intel

APPLICATION NOTE

AP-130

March 1982

Using Operating System Firmware Components to Simplify Hardware and Software Design

John Wharton
Microprocessor Applications

© INTEL CORPORATION, 1982

MARCH 1982
ORDER NUMBER: 210295-001

John Wharton is currently the Technical Director of Applications Research, Sunnyvale, California. Please direct any questions or comments to your local FAE (field applications engineer).

CP/M and CP/M-86 are trademarks of Digital Research Incorporated.

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.

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

BXP, CREDIT, i, ICE, iCS, i_m, iMMX, Insite, Intel, $int_e l$, Intelevision,
Intellec, iOSP, iRMX, iSBC, iSBX, Library Manager, MCS,
Megachassis, Micromainframe, Micromap, Multimodule,
Plug-A-Bubble, PROMPT, RMX/80, System 2000 and UPI.

MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Sciences Corporation.

* MULTIBUS is a patented Intel bus.

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

Intel Corporation
Literature Department SV3-3
3065 Bowers Avenue
Santa Clara, CA 95051

© INTEL CORPORATION, 1982

Using Operating System Firmware Components To Simplify Hardware and Software Design

Contents

INTRODUCTION 1

EVOLUTION OF PROCESSOR EXTENSIONS 2
- Real-Time Operating Systems 2

SYSTEM HARDWARE DESIGN 5
- Basic Functional Blocks 5
- 80130 Pin Functions 6
- Additional System Requirements 9
- Timing Considerations 10
- Example System Design 12

APPLICATION SOFTWARE DEVELOPMENT 12
- Hardware Initialization 13
- Simple Time Delays 15
- Stepper Motor Control 15
- Real-Time Interrupt Processing 15
- Mutual Exclusion 17
- Intertask Communication 18
- Segments, Messages, and Mailboxes 19
- Console Command Interpreter 20
- Initialization Task 21
- Code Translation 22

SOFTWARE CONFIGURATION AND INTEGRATION 22
- Configuring OSP Support Code 23
- Linking and Locating Application Jobs 24
- Creating the Root Job 25
- Programming EPROMS 26

SUMMARY 27

APPENDIX A. EXAMPLE SYSTEM SCHEMATICS A-1

APPENDIX B. SOURCE CODE LISTINGS B-1

APPENDIX C. SYSTEM MEMORY MAP C-1

APPENDIX D. SUPPORT CODE LOCATE MAP D-1

APPENDIX E. APPLICATION JOB LOCATE MAP E-1

APPENDIX F. ROOT JOB LOCATE MAP F-1

INTRODUCTION

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

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

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

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

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

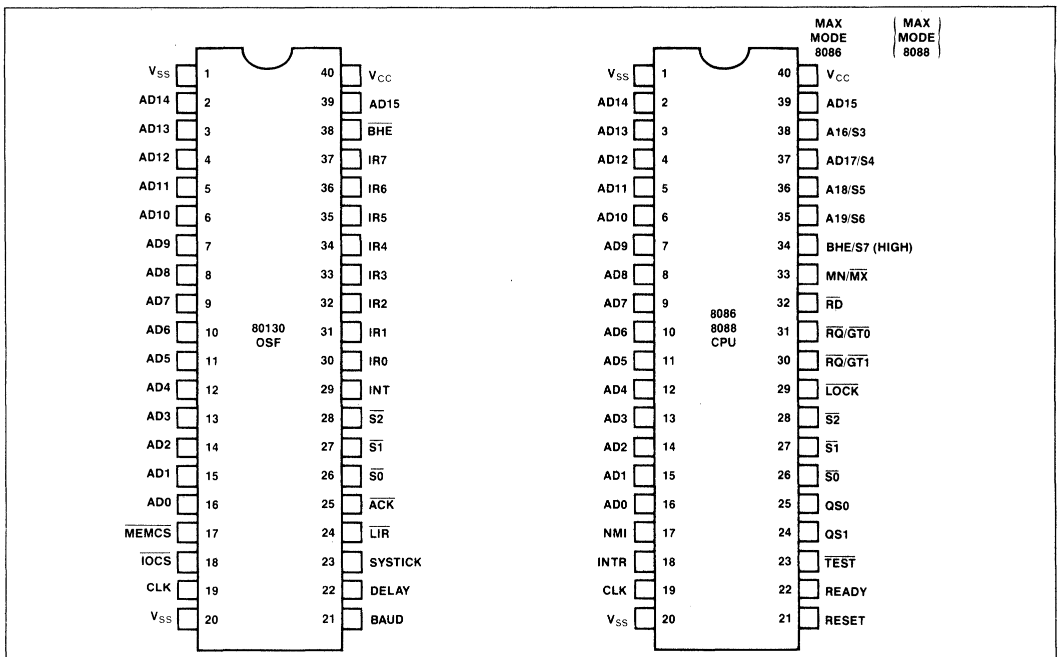


Figure 1. 8086 and 80130 Pinout Diagrams

EVOLUTION OF PROCESSOR EXTENSIONS

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

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

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

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

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

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

1. New applications reveal a need for new types of functionality (in this case, floating-point arithmetic).
2. As common requirements become evident, vendors develop software to serve these needs.
3. Specialized hardware is developed to support the established functions more simply and effectively than software alone.

In time, everything ends up in silicon.

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

What's in a Name?

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

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

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

Real-Time Operating Systems

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

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

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

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

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

A real-time system is one in which the CPU must do many different things (tasks), all more-or-less simultaneously. Unlike the sequential time-sharing of mainframe OSs, though, the tasks are prioritized. Low-priority tasks are preempted if any of higher priority have work to do. The higher-priority task then runs until it must wait for some external event to occur or no longer needs the CPU for some other reason. Thus, the CPU services tasks in their order of importance.

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

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

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

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

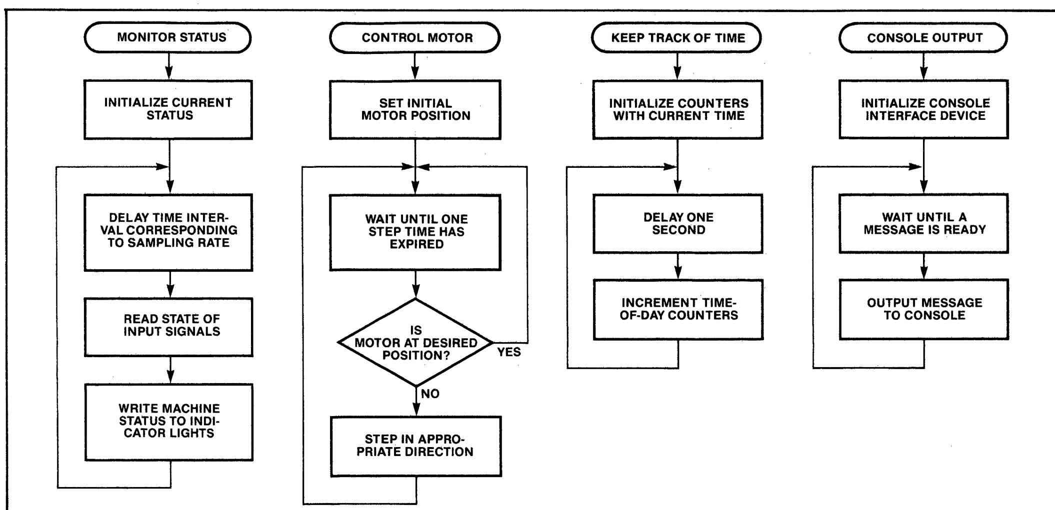


Figure 2. Flowcharts for Concurrent Machine-Tool Tasks

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

Exercise 1. Suppose five tasks are all interrelated. How many communications channels would have to be set up between different processors? If each channel requires two dedicated communication chips, how would the number of peripheral devices compare with the number of CPUs?

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

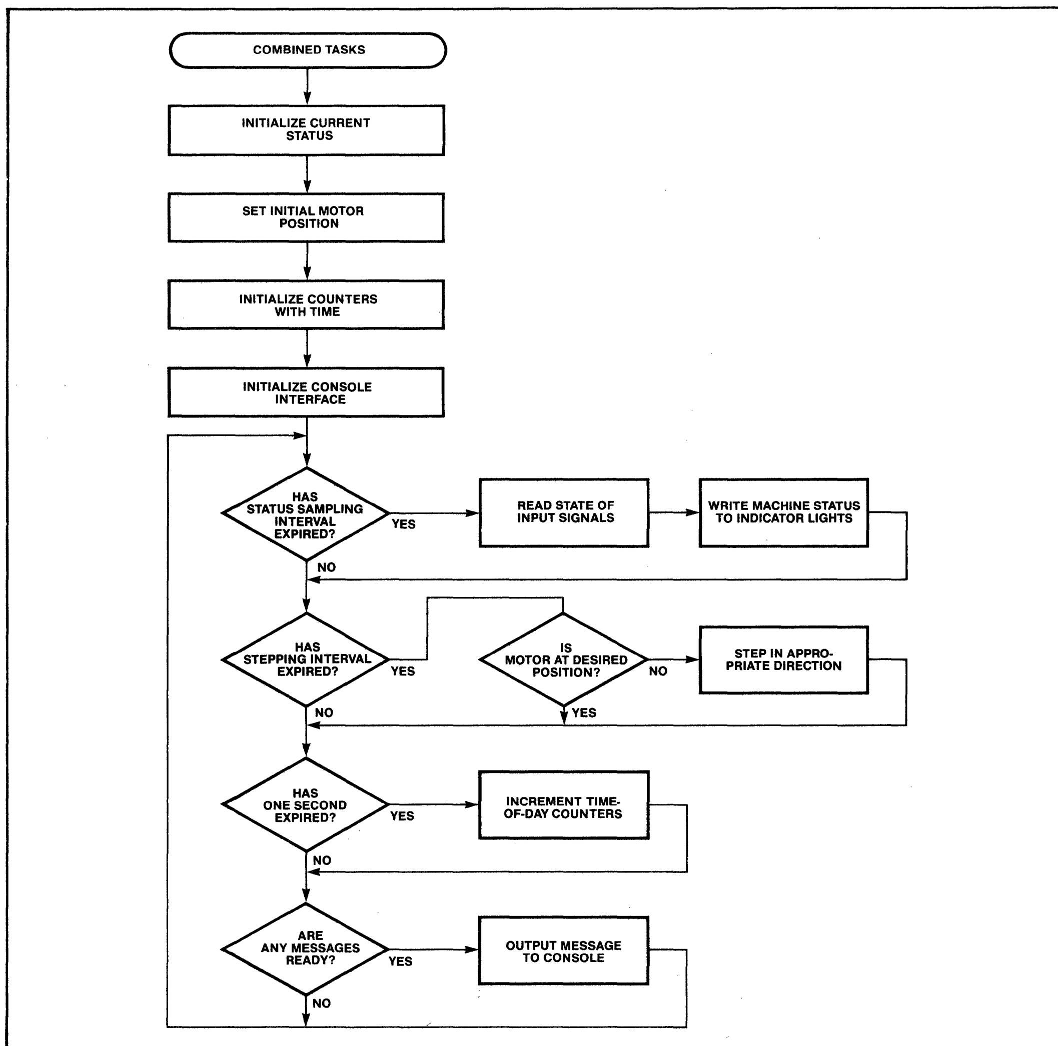


Figure 3. Machine-Tool Tasks Implemented Via Polling Scheme

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

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

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

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

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

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

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

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

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

SYSTEM HARDWARE DESIGN

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

Basic Functional Blocks

In addition to the 80130, the central processing "core" of a typical OSP system would include an 8088 or 8086 operating in maximum mode, an 82843A clock generator, and an 8288 system controller, all connected according to the standard rules. More on the 80130-specific interconnects later.

