perform the initialization of the 8251 USART and the baud rate generators for this channel. The calls in line 6.358 and 6.359 accept an interrupt and a character from the semaphore associated with the receiver just initialized. This is done to clear off an interrupt generated by the 8251 whenever it is initialized.

In the case of a *detach$device call, the code in lines 6.363-6.367 sends the I/O request segment to the
request mailbox twice. This is done to signal two of the
I/O handler tasks to delete themselves. As discussed
earlier in the attach$device section, none of the I/O
handler tasks is any different from any of the others.
There are two created for each 8251 device which is
attached. The protocol set up for their deletion is
shown here. When an I/O handler task receives a
segment of type “detach$device” it will send the
segment to the response mailbox and then delete itself.

The code for the open and close requests is the same.
Both cases are supported but are NOPs since no
specific action needs to be taken by the driver.

Lines 6.379-6.382 contain the code for the cancel$io
procedure. As discussed earlier, this procedure is simply a placeholder and serves no particular purpose.

**INTERRUPT CONTROL MODULE**

The interrupt handler and interrupt task are shown in
lines 7.1-7.329. The interrupt task is the first piece
executed. It is created by the init$534$io procedure. It calls RQ$set$interrupt in line 7.325 to indicate to the
iRMX 86 nucleus that it is an interrupt task.

Once the initialization is complete, the task enters an
infinite loop. The call to RQ$wait$interrupt in line
7.322 causes the task to be put into the asleep state
until an interrupt occurrence is signaled. The task will
be returned to the READY state when an interrupt
occurs, the interrupt handler is started, and the call to
RQ$signal$interrupt is executed at line 7.312. The
current interrupt level is then determined by polling
the 8259 chip on the iSBC 534 board. Using the
encoded level number, a unit is sent to the appropriate
semaphore to indicate that an interrupt is pending.

**I/O TASK**

The final procedure that makes up this driver contains
the code for the tasks that perform the actual I/O to
the iSBC 534 board. The loop executed by each task
starts by waiting at the request mailbox for an I/O
request segment. When the segment is sent by the
queue$534$io procedure, its function code is checked
(line 8.327, 8.332, 8.340). If the function is f$ device,
the task sends the segment to the
response mailbox and then deletes itself.

If the request was for a read, the task fills the buffer
with input data. The call at line 8.334 waits for a unit
at the semaphore which will indicate a receiver ready
on the input line. When the unit is sent by the in-
terrupt task, the character is read in, the pointers and
counts are updated, and another unit is requested.

The last request which is recognized by the I/O task is
for a write operation. The code for this request is
almost identical to the code for a read request. An
interrupt from the transmitter is awaited, a character
is output and the counts are updated in lines 8.341-
8.346.

Once the request is fulfilled, the message is sent to the
response exchange in line 8.350.

The configuration of this system is studied next. The
code for the iSBC 534 driver is linked directly to the
rest of the I/O system libraries. The entry point
addresses for the queue$534$io, cancel$534$io, init$534$io, and finish$534$io procedures are declared in
the IOCNFG.A86 file on the I/O system disk. This file
also contains the device unit information block (DUIB)
structures for the four units on the iSBC 534 board.
The unique information for the iSBC 534 device and
the units on the device is contained in the device and
unit information tables. Pointers to these tables are
contained in the DUIB structures. All of this
information is shown in Figure 24.

The submit file used to build an I/O system using the
iSBC 534 driver is shown in Figure 25. The file
DRV534.LIB contains the object files generated by
PL/M-86 and ASM-86 from the source code shown in
modules 5-9.

**SUMMARY**

This application note is an introduction to the iRMX
86 Operating System. The requirements of operating
systems were studied along with traditional solutions.
Following this, the iRMX 86 Operating System was
introduced and its correlation with the requirements
was studied.

Later in the application note, the topic of system
design was covered. Example solutions were studied to
solidify a methodology for solving application
problems and then the code for these solutions was
discussed to gain insight into the details of implement-
ing iRMX 86 systems.

The purpose of a configurable, real-time, multi-
purpose operating system is to provide a solid founda-
tion for application software. The iRMX 86 system
provides this foundation, giving the software engineer
a means to quickly and easily implement new designs.
In addition, the iRMX 86 architecture is the bridge to
future technology providing the designer with an up-
grade path to future hardware and software products.
extrn init534io: near
extrn queue534io: near
extrn cancel534io: near
extrn finish534io: near

; Duib(8): iSBC 534, unit 1
; define_duib <
& '1534.1', ; name (14)
& 03H, ; suppopt
& 00033H, ; file drivers
& 0, ; granularity
& 0,0, ; device size
& 3, ; device
& 1, ; unit
& 6, ; device_unit
& init534io, ; init5io
& finish534io, ; finish5io
& queue534io, ; queue5io
& cancel534io, ; cancel5io
& dev_534_info, ; device_info
& unit_534_1_info ; unit_info
&
& 534 device info
& dev_534_info dw 48H ; level
db 61 ; priority
db 140H ; base address

; unit info: iSBC 534.0
; unit_534_0_info db 4EH ; usart$cmd
dw 8 ; baud rate

; unit info: iSBC 534.1
; unit_534_1_info db 4EH ; usart$cmd
dw 8 ; baud rate

; unit info: iSBC 534.2
; unit_534_2_info db 4EH ; usart$cmd
dw 8 ; baud rate

; unit info: iSBC 534.3
; unit_534_3_info db 4EH ; usart$cmd
dw 8 ; baud rate

Figure 24. IOCNFg A86 File Entries for iSBC 534™ Driver

; ios(date,origin)
; Sample I/O System .csd file to link and locate an I/O System.
; This file links an I/O System with the timer included.
; This .csd file assumes the I/O System configuration module is
; iocnfag.a86 (found on the release diskette).
; The origin parameter sets the low address of the I/O System;
; all the segments are contiguous in memory.
; asm86 :f1:ios.a86 date(%0) print(:f5:ios.lst)
link86 &
;f1:ios.lib(ioinit), &
;f1:ios.lib(ioconf), &
;f1:ios.lib, &
;f1:dr534.lib, &
;f4:ripfc.lib &
to f1:ios.lnk map print(:f1:ios.mp)
loc86 :f1:ios.lnk to :f1:ios.map sc(3) print(:f1:ios.mp) &
oc(noli,nonpl,nomc,noshb) &
order(classes(code,data,stack,memory)) &
addresses(classes(code(1))) &
segsize(stack(0))

Figure 25. Submit File for Generating an I/O System with the iSBC 534™ Driver
APPENDIX A

Code Listings
Module 1

ISIS-II PL/M-86 V2.0 COMPILATION OF MODULE LISTENER
OBJECT MODULE PLACED IN :Fl:listen.OBJ
COMPILER INVOKED BY: plm86 :Fl:listen.plm PRINT(:Fl:LISTEN.LST)
DEBUG COMPACT OPTIMIZE(3) ROM DATE(5/28/86)

1 listener$module:
do;

************************************************************************
LISTENER: TASK.
This task creates segments, sends them to the input service
job to be filled with input packet info. Upon response
the info is checked to see what action needs to be taken.
If a log$on request is sensed, a worker task, service
mailbox, and response mailbox are created and the packet is
sent along to the worker task. If a log$off is sensed all
local reference to the workstation is deleted and the packet
is sent along to tell the worker to delete himself. If an
I/O request is sensed the station ID is checked to make
sure it is logged on. If it is, the packet is sent along to
the worker. If it isn't an error packet is sent back to the
requesting workstation.
************************************************************************/

$include(:f2:common.lit)
$SAVE NOLIST
$include(:f1:node.lit)
/* literal declaration of node descriptor for list utilities */
1 declare
  node literally 'structure(
    link$f word,
    link$b word,
    work$station$ID word,
    service$mbox$t word,
    worker$task$t word,
    resp$mbox$t word);
$include(:f1:lstutil.lit)
/* external declarations for list manipulation utilities */

$save nolist
$include(:f1:pointr.ext)
/* external declaration of pointerize procedure */
$save nolist
$include(:f1:rqpckt.lit)
/* literal declaration for request packet structure */

24 declare req$segment$struc literally 'structure(
  funct word,
  count word,
  actual word,
  ex$val word,
  work$station$ID word,
  cmd word,
  share word,
  mode word,
  status word,
  file$name (64) byte,
  buf (128) byte);
$include(:f2:nuacprn.ext)
$SAVE NOLIST

