RELATED INTEL PUBLICATIONS

Introduction to the iRMX 86™ Operating System, 501308
iRMX 86™ Nucleus, Terminal Handler, and Debugger Reference Manual, 501300
iRMX 86™ Installation Guide for ISIS-II Users, 501295
iRMX 86™ Configuration Guide for ISIS-II Users, 501297
8086 Family User's Manual, 205885
iSBC 86/12A™ Single Board Computer Hardware Reference Manual, 501180
PL/M 86™ Programming Manual, 402300
MCS-86™ Assembly Language Reference Manual, 402105
MCS-86™ Software Development Utilities Operating Instructions for ISIS-II Users, 402125
iSBC 88/40™ Hardware Reference Manual
AP-86, Using the iRMX 86 Operating System

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

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel’s software license, or as defined in ASPR 7-104.9(a)(9). Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

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

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

ICE Micromap iCS
Intellec iSBC CREDIT
Megachassis UPI iRMX
Prompt iSBX μScope
Insite MULTIMODULE MULTIBUS
Library Manager Intel MCS
RMX

and the combinations of iAPX, ICE, iSBC, MCS, iSBX, RMX 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
## Using the RMX 86™ Operating System on iAPX 86™ Component Designs

### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>OPERATING SYSTEMS FUNCTIONAL OVERVIEW</td>
<td>2</td>
</tr>
<tr>
<td>Multiprogramming</td>
<td>2</td>
</tr>
<tr>
<td>Multitasking</td>
<td>2</td>
</tr>
<tr>
<td>Growth</td>
<td>2</td>
</tr>
<tr>
<td>Scheduling</td>
<td>2</td>
</tr>
<tr>
<td>Communication and Synchronization</td>
<td>3</td>
</tr>
<tr>
<td>Resource Management</td>
<td>3</td>
</tr>
<tr>
<td>Interrupt Management</td>
<td>3</td>
</tr>
<tr>
<td>Initialization</td>
<td>4</td>
</tr>
<tr>
<td>Debugging</td>
<td>4</td>
</tr>
<tr>
<td>Higher Level Functions</td>
<td>4</td>
</tr>
<tr>
<td>Extendibility</td>
<td>4</td>
</tr>
<tr>
<td>iRMX 86 OPERATING SYSTEM ARCHITECTURE</td>
<td>4</td>
</tr>
<tr>
<td>Layers</td>
<td>4</td>
</tr>
<tr>
<td>Configurability</td>
<td>5</td>
</tr>
<tr>
<td>Support Functions</td>
<td>5</td>
</tr>
<tr>
<td>Object Orientation</td>
<td>5</td>
</tr>
<tr>
<td>Task Scheduling</td>
<td>6</td>
</tr>
<tr>
<td>System Hardware Requirements</td>
<td>7</td>
</tr>
<tr>
<td>APPLICATION EXAMPLE</td>
<td>7</td>
</tr>
<tr>
<td>System Design</td>
<td>8</td>
</tr>
<tr>
<td>Jobs and Tasks</td>
<td>9</td>
</tr>
<tr>
<td>Interfaces and Synchronization</td>
<td>10</td>
</tr>
<tr>
<td>Priority</td>
<td>11</td>
</tr>
<tr>
<td>APPLICATION IMPLEMENTATION</td>
<td>11</td>
</tr>
<tr>
<td>Display Functions</td>
<td>11</td>
</tr>
<tr>
<td>Cataloging</td>
<td>12</td>
</tr>
<tr>
<td>Application Code</td>
<td>12</td>
</tr>
<tr>
<td>Supervisor Task</td>
<td>12</td>
</tr>
<tr>
<td>Input Task</td>
<td>12</td>
</tr>
<tr>
<td>FFT Task</td>
<td>13</td>
</tr>
<tr>
<td>Output Task</td>
<td>13</td>
</tr>
<tr>
<td>Terminal Handler</td>
<td>13</td>
</tr>
<tr>
<td>Nucleus Calls</td>
<td>14</td>
</tr>
<tr>
<td>System Configuration</td>
<td>14</td>
</tr>
<tr>
<td>SUMMARY</td>
<td>16</td>
</tr>
<tr>
<td>APPENDIX A — SUPERVISOR TASK LISTINGS</td>
<td>18</td>
</tr>
<tr>
<td>APPENDIX B — INPUT TASK LISTINGS</td>
<td>39</td>
</tr>
<tr>
<td>APPENDIX C — SYSTEM CONFIGURATION</td>
<td>56</td>
</tr>
</tbody>
</table>
INTRODUCTION

An application system based on a custom hardware design will typically perform faster and require less hardware than if it were implemented with “off the shelf” circuit boards. However, these advantages are countered by the disadvantages of custom designs, with one of the largest drawbacks being the custom software required. This software is often unique to the application and specific to the hardware design, requiring a significant and increasing percentage of the development schedule and expense. The cost is multiplied by the need for software tools, standards, and maintenance developed specifically for each application. In addition, much of the application software cannot be used for new applications or hardware. All of these disadvantages can be significantly reduced by using a modular, standardized operating system.

The operating system provides a higher level interface to the system hardware. The hardware characteristics common to most applications, such as memory management and interrupt handling, are handled by the operating system rather than the application software. The operating system provides scheduling and synchronization for multiple functions, allowing application code to be written in independent pieces or modules. The operating system interface can be more standardized than the interface to the hardware components. This allows the application software to be more independent of changing hardware. The application code can be initially implemented and debugged on proven hardware. The software is then easily moved to the final hardware configuration for testing.

The operating system interfaces allow the use of standard software tools, such as debuggers. Operating systems also provide decreased debugging time and increased reliability through error checking and error handling. Perhaps most important, the expertise gained can be carried on to new designs based on the operating system.

Operating systems have generally been described as large and complex, with rigid system requirements. Users have found it difficult to tailor a system to their needs or to use the operating system on more than one hardware configuration. System software has been accepted in large pieces or as a whole, with few system configuration choices in either hardware or software. Those systems small enough to use on component designs have lacked extendibility to larger, more complex designs.

The Intel iRMX 86 Operating System offers users of component hardware all benefits of operating systems while imposing few hardware restrictions. Minimum hardware requirements include 1.8K RAM memory, enough RAM or EPROM memory to hold the Nucleus and the application code, and a handful of integrated circuits. The circuits are an Intel iAPX 86 or iAPX 88 Central Processing Unit, an Intel 8284 Clock Generator, Intel 8282/83 Latches for bus address lines, an Intel 8253 Programmable Interval Timer, and an Intel 8259A Programmable Interrupt Controller. Larger system busses will also require an Intel 8288 Bus Controller and Intel 8286/87 Transceivers for data lines. This basic hardware system is shown in Figure 1.

Figure 1. Basic Hardware System for the iRMX 86™ Operating System
Users with a wide range of applications will find the iRMX 86 Operating System allows them to implement a corresponding range of capabilities, from a minimum iRMX 86 Nucleus to a high level human interface. A complete iRMX 86 Operating System includes extensive I/O capabilities, debugger, application loader, bootstrap loader, and integrated user functions. This flexibility allows one operating system to be used for many projects, minimizing software learning curves for new applications.

This note discusses a relatively small standalone spectrum analysis system based on a subset of the iRMX 86 Nucleus. The intent of the note is to demonstrate advantages of using operating systems in hardware component designs. An overview of operating system functions is given first as background information. Readers familiar with operating systems may wish to skip this section. The overview is followed by a summary of the iRMX 86 Operating System. The summary is brief, as only the iRMX 86 Nucleus is used in this application. A detailed discussion may be found in Application Note AP-86, “Using the iRMX 86 Operating System,” and iRMX 86 System Manuals.

The spectrum analysis system is described after the summary. The system requirements, design and implementation are detailed. The system software is discussed next, followed by configuration and hardware implementation. A summary completes the application note text. Partial code listings of the system software are included in the appendices.

OPERATING SYSTEMS FUNCTIONAL OVERVIEW

Operating system software manage initialization, resources, scheduling, synchronization, and protection of tasks or functions within the system, as well as providing facilities for maintenance, debugging, and growth. In general, operating systems support many of the following:

Multitasking

Multitasking allows functions within a job to be handled by separate tasks. This is particularly valuable when a job is responsible for multiple asynchronous events or activities. One task can be assigned to each event or activity. Tasks are the functional members of the system, executing within the bounds of a job environment. Program code for a multitasking system is modular, with well-defined interfaces and communication protocols. The modular boundaries serve several important purposes. The code for each module can be generated and tested independent of the other modules. In addition, the boundaries confine errors, speeding debugging and simplifying maintenance.

Growth

The modular independence that results from multiprogramming and multitasking gives users the ability to efficiently create new applications by adding functions to old software. Applications can be tailored to specific needs by integrating new modules with previously-written general support code. If care is taken in system design, functions can be added in the field. Documentation for the older software can be carried on to the new applications. This growth path will save completely rewriting expensive custom software for each new application.

Scheduling

Even though a system has multiple jobs and tasks, only one task is actually running on the central processor at any single point in time. Scheduling provides a means of predicting and controlling the selection of the running task from the tasks that are ready to run. Basic scheduling methods include preemptive priority, non-preemptive priority, time-slice, and round-robin. Batch systems often use non-preemptive priority scheduling, in which the highest priority job gets control of the central processor and runs to completion. Preemptive scheduling is typically used in real-time or event-driven systems, where dedicated, quick response is the main concern. A higher-priority waiting task that becomes ready to run will preempt the lower-priority running task. Priority may be either set at task creation (static) or modified during running of the task (dynamic).

Time-slice and round-robin scheduling are used in multiuser or multitask systems that share processing resources and have limits on maximum execution time. Time-slice scheduling gives each task or job a fixed slice of dedicated processor time. Round-robin gives each task or job a turn at using the processor. The time available during the turn depends on system load and task priority.
Communication and Synchronization

Jobs and tasks in a multiprogramming and multitasking environment require a structured means of communication. This communication may be necessary to synchronize processes or to pass data between processes. Two means of providing communication are mailboxes and semaphores. Mailboxes are an exchange place for system messages. The messages may include data or provide access to other system objects, including other mailboxes. Tasks can send objects to a mailbox or wait for objects from a mailbox. Generally, the task has the option of waiting for a specified period of time for the message. The wait time may range from zero for a task that requires immediate response to infinite time for a task that must have a message to continue processing. Multiple mailboxes are used to synchronize multiple tasks.

A communication flow using mailboxes is shown in Figure 2. In this example, the sending task sends a message to a mailbox and specifies a return mailbox. The sending task then waits at the return mailbox. The receiving task obtains the message from the sending mailbox and sends a message to the return mailbox. The first task obtains the second message from the return mailbox, synchronizing the two tasks and passing data.

![Figure 2. Intertask Communications with Mailboxes](image)

If process synchronization is the only requirement, semaphores may be used. Semaphores function like mailboxes except that no data is passed through the semaphore. Instead, semaphores contain "units," with the meaning of the units defined by the sending and receiving tasks. A one unit semaphore may be used as a flag to synchronize the tasks. Multiple unit semaphores can be used for resource control. For example, if tasks require reusable data buffers, a semaphore may be defined as the allocator of the available data buffers. Each unit in the semaphore will represent one available buffer. When a task requires buffers to continue, the task will wait at the semaphore until enough units (representing buffers) are available. The waiting task will receive the units, use the buffers, and return the units (still representing buffers) to the semaphore. Other tasks that require buffers will also have to wait at the semaphore until enough buffers are available.

Resource Management

The operating system is the central guardian of system resources, specifically read/write memory. The memory is made known to the Nucleus at initialization. The Nucleus then gives pieces or segments of the memory to tasks as they request it. This allows the tasks to have no initial knowledge of the actual location and size of system memory. Tasks can share memory if they desire, but the Nucleus allocates memory to each task individually, preventing the tasks from using each others memory. In addition, tasks return memory to the Nucleus when they are through with it, allowing memory to be reused.

Other system resources, such as I/O devices, will also be scheduled by the operating system. The operating system is responsible both for the efficient use of these resources and for providing the tasks with a large measure of independence from the actual I/O hardware requirements. The system may require many types of I/O devices, such as disk drives, tape drives, and printers. I/O is more efficiently accomplished if the operating system provides both asynchronous and synchronous I/O operations. Synchronous operations are those the task starts and waits for completion, doing no other work until the I/O is complete. Asynchronous operations are started by the task, but the actual I/O can take place while the task is doing other work. The overlapped operation of asynchronous I/O provides more user control of the I/O operation at the expense of a more complicated user interface.

Interrupt Management

Real-time software is tightly coupled to hardware functions by interrupts. Interrupts provide rapid notification that the hardware needs attention. The software must respond quickly without corrupting the system environment. System integrity is preserved by preempting the lower priority operating task, saving the task environment, processing the interrupt (including communicating the results to other tasks if necessary), restoring the environment of the operating task, and continuing. All of this must occur in an orderly and efficient manner. The interrupt management of the operating system is responsible for directly interacting with the system hardware that detects interrupts. The interrupt tasks can be ignorant of the detailed interrupt hardware, providing only the system actions to service the event that caused the interrupt.
Initialization

Operating systems create and manage jobs and tasks at initialization as well as run time. Initialization generally must be done in a specific sequence which will depend on the environment existing at that time. An abortive initialization environment may require an orderly shutdown of the system. The operating system has the capability for managing these situations, including communications, access to system resource information, and displaying status of the initialization actions.

Debugging

The system debugger is a window into the internal structures of the operating system. Debuggers allow data structures and memory to be examined, breakpoints to be set, and the user to be notified of abnormal conditions. The debugger may have symbolic debugging, in which system objects such as addresses, tasks, jobs, mailboxes, and memory locations can be assigned names. This gives greater flexibility and accuracy during debugging. The debugger may not be necessary in the final system, so the debugger is often a separate system job. This allows removal of the debugger with no effect on the remaining jobs.

Debugging will be aided if the operating system verifies the parameters required by system actions and also returns status for the requested action. Parameter verification is particularly valuable for new program code in a developing system. Status results other than success are abnormal or exception conditions. Exception conditions may include insufficient memory for the request, invalid input data, inoperative I/O devices, an invalid request for an action, or a request for an invalid action. The operating system may have an exception handler for these conditions and may allow the debugger to be used as the exception handler. The development process will be more efficient if detection of exception conditions takes place for all levels of system actions, from initialization of jobs and tasks to requests for memory or status.