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

Table 1. Operating System Primitives Supported by 80130

Task Management Suspend Task Resume Task Sleep Create Task Delete Task Set Priority Get Task Tokens	**Interrupt Management** Set Interrupt Signal Interrupt Reset Interrupt Enter Interrupt Wait Interrupt Exit Interrupt Enable Disable Get Level
Intertask Communications and Synchronization Send Message Receive Message Create Mailbox Delete Mailbox	**Free Memory Management/System Partitioning** Create Segment Delete Segment Create Job
Mutual Exclusion Control Receive Control Accept Control Send Control Create Region Delete Region	**Misc. Support** Signal Exception Get Type Disable Deletion Enable Deletion Set O.S. Extension Get Exception Handler Set Exception Handler

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

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

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

80130 Pin Functions

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

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

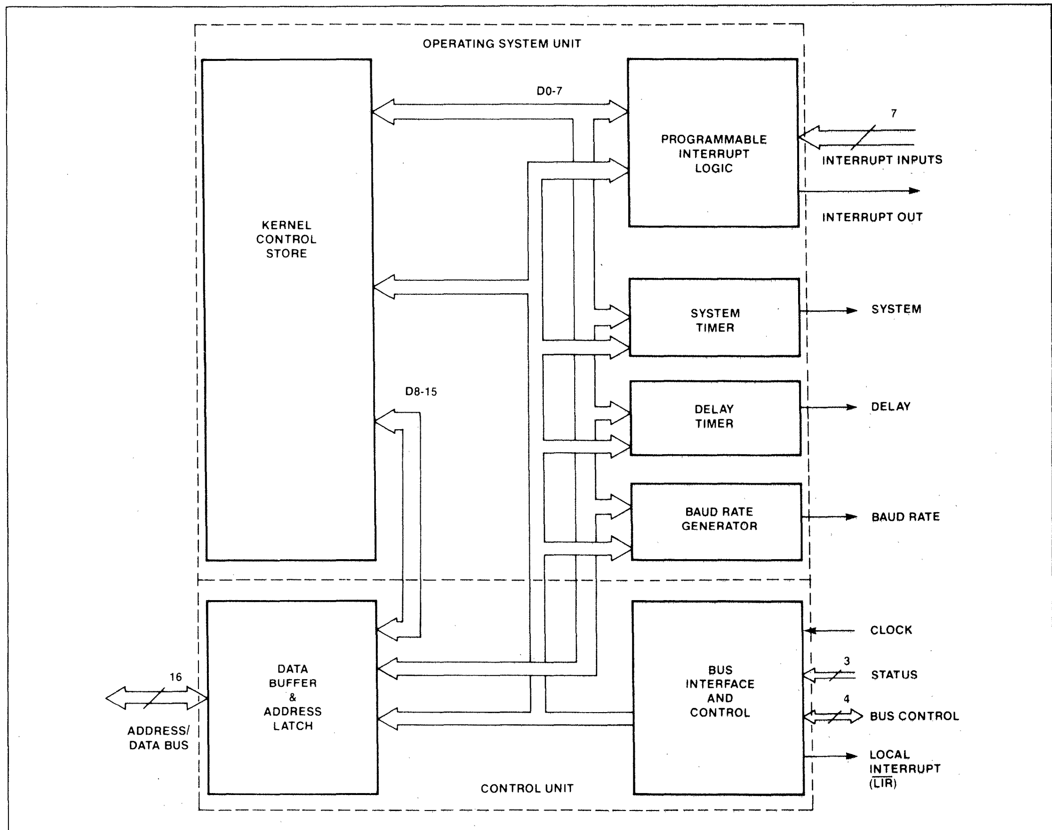


Figure 4. 80130 Internal Block Diagram

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

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

The 80130 also contains dedicated timer logic to provide the OS time base, which is output on SYSTICK. Software operating in conjunction with the 81030 assumes one of the interrupt inputs (INT2 in this case) is driven by SYSTICK, so this connection must be made externally. Routines within the 80130 initialize and perform all bit-level control of the interrupt and timer logic, according to options and parameters specified during the configuration process. Freeing the programmers from this tedium allows them to devote more thought to solving their own unique problems.

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

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

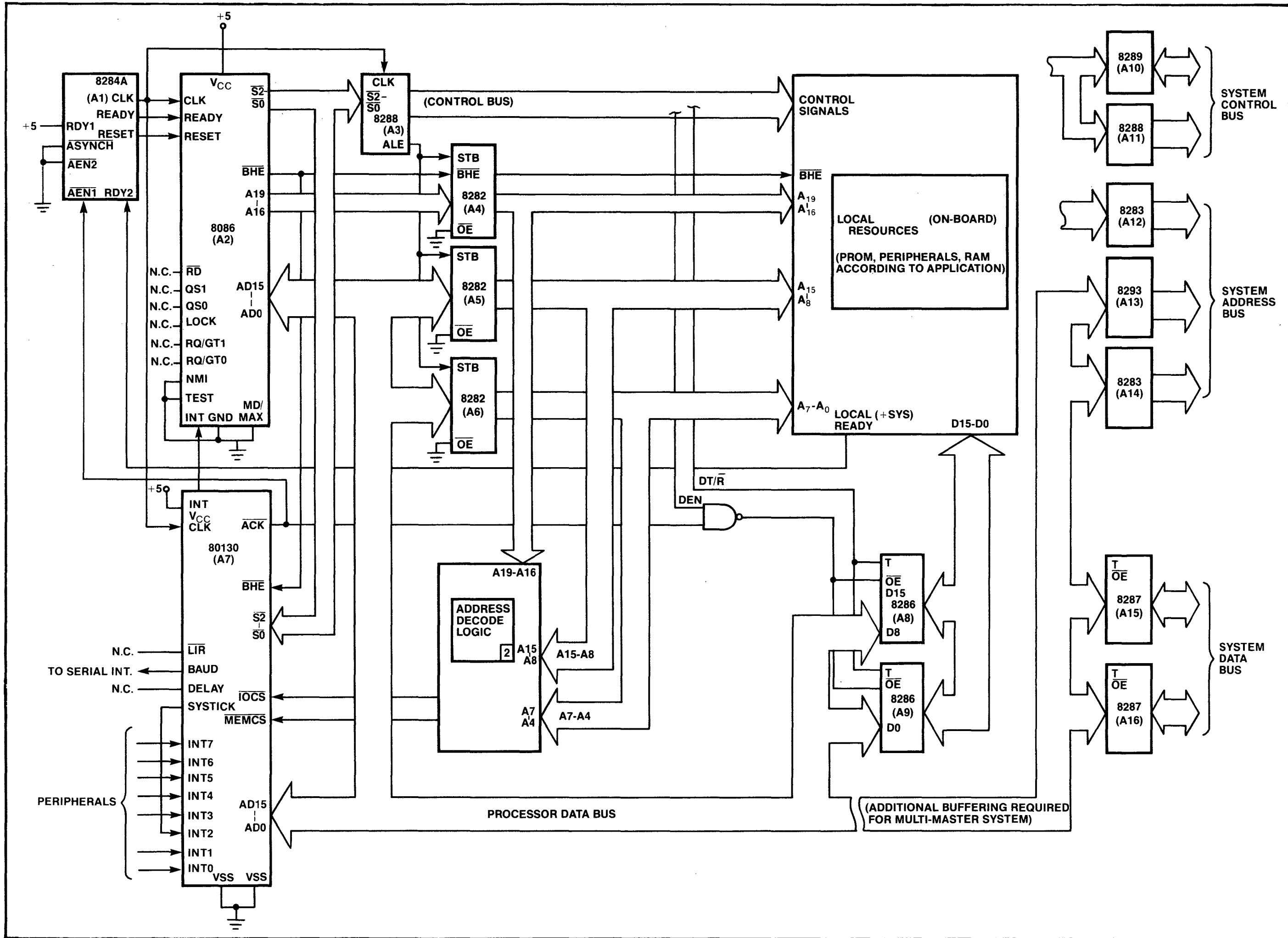


Figure 5. Basic iAPX 86/30 Microcomputer System Block Diagram

AFN-02058A

Additional System Requirements

The OSP requires a certain amount of off-chip memory for its own operation. The system must provide at least 1K bytes of RAM at address 00000H for the CPU interrupt vectors, plus another 1500_{10} bytes for OSP system variables, data structures, stacks, and the like. This RAM may reside anywhere in the 8086 megabyte address space, although it is often contiguous with the interrupt vector up front. Application tasks must each have their own stack, so allow at least an additional 300 bytes of RAM for each.

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

As code is being written, the system designer should count on another 1500 bytes of code from the support libraries being added to his application during the linking and system configuration steps. These memory requirements are shown in Figure 6. In practice, the separate blocks in this figure would be grouped together for more efficient use of RAM and EPROM chips.

The 80130 occupies a 16K-byte block of addresses in the host-processor memory space, so external logic should decode address bits A_{19}–A_{14} to generate $\overline{\text{MEMCS}}$. Similiarly, the timer and interrupt control logic occupy a 16-byte block of addresses in the I/O space; at least some of the bits A_{15}–A_4 must be decoded to generate $\overline{\text{IOCS}}$. The 80130 decodes all the lower-order address bits (14 for memory, four for I/O internally).

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

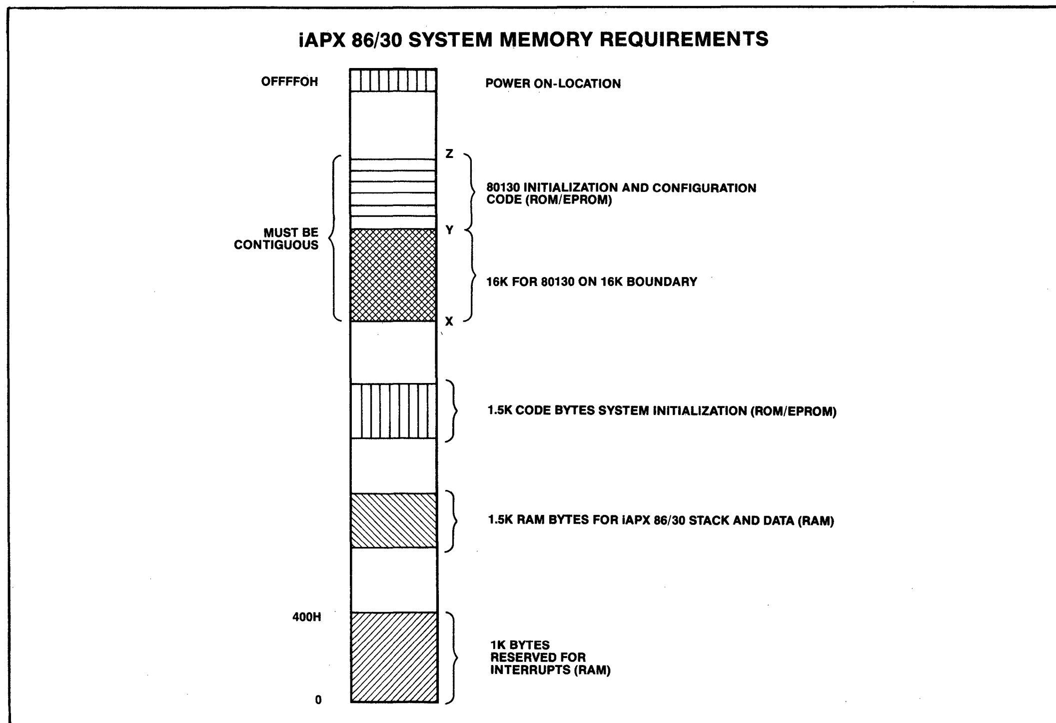


Figure 6. Operating System Processor System Memory Requirements

Timing Requirements

System timing analysis is often the most tedious part of digital hardware design. This discussion can be relatively short, though, because the 80130 timing is quite simple: by design, the part is compatible with the timing of the host processor. Since it interfaces directly with the CPU pins, traditional set-up, hold, and access times no longer matter.