321 worker$task: procedure external;
322 end worker$task;
Module 1, continued

323 1 declare
   begin$listener$task$data byte public,
   begin$worker$task$data byte external,
   log$on$info$seg TOKEN token public,
   ex$val word,
   log$on$seg name (7) byte data(6,'LOG$ON'),
   packet$size literally '132',
   f$read literally '5',
   f$write literally '6',
   log$on literally '0',
   log$off literally '1',
   not$logged$on literally '1',
   (root$job$t,input$request$seg$token) token,
   (output$request$seg$token,resp$seg$token) token,
   (work$station$list$root$t,req$segment$token) token,
   (log$on$info$seg$token,dummy$token,ws$desc$token) token,
   (req$segment$p,work$station$list$root$p) pointer,
   (req$segment based req$segment$p) req$segment$struct,
   (work$station$list$root based work$station$list$root$p) node,
   (log$on$info$seg based log$on$info$seg$p) node,
   data$seg$p$structure(offset word, base word) at(#data$seg$p),
   (ws$desc based ws$desc$p) node;

324 1 return$error$to$WS: procedure;
325 2 req$segment.funct=f$write;
326 2 req$segment.status=not$logged$on;
327 2 call rq$send$message(output$request$seg$token,req$segment$token,0,@ex$val);
328 2 return;

330 1 Listener: procedure public; /* task */
331 2 log$on$info$seg$token=rq$create$seg$mailbox(0,ex$val);
332 2 root$job$t=rq$get$task$tokens(3,ex$val);
333 2 input$request$seg$token=rq$lookup$object(
   /* job */ root$job$t,
   /* name */ @(9,'INPUT$REQ'),
   /* time limit */ @FFFFH,
   /* status ptr */ @ex$val);
334 2 output$request$seg$token=rq$lookup$object(
   /* job */ root$job$t,
   /* name */ @(10,'OUTPUT$REQ'),
   /* time limit */ @FFFFH,
   /* status ptr */ @ex$val);
335 2 resp$seg$token=rq$create$seg$mailbox(0,ex$val);
336 2 work$station$list$root$t=work$station$create$segment(16,ex$val);
337 2 work$station$list$root$p=pointerize(work$station$list$root$t);
338 2 work$station$list$root$link$=work$station$list$root$p;
339 2 work$station$list$root$workstation$ID=0;
340 2 do forever;
341 3 req$segment$token = rq$receive$message(
   /* mbox token */ input$request$seg$token,
   /* time limit */ @FFFFH,
   /* response ptr */ @dummy$p,
   /* status ptr */ @ex$val);
342 3 req$segment$p=pointerize(req$segment$token);
343 3 if req$segment.cmd = log$on then
   do;
Module 1, continued

345  log$on$info$segSt=rq$create$segment(  
 /* size */ 16,  
 /* status ptr*/ @ex$val);  
346  log$on$info$segSp=pointerize(  
 log$on$info$segSt);  
347  log$on$info$seg.service$mboxSt=  
 rq$create$mailbox(0,@ex$val);  
348  log$on$info$seg.resp$mboxSt=  
 rq$create$mailbox(0,@ex$val);  
349  log$on$info$seg.work$station$SID=  
 req$segment.work$station$SID;  
350  data$segSp=#begin$worker$task$data;  
351  log$on$info$seg.worker$taskSt=  
 rq$create$task(  
 /* priority */ 200,  
 /* start addr */ @worker$task,  
 /* data seg ptr */ data$segSp$0.base,  
 /* stack pointer */ 0,  
 /* stack size */ 500,  
 /* task flags */ 0,  
 /* status ptr */ @ex$val);  
352  call rq$send$message(  
 /* mbox token */ log$on$info$mboxSt,  
 /* object token */ log$on$info$segSt,  
 /* response token */ resp$mboxSt,  
 /* status ptr */ @ex$val);  
353  log$on$info$segSt=rq$receive$message(  
 /* mailbox token */ resp$mboxSt,  
 /* time limit */ @FFFFFH,  
 /* response token */ @dummySt,  
 /* status ptr */ @ex$val);  
354  call insert$on$list(work$station$list$rootSt,  
 log$on$info$segSt);  
355  call rq$send$message(  
 /* mbox tok */ log$on$info$seg.service$mboxSt,  
 /* obj tok */ req$segmentSt,  
 /* response */ 0,  
 /* status */ @ex$val);  
356  end;  
357  else if req$segment.cmd = log$off then  
358  do;  
359  ws$descSt=search$list(work$station$list$rootSt,  
 req$segment.work$station$SID);  
360  if ws$descSt = 0 then  
361  call return$error$to$WS;  
362  else  
363  do;  
364  ws$descp=pointerize(ws$descSt);  
365  call delete$from$list(  
 ws$descSt);  
366  end;  
367  end;  
368  else  
369  do;  

Module 1, continued

if ws$desc$t=0 then
  call return$error$to$WS;
else
  do;
    ws$descp=pointerize(ws$desc$t);
    call rq$send$message(
      ws$desc.service$mbox$t,
      req$segment$t,
      @ex$val);
end;

end; /* of do forever */

end; /* of listener task */

end listener$module;

MODULE INFORMATION:

| CODE AREA SIZE | 0281H | 641D |
| CONSTANT AREA SIZE | 0000H | 0D |
| VARIABLE AREA SIZE | 002BH | 43D |
| MAXIMUM STACK SIZE | 0018H | 24D |
| 694 LINES READ |
| 0 PROGRAM ERROR(S) |
Module 2

ISIS-II PL/M-86 V2.0 COMPILATION OF MODULE WORKERTASK
OBJECT MODULE PLACED IN :Fl:worker.OBJ
COMPILER INVOKED BY: plm86 :Fl:worker.plm PRINT(:Fl:WORKER.LST)
DEBUG COMPACT OPTIMIZE(3) ROM DATE(5/28/80)

1       worker$task:
       do;
       /********************************************************************************
       WORKERTASK: TASK.
       This module contains the code executed by the worker tasks.
       When started, the task goes to a mailbox to receive a segment
       containing initialization information. Using this information
       the task services a service mailbox performing any I/O functions
       requested of it. When a log$off request comes in the worker
       task closes and detaches the file and deletes itself.
       ******************************************************************************/
$include(:fl:nucprm.ext)
$SAVE NOLIST
#include(:fl:iosys.ext)
$save nolist
#include(:fl:node.lit)
/* literal declaration of node descriptor for list utilities */
$SAVE NOLIST
#include(:f2:common.lit)
$SAVE NOLIST
#include(:fl:poitr.ext)
/* external declaration of pointerize procedure */
$SAVE NOLIST
#include(:fl:rqpckt.lit)
/* literal declaration for request packet structure */
$SAVE nolist
declare
    read literally '1',
    write literally '5',
    log$on literally '2',
    log$off literally '3',
    (log$on$info$seg$t,log$on$resp$mbox$t,resp$mbox$t,
    root$job$t,user$object$t,prefix$t,iors$t,
    service$mbox$t,conn$t,req$seg$t) token external;
239       worker$task: procedure reentrant public;
240       declare
        (log$on$info$seg$t,log$on$resp$mbox$t,resp$mbox$t,
        root$job$t,user$object$t,prefix$t,iors$t,
        service$mbox$t,conn$t,req$seg$t) token,
        (log$on$info$p,req$seg$p) pointer,
        (req$seg$ based req$seg$p) req$seg$st,
        (log$on$info based log$on$info$p) node,
        (dummy$t,ex$val,work$station$ID) word;
242       log$on$info$seg$t=rgreceive$message(
            /* mbox token */ log$on$info$mbox$t,
            /* time limit */ 0FFFFH,
            /* response ptr */ @log$on$resp$mbox$t,
            /* status ptr */ @ex$val);
243       log$on$info$p=pointerize(log$on$info$seg$t);
244       service$mbox$t=log$on$info.service$mbox$t;
245       resp$mbox$t=log$on$info.resp$mbox$t;
246       work$station$ID=log$on$info.work$station$ID;
Module 2, continued

call rq$sendMessage(
    /* mbox token */ log$on$resp$mbox$t,
    /* object token */ log$on$info$seg$t,
    /* response token */ @ex$val,
    /* status ptr */ @ex$val);

root$job$t=rq$get$task$tokens(3, @ex$val);
u$er$object$t=rq$lookup$object(
    /* job token */ root$job$t,
    /* name */ @(11,'USER$OBJECT'),
    /* time limit */ 0FFFFH,
    /* status ptr */ @ex$val);

pre$fix$t=rq$lookup$object(
    /* job token */ root$job$t,
    /* name */ @6,'PREFIX'),
    /* time limit */ 0FFFFH,
    /* status ptr */ @ex$val);

do forever;

treq$seg$t=rq$receive$message(
    /* mailbox token */ service$mbox$t,
    /* time limit */ 0FFFFH,
    /* response ptr */ @dummy$t,
    /* status ptr */ @ex$val);

    req$seg$p=pointerize(req$seg$t);

    if req$seg$cmd=log$on then
    do;

    end;

    end;

    req$seg$cmd=log$off then
    do;

    end;

    call rq$a$close(
        /* connection */ conn$t,
        /* resp token */ resp$mbox$t,
        /* status ptr */ @ex$val);
Module 2, continued