Higher Level Functions

With the continuing increase in system complexity, more operating systems are providing higher level functions. These functions may include advanced I/O file management, operator console, spooling operations, telecommunications support, multiuser support and access to system resources of increasing size and complexity. Only the largest operating systems provide all of these capabilities, but users of component hardware must be careful their system will integrate higher level functions that may be required in future applications.

Extendibility

In order to provide general purpose support, operating systems must be extendible. New applications may require data structures or system actions not available with the present operating system. The system must be able to integrate these new structures and actions, supporting them in the same manner as existing functions. Choosing an operating system requires a large commitment, both in initial expense and system architecture. Extendibility provides assurance the operating system chosen will not provide built-in obsolescence of that commitment.

IRMX 86™ OPERATING SYSTEM ARCHITECTURE

Layers

The IRMX 86 Operating System architecture is shown in Figure 3. It includes the Nucleus, Basic I/O System, Extended I/O System, Applications Loader, and Human Interface. These major portions of the operating system are designed as layers. Each layer may be added to previous layers as application needs grow. Lower layers may be used without upper layers. All layers may reside in programmable read only memory. Applications have access to all portions of the system, from the Nucleus to all outer layers.

![Figure 3. Architecture of the IRMX 86™ Operating System](image)

The Nucleus is the heart of the system. It includes support for multiprogramming, multitasking, communications, synchronization, scheduling, resource management, extendibility, interrupt handling, and error detection. The Nucleus may be considered as an extended layer of the underlying hardware, giving the hardware system management functions and making the software independent of the detailed hardware. The system environment, including resources, priorities, and placement of program code, is made known to the Nucleus at system initialization. All requests for memory, communication, and creation of basic data structures must
go through the Nucleus. These requests are made by system calls, which are comparable to subroutine calls for system actions.

All higher level functions of the iRMX 86 Operating System are built around a core of the Nucleus. Although outer layers may require a substantial number of the system functions included in the Nucleus, the Nucleus itself is configurable on a call-by-call basis. "Configurable" means the Nucleus may be altered so it contains code only for those functions required by the application. Certain features, such as parameter validation and exception handling, are also configurable. Features and system calls may be included for development and excluded from the final system, giving a Nucleus tailored for each level of application development.

The Basic I/O System is the first layer above the Nucleus. The Basic I/O System provides asynchronous I/O support and format independent manipulation of data. Multiple file types are supported, including Stream, Named, and Physical files. Stream files are internal files for transferring large amounts of data between jobs or tasks. Named files include data files of varying sizes and directories for those files. Named files are designed for random access disk storage. Physical files consider the entire device to be one file. Physical files are primarily used to transfer data to and from printers, tape drivers, and terminals. Device drivers for both floppy and hard disks are provided. Like the Nucleus, the Basic I/O System provides system calls to invoke I/O actions. These calls and the features of the Basic I/O System are fully configurable.

The Application Loader provides the ability to load code and data from mass storage devices into system RAM memory. The Application Loader resides on the Basic I/O System, allowing application code to be loaded from any random access device supported by the Basic I/O System. Application code can be loaded and executed as needed rather than residing in dedicated system memory.

The Extended I/O System supports synchronous I/O, automatic buffering, and logical names. Synchronous I/O provides a simplified user interface for I/O actions. Automatic buffering improves I/O efficiency by overlapping I/O and application operations wherever possible. Logical name support allows applications to access files with a user-selected name, aiding I/O device independence.

The Human Interface uses all lower layers, forming a high level man-machine interface for user program invocation, command parsing, and file utilities. The primary purpose of the Human Interface is to support the addition of interactive commands. The Human Interface is the basis for pass-through language support and multiple user systems.

Configurability

Configurability means the iRMX 86 Operating System can be changed to include only the system calls and features pertinent to the system under development. Smaller applications start with only the iRMX 86 Nucleus. A subset of Nucleus calls, described later in this application note, provide the basis for management of jobs, tasks, memory, interrupts, communication and synchronization, and support for the Debugger and Terminal Handler.

All systems calls may also use parameter validation as a configuration option. Parameter validation verifies that system calls reference correct system objects before the requested action is performed. During debugging and in hostile environments, validation provides error detection for each system call. This error detection does add some overhead to the calls. Debugged application jobs can perform more efficiently without the validation, while new code can use parameter validation to speed development.

Once errors are detected, there are two means available to handle error recovery. The task can either use the status information to perform error recovery actions or the recovery actions may be performed by a specialized error handling program called an exception handler. Applications may use the Debugger as an exception handler, or use one implemented by the application. There are two classes of errors that may cause control to be given to an exception handler; avoidable errors, such as programmer errors, and unavoidable errors, such as insufficient memory. Exception handlers can be selected to receive control for either or both classes.

Support Functions

The iRMX 86 Operating System includes a Debugger, a Terminal Handler, a Bootstrap Loader, and a Patch Facility. The Debugger examines system objects, using execution and exchange (mailbox and semaphore) breakpoints, symbolic debugging, and exception handling. The Terminal Handler provides a line-editing, mailbox-driven CRT communications capability. The Bootstrap Loader is a fully configurable loader for bootstrap loading on reset or command, from any specific random access device. The Patch Facility gives the capability of patching iRMX 86 Object Code in the field.

Object Orientation

The iRMX 86 Operating System is based on a set of system data structures called objects. These objects include jobs, tasks, mailboxes, semaphores, segments, and regions. Users may also define application-specific objects. Object architecture includes the objects, their parameters, and the functions allowed with the objects.
Object orientation is a formal, hardware-independent definition of hardware-dependent system structures that are currently used by most applications. For example, without object orientation, memory is reserved in advance for system buffers. The application code knows buffer sizes and locations. If buffer requirements grow, requiring a new memory layout, much of the application code will change to accommodate the new buffer sizes and locations. Using object orientation, the application requests a segment (buffer) of a particular size when the buffer is needed. The Nucleus allocates the memory and returns a segment object to the application. If the application needs a larger buffer, it returns the old segment and requests a new one of a larger size. The application obtains more buffers by making requests for more segments. If the hardware changes, the iRMX 86 Operating System is made aware of the changes. The application code uses the same system calls to request and return the segment objects regardless of the hardware configuration.

Objects are provided for modular environments (jobs), application code functions (tasks), communication (mailboxes), synchronization (semaphores), memory (segments), and mutual exclusion (regions). Objects are fundamentally a set of standard interfaces between application code and the iRMX 86 Operating System. The standard interfaces have three primary benefits:

1) First, objects provide structures, such as tasks or segments, that are common to all applications. The structures form the basis for a standard set of system calls that make the interface between the application and the operating system more consistent and easier to learn. These calls allow applications to create more objects (segments for buffers, for example), delete them, change them, and inspect them. Development engineers can use their knowledge of the objects on many applications, rather than just the one under development. The common objects allow a common system debugger to be used. The debugger will work for all applications, letting engineers concentrate their efforts on the application itself rather than designing and implementing custom debugging tools.

2) Second, the standard characteristics of the object allow consistent error detection and handling. Requests to alter or use objects can be checked for validity before the Nucleus actually performs the request. Errors can be classed as common to all objects or specific to certain objects, giving more precise error information for effective error handling and faster debugging.

3) Third, the object interface will be preserved on future releases of the iRMX 86 Operating System. Current application code can be split into independent modules. Future applications can use the modules for common functions, preserving the investment in application software.

Task Scheduling

The Nucleus controls task scheduling by task priority and task state. Task priority is specified when the task is created. The priority can be altered dynamically. Tasks are classified into one of five classes: Running, Ready, Asleep, Suspended, or Asleep-Suspended. Tasks that have not been created are considered to be non-existent. The State Transition Diagram is shown in Figure 4.

![State Transition Diagram](image)

Only one task is in the Running state. This task has control of the central processor. The Ready state is occupied by those tasks that are ready to run but have lower priority than the Running task. The Asleep state is occupied by tasks waiting for a message, semaphore units, availability of a requested resource, an interrupt, or for a requested amount of time to elapse. A task can specify the amount of time it will allow itself to spend in the Asleep state, but tasks in the Suspended state must be “resumed” by other tasks. The Suspended state is useful when situations require firm scheduling beyond the control provided by priority and system resource availability. Examples of these situations are system emergencies, controlling tasks in the Ready state for application-dependent scheduling algorithms, and guaranteeing a fixed initialization or shut-down sequence. If another task “suspends” a task already in the Asleep state, the sleeping task goes to the Asleep-Suspended state. This task will enter the Suspended state if the sleep-causing condition is satisfied. The task will go to the Asleep state from the Asleep-Suspended state if it is resumed before the sleep-causing condition is removed. If a task enters the Ready state and has higher priority than the present Running task, the Ready task is given control of the CPU. Control is transferred to another task only when:

1) The Running task makes a request that cannot immediately be filled. The Running task is moved to the
Asleep state. The highest-priority Ready task becomes the Running task.

2) An interrupt occurs, causing a higher-priority task to become Ready. The current Running task goes to the Ready state, allowing the higher-priority task to become the Running task.

3) The Running task causes a higher-priority task to become Ready by releasing the resource for which the higher-priority task is waiting. The current Running task goes to the Ready state. The higher-priority task becomes the Running task.

4) The Running task causes a higher-priority task to become Ready by sending a message or semaphore units to the mailbox or semaphore where the higher-priority task is waiting. The Running task is moved to the Ready state. The higher-priority task becomes the new Running task.

5) The Running task removes a higher-priority task from the Suspended state by “resuming” it, placing the higher-priority task in the Ready state. The current Running task is moved to the Ready state and the higher-priority Ready task becomes the new Running task.

6) The Running task creates a higher-priority task. The new task goes to the Ready state. The current Running task is moved to the Ready state and the higher-priority Ready task becomes the new Running task.

7) The Running task places itself in the Suspended state. The highest-priority Ready task becomes the new Running task.

8) The Running task places itself in the Asleep state. The highest-priority Ready task becomes the new Running Task.

9) The Running task deletes itself, becoming Non-existent. The highest-priority Ready task will be the new Running task.

5) The following RAM:
   a. 1024 bytes from 0 to 1024 for software interrupt pointers (the interrupt vector).
   b. 800 bytes for Nucleus data.
   c. Enough RAM for the application data, code, and system objects.

6) Enough EPROM or RAM to hold the required parts of the iRMX 86 Operating System and the application code.

The Intel iSBC 86/12A Single Board Computer more than meets these minimum requirements. A block diagram of the board is shown in Figure 5. Note in addition to the timer and interrupt controller the board contains an 8251A USART, an 8255 parallel I/O interface, a MULTIBUS™ interface, four sockets for up to 16K bytes of EPROM, and 32K bytes of dual-ported RAM. Even though a user may be developing a custom board for his application, it is recommended that initial system development be accomplished using the iSBC 86/12A Single Board Computer. This will provide a known hardware environment to simplify debugging. In addition, the development hardware system can be adapted to changing application needs by adding Intel MULTIBUS compatible boards to the iSBC 86/12A Single Board Computer. After the software is fully debugged, the application can be moved to the final custom hardware design.

APPLICATION EXAMPLE

A spectrum analyzer is the subject of this application. The analyzer displays the frequency spectrum of an analog signal on a general purpose CRT terminal. The user has control over input signal bandwidth, averaging, and continuous analysis. A fast Fourier transform (FFT) program is used to obtain frequency data from samples of analog data. Fourier transforms provide useful frequency analysis, but the large processing requirements of Fourier transforms have restricted their use. Fast Fourier transforms take advantage of the repetitive nature of the Fourier calculations, allowing the Fourier transforms to be completed significantly faster. The FFT used in this application note is known as “time decomposition with input bit reversal.” Sixteen-bit integer samples of the input signal are placed in frames, with each frame holding 128 complex points. An averaged power spectrum is calculated to sum and square the FFT values, yielding 64 32-bit power spectrum values. These values are displayed on a standard CRT terminal.

The FFT algorithm may be applied wherever frequency analysis of an analog signal is required. Medical applications for FFTs include EEG analysis, blood flow analysis, and analysis of other low-frequency body signals. Industrial uses are production line testing, wear analysis, frequency signature monitoring, analysis in noisy or hostile environments, and vibration analysis. Other applications could cover remote reduction of analog data, frequency correlation, and process control. For this application note, the actual use of the FFT is secondary to its existence as a modular, CPU-intensive task in a general purpose system.

The overall application system characteristics are the following:

1) A user-selectable input signal bandwidth of 120 Hz, 600 Hz, 1200 Hz, 6000 Hz, or 12,000 Hz.

2) The option of averaging frames of samples. The averaging is user selectable, with options of 1 (no averaging), 2, 4, 8, 16, or 32 frames averaged per CRT display.

3) The capability, also user selectable, of repeating the analysis cycle continuously.

4) User capability to abort the analysis.

5) Twelve-bit input sample resolution.

6) A minimum of hardware requirements, including no more than 32K bytes of EPROM memory and 16K bytes of RAM memory.

7) A standard character screen CRT for output.

8) A multitasking structured design that will use a subset of the iRMX 86 Nucleus and exhibit modular application code, formal interfaces, and self-documentation.

System Design

To begin the design, the application is broken up into functional modules, much the same as a hardware block diagram. A SUPERVISOR TASK initializes the system, accepts operator parameters, starts the analysis cycle, and stops the processing upon cycle completion or operator request. An INPUT TASK samples the data and places it in a buffer. An FFT TASK receives the buffer and processes the data. An OUTPUT TASK displays the data received from the FFT TASK. This structure is shown in Figure 6.
The general task functions are:

**SUPERVISOR TASK**: The SUPERVISOR TASK initializes the system by creating the other tasks. The SUPERVISOR TASK then obtains the analysis parameters from the operator. Each parameter is verified as it is received. When all of the parameters are accepted, the operator is asked if they are satisfactory. If the operator agrees, the SUPERVISOR TASK sends frame buffers to the INPUT TASK to initialize the analysis. If not, the operator is asked to input all of the parameters again. During the FFT analysis, the SUPERVISOR TASK waits for an abort request from the operator and for the frame buffer from the OUTPUT TASK. If the abort request is received, the SUPERVISOR TASK terminates the analysis in an orderly fashion and asks the operator for parameters for the next analysis cycle. If the frame buffer is received from the OUTPUT TASK and continuous analysis has been selected, the SUPERVISOR TASK sends the frame buffer to the INPUT TASK to start the next cycle. If the frame buffer is received from the OUTPUT TASK and continuous analysis has not been selected, the current analysis is complete and the SUPERVISOR TASK asks for new parameters.