There are really only two areas of concern in analyzing the timing of most OSP systems, both of which relate to the user-generated chip-select signals. Figure 7 illustrates the relevant timing signals of a standard 8086 four-state Read cycle (memory or I/O), along with the timing responses of the 80130. I/O Write cycle timing is the same. (Full timing diagrams are part of the respective data sheets.)

The first concern is that $\overline{MEMCS}$ and $\overline{IOCS}$ must be active early in a memory or I/O cycle if the 80130 is to respond during T_3. In each case, the chip-select signals must be active T_{CSCL} before the end of state T_2. Assuming wait states aren't desired, addresses generated by the CPU must propagate through the address latches and be decoded during T_1 or T_2.

How much time does this leave the decode logic? As we'll see, ample.

By convention, T_{CLAV} is the delay from the start of T_1 until address information is valid on the CPU pins; T_{IVOV} is the propagation delay through an 8282 latch; and T_{CSCL} is the 80130 chip-select set-up time. The mnemonic T_{OVCS} represents the chip-select logic propagation delay, after the latch outputs are stable. The sum of these four delays must be less than two system clock cycles, reduced by the clock transition time.

$$T_{CLAV} + T_{IVOV} + T_{OVCS} + T_{CSCL} \leq T_{CLCL} + T_{CLCL}$$

$$\begin{aligned} T_{OVCS} &\leq T_{CLCL} + T_{CLCL} - T_{CLAV} - T_{IVOV} - T_{CSCL} \\ &\leq 125 + 125 - 60 - 30 - 20 \text{ (nsec.)} \\ &\leq 140 \text{ nsec.} \end{aligned}$$

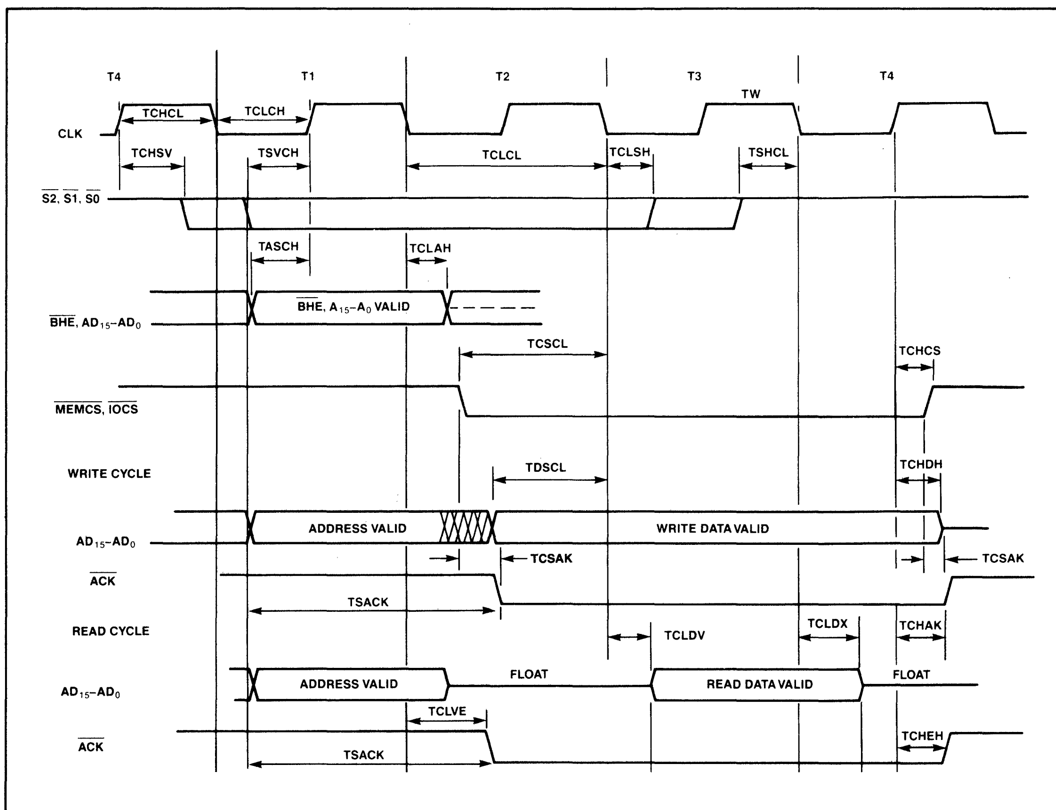


Figure 7. Operating System Processor Timing Diagrams

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

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

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

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

$$T_{CLAV} + T_{OVCS} + T_{CSAK} + R_{R1VCL} \le T_{CLCL} + T_{CLCL}$$

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

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

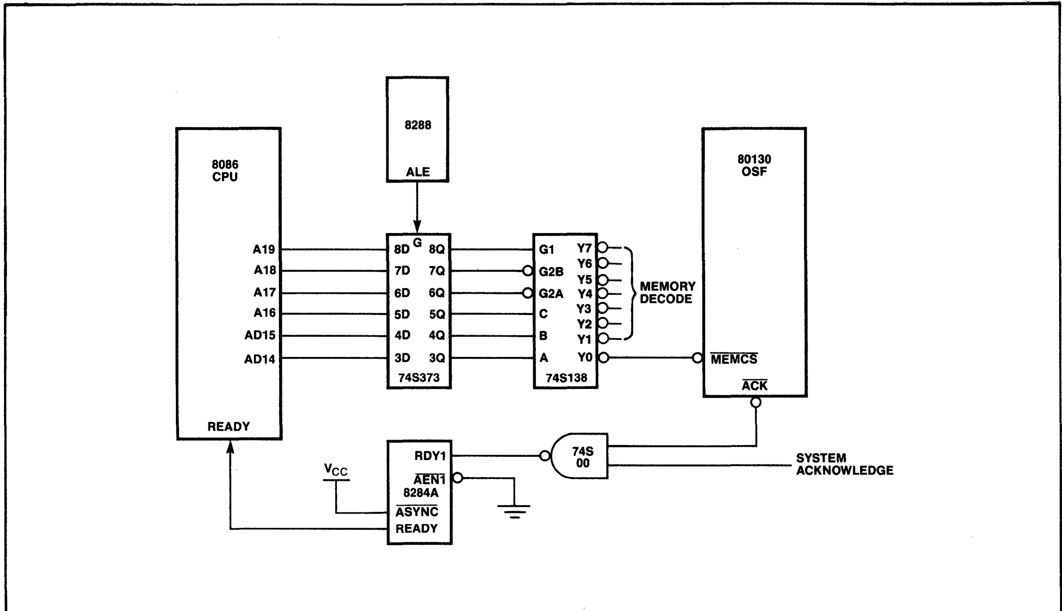


Figure 8. High-Speed Address Decoding Circuit

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

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

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

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

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

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

Example System Design

Appendix A includes full schematics for a complete iAPX 86/30 system providing considerable functionality with only 27 chips. In addition to the OSP, the system has 4K bytes of 2114 RAM (with sockets for another 4K), from 8K to 32K bytes of 2732A or 2764 EPROM, an 8251A USART operating at 9600 baud, and an 8255A Programmable Peripheral Interface with 24 parallel I/O lines. Eight of the inputs read logic values off DIP switches; eight outputs drive small LEDs. Four more outputs connect to the coil drivers of a four-phase stepper motor. A layout diagram of the prototype appears in Figure 9.

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

APPLICATION SOFTWARE DEVELOPMENT

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

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

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

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

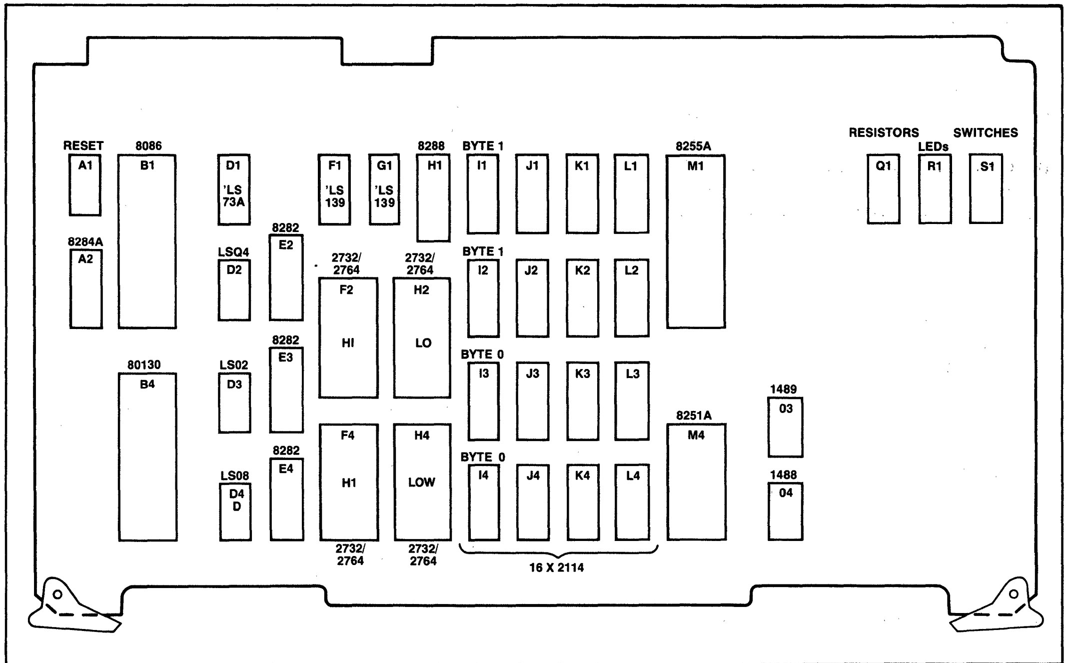


Figure 9. Example System Prototype Layout

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

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

(When a number of tasks cooperate to accomplish some common goal, the collection of tasks is referred to as an application "job." The OSP also allows for an unlimited number of application jobs, though only one is illustrated in the example discussed here. A second similar machine, with different status switches, a different motor, and a different console might make up a second job.)