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* resp ptr */   @dummy$t,
  /* status ptr */ @exSval);

call rq$delete$segment(iors$t,@exSval);

call rq$a$delete$connection(
  /* connection */  connSt,
  /* response ptr */  resp$mboxSt,
  /* status ptr */  @exSval);

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* response ptr */  @dummy$t,
  /* status ptr */  @exSval);

call rq$delete$segment(iors$t,@exSval);

call rq$a$delete$connection(
  /* connection */  connSt,
  /* response ptr */  resp$mbox$t,
  /* status ptr */  @exSval);

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* response ptr */  @dummy$t,
  /* status ptr */  @exSval);

call rq$delete$segment(iors$t,@exSval);

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* response ptr */  @dummy$t,
  /* status ptr */  @exSval);

call rq$delete$segment(iors$t,@exSval);

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* response ptr */  @dummy$t,
  /* status ptr */  @exSval);

call rq$delete$segment(iors$t,@exSval);

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* response ptr */  @dummy$t,
  /* status ptr */  @exSval);

call rq$delete$segment(iors$t,@exSval);

call rq$delete$task(0,@exSval);

do;

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* response ptr */  @dummy$t,
  /* status ptr */  @exSval);

call rq$delete$segment(iors$t,@exSval);

call rq$send$messag~(/* mbox token */ output$request$mbox$t,
  /* object token */  req$seg$t,
  /* resp token */  0,
  /* status ptr */  @exSval)i

call rq$delete$task(0,@exSval);

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* response ptr */  @dummy$t,
  /* status ptr */  @exSval);

call rq$delete$segment(iors$t,@exSval);

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* response ptr */  @dummy$t,
  /* status ptr */  @exSval);

call rq$delete$segment(iors$t,@exSval);

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* response ptr */  @dummy$t,
  /* status ptr */  @exSval);

call rq$delete$segment(iors$t,@exSval);

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* response ptr */  @dummy$t,
  /* status ptr */  @exSval);

call rq$delete$segment(iors$t,@exSval);

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* response ptr */  @dummy$t,
  /* status ptr */  @exSval);

call rq$delete$segment(iors$t,@exSval);

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* response ptr */  @dummy$t,
  /* status ptr */  @exSval);

call rq$delete$segment(iors$t,@exSval);

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* response ptr */  @dummy$t,
  /* status ptr */  @exSval);

call rq$delete$segment(iors$t,@exSval);

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* response ptr */  @dummy$t,
  /* status ptr */  @exSval);

call rq$delete$segment(iors$t,@exSval);

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* response ptr */  @dummy$t,
  /* status ptr */  @exSval);

call rq$delete$segment(iors$t,@exSval);

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* response ptr */  @dummy$t,
  /* status ptr */  @exSval);

call rq$delete$segment(iors$t,@exSval);

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* response ptr */  @dummy$t,
  /* status ptr */  @exSval);

call rq$delete$segment(iors$t,@exSval);

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* response ptr */  @dummy$t,
  /* status ptr */  @exSval);

call rq$delete$segment(iors$t,@exSval);

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* response ptr */  @dummy$t,
  /* status ptr */  @exSval);

call rq$delete$segment(iors$t,@exSval);

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* response ptr */  @dummy$t,
  /* status ptr */  @exSval);

call rq$delete$segment(iors$t,@exSval);

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* response ptr */  @dummy$t,
  /* status ptr */  @exSval);

call rq$delete$segment(iors$t,@exSval);

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* response ptr */  @dummy$t,
  /* status ptr */  @exSval);

call rq$delete$segment(iors$t,@exSval);

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* response ptr */  @dummy$t,
  /* status ptr */  @exSval);

call rq$delete$segment(iors$t,@exSval);

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* response ptr */  @dummy$t,
  /* status ptr */  @exSval);

call rq$delete$segment(iors$t,@exSval);

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* response ptr */  @dummy$t,
  /* status ptr */  @exSval);

call rq$delete$segment(iors$t,@exSval);

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
  /* response ptr */  @dummy$t,
  /* status ptr */  @exSval);

call rq$delete$segment(iors$t,@exSval);

iors$t=rq$receive$message(
  /* mbox token */  resp$mbox$t,
  /* time limit */  $FFFFH,
Module 2, continued

292 4 call rq$send$message( /* mbox token */ output$request$mbox$t, /* object token */ req$seg$t, /* resp token */ 0, /* status ptr */ @ex$val);
293 4 end; /* of do forever */
295 2 end; /* of task */
296 1 end worker$task;

MODULE INFORMATION:

<table>
<thead>
<tr>
<th>CODE AREA SIZE</th>
<th>0288H</th>
<th>648D</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTANT AREA SIZE</td>
<td>0000H</td>
<td>0D</td>
</tr>
<tr>
<td>VARIABLE AREA SIZE</td>
<td>0000H</td>
<td>0D</td>
</tr>
<tr>
<td>MAXIMUM STACK SIZE</td>
<td>0034H</td>
<td>52D</td>
</tr>
</tbody>
</table>

717 LINES READ
0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
Module 3

ISIS-II MCS-86 MACRO ASSEMBLER V2.0 ASSEMBLY OF MODULE POINTR
OBJECT MODULE PLACED IN :F1:POINTR.OBJ
ASSEMBLER INVOKED BY: asm86 :f1:pointr.a86 debug pr(:f5:pointr.lst)

ASSEMBLY COMPLETE, NO ERRORS FOUND
LIST$UTILITIES: PUBLIC PROCEDURES.

This module contains three list manipulation utilities. Insert$on$list takes the given node and inserts it on the list indicated by the root node parameter. Delete$from list unlinks the indicated node from the list it is linked to. Search$on$list scans the list from the root looking for the indicated node. If found, the token for the node is returned. If not found, a zero is returned.

*******************************

Insert$on$list: procedure( root$St,new$descSt) reentrant public;

15 2 declare
16 2  (root$St,new$descSt,fwd$descSt) token,
    (root$Sp,new$descSp,fwd$descSp) pointer,
    (root based root$Sp) node,
    (new$desc based new$descSp) node,
    (fwd$desc based fwd$descSp) node;
17 2 root$Sp=pointerize(root$St);
18 2 new$descSp=pointerize(new$descSt);
19 2 fwd$descSp=root.link$Sf;
20 2 fwd$descSp=pointerize(fwd$descSt);
21 2 root.link$Sf=new$descSt;
22 2 new$desc.link$Sf=fwd$descSt;
23 2 new$desc.link$Sb=root$St;
24 2 fwd$desc.link$Sb=new$descSt;
25 2 return;
26 2 end; /* insert$on$list */

Delete$on$list: procedure(desc$St) reentrant public;

27 2 declare
28 2  desc$St token,
    (desc$Sp,bSdesc$Sp,fSdesc$Sp) pointer,
    (desc based desc$Sp) node,
    (bSdesc based bSdesc$Sp) node,
    (fSdesc based fSdesc$Sp) node;
29 2 desc$Sp=pointerize(desc$St);
30 2 bSdesc$Sp=pointerize(desc.link$Sb);
31 2 fSdesc$Sp=pointerize(desc.link$Sf);
32 2 bSdesc.link$Sf=desc.link$Sf;
33 2 fSdesc.link$Sb=desc.link$Sb;
34 2 return;
Module 4, continued

end; /* delete$fromSlist */

searchSlist: procedure(rootSt, WSSID) word reentrant public;

declare
  (rootSt, WSSID) word,
  (s$desc$p, rootSp) pointer,
  (root based root$Sp) node,
  (s$desc based s$desc$P) node,
  s$desc$SpSo structure (offset word, base word) at(@s$desc$P),
  temp pointer;

s$desc$P=pointerize(rootSt);

next$node:
  if s$desc.work$station$ID=WSSID then
    return s$desc$SpSo.base;
  if s$desc.link$Sf = rootSt then
    return 0;
  temp=pointerize(s$desc.link$Sf);
  s$desc$P=temp;
  goto next$node;

end; /* searchSlist */

end list$utilities$module;

MODULE INFORMATION:

<table>
<thead>
<tr>
<th>CODE AREA SIZE</th>
<th>CONSTANT AREA SIZE</th>
<th>VARIABLE AREA SIZE</th>
<th>MAXIMUM STACK SIZE</th>
<th>114 LINES READ</th>
</tr>
</thead>
<tbody>
<tr>
<td>00FEH</td>
<td>0000H</td>
<td>0000H</td>
<td>0018H</td>
<td>254D</td>
</tr>
<tr>
<td>0D</td>
<td>0D</td>
<td>0D</td>
<td>0D</td>
<td>254D</td>
</tr>
</tbody>
</table>

END OF PL/M-86 COMPILATION
Module 5

ISIS-II PL/M-86 X167 COMPILATION OF MODULE STARTANDFINISH
OBJECT MODULE PLACED IN :Fl:strfin.OBJ
COMPILER INVOKED BY: plm86 :Fl:strfin.plm PRINT (:F5:STRFIN.LST)
DEBUG COMPACT OPTIMIZE(2) ROM DATE(4/28/80)

start$and$finish:
do;

******************************************************************************
INIT$534$IO and FINISH$534$IO: PUBLIC PROCEDURES.