**INPUT TASK**: The INPUT TASK receives the frame buffer from the SUPERVISOR TASK. The input signal is sampled according to the analysis parameters. The actual sample rates are calculated as follows:

1) Multiply the highest frequency of interest by two to obtain the Nyquist sampling rate.
2) Invert this value to obtain time between samples.
3) Scale the value by 60/64. The CRT display is limited to 64 columns. The scaling maps sample values to columns 1 to 60 rather than 1 to 64, giving a more readable x-axis label and display. This method yields sample times of 3.9 milliseconds, 781 microseconds, 390 microseconds, 78 microseconds, and 39 microseconds for frequencies of 120 Hz, 600 Hz, 1200 Hz, 6000 Hz, and 12,000 Hz.

The INPUT TASK samples the data at the required interval and places the samples in the frame buffer. When the frame buffer is full, the INPUT TASK updates the frame buffer number and sends the frame buffer to the FFT TASK. The INPUT TASK sends a status message to the CRT terminal and waits for the next frame buffer.

**FFT TASK**: The FFT TASK receives the frame buffer from the INPUT TASK. A fast Fourier transform is performed on the data contained in the buffer, the power spectrum is calculated, and the data is averaged with previous data if necessary. If the frame buffer is the last one to be averaged prior to display of the frequency data, the frame buffer is filled with the averaged data and sent to the OUTPUT TASK. If the buffer is not the last one to be averaged, the FFT TASK returns the buffer to the INPUT TASK for another frame of data or to the SUPERVISOR TASK if the analysis cycle is nearly complete. The FFT TASK sends status information to the CRT display and waits for the next frame buffer from the INPUT TASK.

**OUTPUT TASK**: The OUTPUT TASK receives the frame buffer from the FFT TASK. The data is formatted and displayed. The OUTPUT TASK sends the frame buffer to the SUPERVISOR TASK and waits for the next frame buffer from the FFT TASK.

**Terminal Handler**: The Terminal Handler serves as the basic I/O device for parameter requests, status data, and frequency displays. The Terminal Handler accepts display requests from all tasks and sends operator input to the SUPERVISOR TASK.

The basic functions of the various tasks in the application have been defined, but system integration has not been discussed. Synchronization of the tasks, scheduling, resource management, mapping to hardware, interrupt handling, and system interfaces have been omitted. No debugging functions have been defined. It is clear the system implementation is just started. The iRMX 86 Nucleus will provide all of the system integration "glue" the application requires, allowing application programmers to concentrate on the actual functional code. In order to use this "glue," the application must be divided into jobs and tasks.

**Jobs and Tasks**

The iRMX 86 Operating System architecture defines jobs as separate environments within which tasks operate. These separate environments allow each job to function with no knowledge of other system jobs. There are two jobs in this application, the Debugger job and the Application job.

The jobs contain functional portions or working programs called tasks. The Application job contains the INIT TASK, SUPERVISOR TASK, INPUT TASK, FFT TASK, OUTPUT TASK, and INTERRUPT TASK. The Debugger job contains the Debugger task and the Terminal Handler task. Tasks provide the ap-
Application goals of modularity, resource constraint boundaries, and functional independence. The structure of the development system is shown in Figure 7.

![Figure 7. Development System Job Structure](image)

The Debugger job is included only for development. When development is completed, the Debugger job is removed from the system. A Terminal Handler job containing only the Terminal Handler task is substituted in place of the Debugger job. The application code is not changed. This structure is shown in Figure 8.

![Figure 8. Final System Job Structure](image)

**Interfaces and Synchronization**

Now that the system is made of jobs and tasks, the primary need is to synchronize the tasks and provide communication interfaces. This will be handled by mailboxes. The messages sent via the mailboxes will be segments, which are pieces of memory allocated by the Nucleus. The frame buffers sent from task to task are these segments. The buffer segments contain all the analysis parameters in addition to the data samples. Communication with the Terminal Handler task is also accomplished by mailboxes, but with a different buffer format. The Terminal Handler format has fields for the operation requested (read or write), the number of characters the Terminal Handler did read or write, a status field for the operation, and the actual data. The buffer format is shown in Figure 9. Figure 12 contains the Terminal Handler segment format.

![Figure 9. Frame Buffer Format](image)

Mailboxes are used to pass the buffer segment from task to task. Tasks can send segments to mailboxes, or receive segments from mailboxes. If there is no segment at a mailbox, a task can elect to wait for the segment, with the wait duration ranging from zero to forever. This provides the simplest system synchronization — each task, upon initialization, waits at its input mailbox for a frame buffer segment. When the task receives the segment, it processes the data, sends the segment on to the mailbox for the next task, and returns to its own mailbox to wait for the next frame buffer. The system is synchronized and controlled by the availability of frame buffers and task priority. It should be clear that multiple frame buffers segments provide overlapped processing, with the segments ultimately "piling up" at the slowest task in the chain. This loose coupling arrangement allows tasks to have radically different execution times. For example, the INPUT TASK has an input sampling time that ranges from 0.6 seconds to 6.3 milliseconds, a range of 100 to 1, and the system requires no special synchronization or scheduling to accommodate this range.

The mailbox interfaces, shown in Figure 10, serve several other important purposes. First, they provide the standardized interface that is a goal of this application. The set of mailboxes and the two buffer segments form all inter-task interfaces. Each task uses only a few mailboxes, making it easy to add or remove tasks by adding or removing mailboxes. The system could easily be expanded to include data reduction tasks, data correlation tasks, or to substitute different tasks for any of the present ones. Dummy tasks were used for real tasks during development to verify overall system execution before the actual tasks were available.
The mailboxes also provide a very convenient window into the application system processing for both debugging and aborting the current cycle. The Debugger can set breakpoints at mailboxes to allow users to examine the frame buffers as they progress from task to task. The Debugger can examine buffer data and control the processing cycle. Tasks wait at the mailboxes in a queue that is either priority or first-in-first-out (FIFO) based. The inter-task mailbox queues are priority based, which means the higher priority SUPERVISOR TASK can intercept segments at the mailboxes ahead of the lower priority waiting tasks, and abort the analysis by removing all of the buffer segments. This method of aborting requires no knowledge of the internal processing of the tasks, making it universally applicable to all the tasks.

A return mailbox may be specified when a segment is sent to a mailbox. The receiving task may send status information, a different segment, or the original segment to the return mailbox. The Terminal Handler will return the buffer segment sent to it if a return mailbox is specified. This is used to synchronize the tasks with the Terminal Handler and to allow multiple tasks to use a single display task. Each sending task waits at a separate return mailbox for the Terminal Handler to return its segment. Each task retains control over its buffer segment and is synchronized with the slower data display function.

**Priority**

In addition to the mailboxes, task execution is governed by priority. In this system, the INPUT TASK has maximum priority to guarantee it can sample the input signal at the precise intervals required for the FFT. The SUPERVISOR TASK, responsible for abort functions, has the next level of priority, with the FFT TASK next, and the OUTPUT TASK lowest.

**APPLICATION IMPLEMENTATION**

**Display Functions**

All application tasks use the iRMX 86 Terminal Handler as an output device. The Terminal Handler is chosen because it provides a standard interface consistent with the application goals, it exists both in the Debugger job and in an independent job, and it is easy to integrate into the application system. The application could also use the same interface with a user-written Terminal Handler. In this application, the Terminal Handler can have messages from four tasks on the screen at one time. To allow this to occur in an orderly fashion, lines on the screens are reserved for each of the tasks. The screen format is shown in Figure 11. Each message to the screen sends the cursor to the upper left corner (the home position), then down to the proper line to display the data.
Cataloging

To aid the debugging process, all system objects, such as mailboxes, segments, and tasks, are cataloged in the directory of the SUPERVISOR TASK. The catalog entries are user-selected, 12-character names. The Debugger can display this directory, giving easy access to objects to aid symbolic debugging. If other tasks know the proper directory and the 12-character name, the tasks can look up the objects in the directory and obtain access to them. This is the method used to find the Terminal Handler mailboxes. For objects that are cataloged only to aid debugging, the system calls that catalog the objects are removed from the code when debugging is complete.

Application Code

The code listings for the SUPERVISOR TASK and the INPUT TASK are included in appendices to this note. Code listings for other tasks are not included, but they are available from the Intel Insite software library. The following discussions reference line numbers in the listings included in this note. The references begin with a first letter for the appendix section (A or B), followed by the actual line number (A.220, for example).

SUPERVISOR TASK

Code listings for the SUPERVISOR TASK are in Appendix A. The actual SUPERVISOR TASK procedure begins at line A.550. After initializing internal buffers and mailboxes (A.551, A.518-A.549), the Supervisor sends an initial screen, one line at a time, to the Terminal Handler (A.553, A.502-A.517). When the screen is complete, the SUPERVISOR TASK creates the INPUT TASK, FFT TASK, and OUTPUT TASK (A.554, A.486-A.501). The order of creation is not important for this application, as each task begins by waiting at its input mailbox for frame buffer segments. The SUPERVISOR TASK requests input parameters from the operator (A.556, A.305-A.485). The actual input parameter loop is found at A.478. The loop consists of asking questions (A.479, A.480) until all answers are satisfactory. The operator is asked to choose the highest frequency of interest, number of frames to average, and single runs or continuous runs (A.331-A.353). If the operator answers with an invalid input, the question is repeated (A.365 and A.415). If the operator wishes to abort the questioning by entering a 99, the questions start over from the first question (A.409-A.413). When all three questions have been answered, the operator is asked to confirm his choice (A.482, A.418-A.467). If the operator does not verify the answers, the question number is set to 0 (A.463) and the parameters are requested again. If he confirms the answers as correct, the SUPERVISOR TASK creates up to three frame buffer segments (A.557, A.279-A.304). The SUPERVISOR TASK places the binary equivalents of the operator answers in the frame segments (A.284, A.292, A.300, A.269-A.278), and sends the segments to the input mailbox for the INPUT TASK (A.287, A.294, A.302).

The SUPERVISOR TASK's background duties are to check its input mailbox for a segment from the OUTPUT TASK and to check a return mailbox for an abort request from the operator (A.558, A.225-A.268). If segments are received at the SUPERVISOR TASK mailbox, the SUPERVISOR TASK sends the segments on to the INPUT TASK if the operator has chosen continuous runs (A.258). Otherwise, the SUPERVISOR TASK deletes the segments (A.242-A.250). When all the segments have been deleted, which halts the FFT analysis, the SUPERVISOR TASK asks for operator input again (A.556).

If an operator abort request is received, the SUPERVISOR TASK, having higher priority than the FFT or OUTPUT TASKS, waits at their mailboxes to intercept the frame segments (A.243, A.249, A.207-A.224). When a segment is received it is deleted (A.219). The SUPERVISOR TASK also checks the INPUT TASK mailbox under abort conditions (A.244). This mailbox is FIFO based to allow the SUPERVISOR TASK to intercept the buffer segment ahead of the higher-priority INPUT TASK (A.523). The SUPERVISOR TASK input mailbox is also checked for frame buffer segments that may have been sent there by the OUTPUT TASK after the abort was requested (A.246). When all of the frame buffer segments have been deleted, the SUPERVISOR TASK asks for operator input (A.556).

INPUT TASK

Listings of the INPUT TASK are in Appendix B. After initializing the buffer for Terminal Handler communications and the mailboxes for communicating with the INTERRUPT TASK and the Terminal Handler (B.335, B.302-B.333), the INPUT TASK waits at its input mailbox for a frame buffer segment (B.338).

When a frame segment is received, the INPUT TASK updates the frame number counter kept by the INPUT TASK (B.340, B.289-B.301), and samples the analog input (B.341, B.231-B.288). The INPUT TASK selects one of two input driver routines, either a software polling loop for faster sampling rates (B.277-B.281), or an INTERRUPT TASK for slower sampling rates (B.251-B.276). If the sampling is driven by interrupts and a Nucleus system call is executing at the time of the interrupt, the time required to respond to that interrupt can vary from 100 to 350 microseconds, depending on the Nucleus call in progress. For the sample rates of 391, 78, and 39 microseconds, corresponding to bandwidths of 1200, 6000, and 12,000 Hz, the system interrupt latency cannot guarantee the precise sampling interval.
required. A simple software polling loop with a delay between samples is used for these rates (assembler code for this loop is included after the INPUT TASK listings in Appendix B). This loop operates at priority 0, the highest priority, to guarantee the loop is not interrupted (B.278, B.280) while the sampling is in progress.

For the longer intervals of 3.9 milliseconds and 781 microseconds, corresponding to bandwidths of 120 and 600 Hz, an Interrupt Handler and INTERRUPT TASK are used (B.251-B.276). Under the iRMX 86 Operating System architecture, an Interrupt Handler is defined as a short procedure with a primary goal of fast interrupt response and limited Nucleus calls. All hardware interrupt levels are masked when an Interrupt Handler is responding to an interrupt. If the interrupt servicing requires higher-level system functions, the Interrupt Handler notifies a waiting INTERRUPT TASK. Higher-level interrupts are enabled when an INTERRUPT TASK is executing. INTERRUPT TASKS can make all system calls.