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

Hardware Initialization

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

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

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

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

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

Example 1. System Hardware Initialization Task

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

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

Example 2. Simple 8251A Input and Output Routines

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

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

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

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

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

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

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

Simple Time Delays

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

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

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

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

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

```
STATUS$TASK:   PROCEDURE;
     DECLARE STATUS$COUNTER BYTE;
     DECLARE STATUS$EXCEPT$CODE WORD;

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

Example 3. Status Polling and Reporting Task

Stepper Motor Control

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

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

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

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

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

Example 4. Stepper-Motor Controller Task

Real-Time Interrupt Processing

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

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

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

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

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

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

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

```
DECLARE AC$CYCLE$COUNT BYTE;

AC$HANDLER:  PROCEDURE INTERRUPT 59;        /*VECTOR FOR 80130 INT3*/
    DECLARE AC$EXCEPT$CODE WORD;

    CALL RQ$ENTER$INTERRUPT(AC$INTERRUPT$LEVEL, @AC$EXCEPT$CODE);
    AC$CYCLE$COUNT=AC$CYCLE$COUNT+1;
    IF AC$CYCLE$COUNT >= 60
        THEN DO;
            AC$CYCLE$COUNT=0;
            CALL RQ$SIGNAL$INTERRUPT(AC$INTERRUPT$LEVEL,
                    @AC$EXCEPT$CODE);
            END;
        ELSE CALL RQ$EXIT$INTERRUPT(AC$INTERRUPT$LEVEL,
            @AC$EXCEPT$CODE);
    END  AC$HANDLER;
```

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

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

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

TIME$TASK:  PROCEDURE;
    DECLARE TIME$EXCEPT$CODE WORD;

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

Example 6. Interrupt Task to Maintain Time of Day

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

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

Mutual Exclusion

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

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

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

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

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

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

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

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

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

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

"Therefore, all CONSOLEOUTTASK has to do to be safe is disable the 60-Hz interrupt around its output routine."

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

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

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

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

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

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

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

```
PROTECTED$CRT$OUT:  PROCEDURE (CHAR) REENTRANT;
    DECLARE CHAR BYTE;
    DECLARE CRT$EXCEPT$CODE WORD;
    CALL RQ$RECEIVE$CONTROL(CRT$REGION$TOKEN,@CRT$EXCEPT$CODE);
    DO WHILE (INPUT(STAT$51) AND 01H)=0;
        /*  NOTHING  */
        END;
    OUTPUT(CHAR$51)=CHAR;
    CALL RQ$SEND$CONTROL(@CRT$EXCEPT$CODE);
    END  PROTECTED$CRT$OUT;
```

Example 7. CRT Output Routine Protected by Region Protocol

As a concession to simplicity, PROTECTED$ CRT$OUTPUT does use a form of the busy waiting method described earlier. The maximum delay at 9600 baud is only one millisecond, however, much shorter than a system tick. Besides, tasks performing character I/O will all have low priority levels, so the OSP would just delay them if anything more urgent comes up.

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

Inter-Task Communication

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

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

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

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

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

length, followed by that number of ASCII characters. Figure 10 shows this format.

```
PRINT$TOD:  PROCEDURE;
    DECLARE TOD$MESSAGE$TOKEN WORD;
    DECLARE TOD$EXCEPT$CODE WORD;
    DECLARE TOD$SEGMENT$OFFSET WORD,
        TOD$SEGMENT$BASE WORD;
    DECLARE TOD$SEGMENT$PNTR POINTER AT (@TOD$SEGMENT$OFFSET);
    DECLARE TOD$TEMPLATE (28) BYTE
        DATA (27,'THE TIME IS NOW hh:mm:ss.',CR,LF);
    DECLARE TOD$STRING BASED TOD$SEGMENT$PNTR (28) BYTE;
    DECLARE TOD$STRING$INDEX BYTE;

    TOD$MESSAGE$TOKEN=RQ$CREATE$SEGMENT(28,@TOD$EXCEPT$CODE);
    TOD$SEGMENT$BASE=TOD$MESSAGE$TOKEN;
    TOD$SEGMENT$OFFSET=0;
    DO TOD$STRING$INDEX=0 TO 27;
        TOD$STRING(TOD$STRING$INDEX)=
                TOD$TEMPLATE(TOD$STRING$INDEX);
        END;
    TOD$STRING(17)=ASCII$CODE(HOUR$COUNT/10);
    TOD$STRING(18)=ASCII$CODE(HOUR$COUNT MOD 10);
    TOD$STRING(20)=ASCII$CODE(MINUTE$COUNT/10);
    TOD$STRING(21)=ASCII$CODE(MINUTE$COUNT MOD 10);
    TOD$STRING(23)=ASCII$CODE(SECOND$COUNT/10);
    TOD$STRING(24)=ASCII$CODE(SECOND$COUNT MOD 10);
    CALL RQ$SEND$MESSAGE(CRT$MAILBOX$TOKEN,
        TOD$MESSAGE$TOKEN,0,@TOD$EXCEPT$CODE);
    RETURN;
    END  PRINT$TOD;
```

Example 8. Subroutine to Send Time-of-Day Message to Output Task

We're coding PRINT$TIME here (see Example 8), while TIME$TASK is fresh in our minds. It will actually be called by (and is therefore considered a part of) KEYBOARD$TASK. Note that while tasks are written as individual procedures, they need not be fully self-contained: outside procedures should be used to help organize and structure the code.

The first thing PRINT$TIME does is have the OSP create a segment of suitable length, and copies a "message template" into the segment, byte by byte. Then it converts the TIME$TASK counter values to ASCII, filling in blanks in the template. Finally, it sends the token for the message to the CRT mailbox.

To repeat, these examples are intended to illustrate use of the OSP routines assuming minimum familiarity with PL/M. Better programming practices might take advantage of PL/M literals, structures and the array LENGTH function to build the message, rather than the inflexible constants shown here. Some of these techniques are suggested by PRINT$STATUS (Example 9), which indicates the binary status of the input switches.

```
PRINT$STATUS:  PROCEDURE;
    DECLARE STATUS$MESSAGE$TOKEN WORD;
    DECLARE STATUS$EXCEPT$CODE WORD;
    DECLARE STATUS$SEGMENT$OFFSET WORD,
        STATUS$SEGMENT$BASE WORD;
    DECLARE STATUS$SEGMENT$PNTR POINTER
        AT (@STATUS$SEGMENT$OFFSET);
    DECLARE STATUS$TEMPLATE (40) BYTE DATA
        (39,'THE SWITCHES ARE NOW SET TO ........B',CR,LF);
    DECLARE STATUS$STRING BASED STATUS$SEGMENT$PNTR (40) BYTE;
    DECLARE STATUS$STRING$INDEX BYTE;
    DECLARE BIT$PATTERN BYTE;

    STATUS$MESSAGE$TOKEN=RQ$CREATE$SEGMENT(40,
            @STATUS$EXCEPT$CODE);
    STATUS$SEGMENT$BASE=STATUS$MESSAGE$TOKEN;
    STATUS$SEGMENT$OFFSET=0;
    DO STATUS$STRING$INDEX=0 TO 39;
        STATUS$STRING(STATUS$STRING$INDEX)=
            STATUS$TEMPLATE(STATUS$STRING$INDEX);
        END;
    BIT$PATTERN=INPUT(PPI$A);
    DO STATUS$STRING$INDEX=29 TO 36;
        STATUS$STRING(STATUS$STRING$INDEX)=
            ASCII$CODE(BIT$PATTERN AND 01H);
        BIT$PATTERN=ROR(BIT$PATTERN,1);
        END;
    CALL RQ$SEND$MESSAGE(CRT$MAILBOX$TOKEN,
        STATUS$MESSAGE$TOKEN,0,@STATUS$EXCEPT$CODE);
    END  PRINT$STATUS;
```

Example 9. Subroutine to Send Status Report Message to Output Task

Exercise 10: One input port is read by both STATUS$TASK and PRINT$STATUS. Does this constitute a shared resource? A critical region?

Exercise 11: PRINT$TIME reads the counts maintained by TIME$TASK, but doesn't alter them. Forced mutual exclusion is generally mandatory when multiple tasks perform read/modify/write sequences on a given variable. Can PRINT$TIME make TIME$TASK malfunction? What about the opposite case? If this failure mode was deemed unacceptable, how could it be protected?

Mailboxes

The data in a message doesn't actually move or get copied from source to destination when the message is sent; this would be too slow with long messages. Rather, the OSP "carries" the message's token from task to task via a data structure cleverly termed a mailbox. If one task must send messages to another, a mailbox must be created to hold them. The sender calls the RQ$SEND$MESSAGE to put a message token into the mailbox. If the receiver isn't ready for the message yet, the OSP puts the message token into an ordered queue. When the receiver calls RQ$

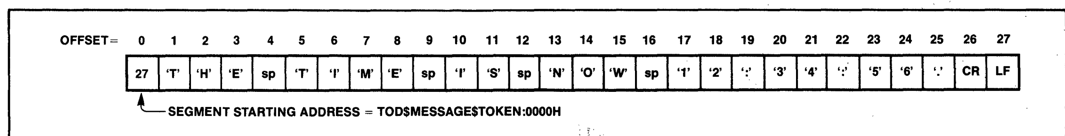


Figure 10. Message Formats Expected by Output Task

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

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

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

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

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

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

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

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

Example 10. Task to Transmit Messages to the CRT

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

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

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

Console Command Interpreter

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

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

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

Initialization Task

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

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

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

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

```
COMMAND$TASK:   PROCEDURE;
    DECLARE CONSOLE$CHAR BYTE;
    DECLARE COMMAND$EXCEPT$CODE WORD;

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

Example 11. Task to Accept and Process Keyboard Commands

```
INIT$TASK: PROCEDURE PUBLIC;
    DECLARE INIT$EXCEPT$CODE WORD;

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

Example 12. Task to Initialize System Software

Table 2. Special Console Commands

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

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

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

Code Translation

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

High-Level Parameter Passing Conventions

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

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

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

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

SOFTWARE CONFIGURATIONS & INTEGRATION

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

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

The three phases of the configuration are:

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

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

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

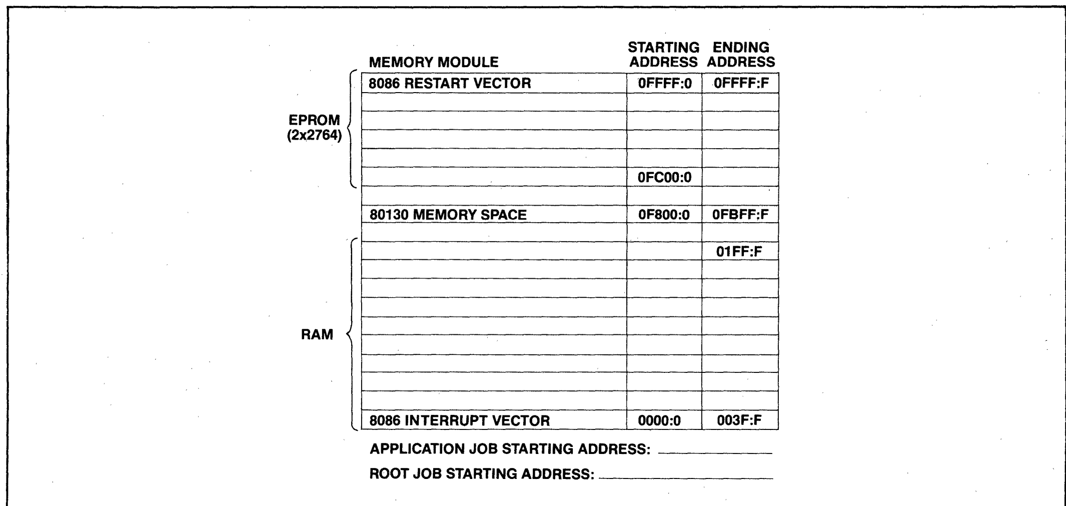
The prototype system memory space has two areas of interest: addresses 00000H through 01FFFH contain RAM, while 0FC000H through 0FFFFFH contain EPROM. Since the CPU uses the first 1K bytes of RAM for the CPU interrupt pointers, and the last 16 bytes for the restart sequence, these areas should be recorded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

Generating the OSP Support Code

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

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

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



	MEMORY MODULE	STARTING ADDRESS	ENDING ADDRESS
EPROM (2x2764)	8086 RESTART VECTOR	0FFFF:0	0FFFF:F
		0FC00:0	
	80130 MEMORY SPACE	0F800:0	0FBFF:F
RAM			01FF:F
	8086 INTERRUPT VECTOR	0000:0	003F:F

APPLICATION JOB STARTING ADDRESS: ____________

ROOT JOB STARTING ADDRESS: ____________

Figure 11. Example System Memory Map