This module contains the INIT$534$IO and the FINISH$534$IO procedures which can be called by the RMX/86 I/O system. START$IO is called just before the first attach$device is performed.
It will create the interrupt task and the eight interrupt$pending semaphores. The FINISH$IO procedure is called just after the last detach$device is performed. It undoes everything the START$IO call did.

******************************************************************************

#include(:f4:nucprm.ext)
= $SAVE NOLIST
#include(:f4:common.lit)
= $SAVE NOLIST
#include(:f1:duib.lit)
= /* duib structure definition */
= $save nolist
#include(:f4:nerror.lit)
= $SAVE NOLIST
#include(:f1:pointr.ext)
= /* external declaration of pointerize procedure */
= $save nolist
#include(:f1:retdata.lit)
= /* literal declaration of ret$data structure for INIT$534$IO */
= $save nolist

314 1 init$534$hw: procedure(data$Sp) external;
315 2 declare data$Sp pointer;
316 2 end init$534$hw; /* initializes 534 hardware */
317 1 int$534$task: procedure external;
318 2 end int$534$task;
319 1 declare
beginInt$534$data byte external,
IOSbase$addr byte public,
int$level word public,
g$ret$data$Sp pointer public,
req$mbox$t token public;

320 1 init$534$IO: procedure(duib$Sp,ret$data$t$Sp,status$Sp) reentrant public;
321 2 declare
(duib$Sp,ret$data$t$Sp,status$Sp) pointer,
(duib based duib$Sp) dev$unit$info$block,
(ret$data$t based ret$data$t$Sp) token,
(status based status$Sp) word,
dev$info$Sp pointer,
device$based dev$info$Sp structure(
  level word,
priority byte,
  IOSbase$addr byte),
Module 5, continued

exSval word,
dataSsegSp pointer,
dataSsegSpSo structure(offset word,base word) at(@dataSsegSp),
(1,j) byte;

322 2 declare
   ret$SdataSp pointer,
   ret$Sdata based ret$SdataSstruc;
323 2 ret$SdataSt=rq$ScreateSsegment(size(ret$Sdata),@exSval);
324 2 if exSval <> 0 then
325 2 goto err0;
326 2 g$ret$SdataSp,ret$SdataSp=pointerize(ret$SdataSt);
327 2 dev$SinfoSp=dub.dev$SinfoSp;
328 2 IO$base$addr,ret$Sdata.IO$base=dev$Sinfo.IO$base$addr;
329 2 int$level,ret$Sdata.int$level=dev$Sinfo.level;
/* create the request mailbox */
330 2 ret$Sdata.request$SmailboxSt,rq$SmailboxSt =rq$ScreateSmailbox(0,@exSval);
331 2 if exSval <> 0 then
332 2 goto err1;
333 2 ret$Sdata.resp$SmailboxSt=rq$ScreateSmailbox(0,@exSval);
334 2 if exSval <> 0 then
335 2 goto err2; /* clean up partial creation */
336 2 dataSsegSp=@begin$int$S34Sdata;
337 2 ret$Sdata.int$taskSt=rq$ScreateStask(
   /* priority */ dev$Sinfo.priority,
   /* entry point */ @int$534Stask,
   /* data segment */ dataSsegSpSo.base,
   /* stack pointer */ 0,
   /* stack size */ 400,
   /* task flags */ 0,
   /* status pointer */ @exSval);
338 2 if exSval <> 0 then
339 2 goto err3; /* can't create. clean up partial creation */
340 2 do i=0 to 7; /* create semaphores */
341 3 ret$Sdata.int$Ssema(i)=rq$ScreateSsemaphore(  
   /* initial value */ 0,
   /* max value */ 1,
   /* priority queue */ 1,
   /* status ptr */ @exSval);
342 3 if exSval <> 0 then
343 3 goto err4; /* clean up partial creation */
344 3 end;
345 2 call init$S534Shw(retSdataSp);
346 2 status=ESOK;
347 2 return;
348 2 err4:
   do j=0 to i;
349 3 call rq$SdeleteSsemaphore(ret$Sdata.int$Ssema(j),statusSp);
350 3 end;
351 2 call rq$Sreset$Sinterrupt(dev$Sinfo.level,statusSp);
352 2 err3:
   call rq$SdeleteSmailbox(ret$Sdata.resp$SmailboxSt,statusSp);
353 2 err2:
   call rq$SdeleteSmailbox(ret$Sdata.request$SmailboxSt,statusSp);
354 2 err1:
   call rq$Sdelete$Ssegment(ret$SdataSt,statusSp);
Module 5, continued

err0:
    status=exSval; /* restore original status condition */
    return;
end; /* of procedure */

finishS53410: procedure(duibSp, retSdataSt) reentrant public;
    declare
        duibSp pointer,
        dev$infoSp pointer,
        dev$info based dev$infoSp structure(
            level word,
            priority byte,
            IOSbase$addr byte),
        retSdataSp pointer,
        ret$data based ret$dataSp structure(ret$data$struct),
        (duib based duibSp) dev$unit$info$block,
        ret$dataSt token,
        i byte,
        exSval word;

    dev$infoSp=duib.dev$infoSp;
    ret$dataSp=pointerize(ret$dataSt);
    call rq$reset$interrupt(dev$info.level, #exSval);
    call rq$delete$mailbox(ret$data.request$mailboxSt, #exSval);
    call rq$delete$mailbox(ret$data.resp$mailboxSt, #exSval);
    do i=0 to 7;
        call rq$delete$semaphore(
            ret$data.int$sem(i),
            #exSval);
    end;
    call rq$delete$segment(ret$data$struct);  
    return;
end; /* of procedure */
end start$and$finish;

MODULE INFORMATION:

    CODE AREA SIZE = 0220H  544D
    CONSTANT AREA SIZE = 0000H   0D
    VARIABLE AREA SIZE = 0000H   9D
    MAXIMUM STACK SIZE = 0034H   52D
    671 LINES READ
    0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
Module 6

ISIS-II PL/M-86 X167 COMPILATION OF MODULE QUEUE534IOMODULE
OBJECT: MODULE PLACED IN :Fl:queio.OBJ
COMPILER INVOKED BY: plm86 :Fl:queio.plm PRINT(:F5:QUEIO.LST)
DEBUG COMPACT OPTIMIZE(2) ROM DATE(4/25/80)

queue534$iomodule:
do;

QUEUE534$IO. PUBLIC PROCEDURE.
This procedure is called by the I/O System to queue an I/O request to the 534 board. The function field in the IORS is used to determine what specific action to take. Module also contains a dummy cancel534$io procedure.

**************************************************************************
#include(:f4:nucprm.ext)
$SAVE NOLIST
#include(:f4:common.lit)
$SAVE NOLIST
#include(:f4:error.lit)
$SAVE NOLIST
#include(:f4:pointr.ext)
/* external declaration of pointerize procedure */
$save nolist
$ include(:f1:duib.lit)
/* duib structure definition */
$save nolist
#include(:f1:iors.lit)
/* literal declaration for iors */
$save nolist
#include(:f1:retdata.lit)
/* literal declaration of ret$data structure for init534$io */
$save nolist

io$534$task: procedure external;
end io$534$task;
declare
begin$io$task$data byte external;

queue534$i: procedure(iors$t,duib$Sp,ret$data$t) reentrant public;
declare
(iors$t,ret$data$t) token,
data$seg$Sp pointer,
data$seg$Sp$o structure(offset word,base word) at(@data$seg$Sp),
IDDR literally '2AH',
(duib$Sp,ret$data$Sp,iors$p) pointer,
(duib based duib$Sp) dev$unit$info$block,
(ret$data based ret$data$Sp) structure(ret$data$struct),
(iors based iors$Sp) IOSrequest$result$segment,
io$task$St token,
unit$info$Sp pointer,
unit$info based unit$info$Sp structure(
    usart$cmd byte,
    baud$rate word),
i byte,
dummy$St token,
ex$val word;
iors$sp=pointerize(iors$t);
ret$data$sp=pointerize(ret$data$t);