The INTERRUPT TASK (B.196-B.203) binds the Interrupt Handler to the hardware interrupt level (B.197) and waits for a signal from the Interrupt Handler (B.199). The Interrupt Handler (B.164-B.195) verifies the interval accuracy (B.166-B.173), samples the data (which automatically starts the next sample) (B.175-B.176), places the data in the frame buffer (B.181-B.184), and notifies the INTERRUPT TASK when the frame buffer is full (B.193). If the buffer is not full, the Interrupt Handler resets the interrupt hardware (B.194). The INTERRUPT TASK notifies the INPUT TASK (B.200) and waits for a return message (B.201). The INPUT TASK disables interrupt level 3 (B.274) and returns the token to the INTERRUPT TASK (B.275). The INTERRUPT TASK enables the Interrupt Handler (B.199), but no interrupts will be received from the freerunning 8253 Timer because hardware interrupt level 3 has been disabled. Sampling for the next buffer is initiated by simply enabling level 3 (B.272). The INPUT TASK sends a status message to the Terminal Handler (B.342, B.219-B.230) and sends the filled frame buffer to the FFT TASK (B.343). The INPUT TASK then returns to the INPUT TASK input mailbox to wait for the next frame segment (B.338).

**FFT TASK**

Listings for the FFT TASK are not included with this application note. The FFT TASK is similar in overall format to the INPUT TASK. The FFT TASK waits at its input mailbox for a frame buffer segment from the INPUT TASK. When one is received, the FFT TASK computes the fast Fourier transform of the data. The auto power spectrum is computed and averaged with previous data. The FFT TASK sends its status message to the Terminal Handler for display. If the frame buffer is the final one to be averaged, the FFT TASK sends the frame buffer to the OUTPUT TASK. If the frame buffer is not the final one in this averaging series, the FFT TASK checks to see if the sampling process is continuous. If so, the frame buffer is returned to the INPUT TASK. If the sampling process is not continuous and the buffer is within two frames of the final frame buffer, the buffer is returned to the SUPERVISOR TASK to prevent unnecessary buffers from going to the INPUT TASK. The FFT TASK then returns to its input mailbox to wait for the next frame buffer.

**OUTPUT TASK**

Listings for the OUTPUT TASK are not included in this application note. The OUTPUT TASK, like the other tasks, waits at its input mailbox for a buffer. When a frame buffer is received from the FFT TASK, the OUTPUT TASK stores the data in an internal buffer and sends the frame buffer to the SUPERVISOR TASK.

The OUTPUT TASK converts each 32-bit frame buffer value to one of 16 levels by taking the base 2 logarithm of the significant 16 bits of sample value. The display screen is sent to the Terminal Handler one line at a time. Each line of the display is composed of 7 characters of label and y-axis data and 64 characters of display data (reference Figure 11). Each line of the display represents a power of two (from 16 down to 1). The character to be displayed at each location is found by comparing the appropriate sample value against the current line value. If the sample value is greater than the line number, a pound sign is displayed at that location. Otherwise, a space is displayed. The x-axis and labels are sent after the data lines to complete the display. The OUTPUT TASK then waits at its input mailbox for another frame buffer.

**TERMINAL HANDLER**

The Terminal Handler interfaces to the application tasks via the two mailboxes and buffer segment format shown in Figure 12. If a task wishes to display data, a segment containing the data is sent to the RQSTHS-NORMSOUT mailbox, specifying a return mailbox. The Terminal Handler displays the data, updates the status fields, and sends the segment to the return mailbox.

Input proceeds in much the same fashion. A task requesting data sends a segment to the RQSTHS-NORMSIN mailbox, again specifying a return mailbox. When the operator terminates the input line with a carriage return, the Terminal Handler puts the data in the segment, updates the status fields, and sends the segment to the return mailbox. This serves two primary purposes: specifying return mailboxes allows multiple tasks to share the display screen while retaining
synchronization and control over their data buffers; and
a user-written Terminal Handler using the same pro­
tocol and mailbox names could easily be integrated into
the application. For this application, the INPUT TASK,
FFT TASK, OUTPUT TASK, and SUPERVISOR
TASK all share the screen for output, but only the
SUPERVISOR TASK uses the Terminal Handler to ob­
tain operator input.

Nucleus Calls

The iRMX 86 Nucleus provides a comprehensive set of
61 system calls. A complete description of these calls
may be found in the iRMX 86 Nucleus Reference
Manual. For most applications, only a subset of the 61
calls will be required. The iRMX 86 Nucleus is configur­
able, which means the final system Nucleus will contain
code only for the system calls required for the applica­
tion. In this case, the following system calls were
required:

RQ$CREATE$MAILBOX, RQ$SEND$MESSAGE,
and RQ$RECEIVE$MESSAGE provide mailbox man­
agement.

RQ$CREATE$SEGMENT and RQ$DELETE$SEG­
MENT are used to create and delete segments for the
frame buffers, internal buffers, and Terminal Handler.

RQ$SET$INTERRUPT, RQ$EXIT$INTERRUPT,
RQ$SIG$INTERRUPT, RQ$WAIT$INTERRUPT,
RQ$ENABLE, and RQ$DISABLE allow the
INPUT TASK to handle hardware interrupts knowing
only the hardware interrupt level (3).

RQ$CREATE$JOB, RQ$CREATE$TASK, and
RQ$SET$PRIORITY are used to create the jobs and
tasks, and set the priority of the input polling loop.

RQ$GET$TASK$TOKENS, RQ$LOOK$UP$OBJECT,
RQ$CAT$ALOG$OBJECT, RQ$DISABLE$DELE­
FIGURE 12. Terminal Handler Interface

TION, RQ$ENABLE$DELETION, RQ$GET$TYPE,
RQ$GET$PRIORITY, RQ$GET$SIZE, and RQ$SIG­
NAL$EXCEPTION are system calls required by the
Debugger and the Terminal Handler. None of these
calls are necessary in this application if a user-written
Terminal Handler is used and debugging is completed.

System Configuration

System Configuration is the integration step in the
development process. It consists of selecting the por­
tions of the iRMX 86 Operating System required in the
application, mapping this code and the application code
to system memory, and creating a Root Job that will ini­
tialize the system. The overall configuration process is
shown in Figure 13. Configuration requires knowledge
of available memory, operating system and application
code entry points, priorities, exception handlers, and
other system parameters. System Configuration consists
of the following steps:

1) Selecting the portions of the iRMX 86 Operating
System required by the application, including the
layers and the specific system calls in each layer.

2) Linking and locating those portions.

3) Assembling or compiling, linking, and locating the
application code.

4) Creating a configuration file that will tell the Nucleus
the locations of available RAM memory, initial char­
acteristics of each system job, and pertinent overall
system parameters. Each job in the system has an en­
try in the configuration file. The order of the entries
is the order of initialization of the jobs.

5) Creating the Root Job by assembling, linking, and
locating the configuration file.
During development of an EPROM-based application such as this one, configuration is accomplished twice: once for the RAM-based development system and once for the final EPROM-based system. These configurations are detailed in Appendix C, System Configuration. In both cases, the Root Job that results from configuration initializes the system jobs. For development, the system job structure is shown in Figure 7. The Root Job creates the Debugger Task in the Debugger Job, which in turn creates the Terminal Handler Task. The Root Job then creates the SUPERVISOR TASK, which creates the INPUT TASK, FFT TASK, and OUTPUT TASK. The INPUT TASK creates the INTERRUPT TASK when necessary.

Software development is completed on the iSBC 86/12A Single Board Computer discussed earlier in this note. After application code is debugged and ready to be placed in EPROM memory, the Debugger Job, which contains both the Debugger and the Terminal Handler, is removed and replaced with the Terminal Handler Job, which contains only the Terminal Handler. This job structure is shown in Figure 8. The Nucleus system calls required only by the Debugger are removed from the iRMX 86 Nucleus. The application code is not changed. The application code and the iRMX 86 Nucleus is configured for the final system, put in EPROM memory, and tested on the final hardware system. The final Nucleus and application code required 30.5K bytes of EPROM, allowing room for future code changes and some expansion within the 32K system limit.

The final application hardware is shown in Figure 14. This system contains an iAPX 86/10 CPU, an 8259A Programmable Interrupt Controller, and an 8253 Programmable Timer. The three chips form the primary hardware requirements for the iRMX 86 Operating System. The system is assembled from Intel components, using standard support circuits and system schematics described in Intel documentation. The analog sampling circuitry is a 12-bit analog to digital converter (ADC) and a sample/hold circuit. Both the sample/hold circuit and the ADC are driven from the on-board local bus. The ADC has a conversion time of 35 microseconds, limiting the overall cycle to approximately 39 microseconds per sample, or a maximum CRT display bandwidth of 12,000 hz.

The hardware system shown in Figure 14 contains components not specifically required for the final configuration. The 8255A Programmable Peripheral Interface and the MULTIBUS multimaster interface are not necessary for a system limited to just spectrum analysis and display via the CRT. However, the flexibility advantages of the iRMX 86 Operating System are supported by this hardware. For instance, the frequency spectrum display is limited by the CRT to a 16-level logarithmic approximation. Accuracy could be improved by using the programmable peripheral interface to drive a plotter or an analog CRT via a digital to analog converter. Software drivers for the plotter or CRT could be new tasks, interfacing to the old tasks through the mailboxes. Or, the OUTPUT TASK could be simply replaced with a new OUTPUT TASK for the plotter and analog CRT. The inclusion of the MULTIBUS interface allows this application to be integrated into a larger system of MULTIBUS-compatible boards. MULTIBUS-compatible memory boards will also aid test and debug. Users of hardware components can include these modular Intel interfaces as required by their application, giving growth and configurability in both hardware and software.
SUMMARY

This application note discussed general operating system functions, the Intel iRMX 86 Operating System, using the iRMX 86 Operating System on hardware component systems, and an example of an application implemented in a component environment. Users of the iRMX 86 Operating System are able to simplify application code development through modularity, standard interfaces, freedom from rigid hardware restrictions, and advanced debugging techniques. The iRMX 86 Operating System can be applied to larger systems by adding other iRMX 86 layers, making the software investment beneficial over a wide range of applications.

Operating systems provide many advantages for hardware component designs, but all of these benefits can be utilized only if the operating system and the development environment are fully supported. Intel's support for this application begins with an Intel MDS 230 Series II Microcomputer Development System. The MDS 230 System interfaces with the development hardware through the iSBC 957A™ Monitor. The development hardware is an Intel iSBC 86/12A Single Board Computer, an Intel iSBC 711™ Analog Input Board, an Intel iSBC 032™ 32K RAM Memory Board, an Intel iSBC 064™ 64K RAM Memory Board, and an Intel iSBC 660™ System Chassis. Final application hardware is debugged using Intel's ICE-86™ 8086 In-Circuit Emulator. Software support is provided by the ISIS-II PL/M 86™ Compiler, MCS-86™ Macro Assembler, and the MCS-86 Utilities LINK86, LOC86, and OH86. The Intel UPP-103™ Universal PROM Programmer is used to convert the final system to PROM memory. This broad support allows expedient development of prototype and final systems based on the iRMX 86 Operating System.

The iRMX 86 Operating Systems and the Intel development tools are valuable only if they translate directly to increased productivity and shortened time to market for new products. This application has 1567 lines of application code. It was developed, from design to final implementation, in approximately 9 man-weeks of effort. This high level of productivity was achieved with the added benefits of modularity, standardization, and ease of application growth.

Intel Corporation is committed to both the continued integration of higher-level functions into hardware and to maintaining compatibility of present software with new hardware. One result of these commitments will be the Intel iAPX 86 and iAPX 286 Processors, which will be compatible with the iRMX 86 Operating System. Another result will be the placement of the iRMX 86 Operating System Nucleus into hardware. This will allow custom hardware applications to have higher-level functions, simplified development, and decreased chip count. Using the iRMX 86 Operating System today will give hardware component users a headstart on Intel's technological innovation for tomorrow.
APPENDIX A

PL/M-36 COMPILER SUPERVISOR TASK FOR AP NOTE 110, OCTOBER 1980
ISIS-II PL/M-86 V2.0 COMPILED MODULE OF MODULE SUPERVISOR
OBJECT MODULE PLACED IN :Fl:superv.OBJ
COMPILER INVOKED BY: plm86 :Fl:superv.p8

Stitle('SUPERVISOR TASK FOR AP NOTE 110, OCTOBER 1980')
$large debug
SUPERVISOR MODULE:
do;
$include(:fl:nuclus.ext)
= $SAVE NOLIST

089 1 declare token literally 'word';

/*****************************************************************
/* The six mailboxes immediately following will */
/* form all of the inter-task communications */
/* interfaces for the five tasks that run in */
/* this system. */
/*****************************************************************/

090 1 declare th_in_mbx token;
091 1 declare th_out_mbx token;
092 1 declare to_input_mbx token public;
093 1 declare to_fft_mbx token public;
094 1 declare to_output_mbx token public;
095 1 declare to_superv_task_mbx token public;

096 1 declare return_th_in_mbx token;
097 1 declare return_th_out_mbx token;
098 1 declare dummy_mbx token;
099 1 declare frame_segment_one token;
100 1 declare frame_segment_two token;
101 1 declare frame_segment_three token;
102 1 declare th_in_segment_token token;
103 1 declare th_out_segment_token token;

104 1 declare general_index byte;
105 1 declare number_of_fft_data_segments byte;

106 1 declare insert_text_pointer pointer;

107 1 declare abort_flag word;
108 1 declare status word;
109 1 declare segment_deleted_tally word;
110 1 declare text_length word;

111 1 declare root_job_token token;
112 1 declare input_task_token token;
113 1 declare fft_task_token token;
114 1 declare output_task_token token;

115 1 declare parameters structure(
    actual_samples word,
...
actual_interval word,
frequency_answer(5) byte,
actual_frames_to_average word,
frames_to_average_answer(5) byte,
continuous_flag word,
continuous_flag_answer(5) byte);

116 1 declare abort literally 'OFFH';
117 1 declare ascii_9 literally '039H';
118 1 declare carriage_return literally '0DH';
119 1 declare forever literally 'while 1';
120 1 declare line_feed literally '0AH';
121 1 declare new_fft_run literally 'OFFH';
122 1 declare no_abort literally '00H';
123 1 declare no_response_requested literally '00H';
124 1 declare null literally '00H';
125 1 declare queue_fifo literally '00H';
126 1 declare queue_priority literally '01H';
127 1 declare rootjob literally '03H';
128 1 declare run_continuous literally '0FFFFH';
129 1 declare size_120_bytes literally '120';
130 1 declare space literally '20H';
131 1 declare supervisor_job literally '00H';
132 1 declare th_read literally '01H';
133 1 declare th_write literally '05H';
134 1 declare wait_forever literally '0FFFH';