```
$TITLE(80130 DEVICE CONFIGURATION TABLE)
NAMEODEVCF

$INCLUDE(:F1:NDEVCF.MAC)

%MASTER_PIC(80130, 2000H, 0, 0)

; SLAVE_PIC( SLAVE_TYPE, BASE_PORT, EDGE_VS_LEVEL, MASTER_LEVEL )

%TIMER(80130, 2008H, 28H, 12500)

; NDP_SUPPORT( ENCODED_LEVEL )

END
```

Figure 12. 80130 Device Configuration Table

```
:F0:ASM86 :F1:SUP130.A86 PRINT(:F1:SUP130.LST) ERRORPRINT &
MACRO(80) PAGEWIDTH(132)
;
:F0:LINK86 &
:F1:OSX.LIB(OSX86,OSXCNF), &
:F1:NUC1.LIB(NBEGIN),      &
:F1:ODEVCF.OBJ,            &
:F1:OSX.LIB,               &
:F1:NUC1.LIB,              &
:F1:OSX.LIB,               &
:F1:NUC2.LIB,              &
:F1:OSX.LIB,               &
:F1:NUC4.LIB,              &
:F1:OSX.LIB,               &
:F1:NURSLV.LIB,            &
:F1:OSX.LIB                &
TO :F1:SUP130.LNK MAP PRINT(:F1:SUP130.MP1) NAME(MINIMAL_80130)
;
:F0:LOC86                                         &
:F1:SUP130.LNK TO :F1:SUP130 MAP PRINT(:F1:SUP130.MP2) SC(3) &
SEGSIZE(STACK(0))                                 &
ADDRESSES(CLASSES(CODE(0F8000H),DATA(00400H)))       &
ORDER(CLASSES(DATA,STACK))                        &
OBJECTCONTROLS(NOLINES,NOCOMMENTS,NOSYMBOLS)
;
```

Figure 13. Support Code Configuration Commands

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

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

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

Application Code Configuration

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

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

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

Creating the Root Job

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

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

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

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

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

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

OH86 :F1:AP130 TO :F1:AP130.H86

COPY :F1:AP130.MP1 TO :LP:

COPY :F1:AP130.MP2 TO :LP:
```

Figure 14. Job Configuration Commands

```
;
;SOURCE PROGRAM DEFINING CHARACTERISTICS OF ROOT JOB FOR
;AP-130 DEMONSTRATION PROGRAM  (JHW - 10/25/81)
;
$INCLUDE(:F1:CTABLE.MAC)
;
%SAB(0,00C0,U)
%SAB(0200,FFFF,U)
%JOB(0,0C0H,100H,0FFFFH,0FFFFH,1,0:0,1,0,100,0FC62:06B5,0,0:0,200H,0)
%OSX     (0F8000H,N)
%SYSTEM(F800,0,4,N,N,1)
;
END
```

Figure 15. Root Job Configuration File

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

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

EPROM Programming

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

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

Next, all the final code modules produced above (e.g., SUP130, AP130, and RJB130) must be loaded into the UPM memory buffer. The three commands in Figure 18 perform this function.

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

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

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

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

```
;
;   LINK AND LOCATE THE iRMX 86 ROOT JOB.
;
;   MODIFIED FOR TWO-DRIVE OPERATION
;   REVISED 10/25 - JHW
;
ASM86 :f1:RJB130.A86 MACRO(75)
;
LINK86                                  &
        :f1:croot.lib(root),            &
        :f1:RJB130.obj,                 &
        :f1:croot.lib                   &
        TO   :f1:RJB130.lnk             &
        MAP PRINT(:f1:RJB130.mp1)
;
LOC86 :f1:RJB130.lnk                    &
       TO :F1:RJB130                    &
       MAP PRINT(:f1:RJB130.mp2)        &
       OC(noli, nopl, nocm, nosb)       &
       PC(noli, pl, nocm, nosb)         &
       SEGSIZE(stack(0))                &
       ORDER(classes(data, stack, memory))          &
       ADDRESSES(classes(code(OFD180H),             &
                          data(00AD0H)))
;
OH86 :F1:RJB130 TO :F1:RJB130.H86
;
COPY :F1:RJB130.LST TO :LP:
;
COPY :F1:RJB130.MP1 TO :LP:
;
COPY :F1:RJB130.MP2 TO :LP:
;
```

Figure 16. Root Job Configuration Commands

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

Figure 17. UPM Initialization Sequence

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

Figure 18. UPM Commands to Load Hex Files

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

Figure 19. UPM Command to Patch Restart Vector

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

Figure 20. UPM Commands to Strip High and Low Bytes

```
program from 4000h to 5fffh start 0

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

Figure 21. UPM Commands to Program EPROMs

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

SUMMARY

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

The 80130 gives the software engineer built-in support for 35 standard operating system primitives. Application problems may now be solved at a higher level than before. It is now possible for concurrent tasks to be dispatched, memory segments allocated, and messages relayed through mailboxes nearly as easily as subroutines, dynamic variables, and I/O ports were used in the past. In effect, Jobs, Tasks, Segments, Mailboxes, and Regions become new OSP data types, manipulated entirely by firmware in the 80130.

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

ACKNOWLEDGEMENTS

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

APPENDIX A
EXAMPLE SYSTEM SCHEMATICS

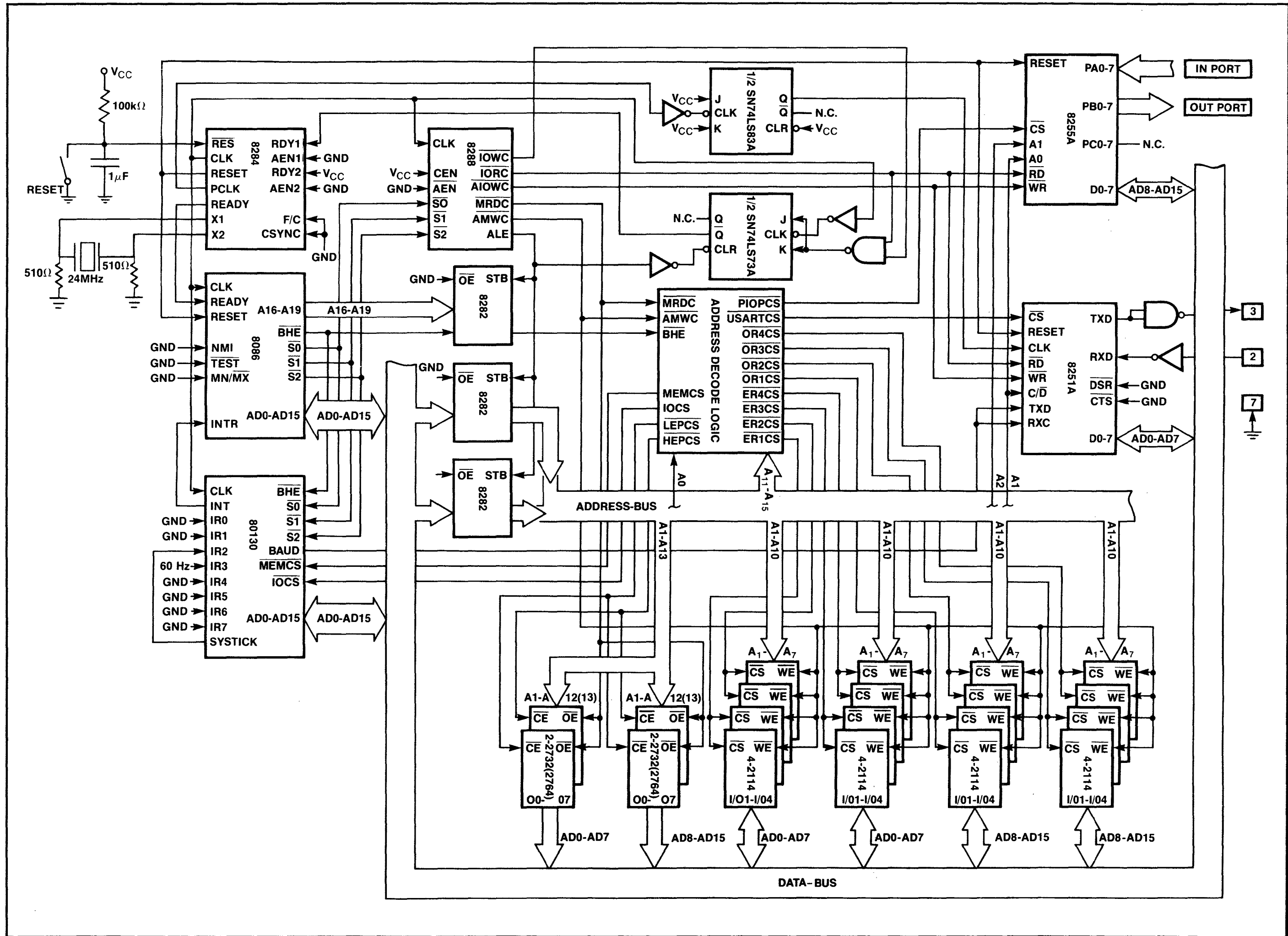


Figure A-1. Example System Schematics

AFN-02058A

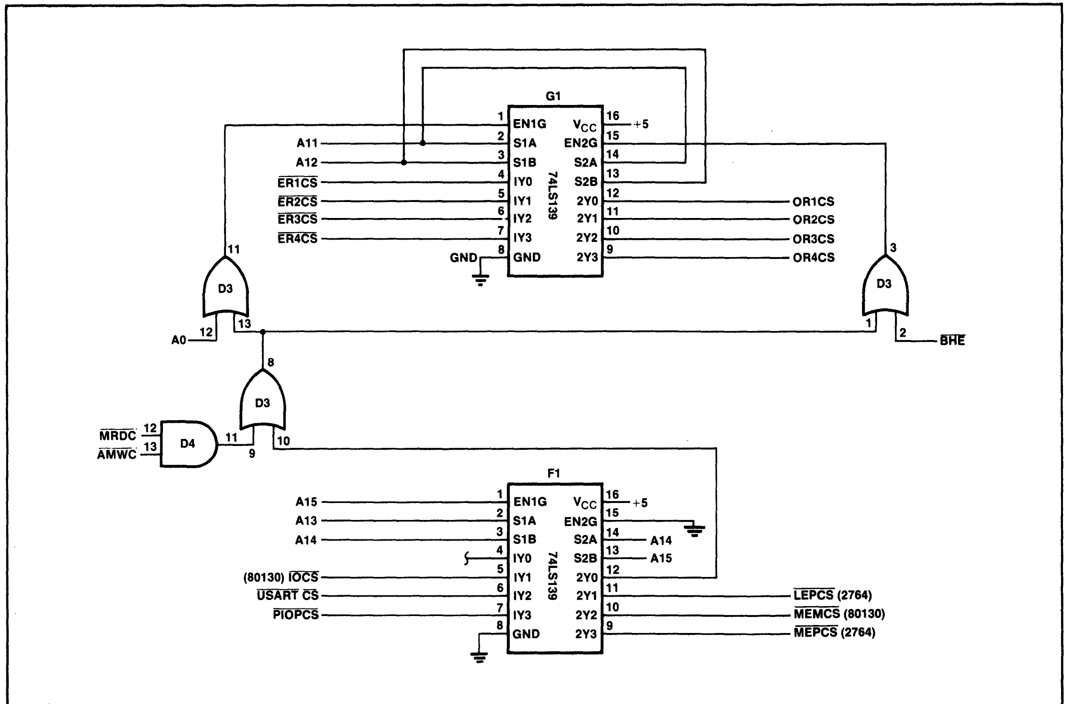


Figure A-1. Example System Schematics (continued)

AFN-02058A

APPENDIX B
SOURCE CODE LISTINGS

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


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

   1            DEMO$130:   DO;

                /*  SYSTEM-WIDE LITERAL DECLARATIONS:  */

   2    1       DECLARE FOREVER LITERALLY 'WHILE 01H';

                /*  I/O PORT DEFINITIONS:  */

   3    1       DECLARE CHAR$51 LITERALLY '4000H',
                    CMD$51 LITERALLY '4002H',
                    STAT$51 LITERALLY '4002H';

   4    1       DECLARE PPI$A LITERALLY '6001H',
                    PPI$B LITERALLY '6003H',
                    PPI$C LITERALLY '6005H',
                    PPI$CMD LITERALLY '6007H',
                    PPI$STAT LITERALLY '6007H';

   5    1       DECLARE TIMER$CMD LITERALLY '200EH',
                    BAUD$TIMER LITERALLY '200CH';

   6    1       DECLARE AC$INTERRUPT$LEVEL LITERALLY '00111000B';

   7    1       DECLARE CR LITERALLY '0DH',
                    LF LITERALLY '0AH',
                    BEL LITERALLY '07H';

   8    1       DECLARE ASCII$CODE (16) BYTE DATA ('0123456789ABCDEF');


                $EJECT

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

                /*  GLOBAL VARIABLE DECLARATIONS:  */

 299    1       DECLARE DATA$SEG$PTR POINTER,
                    DATA$SEG$ADDR STRUCTURE (OFFSET WORD,BASE WORD)
                        AT (@DATA$SEG$PTR);

 300    1       DECLARE HARDWARE$INIT$TASK$TOKEN WORD,
                    STATUS$TASK$TOKEN WORD,
                    MOTOR$TASK$TOKEN WORD,
                    TIME$TASK$TOKEN WORD,
                    AC$HANDLER$TOKEN WORD,
                    CRT$OUT$TASK$TOKEN WORD,
                    COMMAND$TASK$TOKEN WORD,
                    INIT$TASK$TOKEN WORD;

 301    1       DECLARE CRT$MAILBOX$TOKEN WORD,
                    CRT$REGION$TOKEN WORD;
```

```
          $EJECT

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

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

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


          $EJECT

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

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

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


          $EJECT

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

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

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