if iors.funct > 7 then
goto bad$request;
do case iors.funct;
do; /* case 0 -- read */
iors.aux$sp=ret$data$sp;
call rq$send$message(
/* mbox */ ret$data.request$mbox$t,
/* token */ iors$t,
/* resp */ 0,
/* status ptr */ @ex$val);
return;
end;
do; /* case 1 -- write */
iors.aux$sp=ret$data$sp;
call rq$send$message(
/* mbox */ ret$data.request$mbox$t,
/* token */ iors$t,
/* resp */ 0,
/* status ptr */ @ex$val);
return;
end;
do; /* case 2 -- seek (illegal) */
goto bad$request;
end;
do; /* case 3 -- special (illegal) */
goto bad$request;
end;
do; /* case 4 -- attach$device */
/* create two I/O tasks */
data$seg$sp=@begin$IO$task$data;
do i=0 to 1;
io$task$t= rq$create$task(
/* priority */ 150,
/* entry pnt */ @io$534$task,
/* data seg */ data$seg$sp$o.base,
/* stack ptr */ 0,
/* stack size */ 500,
/* task flags */ 0,
/* status ptr */ @ex$val);
end;
unit$info$sp=duib.unit$info$sp;
do i=0 to 3;
output(ret$data.usart$cmd$port(iors.unit))=0;
end;
output(ret$data.usart$cmd$port(iors.unit))=40H;
output(ret$data.usart$cmd$port(iors.unit))=unit$info.usart$cmd;
output(ret$data.usart$cmd$port(iors.unit))=27H;
output(ret$data.timer$cmd$port(iors.unit))=0; /* select cntrl blk */
output(ret$data.timer$load$port(iors.unit))=low(unit$info.baud$rate);
output(ret$data.timer$load$port(iors.unit))=high(unit$info.baud$rate);
Module 6, continued

357 4 output(ret$data.IO$base+0DH)=0; /* select data blk */
/* accept interrupt and character from receiver */
358 4 dummySt=rq$receive$units(
/* sema */ ret$data.int$sema( 2 * iors.unit),
/* units */ 1,
/* timeout */ 0,
/* status */ @ex$val);
359 4 i=input(ret$data.usart$data$port( iors.unit ));
360 4 goto ok$send$resp;
361 4 end;
362 3 do; /* case 5-- detach$device */
/* send two copies of the detach request to the request mailbox.
   This will signal to two of the I/O tasks that they are to
delete themselves */
363 4 call rq$send$message(
/* mbox token */ ret$data.request$mbox$t,
/* object token */ iors$t,
/* response */ ret$data.resp$mbox$t,
/* status */ @ex$val);
364 4 dummySt=rq$receive$message(
/* mbox token */ ret$data.resp$mbox$t,
/* time$limit */ 0FFFFH,
/* response ptr */ @dummySt,
/* status ptr */ @ex$val);
365 4 call rq$send$message(
/* mbox token */ ret$data.request$mbox$t,
/* object token */ iors$t,
/* response */ ret$data.resp$mbox$t,
/* status */ @ex$val);
366 4 dummySt=rq$receive$message(
/* mbox token */ ret$data.resp$mbox$t,
/* time$limit */ 0FFFFH,
/* response ptr */ @dummySt,
/* status ptr */ @ex$val);
367 4 goto ok$send$resp;
368 4 end;
369 3 do; /* case 6-- open */
370 4 goto ok$send$resp;
371 4 end;
372 3 do; /* case 7-- close */
373 4 goto ok$send$resp;
374 4 end;
375 3 end; /* do case */
376 2 return;
377 2 bad$request:
    iors.status=IDDR;
    goto send$resp;
378 2 ok$send$resp:
    iors.status=ESOK;
379 2 send$resp:
        call rq$send$message(iors.resp$mbox,iorsSt,0,@ex$val);
380 2 return;
381 2 end; /* procedure */
382 2 cancel$534$io: procedure(iors$t,duib$p,ret$data$t) public;
383 1 declare
    (iors$t,ret$data$t) token,
    duib$p pointer;
384 2 return;
Module 6, continued

386  2    end;
387  1    end queue$534$io$module;

MODULE INFORMATION:

CODE AREA SIZE = 020CH 524D
CONSTANT AREA SIZE = 0000H 0D
VARIABLE AREA SIZE = 0000H 0D
MAXIMUM STACK SIZE = 0038H 56D
729 LINES READ
0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
Module 7

ISIS-II PL/M-86 V2.0 COMPILATION OF MODULE INTERRUPT534

MODULE

OBJECT MODULE PLACED IN :Fl:int534.obj
COMPILER INVOKED BY: plm86 :Fl:int534.plm PRINT(:Fl:int534.LST)
DEBUG COMPACT OPTIMIZE(2) ROM DATE(5/28/80)

$nointerrupt
1 Interrupt$534$module:
do;

/***********************************************************************
INT$534$TASK and INT$534$HND:
PUBLIC PROCEDURES:

This module contains the interrupt handler and the interrupt
task for the 534 board interrupt. The handler simply calls
signal$interrupt and the task reads the ISR on the 534
board's 8259 and sends a unit to one of eight interrupt$pending semaphores to signal the occurrence of the event.
***********************************************************************/

#include(:f2:nucprm.ext)
#endif
#define NOLIST
#include(:f1:retdata.lit)
#define literal declaration of ret$data structure for init$534$io */
#define $save nolist
#include(:f2:common.lit)
#define $SAVE NOLIST

308 declare
begin
int$534$base byte public,
g$ret$data$Sp pointer external,
IOS$base$addr byte external,int$level word external;

309 int$534$hnd: procedure interrupt 5;

310 declare
l word,
ex$val word;

311 l=rq$get$level(@ex$val);
312 call rq$signal$interrupt(l,@ex$val);
313 return;
314 end;

315 int$534$task: procedure reentrant public;

316 declare
IOS$base byte,int$534$level word,
ret$data$Sp pointer,
ret$data based ret$data$Sp structure(ret$data$struc),
c$level byte,
ex$val word,
eoi literally '20H';

317 IOS$base=IOS$base$addr;
318 int$534$level=int$level;
319 ret$data$Sp=g$ret$data$Sp;
320 call rq$set$interrupt(
/* level */ int$534$level,
/* flags */ 1,
/* entry point */ interrupt$ptr(int$534$hnd),
/* data segment */ 0,
/* status ptr */ @ex$val);
Module 7, continued

321 2 do forever;
322 3 call rq$wait$interrupt(int$534$level,@ex$val);
323 3 output(IO$534$base+8)=0CH;
324 3 c$level=input(IO$534$base+8) and 07H;
325 3 call rq$send$units(ret$data.int$sema(c$level),l,@ex$val);
326 3 output(IO$534$base+8)=EOI;
327 3 end; /* of do forever */
328 2 end; /* of procedure */
329 1 end interrupt$534$module;

MODULE INFORMATION:

CODE AREA SIZE = 00B5H 181D
CONSTANT AREA SIZE = 0000H 0D
VARIABLE AREA SIZE = 0005H 5D
MAXIMUM STACK SIZE = 0026H 38D
541 LINES READ
0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
Module 8

ISIS-II PL/M-86 X167 COMPILATION OF MODULE IO534TASKMODULE
OBJECT MODULE PLACED IN :F1:iotask.OBJ
COMPILER INVOKED BY: plm86 :F1:iotask.plm PRINT(:F5:IOTASK.LST)
DEBUG COMPACT OPTIMIZE(2) ROM DATE(4/25/88)

io$534$task$module:
   do;

/******************************************************************************
IO$534$TASK: TASK.
This task receives IORS segments from the queue$io
procedure and performs the necessary input or
output operations on the iSBC 534 board.
*******************************************************************************/
$include(:f4:common.lit)
$SAVE NOLIST
$include(:f1:pointr.ext)
/* external declaration of pointerize procedure */
$save nolist
$include(:f4:nuccpm.ext)
$SAVE NOLIST
$include(:f4:nerror.lit)
$include(:f1:retdta.lit)
/* literal declaration of ret$data structure for init$534$sio */
$save nolist
$include(:f1:iors.lit)
/* literal declaration for iors */
$save nolist

314 1 declare
   begin$iostask$data byte public,
   reg$mbox$token external,
   f$detach$device literally '5',
   f$read literally '0',
   f$write literally '1';

315 1 IO$534$task: procedure reentrant public;

316 2 declare
   iors$token,
   iors$pointer,
   iors based iors$ as iors$segment,
   ex$word,
   resp$token,
   buff$pointer,
   buf based buff$ (1) byte,
   i word,
   unit byte,
   ret$data$pointer,
   ret$data based ret$data$ as ret$data$segment,
   c$word;

317 2 do forever;
318 3 iors$t=reg$receive$message(reg$mbox$token,0FFFFH,resp$token,ex$word);

   /* check for non-existence of mailbox. If last device has been detached
    the mailbox will be deleted In this case, delete thyself */