/docs/index.html

135 1 declare cursor_home_chars(2) byte data (07EH,012H,);
136 1 declare frame_pointer pointer;
137 1 declare frame_pointer_values structure(  
      offset word,
      base word) at (@frame_pointer);
138 1 declare frame_based_frame_pointer structure(  
      samples_per_frame word,
      sample_interval word,
      frames_to_average word,
      continuous_flag word,
      this_frame_number word,
      number_samples_missed word,
      sample_pointer word,
      reset_flag word,
      sample(256) integer);
139 1 declare th_in_segment_pointer pointer;
140 1 declare th_in_segment_pointer_values structure(  
      offset word,
      base word) at (@th_in_segment_pointer);
APPENDIX A

141 declare th in segment based th in segment pointer structure(
    function word,
    count word,
    exception code word,
    actual - word,
    message(112) byte);

142 declare th out segment pointer pointer;
143 declare th out segment pointer values structure(  
    offset word,  
    base word) at (@th out segment pointer);

144 declare th out segment based th out segment pointer structure(  
    function word,
    count word,
    exception code word,
    actual - word,
    home_chars(2) byte,
    line_index(24) byte,
    message(84) byte);

145 declare frequency question(*) byte data ('Please'  
    ' enter the highest frequency in Hz (120,  
    600, 1200, 6000, OR 12000):');

146 declare frequency answers data(*) byte data (05H,  
    03H,03H,04H,04H,05H,0H,  
    ' 120 600 1200 600012000',  
    42H,0FH,00DH,03H,087H,01H,  
    4EH,00H,027H,00H,00H,000H);

147 declare average question(*) byte data ('Please'  
    ' enter the number of frames to average'  
    ' (1, 2, 4, 8, 16, OR 32):');

148 declare average answers data(*) byte data (06H,  
    01H,01H,01H,01H,02H,02H,  
    ' 1 2 4 8 16 32',  
    01H,00H,02H,00H,04H,00H,  
    08H,00H,10H,00H,20H,00H);

149 declare continuous question(*) byte data ('Please'  
    ' enter "T" for one sample run or "C"'  
    ' for continuous running:');

150 declare continuous answers data(*) byte data (03H,  
    01H,01H,01H,00H,00H,  
    ' l c c ',  
    00H,00H,0FFH,0FFH,0FFH,0FFH,  
    00H,00H,00H,00H,00H,
    00H,00H);

151 declare reject message(*) byte data ('I cannot'  
    ' accept your answer. PLEASE TRY AGAIN.');
do blank line_index = 0 to 78;
  th_out_segment.message (blank_line_index)
    = space;
end;

th_out_segment.message (79) = carriage_return;

END BLANK_LINE;

*******************************************************************************
/* DISPLAY LINE sends the message to the */
/* terminal handler output mailbox and */
/* waits for the segment to be returned. */
*******************************************************************************

DISPLAY_LINE: PROCEDURE;
call rq$sendSmessage (th_out_mbx,
  th_out_segment_token, return_th_out_mbx, @status);

th_out_segment_token = rq$receiveSmessage
  (return_th_out_mbx, wait_forever, @dummy_mbx, @status);

END DISPLAY_LINE;

*******************************************************************************
/* INSERT TEXT fills the output message segment */
/* with the chosen message and pads the rest of */
/* the line with blanks. */
*******************************************************************************

INSERT_TEXT: PROCEDURE(TEXT_POINTER,HOW_MANY);

do insert_text_index = 0 to (how_many - 1);
  th_out_segment.message (insert_text_index) =
    dummy.entries (insert_text_index);
end;

do while insert_text_index < 79;
  th_out_segment.message (insert_text_index) =
    space;
  insert_text_index = insert_text_index + 1;
end;

th_out_segment.message (79) = carriage_return;

END INSERT_TEXT;
APPENDIX A

/**********************************************/
/* Move down line puts the required number of */
/* line-feed characters in the first 4th */
/* through 29th character of the output */
/* message. Each line is displayed on the */
/* screen in its proper location. This allows */
/* multiple tasks to access the screen */
/* without having to blank the line each */
/* time. This technique assumes each message */
/* sends the cursor home each time. */
/**********************************************/

declar skip lines   word;
declar move_down_line_index   word;

if skip lines > 0 then
   do move_down_line_index = 0 to (skip lines - 1);
      th_out_segment.line_index (move_down_line_index) = line_feed;
   end;

   do move_down_line_index = skip lines to 23;
      th_out_segment.line_index (move_down_line_index) = null;
   end;

END MOVE_DOWN_LINE;

/*************************************************/
/** The following procedures are the primary **/}
/** working procedures called by the supervisor **/}
/** procedure and its called procedures. **/}
/*************************************************/

SEND_REJECT_MESSAGE: PROCEDURE;
call move_down_line (21);
text_length     = size(reject_message);
insert_text_pointer   = @reject_message;
call insert_text (insert_text_pointer, text_length);
call display_line;

END SEND_REJECT_MESSAGE;
/* PURGE_MAILBOX removes all tokens from */
/* a mailbox. The purpose is to remove segments */
/* waiting for processing by one of the tasks */
/* if the operator has specified an abort */
/* request. If the segment deletion was */
/* successful, PURGE_MAILBOX updates the */
/* segment_deleted_tally. */
/* ***************************************************/

PURGE_MAILBOX: PROCEDURE (MAILBOX_TO_PURGE);

declare mailbox_to_purge token;

declare for_110_milliseconds literally 'OBH';
declare message_received literally 'OH';

declare contents_token token;
declare purge_dummy_mbx token;
declare purge_status token;

purge_status = message_received;

do while purge_status = message_received;

token = rq$receive$message (mailbox to purge, for_110_milliseconds,
 &purge_dummy_mbx, &purge_status);

if purge_status = message_received then
do;
call rq$delete$segment (contents token, &purge_status);

segment_deleted_tally = segment_deleted_tally + 1;

purge_status = message_received;
end;
end;

END PURGE_MAILBOX;

/* MONITOR MAILBOXES polls the */
/* return_th_in_mailbox and the */
/* to_supervisor_mailbox for messages. The */
/* messages will be abort (from the operator), */
/* and FFT done or OUTPUT done if the runs are */
/* not continuous. If the runs are not */
/* continuous and an FFT done message is */
/* received, MONITOR_MAILBOXES will initialize */
/* the OUTPUT task. */
/* ***************************************************/

MONITOR_MAILBOXES: PROCEDURE;
declare cannot_wait literally '00H';
declare done literally '0FFH';
declare for_400_milliseconds literally '28H';
declare message_received literally '00H';
declare not_done literally '00H';
declare monitor_dummy_mbx token;
declare monitor_token token;
declare monitor_status token;
declare done_flag byte;
declare monitor_index word;
done_flag = not_done;
segment_deleted_tally = 0;
call rq$send$message (th_in_mbx, th_in_segment_token, return_th_in_mbx, @status);
do while done_flag = not done;
/* Check for operator input here. */
monitor_token = rq$receive$message (return_th_in_mbx, for_400_milliseconds, @monitor_dummy_mbx, @monitor_status);
if monitor_status = message_received then
do while segment_deleted_tally < number_of_fft_data_segments;
call purge_mailbox (to_fft_mbx);
if segment_deleted_tally < number_of_fft_data_segments then
call purge_mailbox (to_input_mbx);
if segment_deleted_tally < number_of_fft_data_segments then
call purge_mailbox (to_supervisor_mbx);
if segment_deleted_tally < number_of_fft_data_segments then
call purge_mailbox (to_output_mbx);
done_flag = done;
end;
if done_flag = not done then
do;
monitor_token = rq$receive$message (to_supervisor_mbx, for_400_milliseconds, @monitor_dummy_mbx, @monitor_status);
if monitor_status = message_received then
do;
if parameters.continuous_flag = run_continuous then
call rq$send$message (to_input_mbx, monitor_token, no_response_requested, @monitor_status);
else
APPENDIX A

259 5        do;
260 6        call rq$delete$segment
           (monitor_token, @monitor_status);
261 6        segment_deleted_tally
           = segment_deleted_tally + 1;
262 6        if segment_deleted_tally
           = number_of_fft_data_segments
           then done_flag = done;
264 6        end;
265 5        end;
266 4        end;
267 3        end;
268 2        END MONITOR_MAILBOXES;

/**************************                       /
/* SET_SEGMENT initializes the common parameter */
/* areas of the segment. The pointer to the */
/* proper segment is set up by */
/* INITIALIZE_SEGMENTS. */
/**************************                       /

269 1        SET_SEGMENT: PROCEDURE;
270 2        frame.samples_per_frame = 128;
271 2        frame.sample_interval
           = parameters.actual_interval;
272 2        frame.frames_to_average
           = parameters.actual_frames_to_average;
273 2        frame.continuous_flag = parameters.continuous_flag;
274 2        frame.this_frame_number = 00H;
275 2        frame.number_samples_missed = 00H;
276 2        frame.sample_pointer = 00H;
277 2        frame.reset_flag = 00H;
278 2        END SET_SEGMENT;

/**************************                       /
/* INITIALIZE_SEGMENTS creates the three FFT */
/* data segments and calls SET_SEGMENT for each */
/* segment to initialize the common parameter */
/* areas of the segments. */
/**************************                       /

279 1        INITIALIZE_SEGMENTS: PROCEDURE;
280 2        declare size_528_bytes literally '528';
281 2        frame_pointer_values.offset = 0;
282 2        frame_segment_one = rq$create$segment
           (size_528_bytes, @status);
283 2        frame_pointer_values.base = frame_segment_one;
284 2        call set_segment;
APPENDIX A

frame.reset_flag = new_fft_run;
number_of_fft_data_segments = 1;
call rq$send$message
   (to_input_mb_x, frame_segment_one,
    no_response_requested, @status);

if parameters.actual_frames_to_average > 1 then
   do;
      frame_segment_two = rq$create$segment
         (size_528_bytes, @status);
      frame_pointer_values.base = frame_segment_two;
      call set_segment;
      number_of_fft_data_segments = 2;
      call rq$send$message
         (to_input_mb_x, frame_segment_two,
          no_response_requested, @status);
   end;

if parameters.actual_frames_to_average > 2 then
   do;
      frame_segment_three = rq$create$segment
         (size_528_bytes, @status);
      frame_pointer_values.base
         = frame_segment_three;
      call set_segment;
      number_of_fft_data_segments = 3;
      call rq$send$message
         (to_input_mb_x, frame_segment_three,
          no_response_requested, @status);
   end;

END INITIALIZE_SEGMENTS;

*******************************************************************************/
/* INPUT_PARAMETERS contains three procedures: */
/* set_question_pointers, get_answer, */
/* and verify_answers. The INPUT_PARAMETERS */
/* loop consists of calls to these three */
/* procedures and, as usual, exists at the end */
/* of the procedure. */
*******************************************************************************/

INPUT_PARAMETERS: PROCEDURE;

declare actual_pointer pointer;
declare answer_pointer pointer;
declare answer_display_pointer pointer;
declare question_pointer pointer;
declare answer_actual_value based actual_pointer word;
declare answer_overlay based answer_pointer structure(
   number_of_answers byte,
APPENDIX A

```c
length_of_answer(6) byte,
values_to_match(30) byte,
really_are(6) word);

312 2 declare answer_display based
    - answer_display_pointer structure(
        characters(5) byte);

313 2 declare answer_byte_index byte;
314 2 declare answer_index byte;
315 2 declare answer_match byte;
316 2 declare byte_match byte;
317 2 declare input_byte_index byte;
318 2 declare output_byte_index byte;
319 2 declare question_number byte;
320 2 declare stop_byte byte;

321 2 declare ascii_small_g literally '067H';
322 2 declare ascii_capital_G literally '047H';
323 2 declare average_entry_point literally '0';
324 2 declare continuous_entry_point literally '48';
325 2 declare frequency_entry_point literally '58';
326 2 declare match literally '0FH';
327 2 declare no_match literally '00H';
328 2 declare not_negative literally '<255';
329 2 declare nothing_returned literally '00H';

330 2 set_question_pointers: procedure;
331 3 do case question_number;
332 4 do;
333 5    text_length = size (frequency_question);
334 5    question_pointer = frequency_question;
335 5    answer_pointer = frequency_answers_data;
336 5    actual_pointer = parameters.actual_interval;
337 5    answer_display_pointer = parameters.frequency_answer;
338 5 end;
339 4 do;
340 5    text_length = size (average_question);
341 5    question_pointer = average_question;
342 5    answer_pointer = average_answers_data;
343 5    actual_pointer
        = parameters.actual_frames_to_average;
344 5    answer_display_pointer
        = parameters.frames_to_average_answer;
345 5 end;
346 4 do;
347 5    text_length = size (continuous_question);
348 5    question_pointer = continuous_question;
349 5    answer_pointer = continuous_answers_data;
350 5    actual_pointer = parameters.continuous_flag;
```

28
answer_display_pointer = @parameters.continuous_flag_answer;
end;
end;
end set_question_pointers;

get_answer: procedure;

/* First display the question to be answered */
/* by the operator. */
call insert_text (question_pointer, text_length);
call move down line (19);
call display_line;

/* Then blank the line below for an answer line. */
call blank line;
call move down line (20);
call display_line;

/* Now wait for a response from the operator. */
call rq$send$message
   (th in mbx, th in segment_token, 
   return th in mbx, @status);
th_in_segment_token = rq$receive$message
   (return th in mbx, wait forever, 
   @dummy_mbX, @status);

th_in_segment_pointer_values.base
   = th_in_segment_token;

/* If there is no message returned then send */
/* a reject message. */
if th_in_segment.actual = nothing_returned
   then call send_reject_message;