```
              $EJECT

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

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

344   1       MOTOR$TASK:  PROCEDURE;
345   2           DECLARE MOTOR$EXCEPT$CODE WORD;
346   2           DECLARE MOTOR$POSITION BYTE,
                      MOTOR$PHASE BYTE;
347   2           DECLARE PHASE$CODE (4) BYTE
                      DATA (00000101B,00000110B,00001010B,00001001B);

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


              $EJECT

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

368   1       DECLARE AC$CYCLE$COUNT BYTE;

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

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

```
              $EJECT

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

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


              $EJECT

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

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

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

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


AFN-02058A

```
                $EJECT

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

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

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


                $EJECT

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

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

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

```
            $EJECT

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

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

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


            $EJECT

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

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

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

```
            $EJECT

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

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

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


521   1     END  DEMO$130;


MODULE INFORMATION:

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

END OF PL/M-86 COMPILATION
```

APPENDIX C
SYSTEM MEMORY MAP

EXAMPLE SYSTEM MEMORY MAP

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

INITIALIZATION TASK STARTING ADDRESS: FC62:06B5

ROOT JOB STARTING ADDRESS: FD18:0011

AFN-02058A

APPENDIX D
SUPPORT CODE LOCATE MAP

```
ISIS-II MCS-86 LOCATER, V1.2 INVOKED BY
:F0:LOC86                                                &
:F1:SUP130.LNK TO :F1:SUP130 MAP PRINT(:F1:SUP130.MP2) SC(3) &
SEGSIZE(STACK(0))                                        &
ADDRESSES(CLASSES(CODE(0F8000H),DATA(00400H)))           &
ORDER(CLASSES(DATA,STACK))                               &
OBJECTCONTROLS(NOLINES,NOCOMMENTS,NOSYMBOLS)
WARNING 26:  DECREASING SIZE OF SEGMENT
  SEGMENT:   STACK

SYMBOL TABLE OF MODULE MINIMAL_80130
READ FROM FILE :F1:SUP130.LNK
WRITTEN TO FILE :F1:SUP130

BASE   OFFSET TYPE SYMBOL              BASE   OFFSET TYPE SYMBOL              BASE   OFFSET TYPE SYMBOL

0040H  0000H  PUB  INTERRUPTTASKVEC    0040H  0120H  PUB  DEFAULT_HANDLER     0040H  0144H  PUB  READYLISTROOT
0040H  0148H  PUB  INTERRORENTRY       0040H  014CH  PUB  SYSTEMEXCEPTIONH    0040H  0150H  PUB  DELETIONTASKTOKE
                                                          -ANDLERPTR                                -N
0040H  0152H  PUB  EXTENSIONLISTROO    0040H  0154H  PUB  DELETION_OBJECT_    0040H  0156H  PUB  SYSTEMPOOLTOKEN
                   -T                                     -BASE
0040H  0158H  PUB  ROOTJOBTOKEN        0040H  015AH  PUB  MINTRANSSIZE        0040H  015CH  PUB  LAST_NDP_TASK
0040H  015EH  PUB  NDP_INTERRUPT_LE    0040H  0160H  PUB  PARAM_VALIDATION    0040H  0162H  PUB  REGION_FLAGS
                   -VEL_VAR                               -_VECTOR
0040H  0164H  PUB  TASK_WAITING_FLA    0040H  0166H  PUB  REGION_TOKEN_TAB    0040H  0176H  PUB  SIGNAL_Q_INDEX
                   -GS                                    -LE
0040H  0178H  PUB  SIGNAL_Q            0040H  01E8H  PUB  KERNEL_FLAG         0040H  01E9H  PUB  ACTIVATE_SIGNAL_
                                                                                                -Q
0040H  01EAH  PUB  FILLCHAR            0040H  01EBH  PUB  NUM_SLAVES          0040H  01ECH  PUB  OLD_SLAVE_NUM
0040H  01EDH  PUB  INTMASK             0040H  01F6H  PUB  DISABLEMASK         0040H  01FFH  PUB  LEVEL_SET_TABLE
0040H  0208H  PUB  IMR_PORT            0040H  021AH  PUB  EOI_PORT            0040H  022CH  PUB  ISR_PORT
0040H  023EH  PUB  PIC_INFO            0040H  0247H  PUB  CLOCK_SPEC_EOI      0040H  0248H  PUB  CLOCK_ON
0040H  0249H  PUB  CLOCK_OFF           0040H  024AH  PUB  CLOCK_LEVEL         0040H  05D0H  PUB  END_OF_DATA
F800H  45CCH  PUB  NDP_INTERRUPT_LE    F800H  45C2H  PUB  VALIDATE_PARAMS_    F800H  4542H  PUB  GETDESCRTOKEN
                   -VEL                                   -BODY_DUMMY
F800H  4556H  PUB  GETDESCRPOINTER     F800H  4567H  PUB  GETPOINTER          F800H  453DH  PUB  SCANMEMORY
F800H  4538H  PUB  OVERFLOW            F800H  4533H  PUB  NENTRY_BODY         F800H  452EH  PUB  KSUSPEND
F800H  4529H  PUB  KINITIALIZE         F800H  4524H  PUB  KENABLELEVELNS      F800H  451FH  PUB  KENABLELEVEL
F800H  451AH  PUB  KCREATEREGIONNS     F800H  4515H  PUB  KCREATEOBJECTNS     F800H  4510H  PUB  KCREATEOBJECT
F800H  450BH  PUB  INITNDP             F800H  4506H  PUB  INITIALIZE          F800H  4501H  PUB  FINISHINITIALIZA
                                                                                                -TION
F800H  44FCH  PUB  EOI_ROUTINE         F800H  44F7H  PUB  DIVIDEBYZERO        F800H  44F2H  PUB  DECODE_LEVEL
F800H  44EDH  PUB  COMMON_ERROR        F800H  44E8H  PUB  CLOCKENTRY_BODY     F800H  44E3H  PUB  ARRAYBOUNDS
F800H  44D0H  PUB  SYSTEMEXCEPTIONH    F800H  4472H  PUB  INITIALIZE_TIMER    F800H  43AEH  PUB  INITIALIZE_PICS
                   -ANDLER
F800H  435CH  PUB  INIT_INTERNAL_RE    F800H  434EH  PUB  NDP_INTERRUPT_HA    F800H  433FH  PUB  CLOCKENTRY
                   -GIONS                                 -NDLER
F800H  4336H  PUB  NENTRY              F800H  40FEH  PUB  INITIALIZENUCLEU    F800H  40B6H  PUB  RQWAITINTERRUPT_
                                                          -S                                    -BODY
F800H  40B1H  PUB  RQSIGNALINTERRUP    F800H  40ACH  PUB  RQGETLEVEL_BODY     F800H  40A7H  PUB  RQEXITINTERRUPT_
                   -T_BODY                                                                      -BODY
F800H  40A2H  PUB  RQENTERINTERRUPT    F800H  409DH  PUB  RQDISABLE_BODY      F800H  4094H  PUB  RQWAITINTERRUPT
                   -_BODY
F800H  408AH  PUB  RQSIGNALINTERRUP    F800H  4080H  PUB  RQGETLEVEL          F800H  4076H  PUB  RQENTERINTERRUPT
                   -T
F800H  406CH  PUB  RQEXITINTERRUPT     F800H  4062H  PUB  RQDISABLE           F800H  405DH  PUB  NUNLOCK_DELETION
                                                                                                -_OBJECT
F800H  4058H  PUB  NUNLOCKNS           F800H  4053H  PUB  NUNLOCK             F800H  404EH  PUB  NOPEN_DELETION_O
                                                                                                -BJECT
F800H  4049H  PUB  NOPENNS             F800H  4044H  PUB  NOPEN               F800H  403FH  PUB  NLOCK_DELETION_O
                                                                                                -BJECT
F800H  403AH  PUB  NLOCKNS             F800H  4035H  PUB  NLOCK               F800H  4030H  PUB  NCLOSE_DELETION_
                                                                                                -OBJECT
F800H  402BH  PUB  NCLOSENS            F800H  4026H  PUB  NCLOSE              F800H  4021H  PUB  DELETERUNNINGTAS
                                                                                                -K
F800H  401CH  PUB  DELETEOBJECT        F800H  400AH  PUB  COPYRIGHT           F800H  4000H  PUB  NBEGIN
F800H  4000H  PUB  INIT_NUCLEUS_JUM    FC5DH  0004H  PUB  IMR_START           FC5CH  000EH  PUB
                   -P
FC5CH  000FH  PUB  INIT_CMD1           FC5CH  0010H  PUB  INIT_CMD5_MASTER    FC5CH  0011H  PUB
FC5CH  0012H  PUB  INIT_CMD4_MASTER    FC61H  000EH  PUB  SLAVE_TABLE         FC61H  0003H  PUB