319 3 if ex$word= ESexist then
320 3 call reg$delete$task(0,@ex$word);
321 3 iors$pointerize(iors$t);
Module 8, continued

322 3 buffSp=iors.buffSp;
323 3 unit=iors.unit;
324 3 iors.actual=0;
325 3 i=0;
326 3 ret$dataSp=iors.auxSp;
327 3 if iors.funct = fSdetach$device then
328 3  do;
329 4  call rq$send$message(
330 4  /* mbox token */  respSt,
331 4  /* object token */  iors$t,
332 4  /* response token */  0,
333 4  /* status ptr */  @ex$Val);
334 4  call rq$delete$task(0,@ex$Val);
335 4  end;
336 3  end; /* of do forever */
337 3  end io$534$task$module;
338 3

MODULE INFORMATION:

CODE AREA SIZE = 018DH 397D
CONSTANT AREA SIZE = 0000H 0D
VARIABLE AREA SIZE = 0001H 1D
MAXIMUM STACK SIZE = 0028H 40D
624 LINES READ
0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
Module 9

ISIS-II PL/M-86 X167 COMPILATION OF MODULE INIT534HW

OBJECT MODULE PLACED IN: F1:inithw.OBJ

COMPILER INVOKED BY: plm86 : F1:inithw.plm
PRINT( : F5:INITHW.LST)

DEBUG COMPACT OPTIMIZE(2) ROM DATE(4/25/80)