/* Otherwise it is time to check the response */
/* against the possible answers. */
   else
      do;
      answer_match = no_match;
      /* Set the number of possible answers. */
      answer_index
         = answer_overlay.number_of_answers;
      /* Start a loop to check all of the */
      /* possible answers. */
   */
do while (answer_match = no_match) and (answer_index > 0);
/* Set the starting point for the */
/* byte by byte compare.        */

answer_byte_index = (answer_index * 5) - 1;
/* Set the stopping point for */
/* the compare.                */

stop_byte = answer_byte_index - answer_overlay.length_of_answer
            (answer_index - 1);
/* Start with a "match" so we can */
/* check until "no match" occurs. */

byte_match = match;
/* Set starting point at the right end */
/* of the input data (allows us to */
/* ignore leading blanks and the */
/* ending carriage return).       */

input_byte_index = thIn_segment.actual-2;
/* Scan the bytes until all pertinent */
/* ones are checked or a "no_match" */
/* occurs.                        */

do while (byte_match = match) and
    (answer_byte_index > stop_byte);
    if (input_byte_index not_negative)
        then do;
            if thIn_segment.message
                (input_byte_index) =
                    answer_overlay.values_to_match
                    (answer_byte_index),
                then byte_match = match;
                else byte_match = no_match;
        end;
    else byte_match = no_match;

    answer_byte_index = answer_byte_index-1;
    input_byte_index = input_byte_index -1;
end;
/* A "match" at this point means ALL */
/* bytes matched.                  */

if byte_match = match then
do;
/* Set real values via */
/* answer_actual_value overlay. */

388 6
answer_actual_value
    = answer_overlay.really_are
        (answer_index - 1);
389 6
answer_match = match;

/* Insert displayable values for */
/* later display. */
390 6
answer_byte_index = 4;
391 6
input_byte_index = (answer_index*5)-1;
392 6
do while answer_byte_index not_negative;
    answer_display.characters­
        (answer_byte_index)
        = answer_overlay.values_to_match
            (input_byte_index);
    input_byte_index
        = input_byte_index - 1;
    answer_byte_index
        = answer_byte_index - 1;
396 7
end;

/* We got a match, so be sure the */
/* reject message line is blanked. */
397 6
call move down line (21);
398 6
call blank_line;
399 6
call display_line;
400 6
end;

/* If no match, then let's compare the */
/* input with the next possible answer. */
401 5
else answer_index = answer_index - 1;
402 5
end;

/* If we got a match, then we can move on */
/* to the next question. */
403 4
if answer_match = match
    then question_number = question_number + 1;

/* Otherwise we have to check for an */
/* abort request of '99'. */
405 4
else
    do;
    input_byte_index = th_in_segment.actual-2;
    if (th_in_segment.message(input_byte_index)
        = ascii_9)
        and
        (th_in_segment.message (input_byte_index - 1)
            = ascii_9) then
/* Abort requests are valid, so blank */
/* the reject message line and reset */
/* the question number so we start */
/* asking all over. */

do;
    question_number = 1;
    call move_down_line (21);
    call blank_line;
    call display_line;
end;
else
    /* But if nothing matched and the */
    /* answer was not an abort request, */
    /* then we have to ask the operator */
    /* to try again on this question. */
    call send_reject_message;
end;
end get_answer;

verify_answers: procedure;
    /* First put the output line in the buffer. */
    text_length = size (status_line_one);
    call-insert_text (~status_line_one, text_length);
    /* Then insert the displayable frequency answer. */
    input_byte_index = 0;
    stop_byte = frequency_entry_point + 4;
    do output_byte_index = frequency_entry_point to stop_byte;
        th_out_segment.message (output_byte_index) =
            parameters.frequency_answer (input_byte_index);
        input_byte_index = input_byte_index + 1;
    end;
    call move_down_line(19);
    call display_line;
    /* We have sent the first line, now it is time */
    /* to get the second line. */
    text_length = size (status_line_two);
    call-insert_text (~status_line_two, text_length);
    /* We have to insert the displayable "frames */
/* to average" answer. */

class

input byte index = 0;

stop byte = average_entry_point + 4;
do output byte index
    = average_entry_point to stop byte;
    th out segment.message (output byte index) =
        parameters.frames_to_average_answer (input byte index);
    input byte index = input byte index + 1;
end;

/* The continuous answer is different--we have */
/* to decide if we have continuous runs or */
/* single runs, and insert those words in the */
/* display line. */

input byte index = 0;

stop byte = continuous_entry_point + 15;
if parameters.continuous_flag = Run_continuous
    then
do output byte index
    = continuous_entry_point to stop byte;
    th out segment.message (output byte index) =
        continuous runs (input byte index);
    input byte index = input byte index + 1;
end;
else
do output byte index
    = continuous_entry_point to stop byte;
    th out segment.message (output byte index) =
        single run (input byte index);
    input byte index = input byte index + 1;
end;

/* Then send the message and wait for a response */
/* from the operator. */

call move down line(20);
call display_line;
text length = size (go ahead question);
call insert_text (@go ahead question, text length);
call move down line(21);
call display_line;
call rq$send$message (th in mbx, th in segment_token, return th in mbx, @status);
th in segment_token = rq$receive$message
APPENDIX A

(return_th_in_mbx, wait_forever, @dummy_mbx, @status);

```c
459 3 th_in_segment_pointer_values.base  
     = th_in_segment_token;
460 3 input_byte_index = th_in_segment.actual - 2;
```

/* Check for a "g" or "G" (we aren't fussy). If */
/* we got it, let's quit asking the selection */
/* questions and go. If not, we have to start */
/* at question 1 again rather than try to find */
/* out which of his or her answers wasn't */
/* acceptable. */

```c
461 3 if (th_in_segment.message (input_byte_index)  
     = ascii_small_g) or -  
     (th_in_segment.message (input_byte_index)  
     = ascii_capital_G) then;
463 3 else question_number = 0;
464 3 call blank_line;
465 3 call move_down_line (21);
466 3 call display_line;
467 3 end verify_answers;
```

/* As usual, the actual INPUT PARAMETERS control */
/* loop is at the end. */

```c
468 2 question_number = 0;

/* All we do is get the next question, ask the */
/* question until it is answered successfully, */
/* ask all of the questions, then check all of */
/* the answers. If the operator doesn't like */
/* the set of answers, we loop through them */
/* again. First we make sure the reject message */
/* line and other pertinent lines start out */
/* blanked. */
```

```c
469 2 call blank_line;
470 2 call move_down_line (18);
471 2 call display_line;
472 2 call move_down_line (21);
473 2 call display_line;
474 2 call move_down_line (22);
475 2 call display_line;
476 2 call move_down_line (23);
477 2 call display_line;
478 2 input_loop: do while question_number < 3;
479 3 call set_question_pointers;
480 3 call get_answer;
```
end;
call verify_answers;
if question_number = 0 then goto input_loop;
END INPUT_PARAMETERS;

INITIALIZE_TASKS: PROCEDURE;

declare hardware_interrupt_level_3 literally '038H';
declare no_data_segment literally '00H';
declare nucleus_allocated_stack literally '00H';
declare software_priority_level_67 literally '67';
declare software_priority_level_130 literally '130';
declare software_priority_level_131 literally '131';
declare stack_size_512 literally '512';
declare task_flags literally '00H';

input_task_token = rq$createTask
  (software_priority_level_67, @input_task, 
   no_data_segment, nucleus_allocated_stack, 
   stack_size_512, task_flags, @status);
call rq$catalog$object
  (supervisor_job, input_task_token, 
   @(10,'INPUT TASK'), @status);
fft_task_token = rq$createTask
  (software_priority_level_131, @fft_task, 
   no_data_segment, nucleus_allocated_stack, 
   stack_size_512, task_flags, @status);
call rq$catalog$object
  (supervisor_job, fft_task_token, 
   @(8,'FFT TASK'), @status);
output_task_token = rq$createTask
  (software_priority_level_130, @output_task, 
   no_data_segment, nucleus_allocated_stack, 
   stack_size_512, task_flags, @status);
call rq$catalog$object
  (supervisor_job, output_task_token, 
   @(11,'OUTPUT TASK'), @status);
END INITIALIZE_TASKS;

/***************************************************************************/
/* Initial screen displays the initial two */
/* lines on the screen and sends blank lines */
/* for all the other lines of the first screen. */
/***************************************************************************/

INITIAL_SCREEN: PROCEDURE;

declare initial_screen_index word;

call move_down_line(0);
call blank_line;
call display_line;

call move_down_line(1);
text_length = size(header_line_one);
insert_text_pointer = @header_line_one;
call insert_text (insert_text_pointer, text_length);
call display_line;

call blank_line;
do initial_screen_index = 2 to 23;
call move_down_line (initial_screen_index);
call display_line;
end:

END INITIAL_SCREEN;

/***************************************************************************/
/* INITIALIZE BUFFERS takes care of the */
/* initialization required for general */
/* SUPERVISOR TASK start up. */
/***************************************************************************/

INITIALIZE_BUFFERS: PROCEDURE;

return_th_in_mbx = rq$create$mailbox
(queue_fifo, @status);
call rq$scatalog$object
(supervisor_job, return_th_in_mbx,
@(9,'SUP TH IN'), @status);

return_th_out_mbx = rq$create$mailbox
(queue_fifo, @status);
call rq$scatalog$object
(supervisor_job, return_th_out_mbx,
@(10,'SUP TH OUT'), @status);

to_input_mbx = rq$create$mailbox
(queue_fifo, @status);
call rq$scatalog$object
(supervisor_job, to_input_mbx,
@(12,'TO INPUT MBX'), @status);

to_fft_mbx = rq$create$mailbox
(queue_priority, @status);
call rq$catalog$object
(supervisor_job, to_fft_mbx,
@(10,'TO FFT MBX'), @status);

to_output_mbx = rq$create$mailbox
(queue_priority, @status);

call rq$catalog$object
(supervisor_job, to_output_mbx,
@(10,'TO OUT MBX'), @status);

to_supervisor_mbx = rq$create$mailbox
(queue_priority, @status);

call rq$catalog$object
(supervisor_job, to_supervisor_mbx,
@(10,'TO SUP MBX'), @status);

root_job_token = rq$get$task$tokens
(root_job, @status);

th_out_mbx = rq$lookup$object
(root_job_token, @(11,'RQTHNORMOUT'),
wait_forever, @status);

th_in_mbx = rq$lookup$object
(root_job_token, @(10,'RQTHNORMIN'),
wait_forever, @status);

th_in_segment_token = rq$create$segment
(size 120 bytes, @status);

call rq$catalog$object
(supervisor_job, th_in_segment_token,
@(10,'S THIN SEG'), @status);

th_in_segment_pointer_values.offset = 0;

th_in_segment_pointer_values.base = th_in_segment_token;

th_in_segment.function = th_read;

th_in_segment.count = 82;

th_out_segment_token = rq$create$segment
(size 120 bytes, @status);

call rq$catalog$object
(supervisor_job, th_out_segment_token,
@(11,'S THOUT SEG'), @status);

th_out_segment_pointer_values.offset = 0;

th_out_segment_pointer_values.base = th_out_segment_token;

th_out_segment.function = th_write;

th_out_segment.count = 111;

do general_index = 0 to 2;

th_out_segment.home_chars (general_index) = cursor_home_chars (general_index);

end;

END INITIALIZE_BUFFERS;

/*****************************/
/* At last, the SUPERVISOR TASK! All it does is */
/* call other procedures to initialize the */
/* screen, input the parameters, clean up the */
/* old FFT segments from the mailboxes, set up */
/* new segments, create the tasks, and then wait */
/* for messages from the operator (abort) or */
/* other tasks (FFT or OUTPUT done). */
/****************************************************/

SUPERVISOR_TASK: PROCEDURE PUBLIC;

call initialize_buffers;
call rqSendInitSinit$task;
call initialize_screen;
call initialize_tasks;

do forever;
call input parameters;
call initialize_segments;
call monitor_mailboxes;
end;

END SUPERVISOR_TASK;

END SUPERVISOR_MODULE;

MODULE INFORMATION:

| CODE AREA SIZE | 1032H | 4146D |
| CONSTANT AREA SIZE | 0000H | 0D |
| VARIABLE AREA SIZE | 0084H | 132D |
| MAXIMUM STACK SIZE | 0024H | 36D |

1197 LINES READ
0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
APPENDIX B

ISIS-II PL/M-86 V2.0 COMPILATION OF MODULE INPUT_TASK_MODULE
OBJECT MODULE PLACED IN :Fl:input.OBJ
COMPILER INVOKED BY: plm86 :Fl:input.p86
$title('INPUT TASK FOR AP NOTE 110, OCTOBER 1980')
$large debug
INPUT_TASK_MODULE:
do;
$include(:fl:nucprm.ext)
= $SAVE NOLIST

89 1 declare token literally 'word';
 /* The following two tokens, the FFT sample */
 /* segment format, and the root job directory */
 /* form the entire interface for this task with */
 /* the rest of the system. */
90 1 declare to_input_mbx token external;
91 1 declare to_fft_mbx token external;
92 1 declare ascii_mask literally '30H';
93 1 declare carriage_return literally 'ODH';
94 1 declare done literally 'OFFH';
95 1 declare first_loop literally 'OFFH';
96 1 declare forever literally 'while 1';
97 1 declare frames_to_process_entry literally '50';
98 1 declare hardware_Interruption_level literally '0038H';
99 1 declare interrupt_task_created literally 'OFFH';
100 1 declare interrupt_task_not_created literally '00H';
101 1 declare latch_the_data literally '040H';
102 1 declare line_feed literally '0AH';
103 1 declare new_fft_run literally 'OFFH';
104 1 declare no_response_requested literally '00H';
105 1 declare no_data_segment literally '00H';
106 1 declare not_done literally '00H';
107 1 declare not_first_loop literally '00H';
108 1 declare not_valid literally '00H';
109 1 declare null literally '00H';
110 1 declare processed_so_far_entry literally '33';
111 1 declare queue_fifo literally '00H';
112 1 declare rootjob literally '03H';
113 1 declare run_continuous literally '0FFFFH';
114 1 declare sample_LSB literally '0081H';
115 1 declare sample_MSB literally '0080H';
116 1 declare size_2_bytes literally '?';
117 1 declare size_120_bytes literally '120';
118 1 declare supervisor_job literally '00H';
119 1 declare th_write literally '05';
120 1 declare this_is_the_interrupt_task literally '01H';
121 1 declare timer_one_port literally '00D2H';
122 1 declare timer_mode_control_port literally '00D5H';
123 1 declare valid literally 'OFFH';
124 1 declare wait_forever literally '0FFFFFF';