FC61H  0005H  PUB  CLOCK_0_PORT        FC61H  0007H  PUB  CLOCK_COUNT         FC61H  000AH  PUB
FC61H  000BH  PUB  C_CLOCK_SPEC_EOI    FC61H  000CH  PUB  C_CLOCK_ON          FC61H  0009H  PUB
F800H  4576H  PUB  LEVEL7_HANDLER      F800H  4574H  PUB  PARAM_VALIDATION
                                                          -_PATH

MEMORY MAP OF MODULE MINIMAL_80130
READ FROM FILE :F1:SUP130.LNK
WRITTEN TO FILE :F1:SUP130

SEGMENT MAP

START     STOP      LENGTH ALIGN NAME               CLASS

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

```
00A60H    00A6FH    0010H    G    DELETION_REG_S    DATA   <---- LAST RAM BYTE USED
                                  -EG

00A70H    00A70H    0000H    W    STACK             STACK
00A70H    00A70H    0000H    G    ??SEG
F8000H    FC5CDH    45CEH    W    CODE              CODE
FC5CEH    FC5D2H    0005H    W    PIC_CNF_SEG       CODE
FC5D4H    FC5E5H    0012H    W    _IMR_PORT         CODE
FC5E6H    FC5F7H    0012H    W    _EOI_PORT         CODE
FC5F8H    FC609H    0012H    W    _ISR_READ_PORT    CODE
FC60AH    FC612H    0009H    B    _PIC_INFO         CODE
FC613H    FC61CH    000AH    B    TIMER_CNF_SEG     CODE
FC61EH    FC61EH    0000H    W    CSEG              CODE

FC61EH    FC61FH    0002H    W    SLAVE_SEG         CODE   <---- LAST EPROM BYTE USED

FC620H    FC620H    0000H    W    MEMORY            MEMORY


GROUP MAP

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

           SLAVE_SEG
```

APPENDIX E
APPLICATION JOB LOCATE MAP

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

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

BASE   OFFSET TYPE SYMBOL                BASE   OFFSET TYPE SYMBOL

FC62H  0B3AH  PUB  RQENDINITTASK         FC62H  0B1CH  PUB  RQ_N_C_RETURN_40
FC62H  0B00H  PUB  RQ_N_C_RETURN_20      FC62H  0AE4H  PUB  RQ_N_C_RETURN_14
FC62H  0AC8H  PUB  RQ_N_C_RETURN_12      FC62H  0AACH  PUB  RQ_N_C_RETURN_10
FC62H  0A90H  PUB  RQ_N_C_RETURN_8       FC62H  0A74H  PUB  RQ_N_C_RETURN_6
FC62H  0A58H  PUB  RQ_N_C_RETURN_4       FC62H  0A3EH  PUB  RQERROR
FC62H  0A28H  PUB  RQGETLEVEL            FC62H  0A0EH  PUB  RQSIGNALEXCEPTIO
                                                             -N
FC62H  09F0H  PUB  RQWAITINTERRUPT       FC62H  09DAH  PUB  RQSIGNALINTERRUP
                                                             -T
FC62H  09D4H  PUB  RQDELETESEMAPHOR      FC62H  09CEH  PUB  RQDELETEMAILBOX
                   -E
FC62H  09B8H  PUB  RQEXITINTERRUPT       FC62H  09B2H  PUB  RQUNCATALOGOBJEC
                                                             -T
FC62H  09ACH  PUB  RQSENDUNITS           FC62H  09A6H  PUB  RQSUSPENDTASK
FC62H  09A0H  PUB  RQSETPRIORITY         FC62H  099AH  PUB  RQSETPOOLMIN
FC62H  0994H  PUB  RQSETOSEXTENSION      FC62H  098EH  PUB  RQSENDMESSAGE
FC62H  0988H  PUB  RQSLEEP               FC62H  0982H  PUB  RQSETINTERRUPT
FC62H  097CH  PUB  RQSETEXCEPTIONHA      FC62H  0976H  PUB  RQSENDCONTROL
                   -NDLER
FC62H  0970H  PUB  RQRECEIVEUNITS        FC62H  096AH  PUB  RQRESUMETASK
FC62H  0938H  PUB  RQRECEIVEMESSAGE      FC62H  0932H  PUB  RQRESETINTERRUPT
FC62H  092CH  PUB  RQRECEIVECONTROL      FC62H  0926H  PUB  RQOFFSPRING
FC62H  0920H  PUB  RQLOOKUPOBJECT        FC62H  091AH  PUB  RQINSPECTCOMPOSI
                                                             -TE
FC62H  0914H  PUB  RQGETTASKTOKENS       FC62H  090EH  PUB  RQGETTYPE
FC62H  0908H  PUB  RQGETSIZE             FC62H  0902H  PUB  RQGETPRIORITY
FC62H  08FCH  PUB  RQGETPOOLATTRIB       FC62H  08F6H  PUB  RQGETEXCEPTIONHA
                                                             -NDLER
FC62H  08F0H  PUB  RQFORCEDELETE         FC62H  08EAH  PUB  RQENABLE
FC62H  08D4H  PUB  RQENTERINTERRUPT      FC62H  08CEH  PUB  RQENABLEDELETION
FC62H  08C8H  PUB  RQDELETETASK          FC62H  08C2H  PUB  RQDELETESEGMENT
FC62H  08ACH  PUB  RQDISABLE             FC62H  08A6H  PUB  RQDELETEREGION
FC62H  08A0H  PUB  RQDELETEJOB           FC62H  089AH  PUB  RQDELETEEXTENSIO
                                                             -N
FC62H  0894H  PUB  RQDISABLEDELETIO      FC62H  088EH  PUB  RQDELETECOMPOSIT
                   -N                                        -E
FC62H  0888H  PUB  RQCREATETASK          FC62H  0882H  PUB  RQCREATESEMAPHOR
                                                             -E
FC62H  087CH  PUB  RQCREATESEGMENT       FC62H  0876H  PUB  RQCREATEREGION
FC62H  0870H  PUB  RQCATALOGOBJECT       FC62H  086AH  PUB  RQCREATEMAILBOX
FC62H  0864H  PUB  RQCREATEJOB           FC62H  085EH  PUB  RQCREATEEXTENSIO
                                                             -N
FC62H  0858H  PUB  RQCREATECOMPOSIT      FC62H  0852H  PUB  RQALTERCOMPOSITE
                   -E
FC62H  084CH  PUB  RQACCEPTCONTROL       FC62H  06B5H  PUB  INITTASK

DEMO130:  SYMBOLS AND LINES
FD17H  000CH  SYM  MEMORY                FC62H  0000H  SYM  ASCIICODE
00A7H  0000H  SYM  DATASEGPTR            00A7H  0000H  SYM  DATASEGADDR
00A7H  0004H  SYM  HARDWAREINITTASK      00A7H  0006H  SYM  STATUSTASKTOKEN
                   -TOKEN
00A7H  0008H  SYM  MOTORTASKTOKEN        00A7H  000AH  SYM  TIMETASKTOKEN
00A7H  000CH  SYM  ACHANDLERTOKEN        00A7H  000EH  SYM  CRTOUTTASKTOKEN
00A7H  0010H  SYM  COMMANDTASKTOKEN      00A7H  0012H  SYM  INITTASKTOKEN
00A7H  0014H  SYM  CRTMAILBOXTOKEN       00A7H  0016H  SYM  CRTREGIONTOKEN
FC62H  0084H  SYM  COUT                  STACK  0004H  SYM  CHAR
FC62H  00A1H  SYM  CIN                   FC62H  00B9H  SYM  HARDWAREINITTASK
00A7H  0018H  SYM  HARDINITEXCEPTCO      FC62H  0010H  SYM  PARAM51
                   -DE
00A7H  0040H  SYM  PARAM51INDEX          FC62H  0016H  SYM  SIGNONMESSAGE
00A7H  0041H  SYM  SIGNONINDEX           FC62H  013BH  SYM  STATUSTASK
00A7H  0042H  SYM  STATUSCOUNTER         00A7H  001AH  SYM  STATUSEXCEPTCODE
00A7H  0043H  SYM  CWSTEPDELAY           00A7H  0044H  SYM  CCWSTEPDELAY
00A7H  0045H  SYM  CWPAUSEDELAY          00A7H  0046H  SYM  CCWPAUSEDELAY
FC62H  0172H  SYM  MOTORTASK             00A7H  001CH  SYM  MOTOREXCEPTCODE
00A7H  0047H  SYM  MOTORPOSITION         00A7H  0048H  SYM  MOTORPHASE
FC62H  0039H  SYM  PHASECODE             00A7H  0049H  SYM  ACCYCLECOUNT
FC62H  0256H  SYM  ACHANDLER             00A7H  001EH  SYM  ACEXCEPTCODE
FC62H  029CH  SYM  PROTECTEDCRTOUT       STACK  0006H  SYM  CHAR
STACK  0002H  SYM  CRTEXCEPTCODE         00A7H  004AH  SYM  SECONDCOUNT
00A7H  004BH  SYM  MINUTECOUNT           00A7H  004CH  SYM  HOURCOUNT
FC62H  02CFH  SYM  TIMETASK              00A7H  0020H  SYM  TIMEEXCEPTCODE
FC62H  038BH  SYM  PRINTTOD              00A7H  0022H  SYM  TODMESSAGETOKEN
00A7H  0024H  SYM  TODEXCEPTCODE         00A7H  0026H  SYM  TODSEGMENTOFFSET
00A7H  0028H  SYM  TODSEGMENTBASE        00A7H  0026H  SYM  TODSEGMENTPNTR
FC62H  003DH  SYM  TODTEMPLATE           00A7H  0026H  BAS  TODSTRING
00A7H  004DH  SYM  TODSTRINGINDEX        FC62H  0489H  SYM  PRINTSTATUS
00A7H  002AH  SYM  STATUSMESSAGETOK      00A7H  002CH  SYM  STATUSEXCEPTCODE
                   -EN
```