1.

```
init$534$hw:
do;

******************************************************************************
init$534$hw: PUBLIC PROCEDURE.

This procedure initializes the iSBC 534 hardware and
sets up the device dependent fields of the ret$data
segment which will be used by the queue$io procedures.
*******************************************************************************/

#include(:f4:common.lit)
$save nolist
#include(: f1:retdata.lit)
/* literal declaration of ret$data structure for init$534$io */
$save nolist

init$534$hw: procedure(ret$dataSp) reentrant public;

declare
ret$dataSp pointer,
ret$data based ret$dataSp structure(ret$dataSstruct),
(base,i) byte;

base=ret$data.io$base;
output(base+0FH)=0; /* board reset */
output(base+0DH)=0; /* select data block */
output(base+8)=16H; /* output ICWI */
output(base+9)=0; /* output ICW2 */
output(base+9)=0; /* output mask word */

/* attach$device calls will initialize usarts and timers */
/* set up tables of port addresses for use by queue$io procs */

ret$data.timer$cmd(0),ret$data.timer$cmd(3)=36H;
ret$data.timer$cmd(1)=76H;
ret$data.timer$cmd(2)=0B6H;
do i=0 to 3;
ret$data.usart$cmd$port(i)=base+2*i+1;
ret$data.usart$data$port(i)=base+2*i;
ret$data.timer$load$port(i)=base+i;
end;
ret$data.timer$load$port(3)=base+4;
ret$data.timer$cmd$port(0),
ret$data.timer$cmd$port(1),
ret$data.timer$cmd$port(2)=base+3;
ret$data.timer$cmd$port(3)=base+7;
return;
end init$534$hw;

MODULE INFORMATION:

<table>
<thead>
<tr>
<th>CODE AREA SIZE</th>
<th>CONSTANT AREA SIZE</th>
<th>VARIABLE AREA SIZE</th>
<th>MAXIMUM STACK SIZE</th>
<th>77 LINES READ</th>
</tr>
</thead>
<tbody>
<tr>
<td>00E4H</td>
<td>0000H</td>
<td>0000H</td>
<td>0000H</td>
<td></td>
</tr>
</tbody>
</table>

END OF PL/M-86 COMPILATION
APPENDIX B
Configuration Listings/Worksheets
System Memory Map

; *-*-*-*-*-*-*-*-*-*-*-*-*  NUCLK.CSD  *-*-*-*-*-*-*-*-*-*-*-*-*
;
; THIS SUBMIT FILE LINKS THE NUCLEUS.
;
; F0:LINK86 &
; F1:NUC86.LIB(NENTRY), &
; F1:NUC86.LIB &
; TO :F1:NUCLUS.LNK MAP PRINT(:F1:NUCLUS.MP1) NAME(NUCLEUS)
;
; *-*-*-*-*-*-*-*-*-*-*-*-*  NUCLOC.CSD  *-*-*-*-*-*-*-*-*-*-*-*-*
;
; THIS SUBMIT FILE LOCATES THE NUCLEUS IN MEMORY.
;
; F0:LOC86 &
; F1:NUCLUS.LNK TO :F1:NUCLUS MAP PRINT(:F1:NUCLUS.MP2) SC(3) &
; RESERVE(0 TO 7FFH) SEGSIZE(STACK(0)) &
; ORDER(CLASSES(CODE,DATA,STACK,MEMORY)) &
; OBJECTCONTROLS(NOLINES,NOCOMMENTS,NOPUBLICS,NOSYMBOLS)

Nucleus Link and Locate Commands
Asm86: fl:iocnfg.a86 date(%0)
link86 &
:fl:iocnfg.lib(ioinit), &
:fl:iocnfg.obj, &
:fl:iocnfg.lib, &
:fl:rpifc.lib &
to :fl:iocnfg.lnk map print(:fl:iocnfg.mpl)
loc86 :fl:iocnfg.lnk to :fl:iocnfg map sc(3) print(:fl:iocnfg.mp2) &
oc(noli,nopl,nocm,nosb) &
order(classes(code,data,stack,memory)) &
double(classes(code(0))) &
segsize(stack(0))

I/O System Link and Locate Commands

; Sample I/O System .csd file to link and locate an I/O System.
; This file links an I/O System with the timer included.
; This .csd file assumes the I/O System configuration module is
; iocnfg.a86 (found on the release diskette).
; The origin parameter sets the low address of the I/O System;
; all the segments are contiguous in memory.
;_asm86 :fl:iocnfg.a86 date(%0)
link86 &
:fl:iocnfg.lib(ioinit), &
:fl:iocnfg.obj, &
:fl:iocnfg.lib, &
:fl:rpifc.lib &
to :fl:iocnfg.lnk map print(:fl:iocnfg.mpl)
loc86 :fl:iocnfg.lnk to :fl:iocnfg map sc(3) print(:fl:iocnfg.mp2) &
oc(noli,nopl,nocm,nosb) &
order(classes(code,data,stack,memory)) &
double(classes(code(0))) &
segsize(stack(0))

File Transaction Job; Link and Locate Commands

; Submit file to generate located version of file transaction job
;_link86 &
:fl:ftinit.obj, &
:fl:listen.obj, &
:fl:worker.obj, &
:fl:pointgr.obj, &
:fl:rpifc.lib &
to :fl:apexl.lnk map print(:fl:apexl.mpl)
loc86 :fl:apexl.lnk to :fl:apexl map sc(3) print(:fl:apexl.mp2) &
oc(noli,nopl,nocm,nosb) &
order(classes(code,data,stack,memory)) &
double(classes(code(0))) &
segsize(stack(0))

Communications Job; Link and Locate Commands

; Submit file to generate located version of communications job
;_link86 &
:fl:cminit.obj, &
:fl:comm.lib, &
:fl:pointgr.obj, &
:fl:rpifc.lib &
to :fl:comm.lnk map print(:fl:comm.mpl)
loc86 :fl:comm.lnk to :fl:comm map sc(3) print(:fl:comm.mp2) &
oc(noli,nopl,nocm,nosb) &
order(classes(code,data,stack,memory)) &
double(classes(code(0))) &
segsize(stack(0))
Locate Map for I/O System
(The "→" indicates entries for job macros and memory map)

Locate Map for Communications Job

Locate Map for File Transaction Job
Macro call: SYSTEM (system parameters)

Number of calls required: exactly one

FORMAT:

<table>
<thead>
<tr>
<th>parameter</th>
<th>type</th>
<th>suggested default</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(%SYSTEM nucleus_entry,)</td>
<td>base</td>
<td></td>
<td>80:0</td>
</tr>
<tr>
<td>rod_size,</td>
<td>word</td>
<td>(0)</td>
<td>10</td>
</tr>
<tr>
<td>min_trans_size,</td>
<td>work</td>
<td>(64)</td>
<td>64</td>
</tr>
<tr>
<td>debugger,</td>
<td>see note</td>
<td>(A)</td>
<td></td>
</tr>
<tr>
<td>default__e__h__provided,</td>
<td>see note</td>
<td>(N)</td>
<td></td>
</tr>
<tr>
<td>mode)</td>
<td>word</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

NOTES:

1. Valid entries for the debugger parameter include:
   - A Debugger available
   - N No debugger available

2. Valid entries for the default__e__h__provided parameter include:
   - Y Yes
   - D Debugger
   - N No
**Macro call:** SAB (for system address blocks)

**Number of calls required:** one or more

**CONFIGURATION FILE NAME:** APEX1

**FORMAT:**

<table>
<thead>
<tr>
<th>parameter</th>
<th>type</th>
<th>suggested default</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SAB (start_base, end_base, type)</td>
<td>base</td>
<td>see note 1</td>
<td>U</td>
</tr>
</tbody>
</table>

**NOTES:**

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

2. A SAB is declared between start_base:0 and end_base:F, inclusive.
Macro call: \textbf{JOB} (defines first-level jobs)

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

CONFIGURATION FILE NAME: APEX 1

\textbf{FORMAT:}

<table>
<thead>
<tr>
<th>parameter</th>
<th>suggested type</th>
<th>default</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%JOB (directory_size,</td>
<td>word</td>
<td>(0)</td>
<td>0</td>
</tr>
<tr>
<td>pool_min,</td>
<td>word</td>
<td></td>
<td>\textbf{OFFFF}</td>
</tr>
<tr>
<td>pool_max,</td>
<td>word</td>
<td></td>
<td>\textbf{OFFFF}</td>
</tr>
<tr>
<td>max_objects,</td>
<td>word</td>
<td></td>
<td>\textbf{FFFF}</td>
</tr>
<tr>
<td>max_tasks,</td>
<td>word</td>
<td></td>
<td>\textbf{FFFF}</td>
</tr>
<tr>
<td>max_job_priority,</td>
<td>byte</td>
<td></td>
<td>129</td>
</tr>
<tr>
<td>exception_handler_entry,</td>
<td>addr</td>
<td>(0:0)</td>
<td>0:0</td>
</tr>
<tr>
<td>exception_handler_mode,</td>
<td>byte</td>
<td>(1)</td>
<td>1</td>
</tr>
<tr>
<td>job_flags,</td>
<td>word</td>
<td>(0)</td>
<td>0</td>
</tr>
<tr>
<td>init_task_priority,</td>
<td>byte</td>
<td></td>
<td>\textbf{1713:112}</td>
</tr>
<tr>
<td>data_segment_base,</td>
<td>base</td>
<td>(0)</td>
<td>\textbf{17D6}</td>
</tr>
<tr>
<td>stack_pointer,</td>
<td>addr</td>
<td>(0:0)</td>
<td>0:0</td>
</tr>
<tr>
<td>stack_size,</td>
<td>word</td>
<td>(512)</td>
<td>512</td>
</tr>
<tr>
<td>task_flags)</td>
<td>word</td>
<td>(0)</td>
<td>0</td>
</tr>
</tbody>
</table>

\textbf{NOTE:}

1. \textit{addr} is specified as base:offset

\%JOB Macro Worksheet
Configuration File Apex 1.CNF

; ;**--*--*--*--*--*--*-- CTABLE.CSD --*--*--*--*--*--*--*--**
; ;  SUBMIT :Fx:CTABLE( fsys, fin, fout, config_file, date )
; ;  This submit file assembles the CTABLE module, where:
; ;    fsys = the system disk containing ASM86
; ;    fin  = the source/input disk (FI is assumed)
; ;    fout = the object/listing/output disk
; ;    config_file = the path-name of the configuration file
; ;    date  = the date
; ;  copy %5 to :%1:config.cnf
; ;%0:asm86 :%1:ctable.a86 pr(:%2:ctable.lst) oj(:%2:ctable.obj) date(%4) & xref debug ep

Submit File to Generate Configuration Table

; ; ;**--*--*--*--*--*--*-- CLNKRJ.CSD --*--*--*--*--*--*--*--**
; ;  SUBMIT :Fx:CLNKRJ( fsys, fin, fout )
; ;  This submit file links the Root-Job, where:
; ;    fsys = the system disk containing LINK86
; ;    fin  = the source/input disk
; ;    fout = the object/listing/output disk
; ;%0:link86 :%1:croot.lib(root),&
; ;%2:ctable.obj,&
; ;%1:croot.lib &
; ;to :%2:rootjb.lnk map pr(:%2:rootjb.mpl)
; ;
; ;-- SUBMIT:Fx:CloC RJ( fsys, fin, fout )
; ; This submit file locates the Root-Job, where:
; ; fsys = the system disk containing LOC86
; ; fin = source/input disk
; ; fout = object/listing/output disk
; ;-- NOTE: BE SURE TO REPLACE THE "?????" BELOW WITH THE APPROPRIATE
; ; ADDRESS THE ROOT-JOB IS TO BE LOCATED AT!!
; ;%0:loc86:%2:rootjb.lnk to:%2:rootjb &