125 1 declare data_segment_token token;
126 1 declare dummy_mbx token;
127 1 declare from_Interrupt_task_mbx token;
128 1 declare handler_dummy_mbx token;
129 1 declare handler_status token;
130 1 declare interrupt_status token;
131 1 declare interrupt_task_token token;
132 1 declare interrupt_message_token token;
133 1 declare output_buffer_token token;
134 1 declare return_mbx token;
135 1 declare root_job_token token;
136 1 declare signal_interrupt_token token;
137 1 declare status token;
138 1 declare to_interrupt_task_mbx token;
139 1 declare th_out_mbx token;

140 1 declare sample_data integer;

141 1 declare sample_input_data_structure(  
    LSB byte,  
    MSB byte) at (@sample_data);

142 1 declare current_timer_value word;

143 1 declare timer_values_structure(  
    LSB byte,  
    MSB byte) at (@current_timer_value);

144 1 declare done_flag byte;
145 1 declare first_input_loop_flag byte;
146 1 declare frames_received byte;
147 1 declare general_index byte;
148 1 declare interrupt_task_flag byte;
149 1 declare sample_valid byte;

150 1 declare timer_threshold word;

151 1 declare value_to_convert word;

152 1 declare converted_value_structure(  
    first_digit byte,  
    second_digit byte);

    /* The following declare is for the home */  
    /* characters for the Hazeltine terminals. */  
    /* The sequence is tilde, DC2. */  

153 1 declare cursor_home_chars(2) byte data (07EH,012H);

154 1 declare output_buffer_pointer pointer;
APPENDIX B

155 1 declare output_buffer_pointer_values structure (  
    offset word,  
    base word) at  
    (@output_buffer_pointer);

156 1 declare output_buffer based output_buffer_pointer  
    structure (  
    function word,  
    count word,  
    exception_code word,  
    actual word,  
    home_chars(2) byte,  
    line_index(24) byte,  
    character(30) byte);

157 1 declare data_segment_pointer pointer public;

158 1 declare data_segment_pointer_values structure (  
    offset word,  
    base word) at  
    (@data_segment_pointer);

/* The following is the FFT data segment format. */

159 1 declare data_segment based data_segment_pointer  
    structure (  
    samples_per_frame word,  
    sampleInterval word,  
    frames_to_average word,  
    continuous_flag word,  
    this_frame_number word,  
    number_samples_missed word,  
    sample_pointer word,  
    reset_flag word,  
    sample(256) integer);

160 1 declare input_status line(30) byte data  
    (' - The INPUT TASK has processed ',  
    ' frames out of frames to'  
    ' average. ');

161 1 FAST_INPUT_HANDLER:  
    PROCEDURE (FFT_SEGMENT_POINTER) EXTERNAL;
162 2 DECLARE FFT_SEGMENT_POINTER POINTER;
163 2 END FAST_INPUT_HANDLER;

/*******************************************************************************
/* SLOW_INPUT_HANDLER is an interrupt procedure */
/* that receives an interrupt when the 8253 */
/* interval timer counts to zero. The 8253 is */
/* free running, so it starts counting from the */
/* top again. The 8253 counter is tested */
/* in a polling fashion to be sure it reads the */
/* sample, which resets the conversion, */
/* at a precise time. This aids in removing */
*******************************************************************************/
/* jitter from the sample intervals. When all */ /* of the samples have been taken, */ /* SLOW INPUT_HANDLER calls signal interrupt, */ /* which lets the INTERRUPT_TASK procedure know */ /* the buffer is full. */ /* *********************************************/

SLOW_INPUT_HANDLER: PROCEDURE INTERRUPT 59 PUBLIC;

/* First set the sample to not_valid (we have */ /* to be past the timer threshold before the */ /* sample becomes valid). */
sample_valid = not_valid;

/* Make the timer value stable and read it. */
output (timer_mode_control_port) = latch_the_data;
timer_values.LSB = input (timer_one_port);
timer_values.MSB = input (timer_one_port);

/* If it is not past the threshold, then some */ /* future sample will be valid. */
if current_timer_value > timer_threshold then
  sample_valid = valid;
goto timer_loop;
end;

/* We get to the else only if we are past the */ /* timer threshold. */
else
  do;
    sample_input_data.LSB = input(sample_LSB);
sample_input_data.MSB = input(sample_MSB);
  end;

/* If the sample is valid, we must have come */ /* in before the threshold so we know we */ /* sampled as close to the right time as */ /* possible. */
if sample_valid = valid then
  do;
    /* However, we want to ignore the first */ /* sample (which was started a long */ /* time ago). */
    if first_input_loop_flag = first_loop then
      first_input_loop_flag = not_first_loop;
    else
      do;

74 2
164 1
165 2
166 2
167 2
168 2
169 2
170 2
171 3
172 3
173 3
174 2
175 3
176 3
177 3
178 3
179 4
180 4
181 4
data segment.sample
(data_segment.sample_pointer)
  = sample_data;

data_segmentSample_pointer
  = data_segmentSample_pointer+1;
end;
end;
else
do;

data_segment.number_samples_missed
  = data_segment.number_samples_missed+1;

data_segmentSample_pointer = 0;
end;
sample_valid = not_valid;
end;

/* If we are done, we have to let the */
/* INPUT_TASK know the buffer is full. */
/* Otherwise, we wait for the next interrupt. */

if data_segmentSample_pointer
  >= data_segment.samples_per_frame then
call rqSsignalSinterrupt
  (hardware_interrupt_level_3,
   @handler_status);
else
call rqSexitSinterrupt
  (hardware_interrupt_level_3,
   @handler_status);

END SLOW_INPUT_HANDLER;

*******************************************************************************/
/* INTERRUPT_TASK exists because if an interrupt */
/* task goes to sleep, the level of the */
/* interrupt task, interrupt handler, and lower */
/* levels remain disabled. In order to prevent */
/* this from happening in this application, this */
/* task notifies the INPUT_TASK that the buffer */
/* is full. INPUT_TASK disables Level 3 and */
/* returns the token to INTERRUPT_TASK. */
/* INTERRUPT_TASK will then call wait interrupt, */
/* enabling lower levels. Since INPUT_TASK */
/* disabled Level 3, no Level 3 interrupts will */
/* be serviced until Level 3 is enabled by */
/* INPUT_TASK. */
/* */
/* NOTE THAT PLM/86 REQUIRES THE USE OF THE */
/* BUILT IN INTERRUPTSPTR PROCEDURE TO OBTAIN */
/* THE PROPER INTERRUPT PROCEDURE ENTRY POINT. */
/* */
*******************************************************************************/
INTERRUPT_TASK: PROCEDURE PUBLIC;
APPENDIX B

call rq$set$interrupt
(hardware_interrupt_level_3,
 this is the interrupt task,
 INTERRUPT$PTR(SLOW_INPUT_HANDLER),
 no_data_segment, @interrupt_status);

do forever;
call rq$wait$interrupt
(hardware_interrupt_level_3,
 @interrupt_status);
call rq$send$message
(from_interrupt_task_mbx,
 interrupt_message_token,
 to_interrupt_task_mbx, @interrupt_status);
interrupt_message_token = rq$receive$message
(to_interrupt_task_mbx, wait_forever,
 @dummy_mbx, @interrupt_status);
end;
END INTERRUPT_TASK;

/********************************************
/* CONVERT_DIGITS is a small procedure for */
/* converting a hex number into an ASCII */
/* number, with the advance knowledge that the */
/* hex number will be less than 99 decimal (in */
/* this case, less than 32 decimal). */
/********************************************

CONVERT_DIGITS: PROCEDURE;

converted_value.first_digit = ascii_mask;
converted_value.second_digit = ascii_mask;
done_flag = not done;
do while done_flag = not done;
value_to_convert = value_to_convert - 10;
/* The problem here is we need to check for */
/* a negative value when we have BYTE values */
/* which are, by definition, positive and */
/* modulo 256. So we adapt by checking for */
/* > 200 decimal, which should mean the */
/* value has "wrapped" around zero. If it */
/* has, we can get our previous value back */
/* by adding 10. */

if value_to_convert < 200 then
converted_value.first_digit

= converted_value.first_digit + 1;
else
  do;
    value_to_convert = value_to_convert + 10;
    converted_value.second_digit
    = converted_value.second_digit +
    value_to_convert;
  done_flag = done;
end;
end;

END CONVERT_DIGITS;

/*******************************/
/* SEND_STATUS converts the current frame */
/* number into ASCII and stuffs it into the */
/* previously initialized status line. Then */
/* SEND_STATUS sends the status line to the */
/* terminal handler and waits for the segment */
/* to be returned. */
/*******************************/

SEND_STATUS: PROCEDURE;
value_to_convert = data_segment.this_frame_number;
call convert_digits;
output_buffer.character (processed_so_far_entry)
  = converted_value.first_digit;
output_buffer.character (processed_so_far_entry+1)
  = converted_value.second_digit;
value_to_convert = data_segment.frames_to_average;
call convert_digits;
output_buffer.character (frames_to_process_entry)
  = converted_value.first_digit;
output_buffer.character (frames_to_process_entry+1)
  = converted_value.second_digit;
call rq$send$message
  (th_out_mbx, output_buffer_token,
   return_mbx, @status);
output_buffer_token = rq$receive$message
  (return_mbx, wait_forever,
   @dummy_mbx, @status);
END SEND_STATUS;

/*******************************/
/* INPUT_DATA selects the fast or slow input */
/* handler, initializes the 8253 timer as */
/* necessary, and calls the appropriate input */
APPENDIX B

/* handler. Please note the values of the intervals selected for sampling are scaled by 60/64 so the actual frequency output of the 128 sample FFT algorithm will match up with the base 10 x axis labels. */
/* Base 10 doesn't map too well to a binary x-axis that runs from 1 to 64. */
/* x-axis doesn't map too well to a binary x-axis that runs from 1 to 64. */

231 1 INPUT_DATA: PROCEDURE;

232 2 declare LSB_120Hz_interval literally '058H';
233 2 declare MSB_120Hz_interval literally '002H';
234 2 declare LSB_400Hz_interval literally '078H';
235 2 declare MSB_600Hz_interval literally '000H';

236 2 declare threshold for 120Hz literally '027H';
237 2 declare threshold for 600Hz literally '0048H';

238 2 declare a_3906_microsecond_interval literally '0F42H';
239 2 declare five_places literally '5';
240 2 declare nucleus_allocated_stack literally '00H';
241 2 declare shift_integer_right literally 'SAL';
242 2 declare software_priority_level_0 literally '00H';
243 2 declare software_priority_level_66 literally '66';
244 2 declare stack_size_512 literally '512';
245 2 declare task_flags literally '00H';
246 2 declare this_task literally '00H';
247 2 declare timer_mode_control_word literally '74H';
248 2 declare enable_conversion literally '00';
249 2 declare input_command literally '0080H';

250 2 output(input_command) = enable_conversion;
if data_segment.sample_interval > 391 then
  do;
    output (timer_mode_control_port) = timer_mode_control_word;
    if data_segment.sample_interval = a_3905_microsecond_interval then
      do;
        timer_threshold = threshold_for_120Hz;
        output (timer_one_port) = LSB_120Hz_interval;
        output (timer_one_port) = MSB_120Hz_interval;
      end;
    else
      do;
        timer_threshold = threshold_for_600Hz;
        output (timer_one_port) = LSB_600Hz_interval;
        output (timer_one_port) = MSB_600Hz_interval;
      end;
    first_input_loop_flag = first_loop;
  end;
if interrupt_task_flag = interrupt_task_not_created then
  do;
    interrupt_task_token = rqScreateStask(
      software_priority_level 66, 
      @interrupt_task, no_data_segment, 
      nucleus_allocated_stack, 
      stack_size_512, task_flags, @status);
    call rqScatalogSobject(
      supervisor_job, interrupt_task_token, 
      @(12,'INTERRUPTTSK'), @status);
    interrupt_task_flag = interrupt_task_created;
  end;
else call rqSenable(
  (hardware_interrupt_level_3, @status);
/* Now we wait until the slow handler */
/* fills the buffer. */
signal_interrupt_token = rqSreceiveSmessage(
  from interrupt_task_mbx, wait_forever, 
  @dummy_mbx, @status);
/* If we get the token, we know the buffer */
/* is full, so we disable level 3 */
call rq$disable
   (hardware_interrupt_level_3, @status);

   /* And return the token so the */
   /* INTERRUPT_TASK can enable lower */
   /* interrupt_levels. */

call rq$send$message
   (to_interrupt_task_mbx,
    signal_interrupt_token,
    no_response_requested, @status);

end;
else
   do;

   /* The fast INPUT handler must sample at */
   /* precise intervals that do not allow */
   /* variable interrupt latency. Therefore */
   /* we raise the priority level to 0--the */
   /* highest--and just sample in a polling */
   /* fashion until the buffer is filled. */

   call rq$set$priority
      (this_task, software_priority_level_0,
       @status);

   call FAST_INPUT_HANDLER (data_segment_pointer);

   call rq$set$priority
      (this_task, software_priority_level_66,
       @status);

end;

do general_index = 0 to 127;
   data_segment.sample (general_index)
     = shift_integer_right
       (data_segment.sample (general_index),
        five_places);

end;

do general_index = 128 to 255;
   data_segment.sample (general_index) = 0000H;

end;

END INPUT_DATA;

/* UPDATE_FRAME_NUMBER just updates the frame */
/* number parameter on the data segments. */

UPDATE_FRAME_NUMBER: PROCEDURE;

data_segment.number_samples_missed = 0;
data_segment.sample_pointer = 0;
if frames_received = data_segment.frames_to_average then frames_received = 0;

if data_segment.reset_flag = new_fft_run then
  do;
    frames_received = 0;
    data_segment.reset_flag = 0;
  end;

frames_received = frames_received + 1;
data_segment.this_frame_number = frames_received;

END UPDATE_FRAME_NUMBER;