```
00A7H  002EH  SYM  STATUSSEGMENTOFF        00A7H  0030H  SYM  STATUSSEGMENTBAS
                   -SET                                       -E
00A7H  002EH  SYM  STATUSSEGMENTPNT        FC62H  0059H  SYM  STATUSTEMPLATE
                   -R
00A7H  002EH  BAS  STATUSSTRING            00A7H  004EH  SYM  STATUSSTRINGINDE
                                                              -X
00A7H  004FH  SYM  BITPATTERN              FC62H  052FH  SYM  CRTOUTTASK
00A7H  0050H  SYM  MESSAGELENGTH           00A7H  0032H  SYM  MESSAGETOKEN
00A7H  0034H  SYM  RESPONSETOKEN           00A7H  0036H  SYM  MESSAGEEXCEPTCOD
                                                              -E
00A7H  0038H  SYM  MESSAGESEGMENTOF        00A7H  003AH  SYM  MESSAGESEGMENTBA
                   -FSET                                      -SE
00A7H  0038H  SYM  MESSAGESEGMENTPN        00A7H  0038H  BAS  MESSAGESTRINGCHA
                   -TR                                        -R
FC62H  05AFH  SYM  COMMANDTASK             00A7H  0051H  SYM  CONSOLECHAR

00A7H  003CH  SYM  COMMANDEXCEPTCOD        FC62H  06B5H  SYM  INITTASK  <-- INITIALIZATION TASK STARTING ADDRESS
                   -E

00A7H  003EH  SYM  INITEXCEPTCODE          FC62H  0084H  LIN   302
FC62H  0087H  LIN   304                    FC62H  0093H  LIN   305
FC62H  0096H  LIN   306                    FC62H  009DH  LIN   307
FC62H  00A1H  LIN   308                    FC62H  00A4H  LIN   309
FC62H  00B0H  LIN   310                    FC62H  00B3H  LIN   311
FC62H  00B9H  LIN   312                    FC62H  00B9H  LIN   313
FC62H  00BCH  LIN   319                    FC62H  00C2H  LIN   320
FC62H  00C8H  LIN   321                    FC62H  00CEH  LIN   322
FC62H  00D1H  LIN   323                    FC62H  00E4H  LIN   324
FC62H  00EFH  LIN   325                    FC62H  00F8H  LIN   326
FC62H  010CH  LIN   327                    FC62H  0116H  LIN   328
FC62H  011FH  LIN   329                    FC62H  012CH  LIN   330
FC62H  0139H  LIN   331                    FC62H  013BH  LIN   332
FC62H  013EH  LIN   335                    FC62H  0143H  LIN   336
FC62H  0150H  LIN   337                    FC62H  0150H  LIN   338
FC62H  015CH  LIN   339                    FC62H  0160H  LIN   340
FC62H  016DH  LIN   341                    FC62H  0170H  LIN   342
FC62H  0172H  LIN   344                    FC62H  0175H  LIN   348
FC62H  017AH  LIN   349                    FC62H  017FH  LIN   350
FC62H  0184H  LIN   351                    FC62H  0189H  LIN   352
FC62H  0196H  LIN   353                    FC62H  0196H  LIN   354
FC62H  01A5H  LIN   355                    FC62H  01B0H  LIN   356
FC62H  01BDH  LIN   357                    FC62H  01CDH  LIN   358
FC62H  01D6H  LIN   359                    FC62H  01E6H  LIN   360
FC62H  01F5H  LIN   361                    FC62H  0202H  LIN   362
FC62H  020FH  LIN   363                    FC62H  021FH  LIN   364
FC62H  0228H  LIN   365                    FC62H  0238H  LIN   366
FC62H  023BH  LIN   367                    FC62H  0256H  LIN   369
FC62H  0259H  LIN   371                    FC62H  0266H  LIN   372
FC62H  0270H  LIN   373                    FC62H  0278H  LIN   375
FC62H  027DH  LIN   376                    FC62H  028AH  LIN   377
FC62H  028DH  LIN   378                    FC62H  029AH  LIN   379
FC62H  029CH  LIN   380                    FC62H  02A0H  LIN   383
FC62H  02ACH  LIN   384                    FC62H  02B8H  LIN   385
FC62H  02BBH  LIN   386                    FC62H  02C2H  LIN   387
FC62H  02CAH  LIN   388                    FC62H  02CFH  LIN   390
FC62H  02D2H  LIN   392                    FC62H  02D7H  LIN   393
FC62H  02F3H  LIN   394                    FC62H  0300H  LIN   395
FC62H  030FH  LIN   396                    FC62H  031EH  LIN   397
FC62H  032DH  LIN   398                    FC62H  033AH  LIN   399
FC62H  034EH  LIN   400                    FC62H  0354H  LIN   401
FC62H  035DH  LIN   402                    FC62H  0366H  LIN   403
FC62H  036FH  LIN   404                    FC62H  037CH  LIN   405
FC62H  0389H  LIN   406                    FC62H  038BH  LIN   407
FC62H  038EH  LIN   415                    FC62H  039FH  LIN   416
FC62H  03A7H  LIN   417                    FC62H  03ADH  LIN   418
FC62H  03BEH  LIN   419                    FC62H  03D0H  LIN   420
FC62H  03D9H  LIN   421                    FC62H  03F5H  LIN   422
FC62H  040EH  LIN   423                    FC62H  0427H  LIN   424
FC62H  0440H  LIN   425                    FC62H  0459H  LIN   426
FC62H  0472H  LIN   427                    FC62H  0487H  LIN   428
FC62H  0489H  LIN   429                    FC62H  0489H  LIN   430
FC62H  048CH  LIN   439                    FC62H  049DH  LIN   440
FC62H  04A5H  LIN   441                    FC62H  04ABH  LIN   442
FC62H  04BCH  LIN   443                    FC62H  04CEH  LIN   444
FC62H  04D7H  LIN   445                    FC62H  04DFH  LIN   446
FC62H  04EEH  LIN   447                    FC62H  050BH  LIN   448
FC62H  050FH  LIN   449                    FC62H  0518H  LIN   450
FC62H  052DH  LIN   451                    FC62H  052FH  LIN   452
FC62H  0532H  LIN   460                    FC62H  053FH  LIN   461
FC62H  053FH  LIN   462                    FC62H  055AH  LIN   463
FC62H  0560H  LIN   464                    FC62H  0568H  LIN   465
FC62H  0573H  LIN   466                    FC62H  0588H  LIN   467
FC62H  0592H  LIN   468                    FC62H  059DH  LIN   469
FC62H  05AAH  LIN   470                    FC62H  05ADH  LIN   471
FC62H  05AFH  LIN   472                    FC62H  05B2H  LIN   475
FC62H  05BFH  LIN   476                    FC62H  05BFH  LIN   477
FC62H  05C9H  LIN   478                    FC62H  05D0H  LIN   479
FC62H  05DAH  LIN   480                    FC62H  05E0H  LIN   481
FC62H  05F4H  LIN   483                    FC62H  05FAH  LIN   484
FC62H  0600H  LIN   485                    FC62H  0610H  LIN   486
FC62H  0616H  LIN   487                    FC62H  061CH  LIN   488
FC62H  062CH  LIN   489                    FC62H  063CH  LIN   490
FC62H  064CH  LIN   491                    FC62H  065CH  LIN   492
FC62H  066CH  LIN   493                    FC62H  067CH  LIN   494
FC62H  068CH  LIN   495                    FC62H  069CH  LIN   496
FC62H  06B0H  LIN   498                    FC62H  06B3H  LIN   499
FC62H  06B5H  LIN   500                    FC62H  06B8H  LIN   502
```

```
FC62H  06C4H  LIN    503              FC62H  06D5H  LIN    504
FC62H  06E6H  LIN    505              FC62H  06F6H  LIN    506
FC62H  071FH  LIN    507              FC62H  072CH  LIN    508
FC62H  0755H  LIN    509              FC62H  0762H  LIN    510
FC62H  078BH  LIN    511              FC62H  0798H  LIN    512
FC62H  07C1H  LIN    513              FC62H  07CEH  LIN    514
FC62H  07F7H  LIN    515              FC62H  0804H  LIN    516
FC62H  082DH  LIN    517              FC62H  083AH  LIN    518
FC62H  083DH  LIN    519              FC62H  084AH  LIN    520
FC62H  0084H  LIN    521

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

SEGMENT MAP

START     STOP      LENGTH ALIGN NAME              CLASS

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

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

FC620H    FD17BH    0B5CH    W    CODE             CODE   <--- LAST CODE BYTE OF APPLICATION JOB

FD17CH    FD17CH    0000H    W    MEMORY           MEMORY


GROUP MAP

ADDRESS   GROUP OR SEGMENT NAME
FC620H    CGROUP
          CODE
00A70H    DGROUP
          DATA
```

APPENDIX F
ROOT JOB LOCATE MAP

```
ISIS-II MCS-86 LOCATER, V1.2 INVOKED BY:
LOC86 :f1:RJB130.lnk                &
      TO :F1:RJB130                 &
      MAP PRINT(:f1:RJB130.mp2)     &
      OC(noli, nopl, nocm, nosb)    &
      PC(noli, pl, nocm, nosb)      &
      SEGSIZE(stack(O))             &
      ORDER(classes(data, stack, memory))       &
      ADDRESSES(classes(code(OFD180H),          &
                     data(OOADOH)))
WARNING 26:  DECREASING SIZE OF SEGMENT
  SEGMENT:  STACK

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

BASE   OFFSET TYPE SYMBOL                BASE   OFFSET TYPE SYMBOL

FD18H  0180H  PUB  NUC_INIT_ENTRY        FD18H  0184H  PUB  CODEDATA

FD18H  0011H  PUB  RQSTARTADDRESS  <---- ROOT JOB STARTING ADDRESS   FD18H  0010H  PUB  INTERROR

FD18H  0000H  PUB  CRASH                 FD18H  002AH  PUB  RQROOTJOBVERSION
FD18H  0030H  PUB  ROOTTASK              FD18H  010CH  PUB  SYSTEMSUICIDE
FD18H  0118H  PUB  RQCREATEJOB           FD18H  011EH  PUB  RQGETTASKTOKENS
FD18H  0124H  PUB  RQSUSPENDTASK         FD18H  012AH  PUB  RQ_N_C_RETURN_6
FD18H  0146H  PUB  RQ_N_C_RETURN_40      FD18H  0162H  PUB  RQERROR
00ADH  0000H  PUB  JOBNUMBER             00ADH  0002H  PUB  ROOTTASKSTATUS

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

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

START     STOP      LENGTH ALIGN NAME            CLASS

00AD0H    00AD3H     0004H   W   DATA            DATA

00AD4H    00BFFH     012CH   W   INIT_STACK      STACK  <---- LAST DATA BYTE OF ROOT JOB

00C00H    00C00H     0000H   W   STACK           STACK
00C00H    00C00H     0000H   G   ??SEG
FD180H    FD339H     01BAH   W   CODE            CODE
FD33AH    FD345H     000CH   W   SAB_DESCRIPTOR  CODE
                                 -S

FD346H    FD366H     0021H   W   U_J_DESCRIPTOR  CODE   <---- LAST CODE BYTE OF ROOT JOB
                                 -S

FD368H    FD368H     0000H   W   MEMORY          MEMORY


GROUP MAP

ADDRESS  GROUP OR SEGMENT NAME
00AD0H   DGROUP
         DATA
FD180H   CGROUP
         CODE
         SAB_DESCRIPTORS
         U_J_DESCRIPTORS
```

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, CA 95051 (408) 987-8080

Printed in U.S.A./T-520/20K/0282/CP JL