; map pr(%2:rootjb.mp2) sc(3) &
; name(ROOT JOB) oc(nocm,noli,nopl,nosb) &
; segsize(stack(200h)) &
; order(classes(data,stack,memory,code)) &
; addresses(classes(data(12C00H)))

Submit File to Locate Root Job
REQUEST FOR READER’S COMMENTS

The Microcomputer Division Technical Publications Department attempts to provide documents that meet the needs of all Intel product users. This form lets you participate directly in the documentation process.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this document.

1. Please specify by page any errors you found in this manual.

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

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

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

5. Please rate this document on a scale of 1 to 10 with 10 being the best rating.

NAME ______________________________________________________ DATE ______________________

TITLE ________________________________

COMPANY NAME/DEPARTMENT ____________________________________________

ADDRESS ________________________________

CITY ______________________ STATE ______ ZIP CODE ____________

Please check here if you require a written reply. □

Please mail to: Intel Corporation
Attention: Joe Barthmaier
5200 N.E. Elam Young Parkway
Hillsboro, Oregon 97123
INTERNATIONAL SALES AND MARKETING OFFICES

INTERNATIONAL DISTRIBUTORS/REPRESENTATIVES

ARGENTINA
Micro Sistemas S.A.
9 De Julio 561
Cordoba
Tel: 54-51-32-880
TELEX: 51837 BICC0

AUSTRALIA
A.J.F. Systems & Components Pty. Ltd.
310 Queen Street
Melbourne
Victoria 3000
Tel: TELEX:
Warburton Franki
Corporate Headquarters
372 Eastern Valley Way
Chatswood, New South Wales 2067
Tel: 407-3291
TELEX: AA 21299

AUSTRIA
Bacher Elektronische Geräte GmbH
Rotenmulnngasse 25
A 1120 Viennn
Tel: (0222) 83 63 96
TELEX: (31) 1532
Rekirsch Elektronische Geräte GmbH
Lichtensteinstrasse 97
A1000 Vienna
Tel: (022) 347846
TELEX: 74759

BELGIUM
Inelco Belgium S.A.
Ave. des Croix de Guerre 94
B1120 Brussels
Tel: (02) 216 01 60
TELEX: 25441

BRAZIL
0511-Av. Mutilina 3850
6 Andar
Piratuba-Sao Paulo
Tel: 251-2011
TELEX: (011) 222 ICO BR

CHILE
DIN
Av. Vic. Mc kenna 204
Casilla 6055
Santiago
Tel: 227 504
TELEX: 3520003

CHINA
C.M. Technologies
525 University Avenue
Suite 4-40
Palo Alto, CA 94301

COLOMBIA
International Computer Machines
Adpo. Aereo 19403
Bogotá 1
Tel: 232-6635
TELEX: 43439

CYPRUS
Cyprus Eletron Electronics
P.O. Box 5938
Nicosia
Tel: 21-27982

DENMARK
STL-Lynxko Komponent A/S
Ostmarken 4
DK-2660 Sorbog
Tel: (01) 07 00 77
TELEX: 22990

FINLAND
Oy Fintronic AB
Mellkonkatu 24 A
00201 Helsinki
Tel: 0-4926022
TELEX: 124 224 Flton SF

FRANCE
Cedix S.A.*
53, Rue Charles Freoret
F-94250 Gentilly
Tel: (1) 501 00 20
TELEX: 200 485
Feutrier
Rue des Trois Glorieuses
F-42270 St. Priest-en-Jarez
Tel: (77) 74 67 33
TELEX: 300 0 21
Metrologie*
La Tour d'Asnieres
4, Avenue Laurent Celty
92600-Asnieres
Tel: 791 44 44
TELEX: 611 446
Telekron Astronic*
Cite des Buresier
Rue Carle Vernet
F-92210 Sevres
Tel: (1) 534 75 35
TELEX: 204562

GERMANY
Electronic 2000 Vertriebs GmbH
Neumarker Strasse 75
D-8000 Munich 80
Tel: (089) 4346061
TELEX: 522561
Jermyn GmbH
Postfach 1180
D-6077 CAMBERG
TEL: 45442
Ktron Elektron GmbH
Breslauerstrasse 2
8037 Eching B
D-8000 Munich
Tel: (089) 319.011
TELEX: 522122
Neye Eleotechnik GmbH
Schillerstrasse 14
D-2085 Quickborn-Hamburg
Tel: 041086121
TELEX: 021390

GREECE
American Technical Enterprises
P.O. Box 156
Athens
Tel: 30-1-0811717
30-1-0219470

HONG KONG
Schmidt & Co.
28/F Wing on Center
Connaught Road
Hong Kong
Tel: 450-444
TELEX: 74786 Schmc Hk

INDIA
Micron Devices
104109C, Nirmal Industrial Estate
Sion (E)
Bombay 400022, India
Tel: 486-170
TELEX: 011-5947 MDEV IN

ISRAEL
Eletronics Ltd.*
11 Rozana Street
P.O. Box 3900
Tel Aviv 61360
Tel: 475151
TELEX: 36306

ITALY
Eledra 35 S.P.A.*
Viale Elvezia, 18
I 20154 Milan
Tel: (02) 345.041-3185.441
TELEX: 332332

JAPAN
Asahi Electronics Co. Ltd.
KMM Bldg, Room 407
2-14-1, Asano, Kokura
Kitakyushu City 802
Tel: (093) 511-6471
TELEX: AECKY 7126-16
Hamilton-Atwell Electronics Japan Ltd.
YU and YOU Bldg. 1-4 Horidome-Cho
Nihonbashi
Tel: (03) 662-8911
TELEX: 2920774

Nippon Micro Computer Co. Ltd.
Mutsumi Bldg. 4-5-21 Kojimachi
Chiyoda-ku, Tokyo 102
Tel: (03) 230-0041
Ryoyo Electric Corp.
Konwa Bldg.
1-12-22, Tsukiji, 1-Chome
Chuo-Ku, Tokyo 104
Tel: (03) 543-7711
Tokyo Electric Ltd.
No. 1 Higashihakata-Machi
Hinata-ku, Yokohama 226
Tel: (045) 471-4811
TELEX: 781-4473

KOREA
Koram Digital
Room 411 Ahl Bldg.
494-2 JAG Hoehyong-Dong
Chung-Ku Seoul
Tel: 23-8123
Korea International, Inc.
C.P.O. Box 4046
112-25, Sogong-Dong
Chung-Ku, Seoul 100
Tel: 29-5827
CABLE: "LEEWOOD" Seoul

NETHERLANDS
Inelco Nether. Comp. Sys. BV
Turfsterklaarstraat 63
Aalsmeer 1431 D
Tel: (0775) 26655
TELEX: 14963
Koning & Hartman
Koperwelt 30
2544 EN Den Haag
Tel: (70) 210.101
TELEX: 31528

NEW ZEALAND
W. K. McLean Ltd.
P.O. Box 19-080
Glenn Innes, Auckland, 6
Tel: 587-037
TELEX: NZ2763 KOSFY

NORWAY
Nordisk Elektronik (Norge) AS
Postoffice Box 122
Smedstvingen 4
1364 Hvalstad
Tel: 02 78 62 10
TELEX: 17546

PORTUGAL
Dihram
Componentes E Eletronicas LDA
Av. Miguel Bombarda, 133
Lisboa 1
Tel: (19) 545313
TELEX: 14347 GESPIC

SINGAPORE
General Engineers Associates
Bik 3, 1003-1038, 10th Floor
P.S.A. Multi-Storey Complex
Telok Blangah/Pasir Panjang
Singapore 5
Tel: 271-3163
TELEX: RZ92387 GENERCO

SOUTH AFRICA
Electronic Building Elements
Pine Square
18th Street
Hazelwood, Pretoria 0001
Tel: 789-221
TELEX: 03018A

SPAIN
Interface
Av. Generalisimo 51 9*
E-Madrid 18
Tel: 456 3151
ITT SESA
Miguel Angel 16
Madrid 10
Tel: (1) 4190647
TELEX: 277072461

SWEDEN
AB Gosta Backstrom
Box 1200
10221 Stockholm
Tel: (08) 541 060
TELEX: 10135
Nordisk Elektron AB
Box 27301
S-10525 Stockholm
Tel: (08) 630400
TELEX: 10547

SWITZERLAND
Industrie AG
Gemenkestraasse 2
Postcheck 90 - 21900
CH-8201 Zurich
Tel: (01) 60 22 30
TELEX: 56768

TAWAIN
Taiwan Automation Co.*
444, Shin Hsiang Rd.
Hsing Ho East Road
Taipei
Tel: 771-0940
TELEX: 11942 TAIAMO

TURKEY
Turktelek Electronics
Adapark Boulevard 189
Ankara
Tel: 19483

UNITED KINGDOM
Conway Microsystems Ltd.
Market Street
68-Bracknell, Berkshire
Tel: (0340) 51456
TELEX: 594201
G.E.C. Semiconductors Ltd.
East Lane
North Wembley
Middlesex HA9 7PP
Tel: (01) 904-9303-906-4111
TELEX: 28617
Jerryn Industries
Vestry Estate
Sevenoaks, Kent
Tel: (0732) 501.44
TELEX: 95142
Rapid Recall, Ltd.
6 Soho Mills Ind. Park
Wooburn Green
Bucks, England
Tel: (0629) 24961
TELEX: 649439

Sinnrom Electronics Ltd.*
Airtown Road 2
Reading, Berkshire RG2 0LS
Tel: (07340) 85644
TELEX: 647395

VENEZUELA
Componentes y Circuitos
Electronics TLCA C.A.
Apartado 1013
Caracas 101
Tel: 718-100
TELEX: 21705 TELETIPOS

*Field Application Location
INTERNATIONAL SALES AND MARKETING OFFICES

INTEL® MARKETING OFFICES

AUSTRALIA
Intel Australia
Suite 2, Level 15, North Point
100 Miller Street
North Sydney, NSW, 2060
Tel: 450-847
TELEX: AA 20097

BELGIUM
Intel Corporation S.A.
Rue du Moulin a Papier 51
B-1160 Brussels
Tel: (02) 660 30 10
TELEX: 24814

DENMARK
Intel Denmark A/S
Lynghøj 32 2nd Floor
DK-2100 Copenhagen East
Tel: (01) 182000
TELEX: 19567

FINLAND
Intel Scandinavia
Sentnerikuja 3
SF - 00400 Helsinki 40
Tel: (0) 558531
TELEX: 123 332

FRANCE
Intel Corporation, S.A.R.L.*
5 Place de la Balance
Silico 233
94528 Rungis Cedex
Tel: (01) 687 22 21
TELEX: 270475

GERMANY
Intel Semiconductor GmbH*
Seidlstrasse 27
6000 Muenchen 2
Tel: (089) 53 891
TELEX: 523 177
Intel Semiconductor GmbH
Mainzer Strasse 75
6200 Wiesbaden 1
Tel: (06121) 700874
TELEX: 04186183
Intel Semiconductor GmbH
Wernerstrasse 67
P.O. Box 1460
7012 Fellbach
Tel: (0711) 58082
TELEX: 7254826
Intel Semiconductor GmbH
Hindenburgstrasse 28/29
3000 Hannover 1
Tel: (0511) 82081
TELEX: 926026

HONG KONG
Intel Trading Corporation
99-105 Des Voeux Rd., Central
18F, Unit B
Hong Kong
Tel: 5-450-847
TELEX: 63869

ISRAEL
Intel Semiconductor Ltd.*
P.O. Box 2404
Haifa
Tel: 972/452 4261
TELEX: 92248511

ITALY
Intel Corporation Italia, S.p.A.
Corso Sempione 39
I-20145 Milano
Tel: 2/34.93287
TELEX: 311271

JAPAN
Intel Japan K.K.*
Flower Hill-Shinmachi East Bldg.
1-23-8, Shinmachi, Setagaya-ku
Tokyo 154
Tel: (03) 426-9261
TELEX: 751-26426

NETHERLANDS
Intel Semiconductor B.V.
Cometongebouw
Westblaak 106
3012 Km Rotterdam
Tel: (010) 149122
TELEX: 22283

NORWAY
Intel Norway A/S
P.O. Box 92
Hvamveien 4
N-2013 Sjølten
Tel: (2) 742 420
TELEX: 18018

SWEDEN
Intel Sweden A.B.*
Box 20092
Alpvagen 17
S-16120 Bromma
Tel: (08) 98 53 90
TELEX: 12261

SWITZERLAND
Intel Semiconductor A.G.
Forchstrasse 95
CH 8032 Zurich
Tel: 1-55 45 02
TELEX: 557 89 ich ch

UNITED KINGDOM
Intel Corporation (U.K.) Ltd.
Broadfield House
4 Between Towns Road
Cowley, Oxford OX4 3NB
Tel: (0865) 77 14 31
TELEX: 637203

*Field Application Location