/*******************************************************************************/
/* INITIALIZE_BUFFERS takes care of the usual */
/* trivia of setting up the pointers, creating */
/* the return mailbox, looking up the terminal */
/* handler, and all that other small garbage. */
/*******************************************************************************/

INITIALIZE_BUFFERS: PROCEDURE;

return mbx = rq$create$mailbox
  (queue fifo, @status);
call rq$catalog$object
  (supervisor job, return mbx,
    (9,'I RET MBX'), @status);

from_interrupt_task mbx = rq$create$mailbox
  (queue fifo, @status);
call rq$catalog$object
  (supervisor job, from_interrupt_task mbx,
    (12,'FM INTSK MBX'), @status);

to_interrupt_task mbx = rq$create$mailbox
  (queue fifo, @status);
call rq$catalog$object
  (supervisor job, to_interrupt_task mbx,
    (12,'TO INTSK MBX'), @status);

interrupt_message_token = rq$create$segment
  (size 2-bytes, @status);
call rq$catalog$object
  (supervisor job, interrupt_message_token,
    (10,'INTTSK MSG'), @status);

interrupt_task_flag
  = interrupt_task_not_created;

output_buffer_token = rq$create$segment
  (size 120_bytes, @status);
call rq$catalog$object
  (supervisor job, output_buffer_token,
    (10,'I BUFF SEG'), @status);
APPENDIX B

```
output_buffer_pointer_values.offset = 0;
output_buffer_pointer_values.base = output_buffer_token;
output_buffer.function = th_write;
output_buffer.count = 110;
do general_index = 0 to 6;
   output_buffer.home_chars(general_index) = cursor_home_chars(general_index);
end;
do general_index = 0 to 21;
   output_buffer.line_index(general_index) = line_feed;
end;
do general_index = 22 to 23;
   output_buffer.line_index(general_index) = null;
end;
do general_index = 0 to 78;
   output_buffer.character(general_index) = input_status_line(general_index);
end;
root_job_token = rq$get_task_tokens
   (root_job_status);
th_out_mbx = rq$lookup$object
   (root_job_token, @11,'RQTHNORMOUT'),
   wait_forever, @status);
frames_received = 0;
END INITIALIZE_BUFFERS;

The actual INPUT_TASK begins here. It initializes the buffers to begin things,
then waits forever for the FFT sample segment. It then samples the data, fills
the FFT data segment, and sends it to the FFT_TASK. The INPUT_TASK then updates its
status line, sends it to the terminal handler, and returns to the mailbox to
wait forever.

INPUT_TASK: PROCEDURE PUBLIC;
call initialize_buffers;
data_segment_pointer_values.offset = 0;
do forever;
```
/* Wait forever for an FFT data segment at */
/* the to_input_mbx. */

data_segment_token = rq$receive$message
    (to_input_mbx, wait_forever,
     @dummy_mbx, @status);

data_segment_pointer_values.base
    = data_segment_token;

call update_frame_number;

call input_data;

call send_status;

call rq$send$message
    (to_fft_mbx, data_segment_token,
     no_response_requested, @status);

call rq$send$message
    (to_fft_mbx, data_segment_token,
     no_response_requested, @status);

call input_data;
call send_status;
call rq$send$message
    (to_fft_mbx, data_segment_token,
     no_response_requested, @status);

call update_frame_number;
call input_data;
call send_status;
call rq$send$message
    (to_fft_mbx, data_segment_token,
     no_response_requested, @status);

call update_frame_number;
call input_data;
call send_status;
call rq$send$message
    (to_fft_mbx, data_segment_token,
     no_response_requested, @status);

call update_frame_number;
call input_data;
call send_status;
call rq$send$message
    (to_fft_mbx, data_segment_token,
     no_response_requested, @status);

call update_frame_number;
call input_data;
call send_status;
call rq$send$message
    (to_fft_mbx, data_segment_token,
     no_response_requested, @status);

call update_frame_number;
call input_data;
call send_status;
call rq$send$message
    (to_fft_mbx, data_segment_token,
     no_response_requested, @status);

call update_frame_number;
call input_data;
call send_status;
call rq$send$message
    (to_fft_mbx, data_segment_token,
     no_response_requested, @status);

call update_frame_number;
call input_data;
call send_status;
call rq$send$message
    (to_fft_mbx, data_segment_token,
     no_response_requested, @status);

call update_frame_number;
call input_data;
call send_status;
call rq$send$message
    (to_fft_mbx, data_segment_token,
     no_response_requested, @status);

call update_frame_number;
call input_data;
call send_status;
call rq$send$message
    (to_fft_mbx, data_segment_token,
     no_response_requested, @status);

call update_frame_number;
call input_data;
call send_status;
call rq$send$message
    (to_fft_mbx, data_segment_token,
     no_response_requested, @status);

call update_frame_number;
call input_data;
call send_status;
call rq$send$message
    (to_fft_mbx, data_segment_token,
     no_response_requested, @status);

call update_frame_number;
call input_data;
call send_status;
call rq$send$message
    (to_fft_mbx, data_segment_token,
     no_response_requested, @status);

call update_frame_number;
call input_data;
call send_status;
call rq$send$message
    (to_fft_mbx, data_segment_token,
     no_response_requested, @status);

call update_frame_number;
call input_data;
call send_status;
call rq$send$message
    (to_fft_mbx, data_segment_token,
     no_response_requested, @status);

call update_frame_number;
call input_data;
call send_status;
call rq$send$message
    (to_fft_mbx, data_segment_token,
     no_response_requested, @status);

call update_frame_number;
call input_data;
call send_status;
call rq$send$message
    (to_fft_mbx, data_segment_token,
FAST INPUT HANDLER is an assembler routine that runs at priority level 0 and simply drives an analog to digital convertor and stuffs the samples into a data segment until all of the samples have been taken. FAST INPUT HANDLER has passed to it the address of the data segment, in which the offset is known to be zero. FAST INPUT HANDLER returns nothing to the calling routine.

FAST_INPUT_HANDLER provides the proper timing for 39, 78, and 391 microsecond intervals using timed loops of software instructions. In order to provide an FFT without large amounts of jitter, the sample intervals must be uniform in time. iRMX 86 cannot guarantee this uniformity due to its real time design, so this routine takes complete control of the processor for the (39 times 128) 4.9 milliseconds or (78 times 128) 9.9 or (391 times 128) 50 milliseconds required to complete a frame of 128 samples.

iAPX 86 register usage is the following:

- AX - general
- CX - loop delay counter
- BX - stack index
- DX - sample value
- BP - stack
- SP - stack
- DI - offset index into FFT
- SI - not used
- DS - base for FFT data
- ES - not used

- segment
APPENDIX B

33 ;
34 ;*****************************************************************************************
35 ;
36 ;
37 ASSUME DS:FAST_INPUT_DATA, SS:STACK,  
   CS:FAST_INPUT_CODE, ES:NOTHING
38 ;
39 PUBLIC  FAST_INPUT_HANDLER
40 ;
0080 41 SAMPLE_LSB  EQU  0080H
0081 42 SAMPLE_MSB  EQU  0081H
000E 43 FIRST_PASS  EQU  14
0002 44 SAMPLE_INCREMENT EQU  2
0002 45 SAMPLE_INTERVAL EQU  2
010E 46 SAMPLE_MAX  EQU  270
47 ;
48 ;
49 ----
50 FAST_INPUT_DATA SEGMENT WORD
51 PUBLIC 'DATA'
50 ;
51 LOOP_VALUE  DW  ?
52 ;
53 ----
54 FAST_INPUT_DATA ENDS
55 ;
56 ----
57 STACK SEGMENT STACK 'STACK'
58 ;
59 ;
60 ----
61 STACK ENDS
62 ;
63 ----
64 FAST_INPUT_CODE SEGMENT PARA
65 PUBLIC 'CODE'
64 ;
65 FAST_INPUT_HANDLER PROC FAR
66 ;
0000 66 PUSH DS
0001 5E 67 PUSH BP  
0002 8BEC 69 MOV BP, SP  
0004 8E2E0A 70 MOV DS, [BP + 10]  
0007 BF0200 71 MOV DI, SAMPLE_INTERVAL  
000A 8B05 72 MOV AX, DS:[DI]  
000C BF0E00 73 MOV DI, FIRST_PASS  
000F 3D2700 74 CMP AX, 39
0012 740E 75 JZ SET_39_US

53
APPENDIX B

; TAKES ABOUT 3 US PER DECREMENT
0014 3D4E00 76    CMP AX, 78
; IF AX = 78, SET LOOP_VALUE TO 22--BASIC
0017 7412 77    JZ    SET 78 US
; CYCLE IS ~13 US, PLUS (22 * 3) = 79 US
0019 C70600007E00  R 78  MOV LOOP_VALUE, 126
; 391 IS ONLY ONE LEFT ~13 + (126*3) = 391
001F EB1090 79    JMP INPUT_LOOP
0022 C70600009000  R 80  SET 39 US: MOV LOOP_VALUE, 9
; TIMING IS BY SOFTWARE--SET DELAY COUNT
0028 EB0790 81    JMP INPUT_LOOP
002B C70600001600  R 82  SET 78 US: MOV LOOP_VALUE, 22
0031 88000000  R 83  INPUT_LOOP: MOV CX, LOOP_VALUE
; USE CX TO KEEP TRACK OF DELAY IN AL, SAMPLE LSB
0035 E480 84  IN AL, SAMPLE LSB
0037 8AD0 85  MOV DL, AL
; PUT THE LSB IN DL (8 BIT XFERS ONLY)
0039 E481 86  IN AL, SAMPLE MSB
; SET AL TO MSB OF INPUT SAMPLE
003B 8AF0 88  MOV DH, AL
; PUT AL IN DH TO COMPLETE THE VALUE
003D 83FF0E 89  CMP DI, FIRST_PASS
; WE WANT TO SKIP THE FIRST SAMPLE
0040 7408 90    JZ    SKIP_INPUT
0042 83C702 91  ADD DI, ~SAMPLE_INCREMENT
; INCREMENT DI BY 2
0045 8915 92  MOV DS:[DI], DX
; PUT SAMPLE DATA IN SEGMENT
0047 EB0990 93  JMP DELAY
; AND JUMP TO SOFTWARE DELAY LOOP
004A 83C702 94  SKIP_INPUT: ADD DI, SAMPLE_INCREMENT
; INCREMENT DI BY 2
004D 90 95  NOP
; AND NOP FIVE TIMES FOR EVEN TIMING
004E 90 96  NOP
004F 90 97  NOP
0050 90 98  NOP
0051 90 99  NOP
0052 90 100  DELAY: NOP
; THIS NOP ADDS 3 CLOCKS PER DECREMENT
0053 E0FD 101  LOOPNZ DELAY
; DEC CX AND LOOP--1.5 US PER DECREMENT
0055 81FF0E01 102  CMP DI, SAMPLE_MAX
; COMPARE DI TO SEE IF WE ARE DONE
0059 75D6 103  JNE INPUT_LOOP
; IF NOT, GO BACK FOR ANOTHER SAMPLE
005B 5D 104  POP BP
; OTHERWISE POP BP, DS, AND RETURN
005C 1F 105  POP DS
005D CA0400 106  RET 4H
107 ;

54
APPENDIX B

108  FAST_INPUT_HANDLER  ENDP
109   ;
110  FAST_INPUT_CODE     ENDS
111   ;
112   END

ASSEMBLY COMPLETE, NO ERRORS FOUND
Both the RAM and ROM-based configurations will be discussed in this appendix. They are essentially identical processes. In either case, the first step is to define a map of system memory. Once the map is known, the following sequence is suggested for locating code in memory:

1) Reserve memory $00H$ to $03FFH$ for the Nucleus interrupt vector.

2) If the system is RAM based and the code is loaded by the iSBC 957A Monitor, reserve locations $03FFH$ to $07FFH$ for the monitor’s use.

3) Configure each of the necessary portions of the iRMX 86 Operating System and locate them sequentially in memory.

4) For a RAM-based development system, allow 2K of RAM for the system Root Job. Placing the Root Job after the portions of the iRMX 86 Operating System, which are relatively fixed in size during development, and before the development code will give the Root Job a fixed address. This will prevent having to move the Root Job and reconfigure the system when the development code grows. For final EPROM-based systems, the Root Job should be placed after the development code.

5) Link and locate each of the application code modules sequentially in memory.

6) Define the RAM available to the system.

7) Define memory NOT available to the system. This includes application code, EPROM, and non-existent memory within the 1 megabyte address space.

8) Create the configuration file using the address maps produced by the locate steps and the memory map defined in steps 6 and 7.

9) Create the Root Job from this configuration file.

10) Load and test the system in RAM.

11) If the system has been fully debugged, load the code into EPROM and test the final system.

The above steps are necessary for both the RAM development system and the final EPROM system. Converting this application from RAM to EPROM requires reconfiguring the Nucleus to include only those systems calls required by the application, substituting the Terminal Handler Job for the Debugger Job, removing any remaining system calls to catalog objects for debugging, and remapping the system to the EPROM address space. The memory maps for the development and final application are shown in Figures C-1 and C-2.

System configuration is a straightforward but exacting process. As with any such processes, there are some hints that can make development easier. In addition to care in locating the Root Job in memory, users should fix the initialization job entry point and the data RAM addresses.
The Intel PL/M 86 programming language does not allow a procedure to be used until after it has been declared. This requires the initialization procedure to be declared after all the other procedures. Since the initialization is last, changing the other procedures will change the location of the initialization procedure. If the system entry point changes, the system must be reconfigured. The moving entry point can be circumvented by writing a separate initialization task. The Root Job will create only the initialization task which will then initialize the system jobs. The initialization task entry point is fixed by linking it ahead of the other application tasks and by not changing the initialization task during development. The actual system entry points will be bound to the initialization task during linking and locating. The linking and locating steps are a natural consequence of changing the application code, so binding the fixed system entry point is done automatically during development. The fixed initialization task entry point is used in the configuration file, giving the Root Job an unchanging system entry point.

The remaining moving target during development is the RAM area for data and stack use. If the data and stack RAM is located before or after the application code, with enough extra memory in between for growth during development, the data and stack locations can stay constant. Fixing both the application entry point and the locations of the stack and data segments will allow development of the application code to proceed without requiring frequent reconfigurations.